國立交通大學

資訊工程學系

碩 士 論 文

Linux 平台上驅動程式層網路事件通知機制之設計與實作

Design and Implementation of Driver-Level Network Event Notification Mechanism in Linux

研 究 生：周大鈞

指導教授：曾建超 教授

中 華 民 國 九 十 四 年 六 月

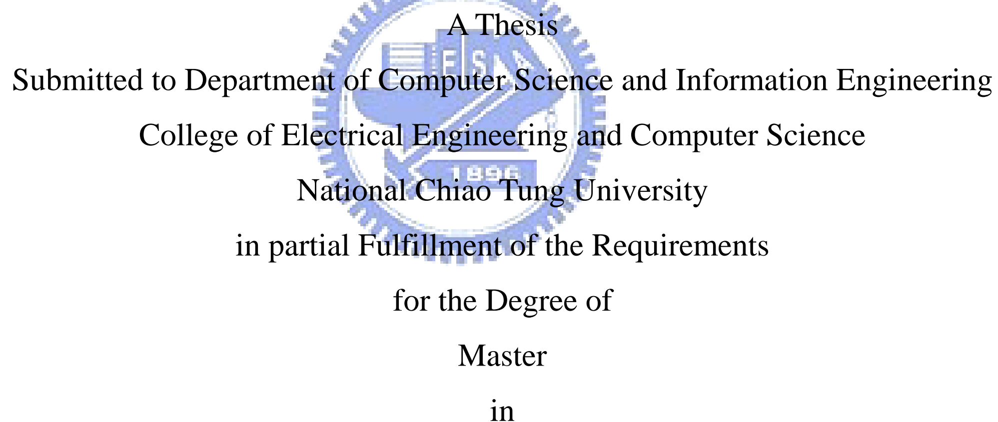
Linux 平台上驅動程式層網路事件通知機制之設計與實作

Design and Implementation of Driver-Level Event

研 究 生：周大鈞　　Student：Da-Juan Chou

指導教授：曾建超　　Advisor：Chien-Chao Tseng

國 立 交 通 大 學
資 訊 工 程 系
碩 士 論 文



A Thesis
Submitted to Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

中華民國九十四年六月

Linux 平台上驅動程式層網路事件通知機制之設計與實作

研究生：周大鈞　　　　　　　　　　　　　　　　　　　指導教授：曾建超

國立交通大學資訊工程學系碩士班

摘　　要

近年由於各種無線網路(GPRS、3G、Wireless LAN 等)接取技術的成熟，在行動設備(例如手機、個人數位助理或是筆記型電腦)上同時配備多種無線網路接取介面也成為潮流，但是，由於各種無線網路接取技術的特性不同，因此，我們通常需要一套介面管理的機制使得使用者能夠在不同環境下，根據自己的喜好設定，選擇適當的網路接取方法。

介面管理通常在應用層(Application Layer)處理，傳統上，應用層的介面管理程式會週期性的發出系統呼叫，以取得底層介面的資訊(例如網路介面的種類、頻寬或連線狀況)，但是，由於網路介面的狀態改變可能不會如此的頻繁，因此頻繁的發出系統呼叫可能會造成系統多餘的負荷，此外，在需要快速換手的情況下，介面管理程式必須對網路介面的狀態改變做出迅速的反應，而在傳統的狀況下，網路介面狀態更新的頻率取決於程式發出系統呼叫的頻率，而且受到作業系統排程的影響，因此可能無法對網路設備狀態的改變作出即時的處理。

為了解決上述的問題，我們設計並且實作了一套「驅動程式層網路事件通知機制」，將網路介面相關的狀態改變利用類似傳統 UNIX 作業系統上的信號(signal)機制通知使用者空間的程式，並且修改了排程的演算法，使得使用者空間程式不必再發出無意義的系統呼叫，加快對於網路介面狀態

改變的處理，增進系統效能，並且使得利用本論文提出的機制所實作出來的應用程式更適用於多網路的環境之下。

Design and Implementation of Driver-Level Network Event Notification Mechanism in Linux

Student：Ta-Juan Chou Advisor：Dr. Chien-Chao Tseng

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

As the wireless technologies advance, it is now a trend to equip a handheld device with multiple wireless interfaces of different wireless accessing technologies, such as GPRS, 3G, and Wireless LAN. Because different wireless access technologies have different communication properties and constraints, the multi-interfaced mobile devices need an interface manager that can select the most appropriate interface according to the user preferences and statuses of wireless networks.

Current Linux operating system is insufficient for the interface manager to react promptly according to the interface statuses. Interface management is normally accomplished by user programs running in the application layer. Therefore, application programs that manage interfaces need to issue system calls periodically to retrieve interface statuses and reacts to the changes accordingly. However, interface statuses may remain the same for a long period but change suddenly depending on the network deployment and environmental conditions. If the interval of system calls is short then these system calls will result in wasting system resources for useless information. On the other hand, if the system call interval is long then interface management applications may not react promptly to the changes of interface statuses. Furthermore, even with frequent system calls, interface management applications may still not be able to

gather the statuses in time due to the scheduling policy of the operating system. Therefore in order for interface management applications to react promptly to the changes in interface statuses, some notification mechanism is required for the operating system to notify interface management applications of the changes in interface statuses.

In this thesis, we proposed and implemented a new mechanism called "Driver-level Network Event Notification Mechanism" for the Linux operating system. With the mechanism, the Linux operating system can notify user-mode processes of the changes in interfaces statuses to eliminate intensive system calls. The mechanism also provides a signal-like method in UNIX for processes to register handler functions. In order to speedup the handling of the interface status changes, we also modify the scheduling process of the Linux kernel. Experimental results show that our mechanism can indeed shorten the time an interface management program reacts to the changes in interface statuses and eliminate unnecessary system calls invoked by the interface management program.

誌　　　謝

首先我要感謝我的指導教授—曾建超博士，提供這篇論文的研究方向以及提供我一個良好且自由的研究環境，此外，也要感謝楊人順學長在我論文寫作期間給我的建議以及支持，最後，還要感謝實驗室的同學、學長姐以及學弟妹在我碩士生涯中給我支持及鼓勵，謝謝你們。

此外，我要感謝我的家人，他們在我最困難的時候給我的支持是我奮鬥下去的原動力。

目錄

中文摘要……………………………………………………………………………i
英文摘要…………………………………………………………………………..iii
誌謝……………………………………………………………………………….v
目錄…………………………………………………………………………..vi
圖目錄……………………………………………………………………………viii
表目錄………………………………………………………………………………x
第一章 緒論 1
1.1 研究動機..1
1.2 研究目標..2
1.3 章節簡介..2
第二章 背景與相關研究 4
2.1 POSIX信號 4
2.1.1 傳統信號 4
2.1.2 即時信號 5
2.2.3 信號的產生以及傳達 5
2.2 IOCTL系統呼叫 5
2.3 網路驅動程式簡介...8
2.4 行程敘述子簡介..10
2.5 核心模式堆疊與使用者模式堆疊..12
2.6 通知者串鍊...13
2.7 多網路的整合...15
2.8 HUT VHO DAEMON...16
第三章 驅動程式層網路事件通知機制設計與架構 18
3.1 目標與問題定義...18
3.2 系統概觀...19
3.3 系統呼叫的設計...20
3.4 網路相關事件之定義..20
3.4.1 驅動程式和協定堆疊間的事件 21
3.4.2 其它使用者層關心的事件 21
3.5 網路事件的紀錄...22
3.6 事件的產生...22
3.7 事件的傳達...23
3.7.1 使用者和核心空間之間的切換 23
3.7.2 將事件即時傳達至使用者空間程式 24
第四章 驅動程式層網路事件通知機制之實作 26
4.1 軟硬體需求...26

4.2 事件及其相關處理的實作......26
4.2.1 網路位址改變事件之網路通知者串鍊支援......27
4.2.2 載體出現與消失事件之網路通知者串鍊支援......27
4.2.3 事件的格式化以及記錄......27
4.3 行程管理實作......29
4.4 事件處理之時機......31
4.4.1 檢查事件的發生......31
4.4.2 排程器的修改......34
4.5 系統呼叫的重新執行......36
4.6 使用者和核心的切換......39
第五章 操作實例......43
5.1 操作實例......43
第六章 結論與未來工作......51
6.1 結論......51
6.2 未來工作......51
參考文獻......53

圖目錄

Fig.2- 1 ioctl之宣告 5
Fig.2- 2 ioctl程式範例 6
Fig.2- 3 ifreq資料結構 7
Fig.2- 4 狀態基礎和事件基礎機制的差異 8
Fig.2- 5 Linux網路介面架構 9
Fig.2- 6 行程描述子 11
Fig.2- 7 thread_info資料結構 12
Fig.2- 8 核心堆疊 13
Fig.2- 9 notifier block資料結構 14
Fig.2- 10 netdev_chain示意圖 14
Fig.2- 11 跨層設計概念示意圖 15
Fig.2- 12 HUT VHO Daemon之系統架構 17

Fig.3- 1 系統元件架構和關係 20
Fig.3- 2 系統呼叫相關宣告 20
Fig.3- 3 一般在核心空間時的的核心模式堆疊和使用者模式堆疊 24
Fig.3- 4 欲執行回叫函數前的核心模式堆疊和使用者模式堆疊 24
Fig.3- 5 中央處理器控制權的交替實例 25

Fig.4- 1 事件區塊資料結構 27
Fig.4- 2 事件區塊的記錄方式 28
Fig.4- 3 事件處理者之流程 29
Fig.4- 4 對行程敘述子的修改 30
Fig.4- 5 事件的發生及儲存之實例 31
Fig.4- 6 偵測是否有未處理事件的程式碼 32
Fig.4- 7 未修改前由核心返回的流程圖 33
Fig.4- 8 修改過後的核心返回流程圖 34
Fig.4- 9 修改過後的排程演算法 35
Fig.4- 10 將行程加入某個等待佇列的程式碼片段。 36
Fig.4- 11 signal_pending()函數之修改。 38
Fig.4- 12 重新執行系統呼叫。 38
Fig.4- 13 沒有註冊處理函數時的系統呼叫重新執行。 39
Fig.4- 14 正常狀態下的核心模式堆疊和使用者模式堆疊 40
Fig.4- 15 執行回叫函式前的核心模式堆疊和使用者模式堆疊 41
Fig.4- 16 置放於堆疊中的程式返回碼 42

Fig.5- 1 「驅動程式層網路事件通知機制」範例程式。 43
Fig.5- 2 「驅動程式層網路事件通知機制」的標頭檔。 44
Fig.5- 3 範例程式開始執行 45
Fig.5- 4 範例程式註冊兩個處理函數 45
Fig.5- 5 啓用網路介面wlan0 46

Fig.5- 6 介面改變傳達至範例程式，印出啓用的介面 47
Fig.5- 7 停用網路介面wlan0....48
Fig.5- 8 介面改變傳達至範例程式，印出停用的介面 48
Fig.5- 9 按下Control-C，程式停止執行 49
Fig.5- 10 範例程式註冊的處理函數被移除 50

表目錄

Table.2- 1 事件處理函數回傳值 15

Table.3- 1 系統內定義的網路驅動程式事件 21

Table.4- 1 系統定義事件和網路通知者串鍊定義事件交互參照 26
Table.4- 2 行程對信號動作和系統呼叫回傳值對系統呼叫執行影響之列表 37

第一章 緒論

1.1 研究動機

近年來 WLAN、GPRS、3G 等等無線網路接取技術漸漸成熟，使用者利用無線網路在各種公共場合中存取網際網路上資源已然成爲未來的趨勢，除此之外，使用者的網路接取裝置也常配置多種無線網路的接取設備，這些設備通常具有可動態移除的特性 (例如 PCMCIA 網路介面卡、SD 網路介面卡等) ，此外，不同的網路接取技術有其不同的優缺點，舉例來說，無線網路具有傳輸速率高的優點，但其涵蓋範圍卻較小；GPRS 和 3G 等電信網路無線接取技術雖然有較大的涵蓋範圍，但是其傳輸速率卻較低。

由於我們之前提到的動態移除特性以及其涵蓋範圍和傳輸速率上的限制，因此，網路介面的管理成爲行動性支援的一項重要議題，舉例來說，當使用者插入新的網路介面卡或是移除舊有的網路介面時，系統內的行動管理員必須能要及時的得知並且爲這些網路介面套用網路設定。

目前在 Linux 的系統中得知網路介面相關資訊的方法主要是利用 ioctl[1]系統呼叫對下層的驅動程式下達命令，使用這個方法有以下的缺點：

- 無法得知網路介面的新增

  使用 ioctl 系統呼叫必須先得知網路介面的相關資訊，例如網路介面的名稱等等，而網路介面在新增之前我們無法得知其網路介面的名稱，因此我們也無法利用 ioctl 得知網路介面的出現。

- 輪詢化的程式結構

  ioctl 系統呼叫必須由使用者空間程式下達，因此程式想要及時的對某些網路介面的事件反應，則必須使用類似輪詢的方法，在迴圈內不停的使用 ioctl 系統呼叫。

- 需要紀錄網路介面的相關資訊

  由於使用 ioctl 系統呼叫只能得到目前網路介面的資訊，因此若是要知道網路介面的狀態和之前的狀態有何不同，程式必須記錄下網路介面之前的資訊作爲比對的依據，如此一來，使得程式的結構複雜化。

1.2 研究目標

本論文主要的目標如以下所示：

- 將網路介面發生的事件傳達給使用者空間的程式。
- 當事件發生時由核心直接傳達至使用者空間。
- 事件必須不可遺失，且保留事件的先後發生順序。
- 盡快的將事件的發生傳達至使用者空間。

根據以上的目標，本論文提出了一個可用於整合多個網路介面的應用程式平台，我們將核心內定義的各種網路驅動程式傳達至是用者空間的程式，此外，並額外定義了一些使用者空間行程感到興趣可是目前核心內定義沒有實作的事件，使得多網路介面管理的支援更加完整。

在核心內的事件傳遞我們利用並且擴充了原本核心內用來通知事件發生的通知者串鍊機制 (notifier_chain) 來達到，當有事件發生的時候利用該機制以通知我們所實作的事件處理者程式。

關於使用者空間的行程如何得到事件發生的通知我們是採用傳統信號的作法，由使用者對於其感興趣的事件作註冊，並且傳入一個函式指標作爲回叫函式，當系統有事件發生時，便會自動執行該行程的回叫函式。

傳統信號爲不可靠的信號，爲了要讓重複發生的相同事件能夠傳達至使用者空間，我們實作了事件的處理者，該事件的處理者找到對該事件感興趣的行程，並且封裝以及記錄事件的發生。

爲了要能夠使使用者空間的事件能夠盡快的對其感興趣之事件的發生做出相對應的處理，我們修改了核心的程式碼中由核心空間 (kernel space) 返回使用者空間 (user space) 的程式碼以及排程器的處理，使得事件的傳達能夠盡快達成，並且給予註冊過的較高的程式執行權。

1.3 章節簡介

本篇論文各章節簡述如下：

第一章： 描述本篇論文的研究動機，以及本篇論文希望達到的目標。

第二章： 說明本篇論文中相關的研究背景，包含目前信號機制、ioctl 系統呼叫的

介紹、網路驅動程式和行程管理的簡介以及網路整合相關討論及研究。

第三章： 提出本論文中各個元件的設計理念以及其欲達到的功能。

第四章： 介紹我們實作本系統的各個細節，包含了軟硬體架構、新增資料結構和修改過的系統元件及修改過後的演算法。

第五章： 執行結果。

第六章： 對本論文做出總結，以及未來的研究方向。

第二章 背景與相關研究

本章中將介紹和本論文相關的知識背景以及相關的研究，首先將介紹在 POSIX[6] 定義的傳統信號和即時信號的機制；接下來介紹 ioctl 系統呼叫，這是 UNIX 系統中用來取得下層驅動程式資訊的標準界面；然後則是 Linux 中網路驅動程式架構以及行程管理的簡介；接著介紹 Linux 系統中通知者串鍊機制，然後則是一些網路整合以及跨層設計 (cross layer design) 的相關討論，最後介紹稱為 HUT VHO DAEMON[8]，這是由赫爾辛基大學所發展的異質網路整合平台，我們將介紹其作法以及和我們平台的異同。

2.1 POSIX 信號

2.1.1 傳統信號

在 Linux 系統以及其他一般 UNIX 系統中的標準事件通知機制以及介面為 POSIX 信號，POSIX 信號被定義於為 POSIX.1b [9]中。信號機制提供了使用者一個類似事件驅動的環境，由系統或是其他使用者層的行程發出信號，以告知一些特殊狀況的發生。

在傳統信號機制中，使用者對於信號的發生可能有以下三種處理方式：

- 忽略該信號。
- 停止執行。
- 執行先前註冊過的信號處理函式。

不同的信號有不同的預設處理方式，除了 SIGKILL 以及 SIGSTOP 之外，使用者程式可以利用 sigaction()[2]系統呼叫改變預設的信號處理方式，除此之外，使用者也可以利用 sigprocmask()[3]系統呼叫來指定哪些信號被阻擋 (blocked) 。

大部分傳統信號中的每個信號都已經代表了系統的某些事件，舉例來說，當程式收到 SIGILL 信號時代表程式執行了不合法的指令，而當程式接收到了 SIGINT 則表示由鍵盤接收到了 CTRL+C；註冊的信號處理函式除了得到一個數字以辨識發生的信號原因之外，並不會得到額外的訊息資訊。

在 Linux 系統下，傳統信號指的是信號代碼大於 0 且小於 SIGRTMIN (32) 的信號，傳統信號的缺點有以下幾個[10]：

- 大部分的信號都有其專門的用途，用戶無法自行定義。
- 信號沒有附加信息，僅能得知某個事件的發生。

- 爲不可靠信號，重複發生的信號可能會遺失。

2.1.2 即時信號

早期的 UNIX 系統只支援傳統信號，而 Linux 在 2.3 版本以後的核心中也支援了 POSIX 1.b[9]標準標準所訂定的即時信號 (Real-Time Signals, RT signals) ，在 Linux 系統中信號代碼大於 SIGRTMIN (32) 且小於 SIGRTMAX (63) 的爲即時信號，即時信號改善了傳統信號的兩大缺點：

- 即時信號的信號不會遺失。
- 即時信號沒有特定的用途，使用者可以自由使用。

即時信號通常會和某個檔案描述子結合使用以偵測是否該檔案描述子的狀態是否有改變，當檔案敘述子的狀態有改變的時候便會驅動與該檔案敘述子結合 (bind) 的即時信號，達到和 select()[4]以及 poll()[5]系統呼叫相同的效果。

關於即時信號的應用最有名的就是 phhttpd[7]網頁伺服器加速器，phhttpd 處理靜態網頁的要求，並且將其它的要求轉交由 Apache 等網頁伺服器處理。

2.2.3 信號的產生以及傳達

一般來說，在核心內信號的傳送我們可以分爲兩個步驟，第一步驟爲「信號的產生」，第二步驟稱爲「信號的傳達」[10]；當系統有某些事件發生時，作業系統會更新核心內行程的資料結構，代表著某個信號已經發生，這個步驟稱爲「信號的產生」。

而「信號的傳達」則是指當行程對信號做出反應，例如核心改變行程的狀態，或者是核心改變行程的硬體環境，以執行系統的信號處理函式，已經產生，但尚未傳達的信號稱爲未決信號 (pending signal) ，在傳統信號當中，每個種類的信號只能有一個未決信號，在即時信號以及本系統的架構中，同種類的信號可以存在多個未決信號。

2.2 ioctl 系統呼叫

ioctl()在系統中的宣告如圖 Fig.2-1 所示。

```
#include        <sys/ioctl.h>
int ioctl(int fd, int request, ...);
```

Fig.2- 1 ioctl 之宣告

ioctl 系統呼叫處理和某個檔案相關的底層設備參數。在圖 Fig.2-1 中的 fd 是一個檔案描述子，request 則是和設備相依的一個命令碼，最後一個參數一般來說是一個 char *

的指標，指向一個記憶體位置。

圖 Fig.2-2 爲 ioctl 的使用範例，本範例中的函數將傳數的參數 flag 設定至名稱爲 ifname 的網路設備驅動程式的 flags 欄位，而 struct ifreq 則是一個宣告在 /usr/include/net/ifreq.h 的資料結構，網路相關的 ioctl 控制都是利用這個資料結構達到的，除此之外，設定網路卡的 IP 位址，網路遮罩等等也都經由 ioctl 系統呼叫成。

```
static int set_flag(char *ifname, short flag)
{
   struct ifreq ifr;

   safe_strncpy(ifr.ifr_name, ifname, IFNAMSIZ);
   if (ioctl(skfd, SIOCGIFFLAGS, &ifr) < 0) {
         fprintf(stderr, _("%s: unknown interface: %s\n"),
                  ifname,strerror(errno));
         return (-1);
   }
   safe_strncpy(ifr.ifr_name, ifname, IFNAMSIZ);
   ifr.ifr_flags |= flag;
   if (ioctl(skfd, SIOCSIFFLAGS, &ifr) < 0) {
         perror("SIOCSIFFLAGS");
         return -1;
   }
   return (0);
}
```

Fig.2- 2 ioctl 程式範例

圖 Fig.2-3 中顯示了 ifreq 資料結構的詳細內容，例如當要下達 ioctl 給下層的驅動程式前，必須設定好其中的 ifrn_name 欄位，這個欄位應該被設定爲所要下達 ioctl 的網路介面名稱，若是要設定網路介面的話，除了網路介面的名稱之外，還要依據欲設定的資訊填入 ifr.ifr_ifru 中的特定欄位，若是要由下層設備取得其狀態的話，則依據不同的命令代碼，核心會將所感興趣的參數填入特定的欄位。

```
struct ifreq
  {
# define IFHWADDRLEN    6
# define IFNAMSIZ   IF_NAMESIZE
    union
      {
        char ifrn_name[IFNAMSIZ]; /* Interface name, e.g. "en0".  */
      } ifr_ifrn;

    union
      {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        struct sockaddr ifru_broadaddr;
        struct sockaddr ifru_netmask;
        struct sockaddr ifru_hwaddr;
        short int ifru_flags;
        int ifru_ivalue;
        int ifru_mtu;
        struct ifmap ifru_map;
        char ifru_slave[IFNAMSIZ]; /* Just fits the size */
        char ifru_newname[IFNAMSIZ];
        __caddr_t ifru_data;
      } ifr_ifru;
  };
```

Fig.2- 3 ifreq 資料結構

一般而言，當常駐的應用程式想要得知網路設備的狀態是否有改變的話，該應用程式就必須要定期的對其感興趣的網路介面下達特定的 ioctl 命令，並且在程式中維護一份網路介面的資訊，以便得知網路介面的改變，類似輪詢的作法，爲狀態基礎 (state-based) 的方法。

本論文所使用的方法則是使用事件爲基礎的方法，當網路設備的狀態有改變的時候，由核心主動使用類似信號的機制告知應用程式事件的發生，這兩者的不同點可由圖 Fig.2-4 顯示。

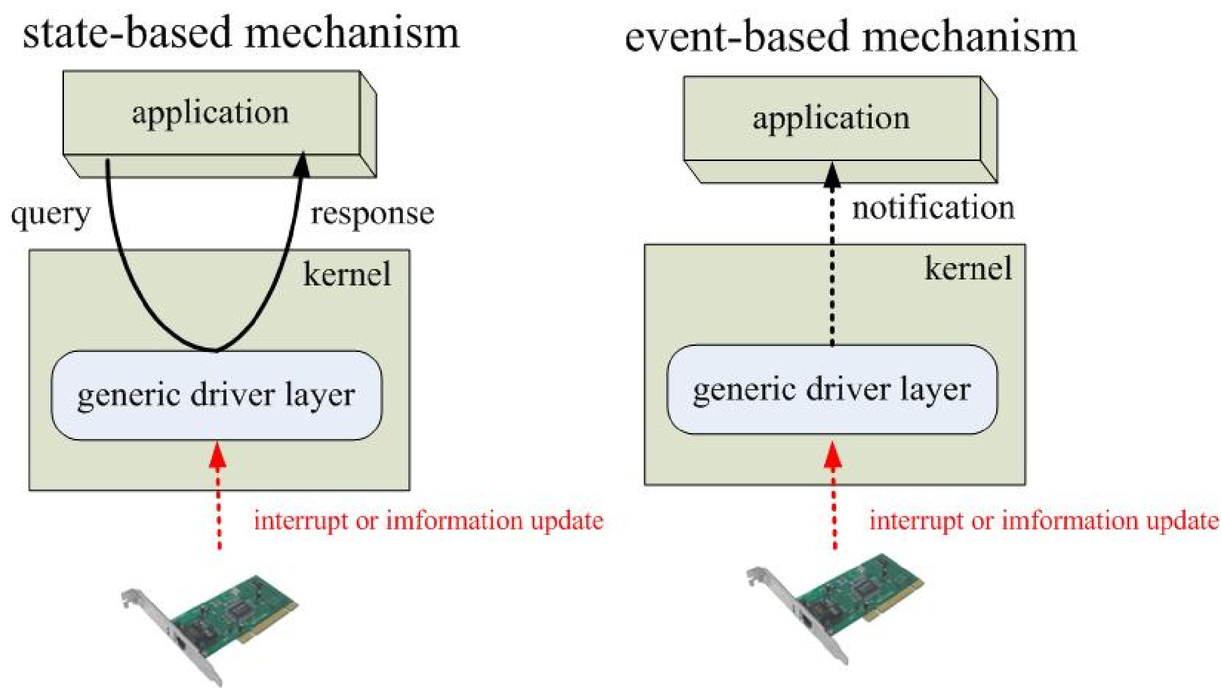


Fig.2- 4 狀態基礎和事件基礎機制的差異

2.3 網路驅動程式簡介

Linux 中，驅動程式主要有兩個作用[11]：

1. 抽象化硬體的技術細節。
2. 提供上層協定一個統一的介面。

驅動程式在核心內的架構如圖 Fig.2-5 所示，在 Linux 中網路驅動程式被實體化為 net_device 資料結構，一般而言，每個網路介面對應到一個 net_device 資料結構，網路驅動程式和一般驅動程式最大的不同點在於在/dev 檔案系統中並不會出現和網路驅動程式對應的檔案；在圖中我們可以看到，當上層的協定堆疊需要下層驅動程式的服務時，上層協定會先呼叫經由通用驅動程式層（定義於核心程式碼/net/core/dev.c 中）的統一界面，例如 dev_close()或是 dev_open()，而通用程式層的函數則會依據 net_device 資料結構中的資訊呼叫驅動程式的相關函式，最後呼叫到驅動程式內的函式，進行硬體相關的處理。

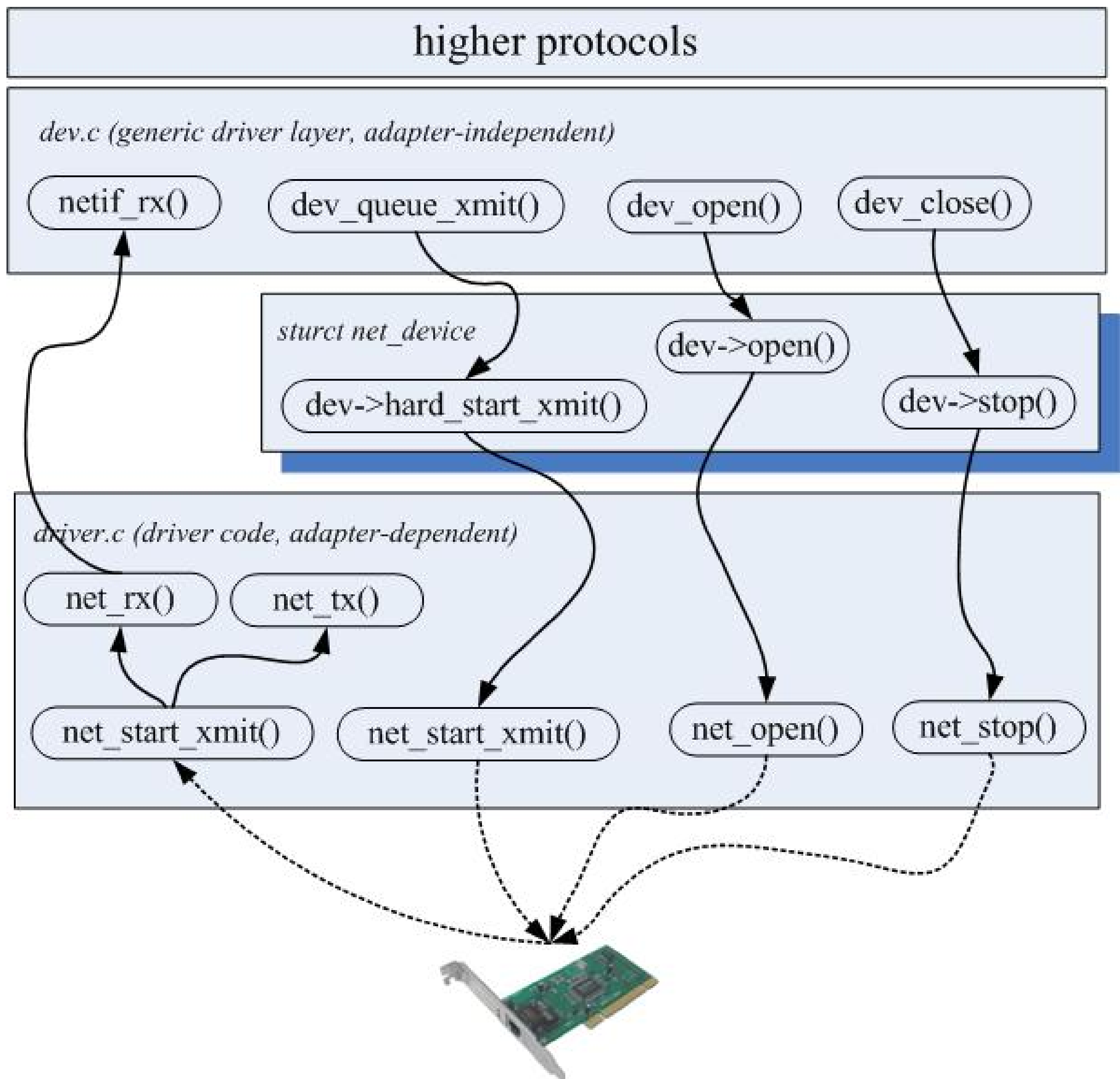


Fig.2- 5 Linux 網路介面架構

由於網路驅動程式不會有對應的設備檔案，因此，當網路驅動程式被插入核心中時，其對應的資料結構 net_device 是由一個靜態宣告的串列 dev_base 所紀錄；net_device 資料結構被定義在/include/linux/netdevice.h 中，其記錄的資訊十分複雜，可以大致上分類為以下幾大部分：

- 一般欄位：

  記錄了一般性的資訊，這些資訊和網路堆疊以及硬體沒有直接相關性，多半是核心用來管理網路介面，這些欄位包含了 **name** (網路設備名稱) ，**next** (指向下一個網路驅動程式的指標) ，**ifindex** (名稱之外的另一個網路識別) 以及 **state** (目前驅動程式的狀態) 等等。

- 硬體相關欄位：

包含了和硬體設施相關的欄位，其中包含了 **mem_start**、**mem_end** (共享的記憶體位置) ，**base_addr** (I/O 位址) ，**irq** (irq 代碼) ，**dma** (dma 通道數) 等欄位。

- 實體層相關欄位：

  包含一些實體層的相關欄位，就特定種類的網路卡而言 (例如以太網路) ，這一個分類的值通常都是相同的，其中包含了 **hard_header_length** (鍊結層封包表頭的長度) ，**mtu** (最大傳輸單位) ，**tx_queue_len** (輸出串列的長度) ，**type** (硬體種類) ，**dev_addr[MAX_ADDR_LEN]** (硬體位置) ，**broadcast[MAX_ADDR_LEN]** (廣播位置) 等欄位。

- 網路層相關欄位：

  網路層相關資訊記錄於這裡，通常是一個資料指標，其指向的資料隨著網路層的協定不同而不同，包含的欄位有 **ip_ptr** (IP 網路協定相關資料) ，**ip6_ptr** (Ipv6 網路協定相關資料) ，**ax25_ptr** (AX.25 網路協定相關資料) 等欄位。

- 驅動程式處理函式指標：

  包含了一些設備相關函式或是硬體相關函式的指標，包括 **init()**，**uninit()**，**open()**，**close()**，**hard_start_xmit()**，**do_ioctl()**等等。

2.4 行程敘述子簡介

行程為正在執行的程式，對於核心來說，行程代表著一個核心必須為其配置資源的實體 (例如中央處理器時間以及記憶體) ，除此之外，核心也必須維護行程的一些資訊，例如未處理的信號，位址空間，以及開啓的檔案等等。

為了要管理行程，作業系統必須清楚的知道行程的狀態以及其資源的配置，Linux 將這些資訊儲存在 task_struct 資料結構 (定義於核心程式碼 include/linux/sched.h) 內，我們稱之為行程描述子 (process descriptor) ，該資料結構儲存了所有行程相關的資訊，例如行程狀態、排程資訊以及行程關係的連結等等，因此是個龐大而複雜的資料結構，除了和行程相關的屬性之外，該資料結構還包含了許多指標，指向許多資源的描述子，其示意圖可參照圖 Fig.2-6。

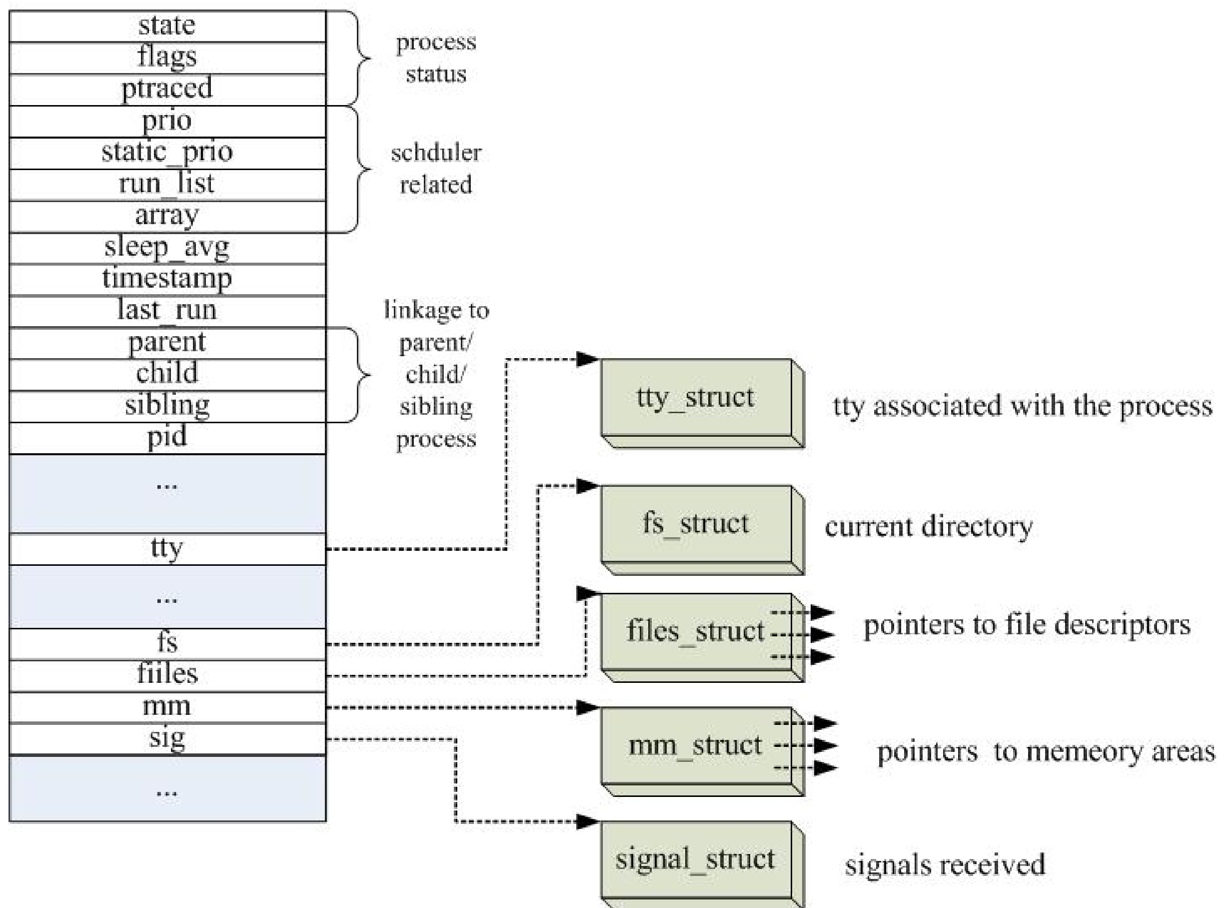


Fig.2- 6 行程描述子

其中的 state 欄位代表著行程目前的狀態，其值可能爲：

- TASK_RUNNING：此時行程正在執行或者是等待被排程器選擇執行
- TASK_INTERRUPTIBLE：此時行程爲不可執行的狀態並且等待某個狀況發生。此時，中斷發生、等待的資源已經準備完畢或者是信號的傳達都可能使其狀態回復至 TASK_RUNNING，在本論文的系統中，當網路驅動程式事件發生時，也會造成系統狀態回複至 TASK_RUNNING。
- TASK_UNINTERRUPTIBLE：和 TASK_INTERRUPTIBLE 狀態相同，除了此時信號的傳達不會將狀態回復至 TASK_RUNNING。
- TASK_STOPPED：行程已經停止，當行程收到 SIGSTOP、SIGTSTP、SIGTTIN 以及 SIGTTOU 時會進入此一狀態。
- TASK_ZOMBIE：行程已停止執行但其父行程尚未執行 wait()或是其他和 wait (例如 waitpid()、wait3()、wait4()) 功能相同的系統呼叫清除其對於已停止行程的資訊。

2.5 核心模式堆疊與使用者模式堆疊

在 Linux 系統中，每個應用程式都會有兩個堆疊，一個稱爲使用者模式堆疊，另外一個則是核心模式堆疊，使用者模式堆疊就是應用程式在使用者空間執行的時候所使用的堆疊，而核心模式堆疊則是當應用程式進入核心空間時所使用的堆疊，當應用程式由使用者空間進入核心空間時，SP 暫存器的值從使用者模式堆疊頂端的記憶體位置被置換爲核心模式堆疊的頂端的記憶體位址。

在 2.6 版本的核心之中，和過去不同的地方是，在過去版本的核心中，核心模式堆疊的底端記錄的是該行程的行程描述子，而在新版本的核心之中，核心堆疊的底端只放置了進入核心或返回使用者空間時必要的資料結構，這些資料結構在核心內被放在行程敘述子的 thread_info 資料結構內，該資料結構可見圖 Fig.2-7。

```
struct thread_info {
        struct task_struct      *task;              /* main task structure */
        struct exec_domain      *exec_domain;       /* execution domain */
        unsigned long           flags;              /* low level flags */
        unsigned long           status;             /* thread-synchronous flags */
        __u32                   cpu;                /* current CPU */
        __s32                   preempt_count;      /* 0 => preemptable, <0 => BUG */


        mm_segment_t            addr_limit;         /* thread address space:
                                                    0-0xBFFFFFFF for user-thead
                                                     0-0xFFFFFFFF for kernel-thread
                                                    */
        struct restart_block    restart_block;

        unsigned long           previous_esp;       /* ESP of the previous stack in case
                                                    of nested (IRQ) stacks
                                                    */
        __u8                    supervisor_stack[0];
};
```

Fig.2- 7 thread_info 資料結構

核心模式堆疊主要的作用有以下三點：

- 紀錄行程描述子中的線行程資訊 (thread_info) 。
- 儲存行程硬體環境 (hardware context) 。
- 核心函數呼叫及一般使用。

一般而言，核心堆疊的大小爲 8k，其配置如圖 Fig.2-8 所示。

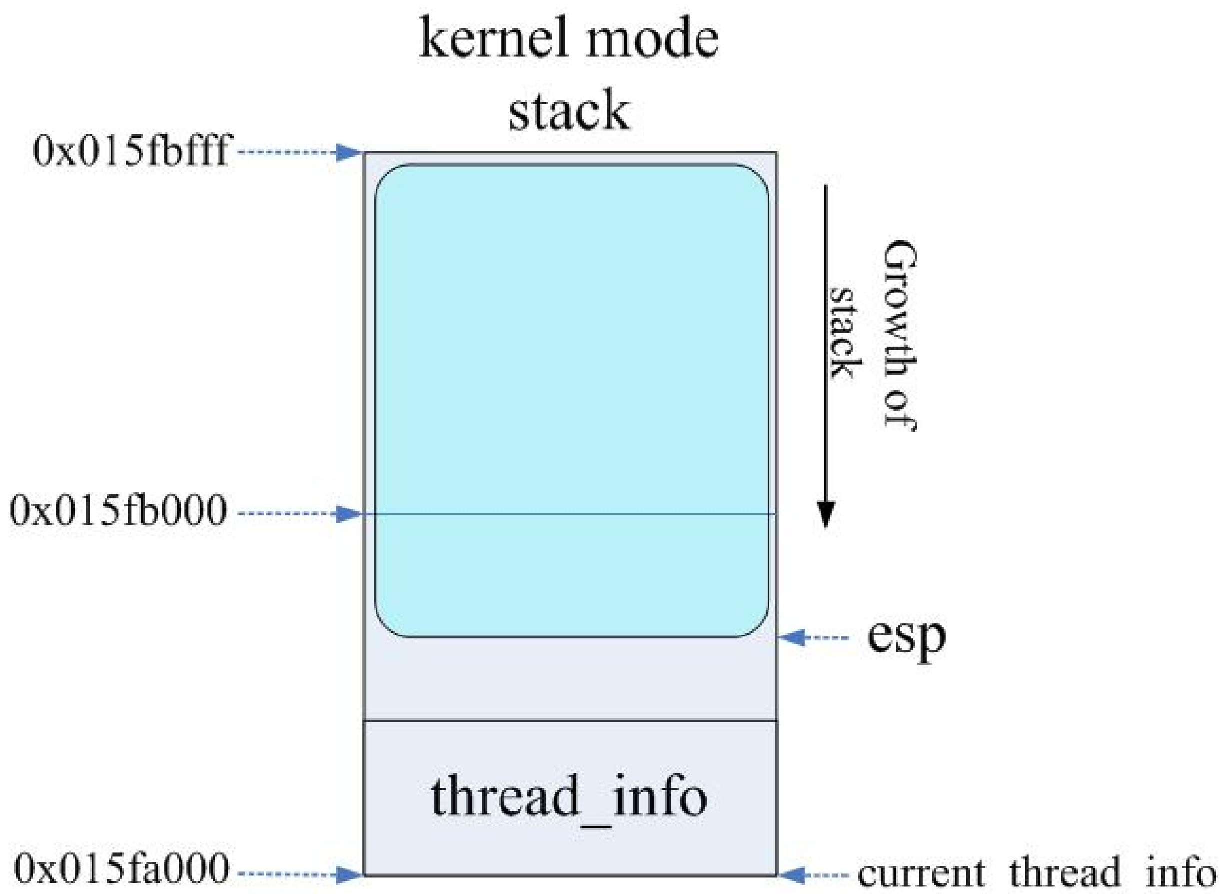


Fig.2- 8 核心堆疊

2.6 通知者串鍊

在 Linux 環境中網路介面卡驅動程式可能改變其名字或是一些旗標，甚至動態地插入或是移除自核心之中（當被編譯成模組時），在驅動程式而言，作這些動作是很單純的事情，但是，當上層的協定堆疊或是核心程式有參照使用到這些驅動程式的時候，協定堆疊或是其它核心程式就必須知道驅動程式的狀態改變，在 Linux 中通知核心其它部分驅動程式有所改變所利用的機制稱爲「通知者串鍊」(notifier chain) [12]。

通知者串鍊爲一個通用的核心內事件通知機制，很多事件都經由這個機制通知核心的不同部分，圖 Fig.2-9 爲其核心內的資料結構宣告，該結構表示了一個單向連結的串鍊，其中：

- notifier_call 欄位是一個函數指標，爲核心對於該事件的處理函數，其接受三個參數：
    - self 爲一個指向自身 notifier_block 的指標。
    - event 爲事件的代碼

- priv 爲一個通用指標，所指的資料結構隨著串列種類的不同而定。
- next 爲連結下一個串鍊的指標。
- 最後的 priority 欄位決定了你在串鍊中的位置，priority 欄位值設定的越大，會被插入在串鍊的越前面，當事件發生的時候，註冊的處理函式也會越早被呼叫。

```
struct notifier_block{
        int (*notifier_call)(struct notifier_block *self, unsigned long event,
void* priv)
        struct notifier_block *next;
        int priority;
};
```

Fig.2- 9 notifier block 資料結構

網路事件使用的串鍊名稱爲 netdev_chain，其在核心中的宣告爲一個 notifier_block 的指標，當協定堆疊或是核心其它參照使用了驅動介面的部分想要得知某些事件的發生時，便對該串列進行註冊，並且將自己的事件處理函數的指標放入 notifier_block 的 notifier_call 欄位，當事件發生後，便會照著串鍊順序依序呼叫其註冊的函數，網路使用串鍊的示意圖如圖 Fig.2-10 所示。

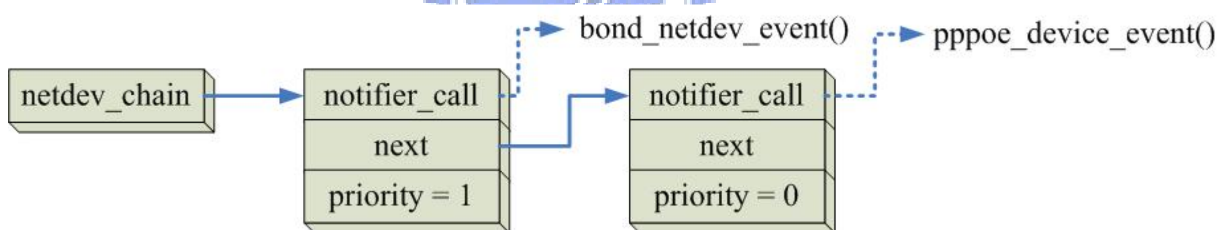


Fig.2- 10 netdev_chain 示意圖

核心中針對 netdev_chain 也提供一些處理通知者串鍊的函數，這些函數包含了 call_netdevice_notifiers()、register_netdevice_notifier()以及 unregister_netdevice_notifier()；call_netdevice_notifiers()會依序呼叫向 netdev_chain 註冊過的每一個處理函數；register_netdevice_notifer()會向 netdev_chain 註冊，將傳入的通知者區塊 (notifier block) 加入串鍊當中；unregister_netdevice_notifier()則進行解除註冊的動作。

事件處理函數有固定的回傳值，call_netdevice_notifiers()接收到不同的回傳值後對於之後的串鍊有不同的處理，各個回傳值的定義和串鍊的處理如表 Table.2-1 所示。

回傳值	描述
NOTIFY_DONE	已得到通知，但該事件無關緊要。
NOTIFY_OK	已得到通知，並且已處理完畢。
NOTIFY_STOP_MASK	已得到通知，並且停止對之後串鍊的呼叫。
NOTIFY_BAD	錯誤的通知。

Table.2- 1 事件處理函數回傳值

2.7 多網路的整合

現今的無線網路擷取技術大概可以分成兩種型態：無線區域網路(Wireless LAN，WLAN)及個人行動電話的數據服務(如 GPRS、3G、和 PHS 等)。無線區域網路可提供較高的傳輸速率，但移動性差和涵蓋範圍小，而數據服務提供較佳的移動性和較大服務範圍，但傳輸速率較小。

由於目前各種無線接取技術的陸續出現，整合各種無線網路接取設備（例如 WLAN、GPRS、3G、PHS）於行動裝置之上已經是不可避免的趨勢；在 Cross Layer Design In 4G Wireless Terminals[13]此篇文章中提到在目前現有在使用的協定堆疊以及網路架構都沒有納入多重介面的特性作爲考量，在該篇文章之中也提及，在進行垂直換手的時候（vertical handoff），由於不同網路接取技術的特性相差很多，因此會對系統效能造成很大的傷害，也由於如此，該篇文章認爲應該存在著跨層（cross-layer）的通知機制，使得驅動程式的事件可以傳達至網路堆疊的各層，如圖 Fig.2-11。

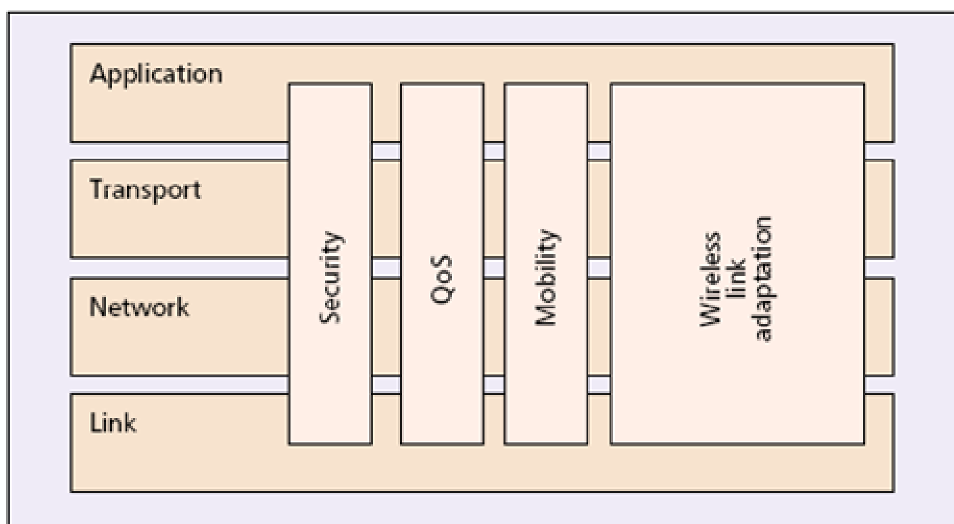


Fig.2- 11 跨層設計概念示意圖

根據 Mobile IP Applicability: When Do We Really Need It[14]的分類，行動性支援

(mobility support) 包含了：

- 位置管理 (location management) 。
- 換手管理 (handoff management) ，包含垂直和水平的換手。

一般來說，因爲介面的控制和選擇是一種策略 (policy) ，通常由使用者空間的應用程式來處理，而核心只提供控制介面的方法和機制，而在 Mobile IP Applicability: When Do We Really Need It 中則甚至主張行動性的支援應該由網路協定中更高層的部分來處理，例如傳輸層 (transport layer) 或是應用層 (application layer) 。

而當位置管理或者是換手管理在應用層處理時，網路介面資訊的取得是不可或缺的，例如網路位置的取得與設定，網路介面的偵測等等，而本篇論文所提出的機制則是使用類似 UNIX 系統中傳統信號的機制將底層網路介面的資訊或是動作主動的由核心通知使用者層的程式，將核心空間內的事件通知機制延伸到是使用者空間，使得使用這空間應用程式對於位置管理以及換手管理的處理能夠更加的簡單。

2.8 HUT VHO DAEMON

芬蘭的赫爾辛基技術大學 (Helsinki University of Technology, HUT) 在 2005 年的 3 月 8 日的一個 study cluster 中也提出了和本論文類似概念的系統，該系統的架構圖如圖 Fig.2-12 所示。

該系統提供了一套使用者函式庫和應用程式介面 (API) 給應用程式，而在訊息的傳達方面主要是利用一個使用者空間的常駐程式 (daemon) ，應用程式使用該系統提供的應用程式介面和 VHO Daemon 之間建立 IPC (Inter-process Communication) 的通道，當下層驅動程式有事件發生時，VHO Daemon 會先得知，之後利用 IPC 的方式通知應用程式，而其提供的函式庫會依據發生的事件種類呼叫應用程式呼叫程式的回叫函數。

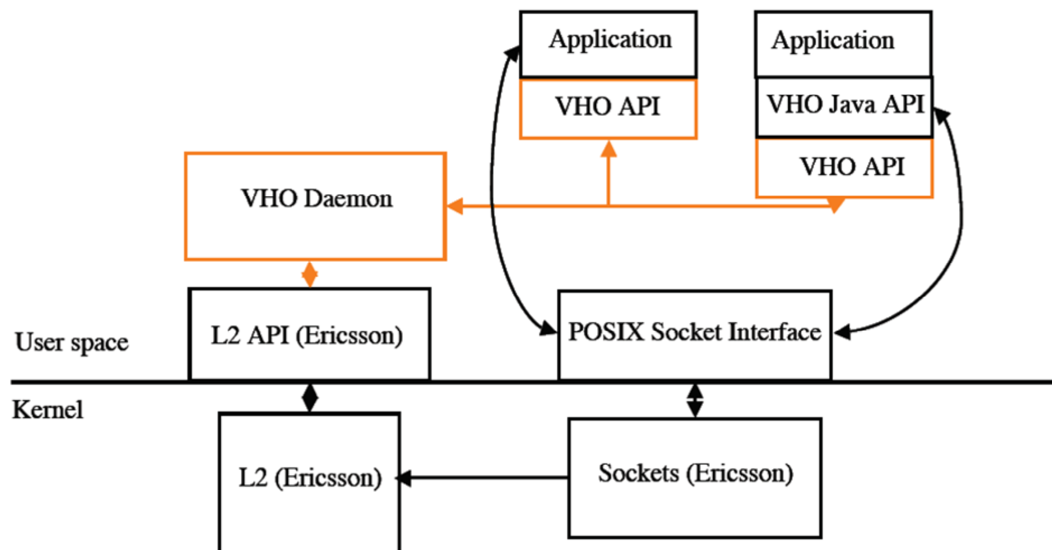


Fig.2- 12 HUT VHO Daemon 之系統架構

該系統中定義的事件有以下四大項：

- 系統驅動程式相關事件，其中包含了：
  - 設備的新增以及移除
  - 設備的連線狀態
  - 其他設備資訊
- 換手通知。
- 其他 VHOAPI (Vertical Handoff API) 事件。
- Daemon 事件。

該套系統目前正在發展中，能找到的資料並不多，關於其驅動程式層的支援則是使用 Ericsson 發展的 L2 機制，不過相關的資料我們無法取得，該系統和本論文所發展的系統相異之處在於：

- 使用 IPC 作爲事件通知機制的基礎。
- 處理函式的呼叫全在使用者空間完成。
- 支援 Java 平台

第三章 驅動程式層網路事件通知機制設計與架構

在本章中，將會敘述實作本論文之系統概觀以及各元件的設計原理以及原則，其中，包含了網路相關事件的定義以及其代表的意義，這些事件何時發生及核心何時對其作相關的處理，訊息的產生 (generation) 和傳達 (delivery) ，以及在使用者空間和核心空間之間切換所使用的方法。

3.1 目標與問題定義

本論文欲達成的目標有以下三點：

1. 由作業系統主動通知上層應用程式下層硬體設備狀態的改變。

   本論文希望提出一套機制，改變傳統由上層發出系統呼叫得知下層設備狀態的機制，改由作業系統告知使用者程序下層設備狀態的改變。

2. 使應用程式對下層硬體事件能儘快反應。

   除了要能夠由作業系統告知使用者下層狀態的改變之外，我們也希望使用者程序能夠對已經發生的事件能夠快速的反應，使得使用者能夠對發生的事件作最迅速的處理。

3. 使用者使用類似傳統信號的方式，對其感興趣的事件註冊。

   本論文希望將底層驅動程式的狀態改變以事件的概念呈現，並且能夠提供程式設計者一個類似傳統信號的系統函示介面，使得程式設計者可以使用和傳統信號類似的程式設計方式來設計其應用程式。

為了要完成以上目標，我們必須解決以下的問題：

- 何時記錄下事件的發生？
- 如何記錄事件的發生？
- 何時使用者程式會得知發生的事件？
- 如何對發生事件的程序排程？
- 如何保存原本的使用者本文 (當在執行事件處理函數時)？
- 如何回復原本的使用者本文 (當事件處理函數執行結束後)？

接下來的第三章其他章節將會介紹我們為了解決上面所提及的六點問題，所提出的系統之設計以及設計的原則，第四章的各章節則是詳細描述了我們如何依據第三章所提

及的設計於 Intel x86 平台實做出一套完整的系統。

3.2 系統概觀

UNIX 系統上傳統的訊息通知機制有兩大功能：

1. 系統通知特定的程序某個與系統相關的事件的發生
2. 使程序對該系統事件做出相對應的反應

有關傳統訊號已經在章節 2.1.1 中介紹過了，其本身架構上的缺點主要如下：

- 低擴充性：

  訊號數目總共只有 32 個，大部分的訊號都已經代表某個特定的系統資訊，無法另行定義。

- 訊息遺失：

  同樣類型的信號在被傳達之前重複發生的同樣事件將會遺失，系統不會知道有事件重複發生。

此外，POSIX.1b[]也定義了即時信號(Real-Time Signals)，可以用來作非同步輸出輸入（Asynchronous I/O）且相容於 POSIX.1 所定義的訊號介面。

雖然存在了上述的訊息通知機制，但是這些訊息主要是針對系統事件（POSIX.1 定義的傳統訊號）或是針對某個檔案敘述（file descriptor）的訊息作輸入輸出方面之通知，使用者程式想要得到網路介面的狀態仍然要經由傳統的 ioctl()系統呼叫得知，如此一來，不僅沒有效率，並且增加程式的複雜度。

本論文所提出的機制可以將網路驅動程式所發生的事件（例如有新的網路介面卡插入，或是網路介面卡的 IP 更改）即時且不會遺失地利用類似傳統信息機制通知使用者空間的程式，並且使系統的控制權交至使用者程式利用系統呼叫（system call）註冊的回叫（callback）函數，當有事件發生時，系統的排程器（scheduler）便會優先先執行註冊過網路事件的行程。

本系統之實作可以分為三個主要部分：

- 事件通知部分（Event notification）。
- 行程管理（process management）。
- 排程器（scheduler）。

這三大部分在系統核心內的關係如圖 Fig.3-1 所示。

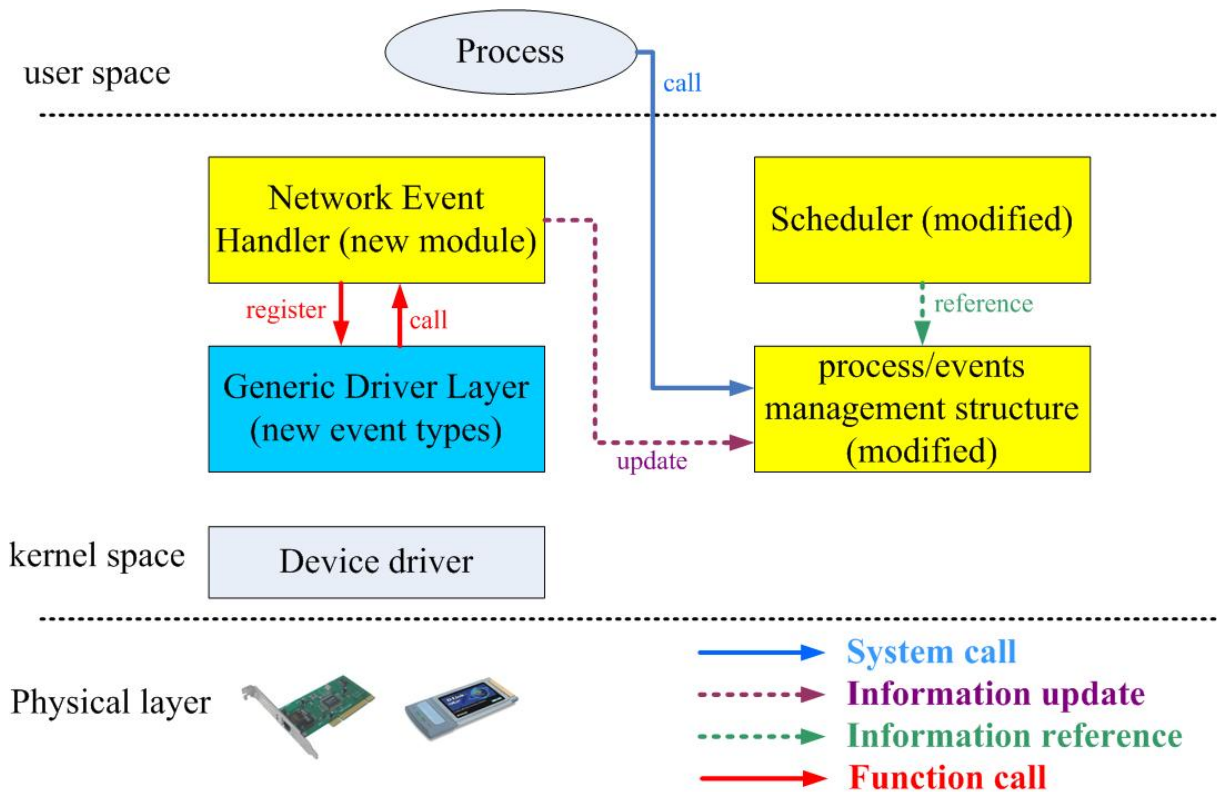


Fig.3- 1 系統元件架構和關係

3.3 系統呼叫的設計

本系統允許使用者利用系統呼叫註冊的回叫函數，以下敘述該系統呼叫的使用方法以及設計。

本系統所實作的系統呼叫之相關宣告如圖 Fig.3-2 所示。

```
typedef void (*__iwsig_handler_t) (int);
int iwsignal(int signo, __iwsig_handler_t iwhandler, char *namebuf, unsigned long bufsize)
```

Fig.3- 2 系統呼叫相關宣告

其中，第一個參數代表系統呼叫所欲註冊的事件代碼，第二個參數則是一個沒有回傳值的函數指標，當此參數被設定為空指標 (NULL value) 時，代表解除註冊該事件的監聽，第三個參數為一個全域變數的緩衝區 (buffer) 位址，當相關事件發生時，核心會將發生事件的網路介面名字填入這個緩衝區中，最後則是該緩衝區的大小。

3.4 網路相關事件之定義

當程式寫作者想要寫作一個異質網路的管理平台時或這是當程式寫作者單純的想要將其原本的應用程式加上行動性的支援時，最難處理的其中一個部分便是網路介面的管理。使用者層的程式為了要得知下層驅動程式的狀況，必須不停的利用系統呼叫詢問

下層的驅動程式狀態，在本系統中，將這些類似輪詢的下層驅動程式得知方式改變為事件觸發的方式，在本節中，將會介紹本系統中所實作的網路驅動程式相關事件。

3.4.1 驅動程式和協定堆疊間的事件

在章節 2.6 中我們知道了核心各部分的相互通知機制是通過通知者串鍊（notifier chain）機制，在針對網路驅動程式的部分核心定義了一個靜態宣告的通知者串鍊 netdev_chain，並且事先定義了核心其它部分或是協定堆疊程式會對網路驅動程式感到興趣的事件，這些事件以及其代表的意義如表 Table.3-1。

事件名稱	代碼	描述
NETDEV_UP	0x0001	啟動某個網路介面。
NETDEV_DOWN	0x0002	關閉某個網路介面。
NETDEV_REBOOT	0x0003	當網路介面偵測到硬體錯誤且需要重新啟動。
NETDEV_CHANGE	0x0004	驅動程式旗標有改變。
NETDEV_REGISTER	0x0005	網路驅動程式被插入核心中。
NETDEV_UNREGISTER	0x0006	網路驅動程式由核心中移除。
NETDEV_CHANGEMTU	0x0007	網路介面更改 MTU (Maximum Transfer Unit) 。
NETDEV_CHANGEADD	0x0008	網路介面之硬體位址改變。
NETDEV_GOING_DOWN	0x0009	在驅動程式的 IFF_UP 旗標被清除之前。
NETDEV_CHANGENAM	0x000A	驅動程式改變其名字。

Table.3- 1 系統內定義的網路驅動程式事件

本系統將原本僅能在核心空間中使用的網路通知者串鍊的機制延伸至使用者空間，系統內定義的 10 種網路驅動程式事件將能夠被傳達給使用者層的程式，使得使用者層程式能夠不需要利用類似輪詢的方法得知網路驅動程式的狀態。

3.4.2 其它使用者層關心的事件

前小節中所提及的核心內訂的 10 種事件大部分都是和核心的驅動程式直接相關的部分，但是除了核心內訂的 10 種網路介面相關事件之外，使用者還有一些額外關心的事件，而這些事件是核心驅動程式事件所未定義的，因此，在本系統的實作中我們額外定義了三個事件：

1. 網路卡載體開通（carrier on）。
2. 網路卡載體關閉（carrier off）。
3. 和網路卡相結合之 IP 位址改變。

這三個事件對於核心來說並沒有定義在網路通知者串鍊的事件之中，原因是因為前

兩者對於核心的網路協定堆疊來說並沒有特別意義，而第三者在核心的定義來說並非驅動程式和網路硬體直接相關的部分，但是，對於使用者或者是應用程式的寫作者來說，這些事件卻和使用者對於網路的使用或者網路介面的管理息息相關，因此，在本系統中也將這些事件納入我們的系統之中。

3.5 網路事件的紀錄

前一節敘述了所有定義的網路事件，當這些事件發生的時候，核心必須將其記錄下來，傳統訊號處理在訊號的紀錄方面的一大缺失就是其使用當有兩個以上重複同類型的事件發生時，使用者程式只會被通知一次；即時信號在這一方面作了一些改進，解決了信號可能遺失的問題，但是，即時信號並不能對網路驅動程式的相關事件作註冊，因此難以用來作多網路介面的整合的應用。

在本系統中，針對每個行程所註冊想要收聽的訊息，將會被轉換成一個位元陣列 (bit array) 儲存，而對於發生事件的紀錄，我們在系統的行程敘述子 (process descriptor) 中新增一個事件串列，該串列用以記錄行程所註冊過，且已經發生過的網路驅動程式相關事件。

當系統中有網路驅動程式事件發生後，程式將會將對應的事件及其相關訊息轉換成一個事件描述子 (event descriptor)，並且依據事件發生的順序將其插入對該事件註冊過之行程敘述子中的事件串列中，如此一來，便可以得知重複事件的發生以及其先後發生的順序。

3.6 事件的產生

在章節 2.2.3 中定義了事件的產生 (generation) 以及傳達 (delivery) 的不同，對於網路事件的產生方面，我們利用了系統內的網路驅動程式通知者串鍊的機制並且對其擴充，除了核心內訂的 10 種網路驅動程式事件之外，我們為其加入了在章節 3.2.2 中所定義的三個使用者關心的事件。

在 Linux 架構下，所有對於網路卡狀態的改變都要先經過核心內的通用驅動程式層處理 (這部分的處理通常定義於核心程式碼的 *net/core/dev.c*) ，舉例來說，當使用者或是其它系統程式想要暫停某個網路卡的使用時，便會使用 ioctl()系統呼叫向驅動程式要求清除 IFF_UP 旗標，並且呼叫通用驅動程式層的 *dev_close()*函數，在這些動作完成之後，通用網路驅動程式層中的 *dev_close()*函數會使用通知者串鍊通知核心其它部分該事

件的發生。

在本系統中，仍然利用通知者串鍊來達到事件提醒的基礎，也就是說，我們實作了一個事件處理者 (event handler) ，該事件處理者會將核心內用來通知其它網路協定部分的網路通知者串鍊機制延伸至使用者空間使用；當有網路驅動程式相關事件發生後，事件處理者就會得到通知，並且根據之前註冊的狀態更新相關的資料結構，等待之後的事件傳達相關程式碼來作處理，最後跳至使用者註冊的回叫函數執行。

3.7 事件的傳達

在章節 3.4 中已經介紹了事件的產生 (generation) ，在事件的產生之後接下來的處理就是事件的傳達 (delivery) ，關於事件的傳達，我們希望達到的目標有：

1. 能夠由核心空間直接切換到使用者空間。
2. 能夠直接執行應用程式所註冊的事件處理函式。
3. 對於事件的發生能夠盡快反應。

3.7.1 使用者和核心空間之間的切換

當網路驅動程式相關訊息發生且記錄了以後，我們的目標就是由核心空間直接切換至使用者空間來執行，也就是說，在行程回到使用者空間之前，我們會去檢查是否有該行程所註冊的事件發生，當有註冊的事件發生時，系統應該要先執行該行程所註冊的對應處理函式，接下來，利用系統呼叫返回核心空間，並且在該系統呼叫內回覆原來所應該執行行程的硬體環境 (hardware context) 。

關於使用者空間和核心空間的切換，我們利用了 Linux 下的兩種堆疊 – 核心模式堆疊 (kernel mode stack) 和使用者模式堆疊 (user mode stack) ；核心模式堆疊主要作用有兩點，記錄使用者硬體環境 (hardware context) 以及存放程序敘述子。

當要跳至使用者空間執行行程註冊的回叫函數之前，因為在 Linux 中每次由核心空間返回使用者空間都會將核心模式堆疊清空，如此一來，當使用者由回叫函數返回核心空間時，原本用以執行一般程式碼的硬體環境會消失；因此我們必須將核心模式堆疊中的硬體環境存至別的地方，以免原本的硬體環境消失，因此，我們將原本的硬體環境儲存在使用者模式的堆疊中，等到由回叫函數返回後，再將其復原，其過程如圖 Fig.3-3 與 Fig.3-4 所示。

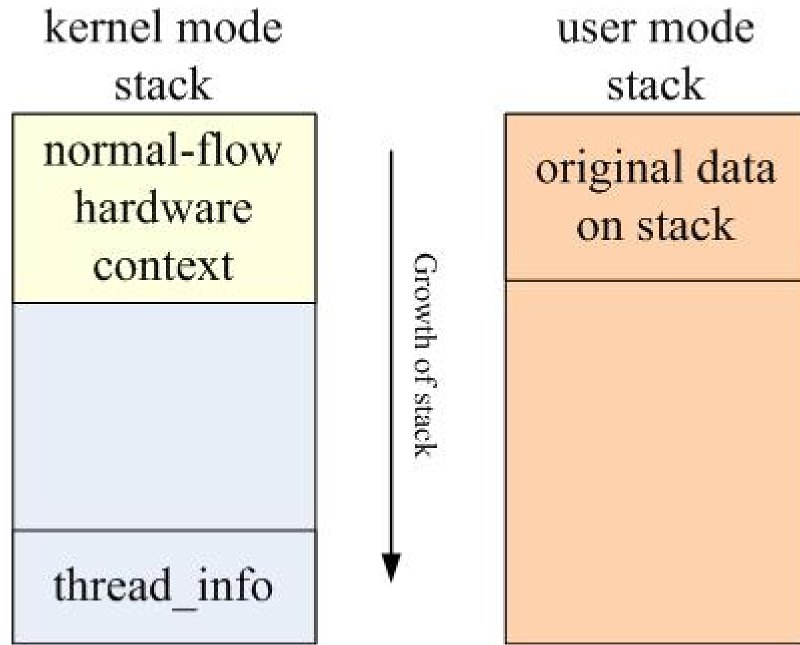


Fig.3- 3 一般在核心空間時的的核心模式堆疊和使用者模式堆疊

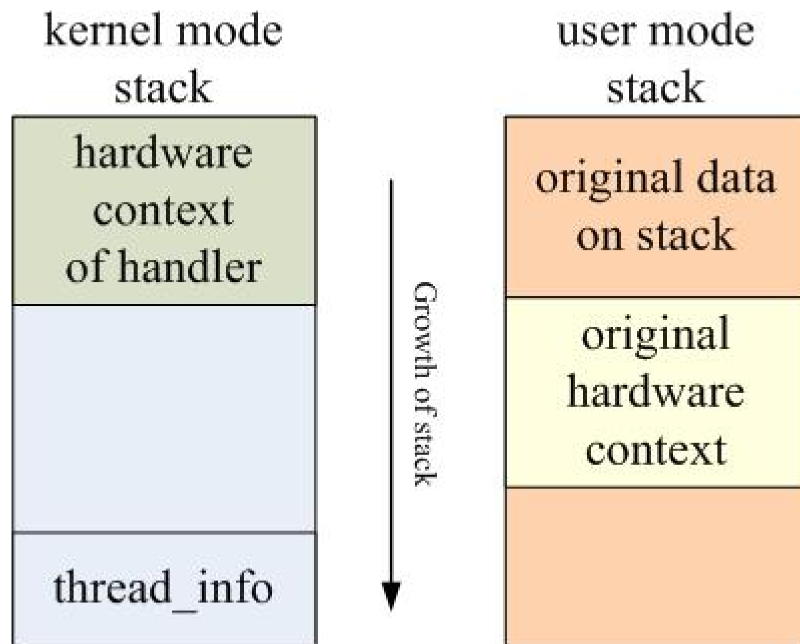


Fig.3- 4 欲執行回叫函數前的核心模式堆疊和使用者模式堆疊

3.7.2 將事件即時傳達至使用者空間程式

爲了能將事件及時的傳達給使用者空間的程式，在我們的系統之中使用兩個設計以加速信息的傳達：

1. 每次由中斷 (interrupt) 處理或例外 (exception) 處理返回以及新生行程 (fork) 之後且要返回行程前檢查是否該行程註冊過的事件是否發生。
2. 每次行程執行完系統呼叫且要返回使用者空間之前。

3. 當每次啓動系統排程器 (scheduler) 時，優先執行註冊過網路驅動程式事件且事件發生過的行程。

當某個行程正在執行時，如果有某個註冊過的網路驅動程式事件發生，則至少會在一個計時器中斷後得知；若是當註冊過的網路驅動程式事件發生，且當時中央處理器的控制權並不在註冊過的行程上時，我們需要讓有註冊過網路程式驅動程式事件的行程先執行，以便能夠即時的把事件傳達。

在我們的設計之中，我們針對每一個註冊過的行程都保留了一份記錄，因此我們可以很容易的知道目前系統上有哪些行程註冊過了事件，參照章節 3.4 可知，由於發生的事件都會被紀錄在行程描述子的一個串列中，因此我們可以容易的得知註冊過的行程是否有尚未處理的事件。

參照 Fig.3-5，假設核心內預設的排程法爲 Process#1，Process#2，然後 Process#3，此時 Process#2 已向核心註冊欲得知網路驅動程式事件，當 Process#3 排程結束之後，原本應該要輪到 Process#1 獲得中央處理器的控制權，可是此時核心檢查到有相對應事件的發生，因此核心排程器將會優先執行 Process#2 而非 Process#1。

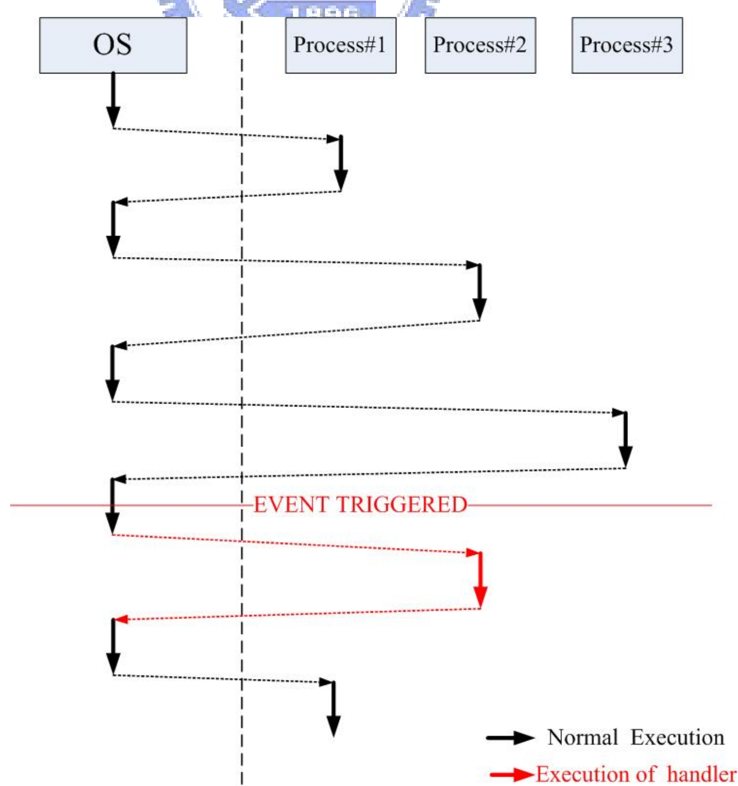


Fig.3- 5 中央處理器控制權的交替實例

第四章 驅動程式層網路事件通知機制之實作

4.1 軟硬體需求

以下是本論文實作之軟硬體平台：

- ✓ 硬體需求：IA-32 Intel architecture compliant computer
- ✓ 系統核心：Linux kernel version 2.6.10
- ✓ 發行套件：Debian Linux sarge
- ✓ 發展工具：
  - ◆ GNU gcc 3.3.5 (Debian 1:3.3.5-12)
  - ◆ GNU ld 2.15
  - ◆ GNU make 3.80
  - ◆ GNU Debugger GDB

4.2 事件及其相關處理的實作

本系統所支援的網路驅動程式事件總共有 13 種，其在本系統內的定義以及和核心內定義的網路通知者串鍊定義事件之對應如 Table.4-1 所示。

系統定義事件	網路通知者串鍊定義事件	代表值
IWEV_DEVUP	NETDEV_UP	0x0001
IWEV_DEVDOWN	NETDEV_DOWN	0x0002
IWEV_REBOOT	NETDEV_REBOOT	0x0003
IWEV_DEVSTATCHG	NETDEV_CHANGE	0x0004
IWEV_DEVREG	NETDEV_REGISTER	0x0005
IWEV_DEVUNREG	NETDEV_UNREGISTER	0x0006
IWEV_DEVCHGMTU	NETDEV_CHANGEMTU	0x0007
IWEV_DEVCHGADDR	NETDEV_CHANGEADD	0x0008
IWEV_DEVGODOWN	NETDEV_GOING_DOWN	0x0009
IWEV_DEVCHGNAME	NETDEV_CHANGENAM	0x000A
IWEV_DEVCHINADDR	—	0x000B
IWEV_DEVCARON	—	0x000C
IWEV_DEVCAROFF	—	0x000D

Table.4- 1 系統定義事件和網路通知者串鍊定義事件交互參照

事件的發生都需要呼叫作業系統的一些應用程式編程介面（API），在 Linux 中，核

心事先定義了一些通用的處理函數，供發展者使用，這些通用的處理函數稱爲核心編程介面 (kernel API) ，網路通知者串鍊的實作原理就是當某些核心的編程介面被呼叫且完成前，循序地呼叫註冊在串鍊上的回叫函數。

原來系統中就有定義的的 10 種事件原本就被系統所支援，當對應事件發生的時候便會呼叫串鍊中的函數，另外三個事件則需要我們自行加入使用者通知串鍊的機制。

4.2.1 網路位址改變事件之網路通知者串鍊支援

在 Linux 系統當中，網路位址的改變通常是利用 ioctl 系統呼叫，對驅動程式下達 SIOCSIFADDR 指令，我們對系統核心所做的修改就是當該指令被下達且處理完成之後，使用系統的 *call_netdevice_notifiers()*介面通知對網路通知者串鍊註冊過的核心函式，相關函式的處理在 Linux kernel 2.6.10 中存在於核心程式碼 net/ipv4/devinet.c 中 *devinet_ioctl()*函式中。

4.2.2 載體出現與消失事件之網路通知者串鍊支援

網路驅動程式載體的偵測通常是由網路驅動程式負責偵測，並且呼叫核心程式介面 *netif_carrier_on()*以及 *netif_carrier_off()*來設定或是清除驅動程式資料結構的 state 欄位的__LINK_STATE_NOCARRIER 位元。

我們修改了核心內的 *netif_carrier_on()*以及 *netif_carrier_off()*函式，在這兩個函數結束返回之前呼叫 *call_netdevice_notifiers()*以通知所有對網路通知者串鍊註冊過的核心函式，相關函式的處理在 Linux kernel 2.6.10 中存在於 include/linux/netdevice.h。

4.2.3 事件的格式化以及記錄

對於網路驅動程式事件的發生，我們利用核心的網路通知者串鍊通知本系統的網路事件處理模組，當網路事件處理模組收到該事件之後，我們會先將事件實體化爲一個事件區塊 (event block) ，事件區塊的資料結構如圖 Fig.4-1 所示。

```
struct event_blk {
        char ifname[IFNAMESIZ];
        unsigned int event_num;
        struct list_head proc_list;
};
```

Fig.4- 1 事件區塊資料結構

當有網路驅動程式事件被實體化爲事件區塊之後，網路事件處理模組便去檢查註冊過的行程是否有對於該事件感興趣，若是，便會將該事件加入該行程描述子的事件串列之中，行程描述子和事件區塊的關係如圖 Fig.4-2 所示。

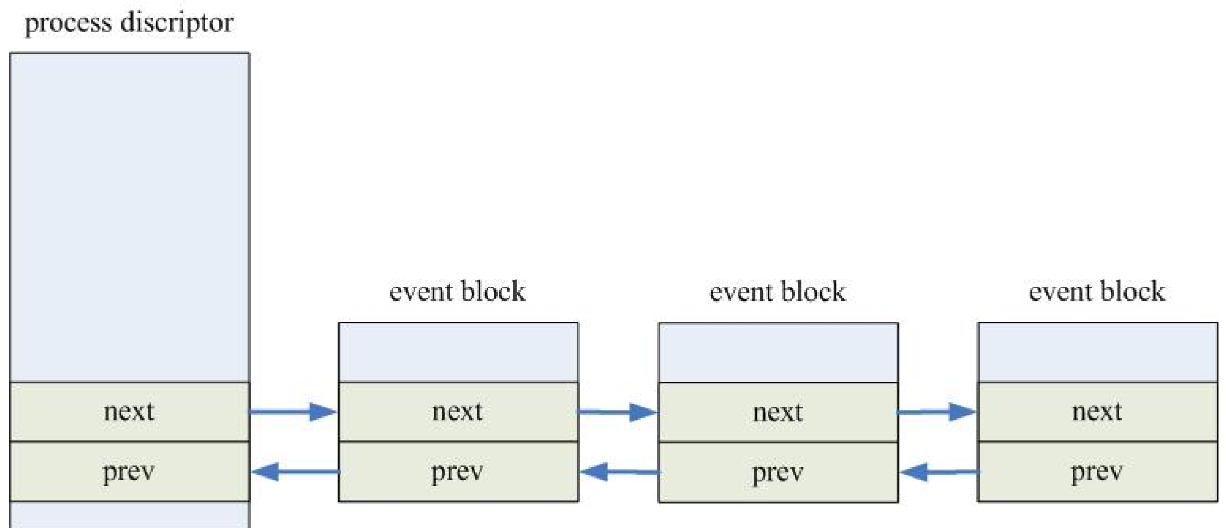


Fig.4- 2 事件區塊的記錄方式

圖 Fig.4-3 為事件處理者的處理流程，其流程的核心部分為一個 for 迴圈，迴圈中每次選取一個註冊過的行程，檢查其是否對目前發生的事件感興趣，若感興趣，則將事件格式化，並且插入行程敘述子的事件串列中，若非，則選取下一個註冊過的行程，直到所有行程檢查完畢。

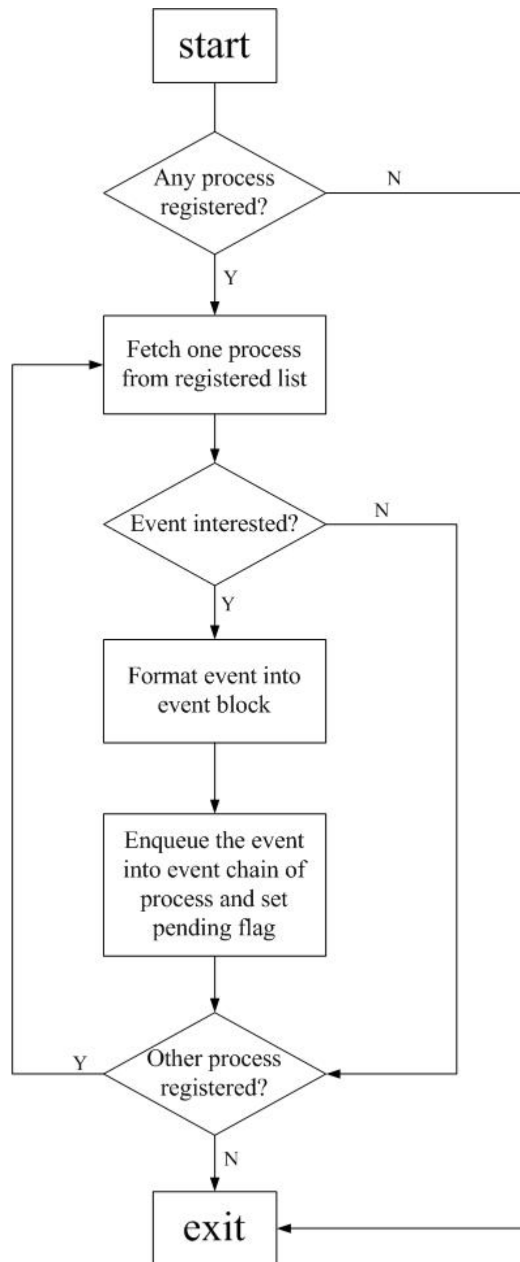


Fig.4- 3 事件處理者之流程

4.3 行程管理實作

爲了要記錄和管理註冊過行程以及發生過的事件，我們必須對原本的行程資料結構作一些改變，在核心之中，每一個使用者空間行程都由一個行程描述子代表，行程描述子記錄了行程的所有資訊，例如記憶體空間，開過的檔案資訊等等…，在本系統中，爲了要支援程式能夠註冊其感興趣的事件，因此我們必須對行程敘述子增加一些欄位。

```
struct task_struct {
        …
        …
        char *devname_buf;
        unsigned int event_blocked
        __iwsig_handler_t iw_handler[NR_EVENT];
        struct list_head iw_chain;
        struct list_head evt_blk_chain;
};

struct thread_info{
        …
        …
        iw_flag;
};
```

Fig.4- 4 對行程敘述子的修改

圖 Fig.4-4 是我們針對本系統增加於行程敘述子的資料結構，其個別的意義如下：

- struct task_struct 內新增的欄位：
  - devname_buf：使用者空間的字串指標，用來儲存發生事件的設備名稱。
  - event_blocked：一個位元遮罩，表示現在不想收到的信號。
  - iw_handler[]：一組使用者空間的回叫函數指標
  - iw_chain：用來記錄目前註冊過的行程的串鍊
  - vt_blk_chain：發生過的感興趣事件。
- struct thread_info 內新增的欄位：
  - iw_flag：表示目前的行程是否尚有未處理的事件，當事件串列清空時，將會清除這個旗標。

當使用者空間行程呼叫系統呼叫註冊其對某些事件感興趣時，若是該行程之前沒有註冊過時，系統便會將其加入一個在系統內靜態宣告 (static declare) 的串列中，之後每次有事件發生便會去該串鍊檢查是否有行程對事件感興趣，若有，則將事件區塊加入該事件的事件區塊串鍊中。

以圖 Fig.4-5 為例，當要啓用某個網路介面的時候，使用者會通過 ioctl 系統呼叫控制下層的網路介面，在本例中使用者下達了 ioctl 命令設定網路介面的 IFF_UP 旗標，該

命令在核心內最後會由通用驅動程式層來處理，通用驅動程式層會先去呼叫驅動程式的處理函式 (在本例中為 *dev->open()*) ，當處理函式執行完了之後，通用驅動程式層便會呼叫 call_netdevice_notifiers()函數，該函數會去對 netdev 通知者串列註冊的回叫函數一一呼叫，當呼叫到我們註冊的事件處理通知者區塊時，便會呼叫事件處理者函式，該函式會將事件實體化，並且將其加入對該事件感到興趣的行程敘述子之發生事件串鍊中，等待之後核心的處理。

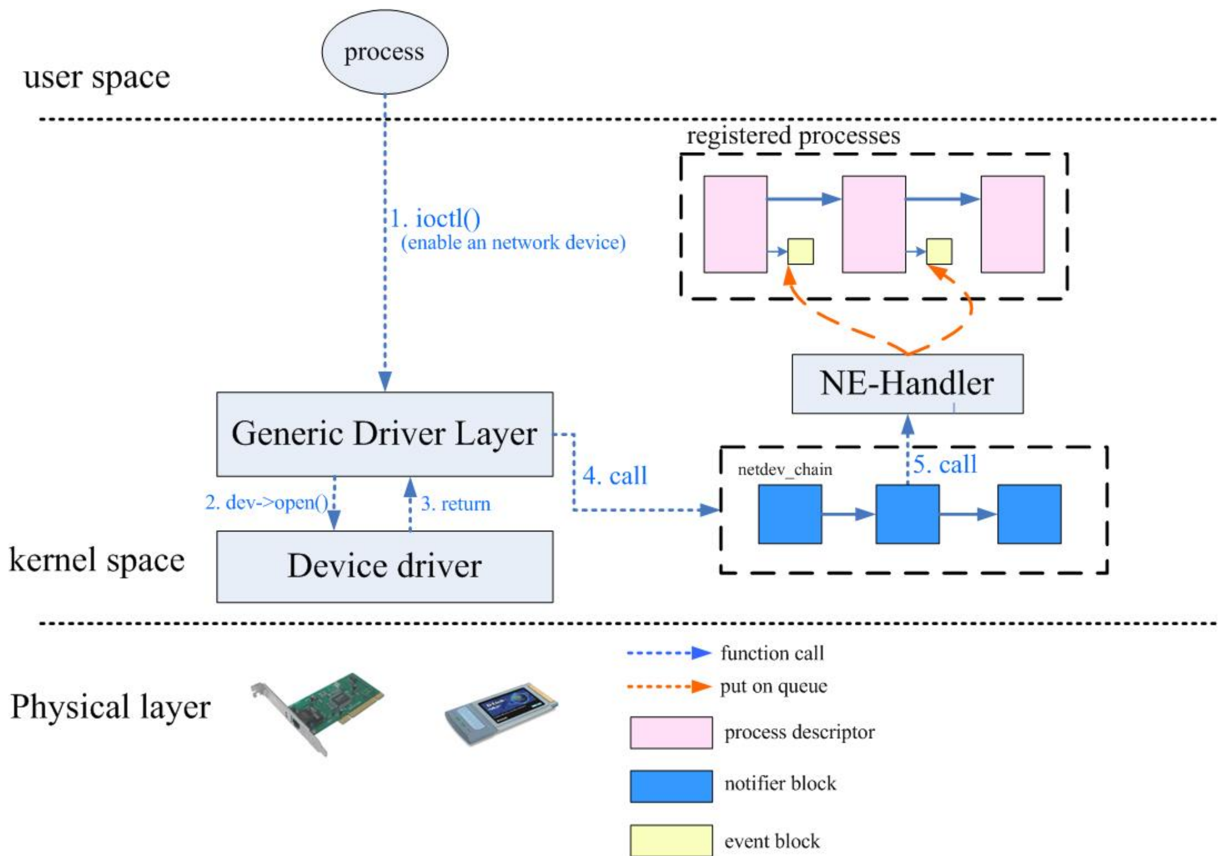


Fig.4- 5 事件的發生及儲存之實例

4.4 事件處理之時機

4.4.1 檢查事件的發生

為了要能使使用者空間行程能夠快速的得知事件的發生，並且對其採取對應動作，核心必須盡快的對已發生的事件反應，並且將中央處理器的控制權交回使用者空間的程式，在我們的處理中，將在每次由中斷處理或例外處理返回以及新生行程返回且執行該行程時，核心將會檢查欲返回的行程是否有未處理的事件，若有，則對該事件作處理，將至中央處理器的控制權交回至使用者空間，並且執行對應回叫函數。

關於檢查事件的程式碼，可以參照 Fig.4-6，該段檢查的程式碼會被插入在核心返回

使用者空間程式碼中中斷處理和例外處理之後，以及系統呼叫執行之後。

```
ENTRY(resume_userspace)
        cli
        movl TI_IWFLAG(%ebp), %ecx
        jnz no_iw_event
        movl %esp, %eax
        call handle_iw_event_sig
        jmp restore_all
no_iw_event:
        …
        ...
```

Fig.4- 6 偵測是否有未處理事件的程式碼

圖 Fig.4-7 是未修改前由 Linux 2.6.10 核心返回的使用者空間的處理流程，當要由核心返回使用者空間有三個狀況，由例外中返回，由中斷返回，以及由系統呼叫返回，在圖中代表這三個情況發生的區塊分別爲 ret_from_exception、ret_from_intr 以及 syscall_exit，當由 ret_from_exception 以及 ret_from_intr 返回時，若是要回到使用者空間行程執行，則要先經過 work pending 判斷方塊的檢查，若爲真表示有信號尚未處理，此時，若不到重新排程時，便會呼叫 *do_signal()*函式處理尚未處理的信號。

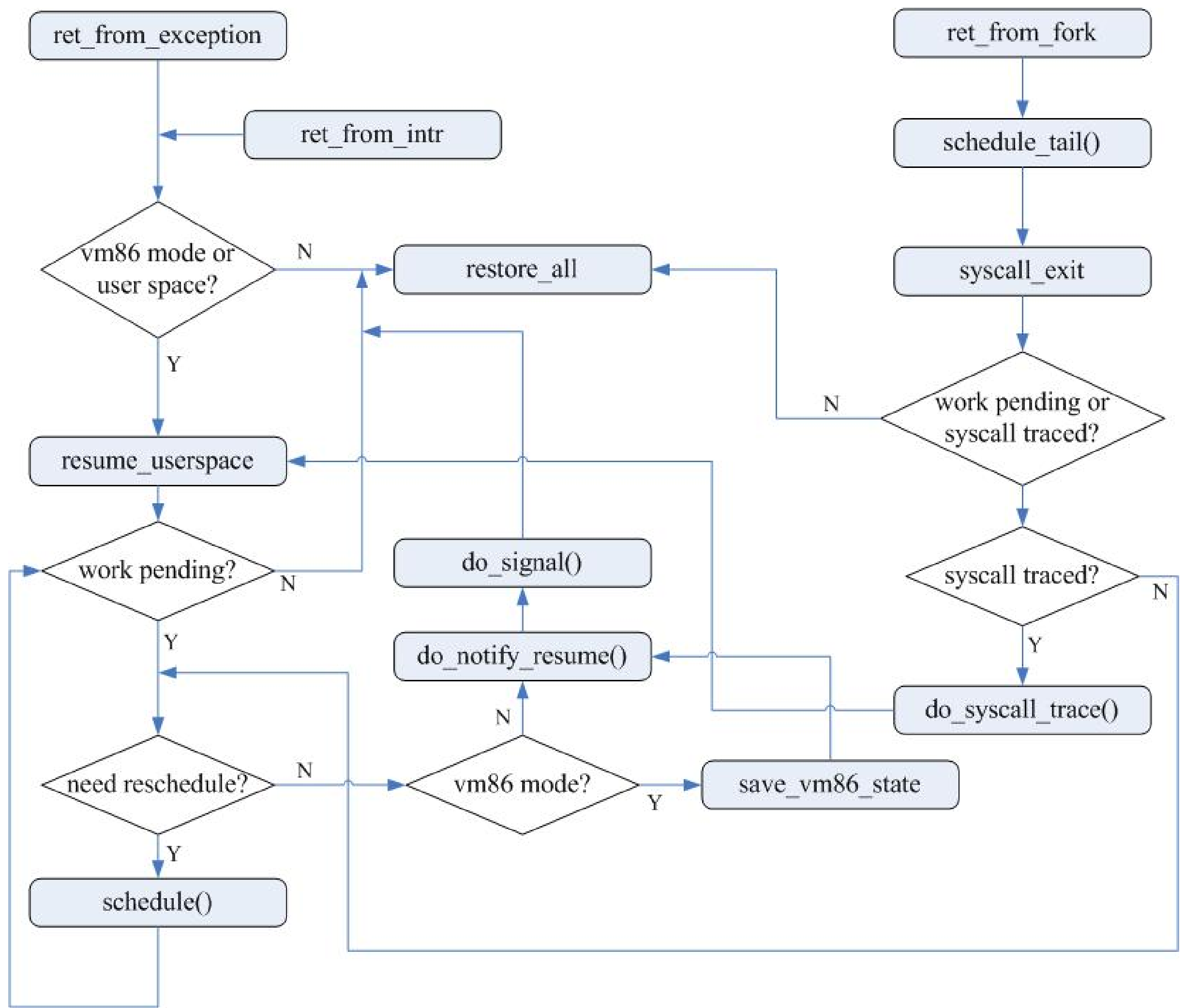


Fig.4- 7 未修改前由核心返回的流程圖

圖 Fig.4-8 爲修改過後由核心返回的流程圖，我們加入的處理爲橘色的部分，主要分爲兩個部分，第一個部分是在 ret_from_intr 以及 ret_from_exception 中返回且進入 resume_userspace 的處理後，我們會在判斷是否有信號尚未處理之前（work pending 之判斷）先判斷，是否有未處理的網路驅動程式事件，若有，則呼叫 handle_event_sig()函式，該函式會執行使用者空間和核心空間之切換的相關設定，接下來跳至核心的 retore_all 函數處理，該函數會將硬體環境還原，並且跳至使用者空間執行使用者空間的程式碼；另外一個檢查點則是在由系統呼叫返回且檢查是否有信號未處理或是系統呼叫被追蹤前，發現有未處理的事件時也一樣會呼叫 handle_event_sig()函數，然後跳至 restore_all 內回復硬體環境。

由於每次由系統呼叫，中斷，或者是例外中返回時都會檢查是否有感興趣的事件發生，因此，使用這個方法可以保證「在該行程握有中央處理器的控制權時，至少在兩次記時器中斷（timer interrupt）其間的事件，可被傳達至使用者空間的行程」。

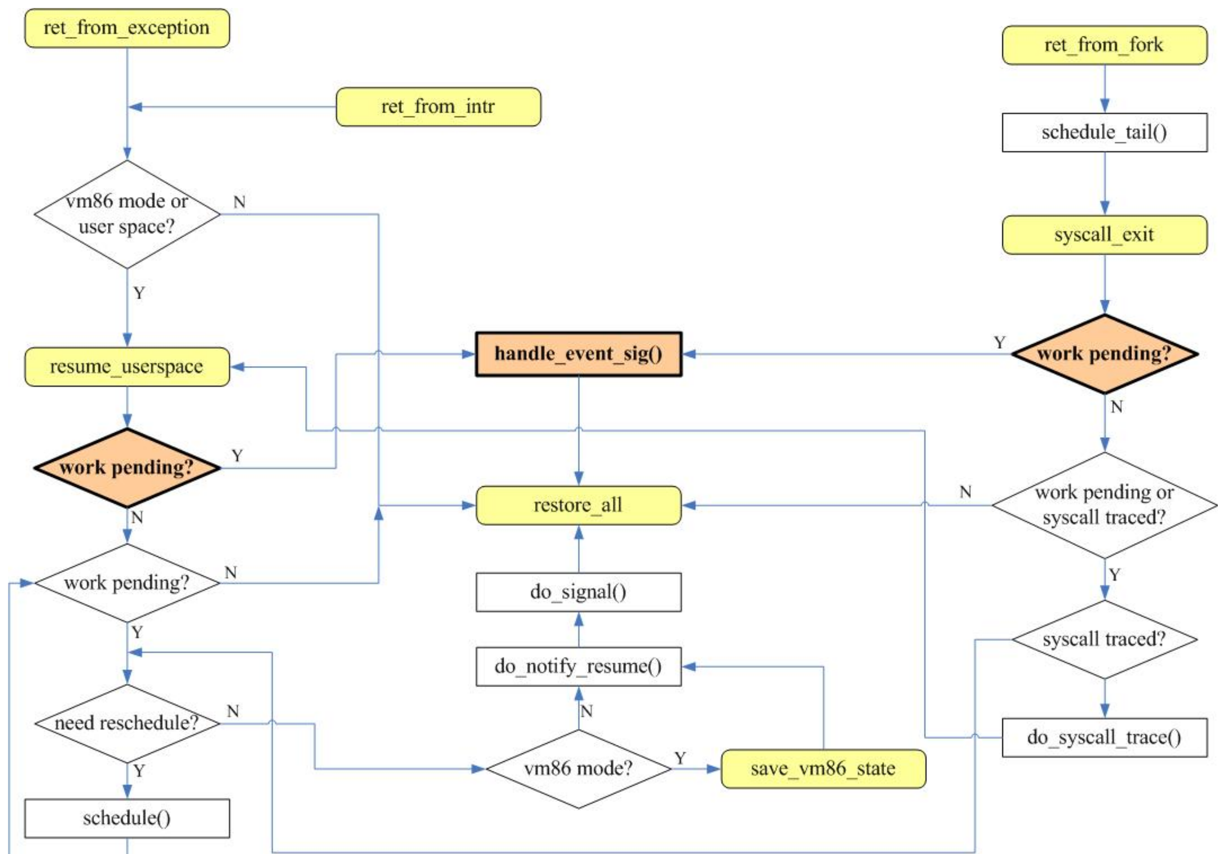


Fig.4- 8 修改過後的核心返回流程圖

4.4.2 排程器的修改

章節 4.4.1 中所提及的處理方法只有在該使用者取得中央處理器的佔有權時，才會執行相關的檢查，若當時有某個行程感到興趣的事件發生，可是此時中央處理器並不由該行程佔有時，則事件則無法傳達至該行程。

爲了要能夠及時的讓使用者空間的行程得知其感興趣的事件已經發生，由於當事件發生的時候中央處理器的控制權可能不已註冊監聽事件上，因此我們希望控制權能盡快轉移至有註冊事件發生的行程上，因此我們對\核心的排程器做了一些修改。

圖 Fig.4-9 爲修改過後的核心排程演算法，其中黃色和加粗的線條部分是我們加入的判斷以及處理，該處理主要爲以下兩個部分：

- 當目前行程爲 TASK_INTERRUPTIBLE 狀態時，若發現有該行程感興趣的事件已發生，則將其狀態設定爲 TASK_RUNNING。
- 在依正常程序選擇下一個執行的行程前，若已註冊的行程有感興趣的事件發生，則優先執行已註冊過的行程執行。

對於選擇註冊過的行程方面，由於所有註冊過的行程都會由一個靜態宣告的串列記

錄下來，因此我們檢查該串列中是否有發生興趣的行程，若是，則優先執行該行程，且在檢查的同時，若是該事件只有一個未處理的事件，則將其移至串列的後端。

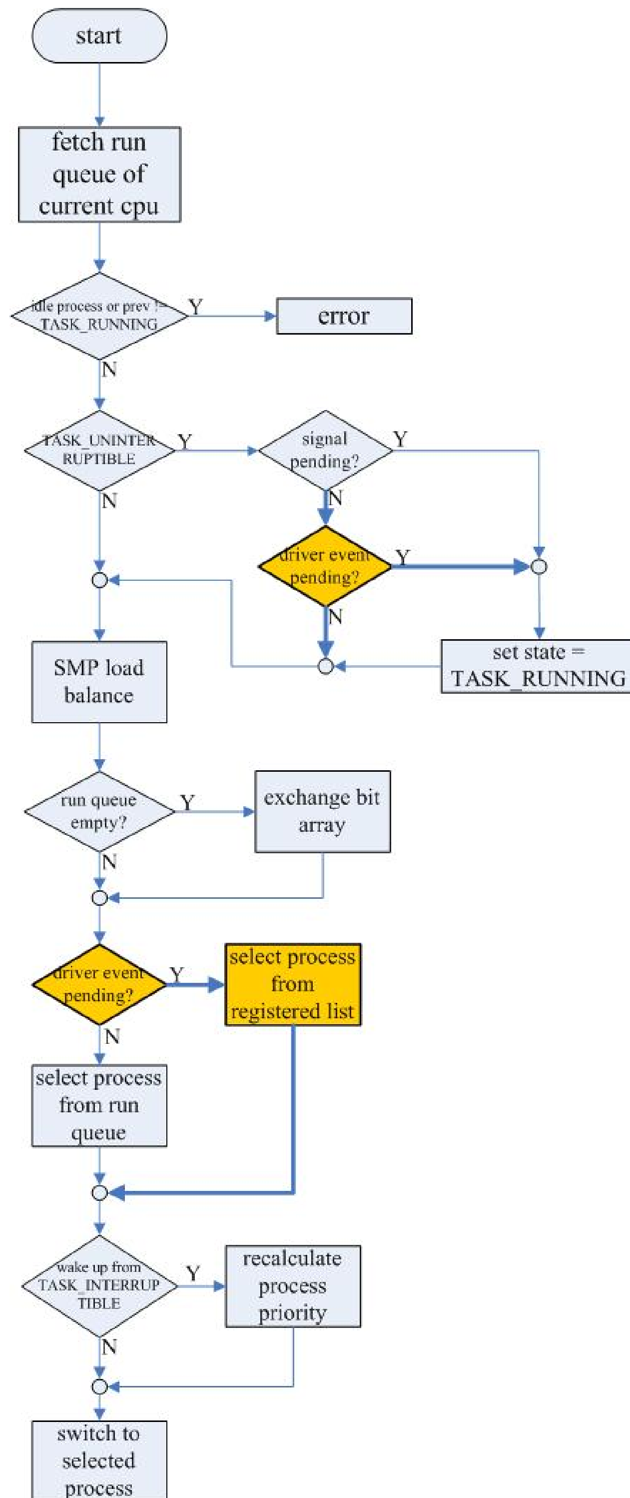


Fig.4- 9 修改過後的排程演算法

4.5 系統呼叫的重新執行

章節 4.4.2 中我們曾經提及，當發現當目前行程為 TASK_INTERRUPTIBLE 狀態時，若發現有該行程感興趣的事件已發生，則將其狀態設定為 TASK_RUNNING，如此一來，若是目前行程尚有多餘的時間切片的話，便可以繼續執行，以便對該發生的事件進行處理。

初看之下，以上這段處理似乎十分自然，但是，我們知道，行程會由 TASK_RUNNING 狀態轉移到 TASK_INTERRUPITBLE 狀態，必然是因為該行程的要求無法即時的被完成，而使用者空間的行程要向核心做出要求的話，必須通過系統呼叫的方式，因此，當我們將行程的 TASK_INTERRUPTIBLE 狀態改變為 TASK_RUNNING 時，其實之前行程向核心，也就是作業系統，經由系統呼叫所做的要求並未被完成 (例如，寫入檔案，要求更多的記憶體...)，而行程的狀態卻被換成 TASK_RUNNING 了。

現在，讓我們來看一下如果我們只將現在行程的狀態由 TASK_INTERRUPTIBLE 換成 TASK_RUNNING 的話會發生什麼樣的狀況？

當使用者空間的行程執行了一個系統呼叫後，IP 暫存器將會指向該系統呼叫的下一個指令 (此在核心空間時將被儲存在核心堆疊內)，接下來，系統便進入核心空間執行以便完成系統的要求，當使用者的要求無法被馬上滿足時，核心程式會執行如以下的程式碼，將該行程加入一個等待佇列當中，並且呼叫排程器排程：

```
DECLARE_WAITQUEUE(wait, current);
…
…
add_wait_queue(q, &wait);
while(!condition){
        set_current_state(TASK_INTERRUPTIBLE);
        if(signal_pending())
        /* handle signal */
        schedule()
}
set_current_state(TASK_RUNNING);
remove_wait_queue();
```

Fig.4- 10 將行程加入某個等待佇列的程式碼片段

當我們將行程的狀態由 TASK_INTERRUPTIBLE 換成 TASK_RUNNING 時，該程

式碼會由 schedule()的下一行開始執行，此時，程式進入 while 迴圈的判斷，若是等待的狀況(在圖 Fig.4-10 中以 condition 變數作爲代表)沒有成立，換句話說，也就是行程的要求無法被立即滿足時，程式將會重新進入 while 迴圈當中執行，又將行程的狀態設定爲 TASK_INTERRUPTIBLE，然後又呼叫 schedule()，形成類似無窮迴圈。

瞭解了只改動行程的狀態會發生什麼事情後，我們來說明要如何避免前段所述的情況發生；根據 Linux 系統在傳統訊號的處理上，當系統呼叫無法完成，且有訊號傳達時，圖 Fig.4-10 中的 signal_pending()的回傳值爲一個非零值，此時，系統呼叫必須回傳 EINTR、ERESTARTNOHAND、ERESTARTSYS 或 ERESTARTINTR。

在傳統的信號機制當中，根據行程各種信號的不同動作(忽略、捕捉或預設動作)，在信號處理的核心函數內系統會做出對應的不同處理，表 Table.4-2 表示了行程對信號的動作和系統呼叫回傳值的對應，以及核心信號處理函數會採取的動作[10]。

行程對信號動作	系統呼叫回傳值			
	EINTR	ERESTARTSYS	ERESTARTNOHAND	ERESTARTNOINTR
預設	*終止*	*重新執行*	*重新執行*	*重新執行*
忽略	*終止*	*重新執行*	*重新執行*	*重新執行*
捕捉	*終止*	*不一定*	*終止*	*重新執行*

Table.4- 2 行程對信號動作和系統呼叫回傳值對系統呼叫執行影響之列表

其中，表 Table.4-2 中所指的「不一定」表示要根據 SA_RESTART 這個旗標的設定與否來判斷(使用者空間行程可以利用 sigaction 系統呼叫設定此旗標)，若該旗標被設定，系統呼叫將會被重新執行。

在本系統當中，根據系統呼叫的回傳值的處理大致上仍然依照表 Table.4-2 所示的規則來進行，此外，當系統發現目前行程有設備信號尚未被處理時，。

我們根據行程是否註冊處理函數來討論，如果使用者空間的行程沒有註冊處理函數，則當系統呼叫回傳 EINTR 時會終止，而系統呼叫回傳 ERESTARTSYS、ERESTARTNOHAND、ERESTARTNOINTR 時會重複執行系統呼叫；當使用者空間行程註冊了處理函數時，當系統呼叫回傳 EINTR 以及 ERESTARTNOHAND 時，系統呼叫將被終止，而當系統呼叫回傳 ERESTARTSYS 以及 ERESTARTNOINTR 則會重新執行中

斷的系統呼叫。

首先，我們將 signal_pending()這個函數的判斷加入判斷設備信號是否尚未處理的判斷，如下所示：

```
static inline int signal_pending(struct task_struct *p)
{
        if(p->thread_info->iw_flag == 1){
                return 1;
        }else{
                return unlikely(test_tsk_thread_flag(p, TIF_SIGPENDING));
        }
}
```

Fig.4- 11 signal_pending()函數之修改

在設備信號的處理函數 handle_iw_event_sig()當中，我們必須判斷是否欲處理的信號，如果有未處理的信號，而且又有註冊處理函數時，則以如下的程式碼來處理：

```
if(regs->orig_eax > 0){
        switch(regs->eax){
                case -ERESTARTNOHAND:
                        regs->eax = EINTR;
                        break;
                case -ERESTARTSYS:
                case -ERESTARTNOINTR:
                        regs->eax = regs->orig_eax;
                        regs->eip -= 2;
        }
}
```

Fig.4- 12 重新執行系統呼叫。

如果行程並未對發生的設備事件註冊處理函數時，則會在設備信號的處理函數 handle_iw_event_sig()中執行以下的程式碼，除了回傳值爲 EINTR 之外的情況都需要重複執行系統呼叫：

```
if(regs->orig_eax > 0){
        if(regs->eax == -ERESTARTNOHAND ||
                regs->eax == -ERESTARTSYS) ||
                regs->eax == -ERESTARTNOINTR){
                regs->eax = regs->orig_eax;
                regs->eip -= 2;
        }
}
```

Fig.4- 13 沒有註冊處理函數時的系統呼叫重新執行

4.6 使用者和核心的切換

根據章節 4.4.1 中所介紹的系統返回程式碼，當核心發現欲返回的使用者空間行程有感興趣的事件發生時，會呼叫 handle_iw_event_sig()函式以處理，在這個函式中將會做所有切換到使用者空間行程的設定及動作。

本系統主要實作的平台爲英特爾 32 位元平台 (Intel Architecture 32-bits，IA32) ，在使用者和核心空間的切換方面，我們採用 Linux 對於傳統信號處理中執行信號處理函式之方法，該方法可分爲兩大步驟：

1. 將目前儲存在核心模式堆疊的使用者空間硬體環境儲存到使用者模式堆疊。
2. 置換目前核心模式堆疊中的使用者空間硬體環境。

圖 Fig.4-14 爲一般狀況下行程由核心空間返回使用者空間前核心模式堆疊和使用者模式堆疊的狀態，當行程由使用者空間跳至核心空間時，核心會先將使用者空間的硬體環境儲存在核心堆疊中，其中的 IP 暫存器值存著下一個要執行指令的記憶體位址，SP 暫存器儲存著堆疊端的位置。

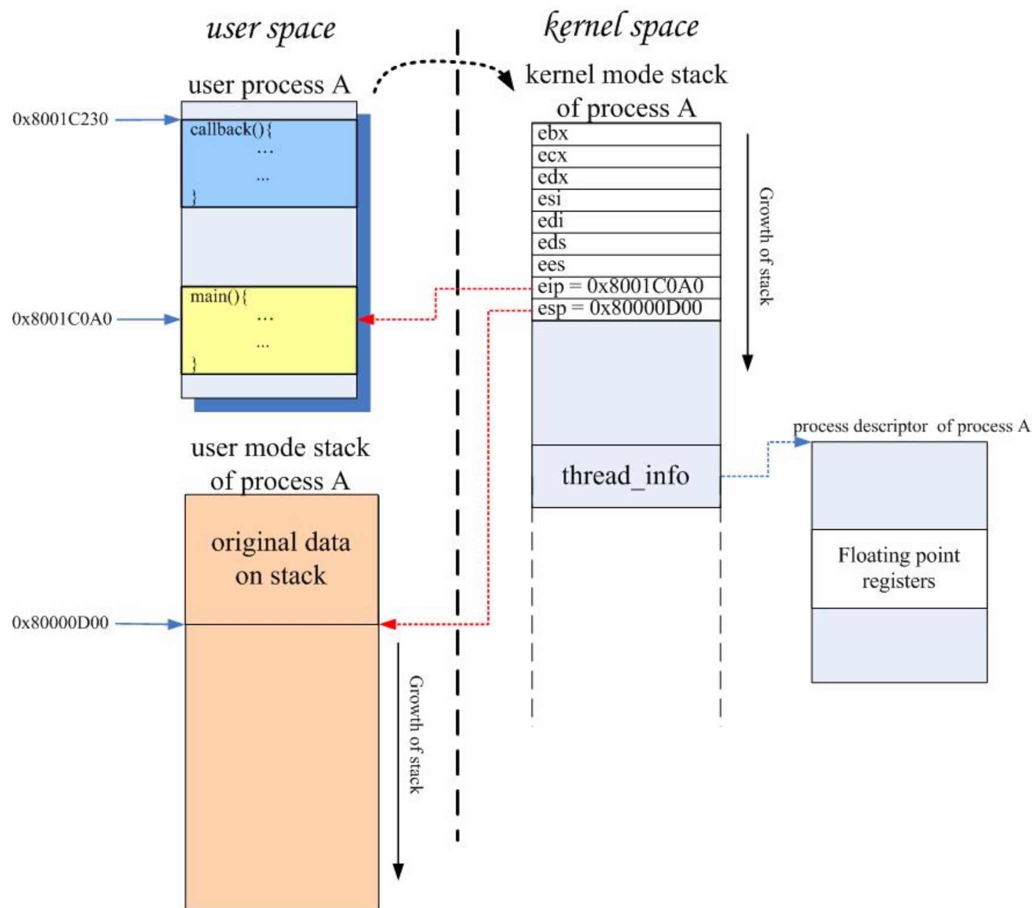


Fig.4- 14 正常狀態下的核心模式堆疊和使用者模式堆疊

我們的作法是先將一段系統之後欲執行的組語碼寫在使用者模式堆疊的頂端，再將原本儲存於核心模式堆疊的硬體環境 (hardware context) 儲存在使用者模式堆疊中，最後將回存位址 (return address) 寫入使用者模式堆疊的頂端，並且使用核心堆疊內的行程所註冊的信息處理函數的硬體環境 (包含 IP 暫存器) 取代原來核心模式的硬體環境。

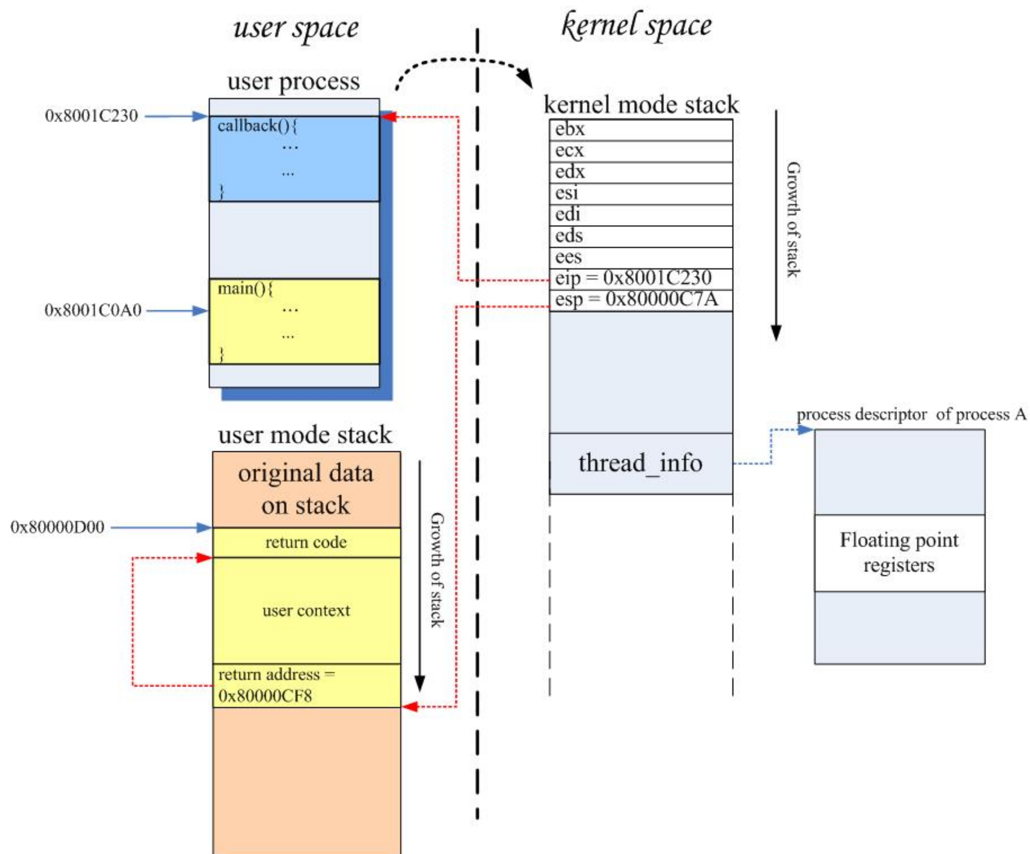


Fig.4- 15 執行回叫函式前的核心模式堆疊和使用者模式堆疊

Fig.4-15 爲執行回叫函數前的核心模式堆疊和使用者模式堆疊的狀態，此時原本的硬體環境已經被儲存到使用模式堆疊上，且核心模式堆疊上的硬體環境已經被置換爲欲執行回叫函數的硬體環境，其中相對應 IP 暫存器的記憶體位置儲存著回叫函數的記憶體位置，而相對應 SP 暫存器的位址則儲存著目前使用者模式堆疊的頂端位址，經過這樣設定使用者模式堆疊以及核心堆疊之後，核心的處理便返回 entry.S，根據圖 Fig.4-8 所示，程式將會跳至 restore_all 函式處理，將核心模式堆疊中的使用者硬體環境回復，並且開始執行使用者的回叫函式。

當回叫程式執行完之後，會由使用者模式堆疊的頂端找尋要跳回的位址，由圖 Fig.4-15 中我們可以看出，堆疊頂端所存放的位址爲指向堆疊中一個固定的位址 (原本的 SP 暫存器值減 8) ，該位址存放著我們欲執行的程式碼 (化爲二進位碼存在堆疊上) ，因此當回叫函式執行完之後，便會執行堆疊中我們放入的程式碼，該段程式碼相當於圖 Fig.4-16，也就是呼叫我們所新增的一個系統呼叫，iw_sigreturn()。

```
popl    %eax
movl    NR_iwsigreturn  %eax
int     $0x80
```

Fig.4- 16 置放於堆疊中的程式返回碼

iw_sigreturn()系統呼叫主要的任務是將之前存放入使用者模式堆疊中的硬體環境，回復至圖 Fig.4-14 的狀態，系統呼叫之後若沒有感興趣的事件發生，會進入 entry.S 中執行，進入系統呼叫後的判斷。

第五章 操作實例

5.1 操作實例

圖 Fig.5-1 為一個簡單的程式碼，用以驗證本論文所實作出來系統的可行性，該程式碼的主體部分為一個無窮空迴圈，程式碼的起始部分使用本論文實作的系統呼叫介面 iwsignal()註冊事件處理函數，而處理函數中會將發生的事件印出在終端機上，在本例中我們註冊了 devup()以及 devdown()作為介面啟用以及停用的處理函數。

```
#include <stdio.h>
#include <stdlib.h>
#include <asm/unistd.h>
#include <signal.h>
#include  "iwsig.h"

char  device[IWNAMESIZ];

void devup(int sig)
{
        printf( "devup %s\n" , device);
        return;
}

void devdown(int sig)
{
        printf( "devdown %s\n" , device);
        return;
}

int main()
{
        iwsignal(IWEV_DEVUP, devup, device, IWNAMESIZ);
        iwsignal(IWEV_DEVDOWN, devdown, NULL, 0);
        while(1);
        return 1;
}
```

Fig.5- 1 「驅動程式層網路事件通知機制」範例程式

而圖 Fig.5-2 爲使用本機制所需要的定義檔，主要是定義了系統中網路事件的代碼代碼以及我們所新增的系統呼叫介面 iwsignal()的相關宣告。

```
#include <sys/types.h>
#include <linux/unistd.h>
#include <asm/sigcontext.h>

#define IWEV_DEVUP            0x1
#define IWEV_DEVDOWN          0x2
#define IWEV_REBOOT           0x3
#define IWEV_DEVSTATCHG       0x4
#define IWEV_DEVREG           0x5
#define IWEV_DEVUNREG         0x6
#define IWEV_DEVCHGMTU        0x7
#define IWEV_DEVCHGADDR       0x8
#define IWEV_DEVGODOWN        0x9
#define IWEV_DEVCHGNAME       0xA
#define IWEV_DEVCHINADDR      0xB
#define IWEV_DEVCARON         0xC
#define IWEV_DEVCAROFF        0xD

#define IWNAMESIZ             16
typedef void (*__iwsig_handler_t)(int);

__syscall4(int, iwsignal, int, signo, __iwsig_handler_t, iw_handler, char *, namebuf,
unsigned long, bufsize);
__syscall1(int, iwsigreturn, struct sigcontext *, sigcontext);
```

Fig.5- 2 「驅動程式層網路事件通知機制」的標頭檔

接下來則是程式執行的結果：

[步驟一]

- [執行動作]：開始執行程式。
- [程式動作]：執行實作函數 iwsiscall，註冊處理函數，如圖 Fig.5-3。
- [系統反應]：處理函數已註冊，如圖 Fig.5-4。

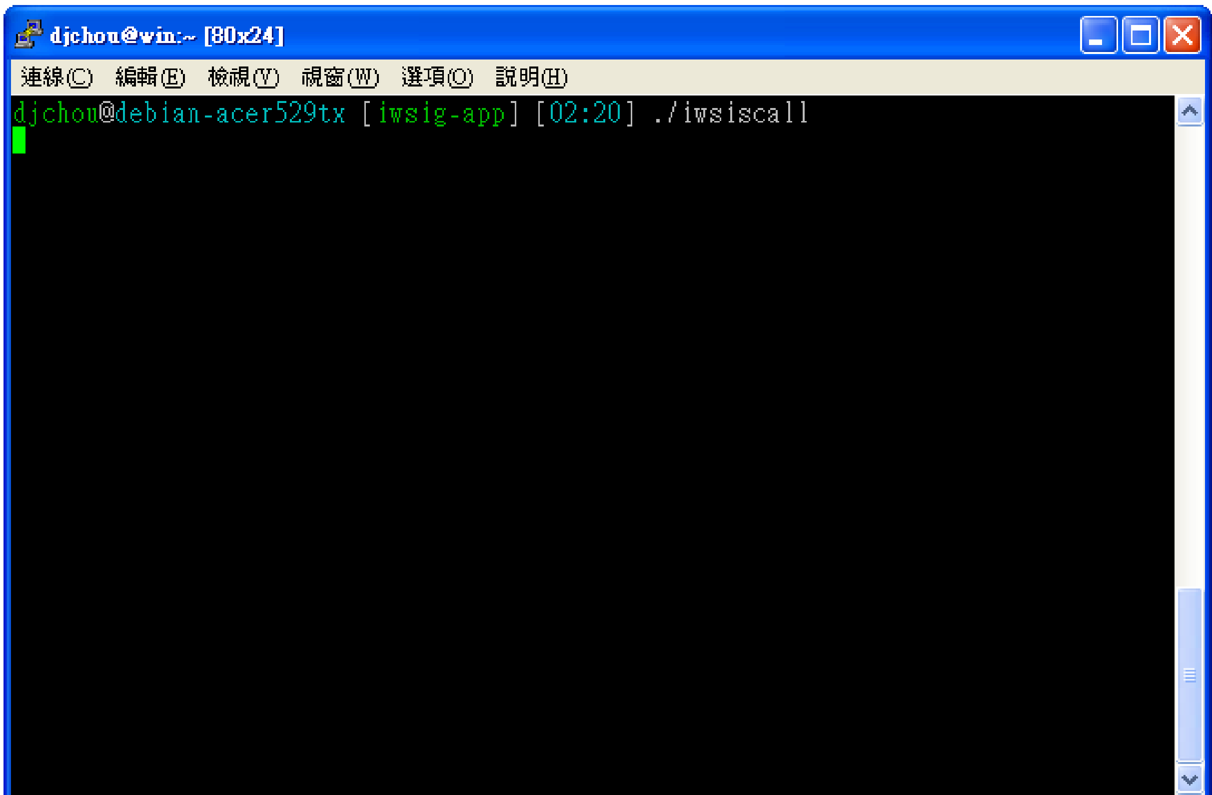


Fig.5- 3 範例程式開始執行

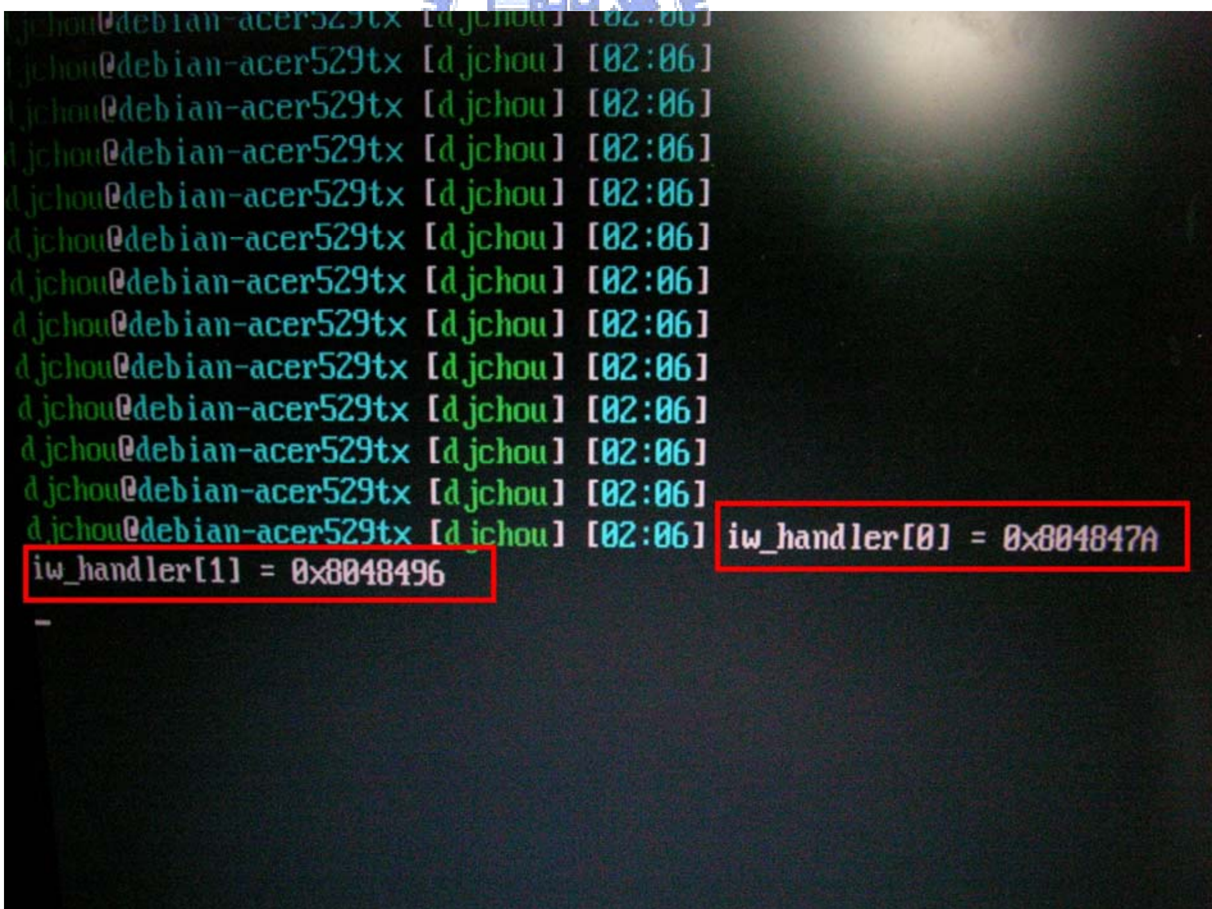


Fig.5- 4 範例程式註冊兩個處理函數

[步驟二]

◆ [執行動作]：啓用 wlan0 網路介面，如圖 Fig.5-5。

◆ [程式動作]：印出對應訊息，表示事件已經傳達，如圖 Fig.5-6。

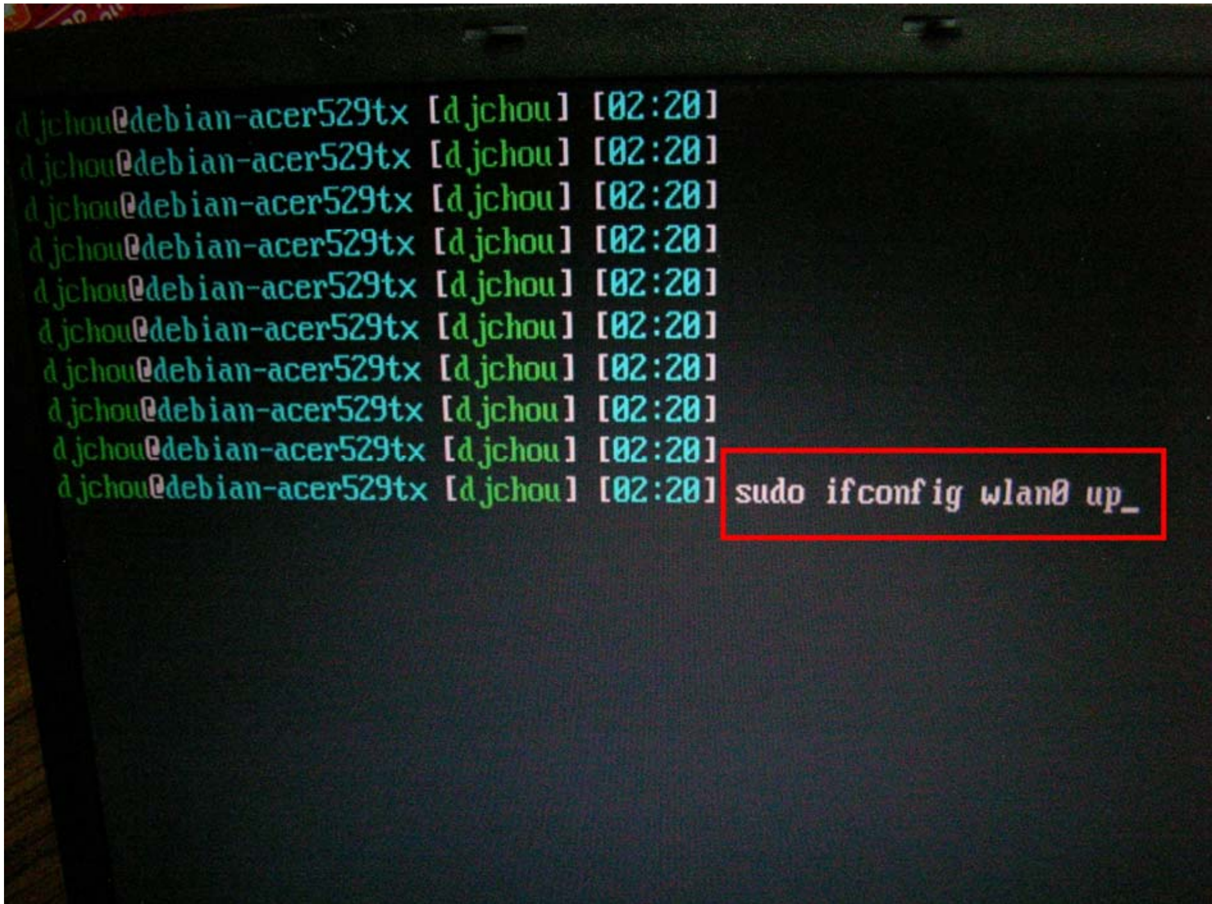


Fig.5- 5 啓用網路介面 wlan0

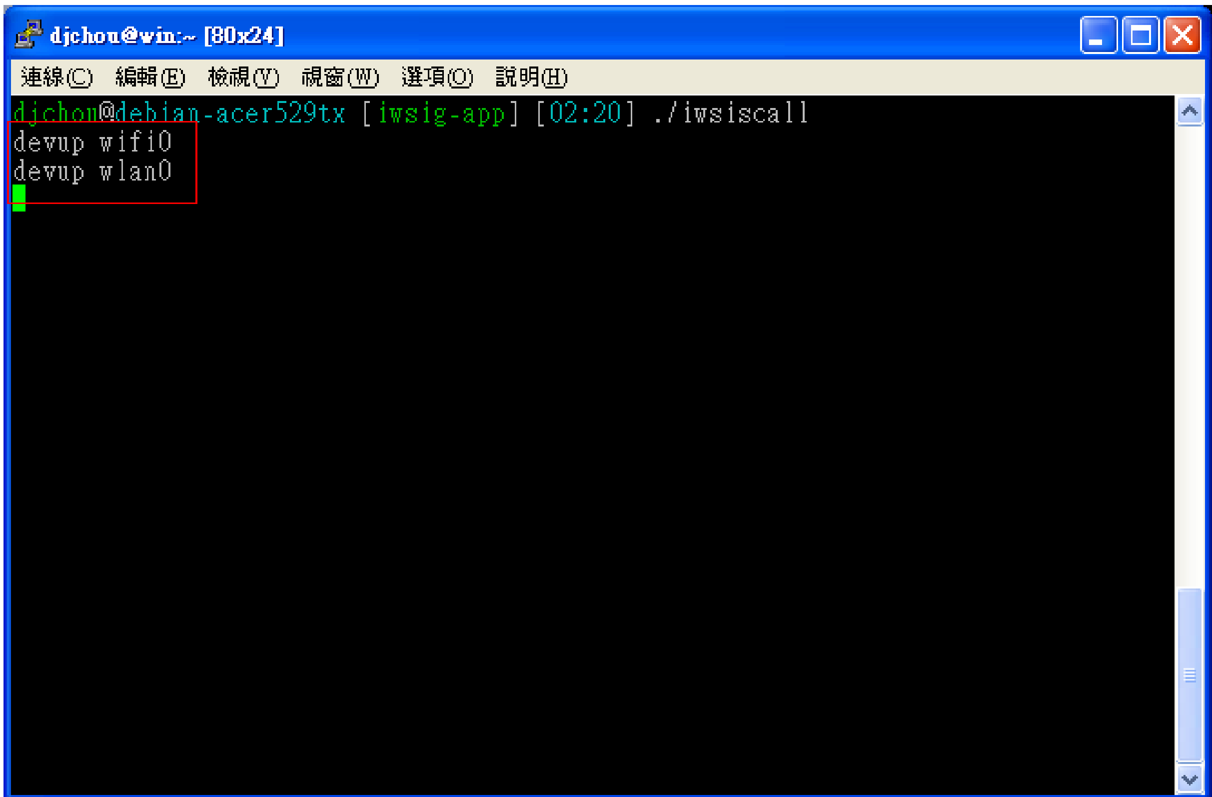


Fig.5- 6 介面改變傳達至範例程式，印出啓用的介面

[步驟三]

- [執行動作]： 停用 wlan0 網路介面，如圖 Fig.5-7。
- [程式動作]：印出對應訊息，表示事件已傳達，如圖 Fig.5-8。

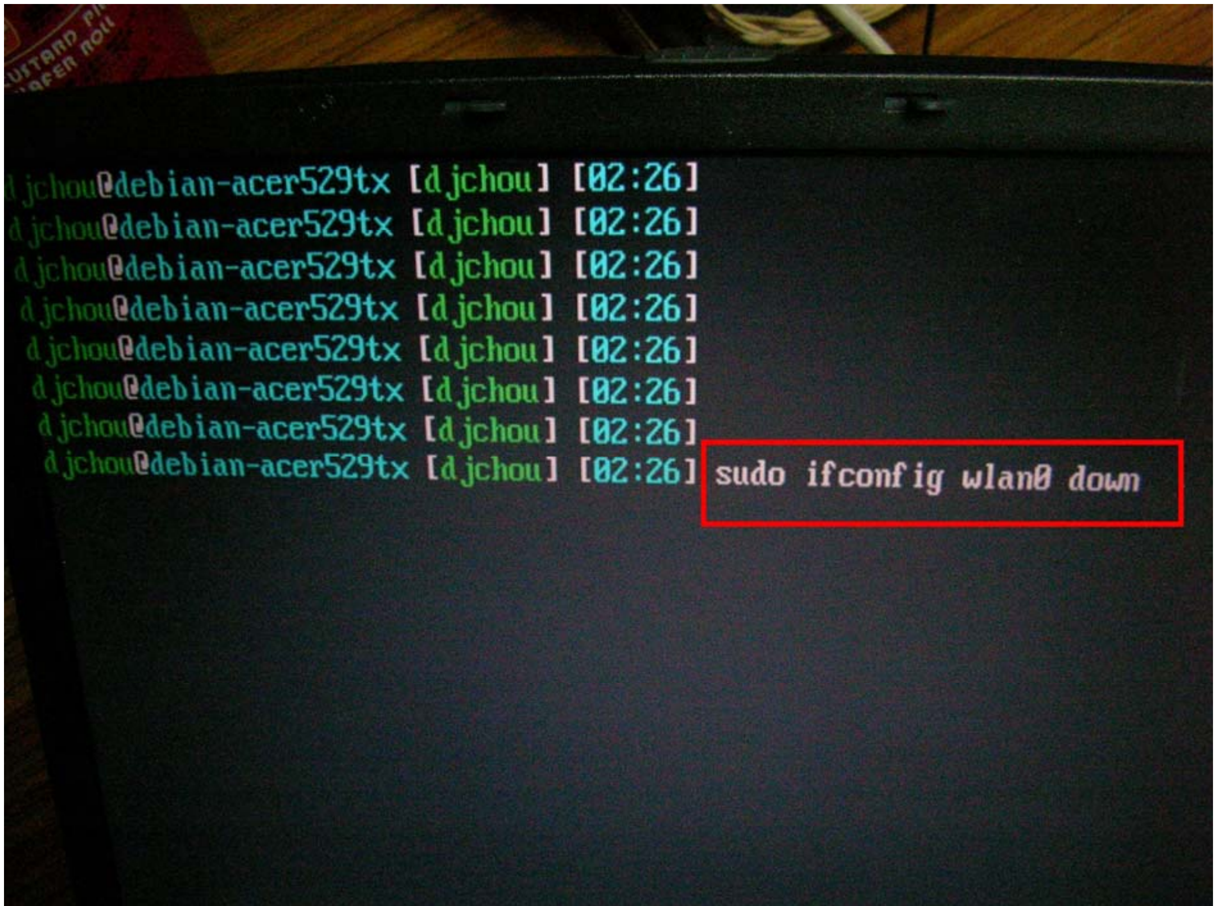


Fig.5- 7 停用網路介面 wlan0

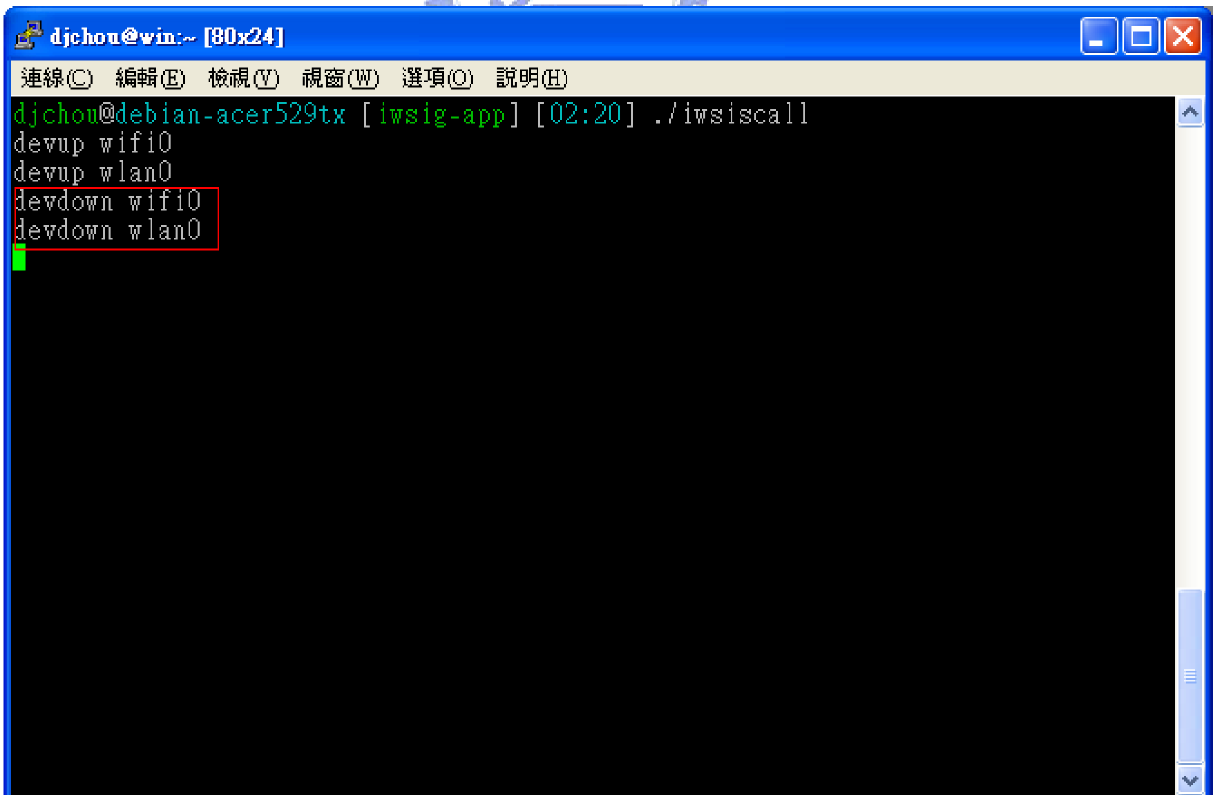


Fig.5- 8 介面改變傳達至範例程式，印出停用的介面

[步驟四]

- ◆ [執行動作]：程式停止。
- ◆ [程式動作]：程式停止，如圖 Fig.5-9。
- ◆ [系統反應]：顯示註冊函數已經移除，如圖 Fig.5-10。

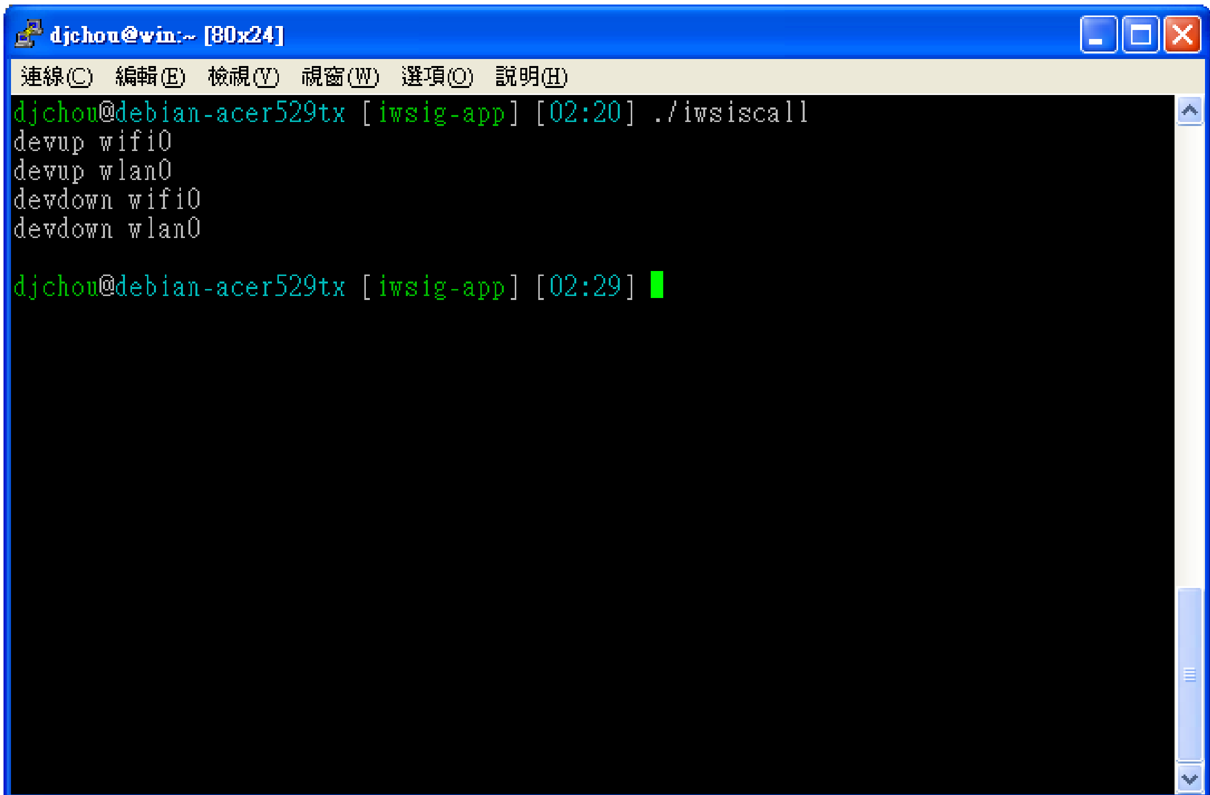


Fig.5- 9 按下 Control-C，程式停止執行

```
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29]
djchou@debian-acer529tx [djchou] [02:29] Removing [2705][0x3] from chain
_
```

Fig.5- 10 範例程式註冊的處理函數被移除

第六章 結論與未來工作

6.1 結論

原本 UNIX 或是類 UNIX 系統 (UNIX clone) 中，使用者空間的程式想要取得下層網路介面的狀態時，必須利用系統呼叫取得，當使用者空間程式欲對網路介面的狀態改變作相對應處理時，程式必須不斷的使用系統呼叫取得網路介面的狀態，並且和前一次取得的網路介面狀態作比對，也就是說，就算網路介面的狀態都沒有改變，程式仍然要不斷的發出系統呼叫取得網路介面的資訊，cpu 的時脈因此會浪費在這些多餘的系統呼叫上。

在本論文中，我們提出了一套稱為「驅動程式層網路事件通知機制」的新方法，並且在 Linux 平台上實作，結合了 Linux 中的核心通知者串鍊機制以及傳統 UNIX 信號的傳訊機制，由作業系統通知使用者層程式網路介面狀態的改變，此外，也對原本的排程方法做出了些微的改變，以加速信號的處理；當網路介面狀態沒有改變的時候，程式可以執行其想要的動作或是呼叫會進入睡眠狀態的系統呼叫，因此 cpu 的時脈不會被浪費在取得網路介面資訊的系統呼叫上。

使用者空間的程式將自己處理網路介面狀態改變的程式碼包裝成函數，並且利用類似 UNIX 中的 signal()系統呼叫介面註冊處理函數以及相對應的事件代碼，當網路介面狀態發生改變時，在核心中會利用核心通知者串鍊處理該事件，而將事件傳達給使用者則是利用傳統 UNIX 的信號機制達成。

6.2 未來工作

目前該機制只在 386 及其相容平台的 Linux 作業系統上實作，由於現在配有多種網路接取介面的手持式設備逐漸風行，而且手持式設備例如 PDA 等的 cpu 時脈比個人電腦來得珍貴（因為時脈較慢），因此，該機制在一些嵌入式或是運算能力不強的架構中顯得更為有用，所以，將該機制移植到一些手持式裝置常用的平台上，例如 ARM 平台或是 MIPS 平台為一個未來可以努力的方向。

此外，本論文僅提出了機制，未來希望能夠以此機制為基礎，實作一套完整的介面管理程式，幫助使用者在多網路介面的情況下管理網路介面；對於其他的應用程式方

面，使用該機制加入網路介面相關的處理，使得應用程式本身即具有網路介面的處理能力也是一個選擇。

最後，本論文對於排程器的修改方面爲優先執行有事件發生的行程，也就是說，有事件發生的行程執行優先權爲最大，當系統中有許多使用本機制的行程在執行，且網路介面狀態頻繁的改變時，可能會阻礙到系統其他的行程的執行，比較好的方法應該是將「行程有未處理的網路事件」這個條件加入系統執行優先權的計算因素之一，使得該種行程的執行優先權較高，然後以一般的排程來處理，如此一來對於系統其他的行程影響較少，但是隨之而來的負面影響就是網路事件的傳達會變得較慢。

參考文獻

[1] Linux Programmers Manual (Linux man pages), ioctl(2), 2000-09-21

[2] Linux Programmers Manual (Linux man pages), sigaction(2), 2001-12-29

[3] Linux Programmers Manual (Linux man pages), sigprocmask(2), 2001-12-29

[4] Linux Programmers Manual (Linux man pages), select(2), 2001-02-09

[5] Linux Programmers Manual (Linux man pages), poll(2), 1997-12-07

[6] Portable Operating System Interface, http://www.knosof.co.uk/posix.html

[7] phhttpd home page, http://www.zabbo.net/phhttpd

[8] Raimo Nikkilä, "Vertical Handover Study Cluster: VHO DAEMON", Product Modelling and Realisation Group, Helsinki University of Technology, 2005-03-08

[9] Andrew Josey, editor, "Go solo2: The Authorized Guide to Version 2 of the Single UNIX Specification", Chapter 9 and Chapter 10, May 1997.

[10] Danial P.Bovert, Marco Cesati, "Understanding The Linux Kernel, 2nd edition". Chapter 10, December 2002.

[11] Alessandro Rubini, Johathan Corbet, "Linux Device Driver, 2nd edition", Chapter 1, June 2001.

[12] Klaus Whhrle, Frank Pählke, Hartmut Ritter, Daniel Muller, Marc Bechler, "The LINUX Networking Architecture – Design and Implementation of Network Protocols in the Linux Kernel", Chapter 5, 2002.

[13] G. Carneiro, J. Ruela, M. Ricardo, "Cross-layer design in 4G Wireless Terminals". IEEE Wireless Communications, pp. 7-13, , April 2004.

[14] Jun-Zhao Sun, Jaakko Sauvola,"Mobile IP Applicability: Do We Really Need It?", IEEE Internation Conference on Parallel Processing Workshops, pp. 116-123, 2004