追蹤 Linux TCP/IP 核心 － 使用遠端除錯

論文領域：其它
蔡品再　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：ie855160@csie.fju.edu.tw , ydlin@cis.nctu.edu.tw
主要聯絡人：蔡品再 TEL：(03) 5712121 EXT. 87605

摘 要

Linux 可說是開放程式碼中最普遍的作業系統，可是想要了解為數眾多的核心程式碼卻是令人傷透腦筋，在這裡我們提供一個不錯的方法來接近它，一步步的了解它。我們會告訴大家如何在 x86 上使用 remote debug(由 host 對 target 進行動態 debug)來解剖 Linux 核心程式碼。並以 Linux TCP/IP 協定驅動程式為例，描述核心程式碼的流程。我們使用 KGDB 與 GDB 來實現 remote debug，並利用工具 KernProf 來剖析核心函式間相互的關係。另外還有能在單機上執行追蹤與中斷的工具 KDB。

關鍵字：Linux、Protocol Driver、Kernel Remote Debug、KernProf、Kdb 、Kgdb

一、動機

Linux 可說是當紅炸子機，打著 Open Source 的名號，號招天下英豪，一起建構一個好用、穩定、免費的作業系統。Linux 在網路的支援可說是非常的完整；網路功能內建於核心，不僅支援現行的 ipv4 又支援最新的 ipv6 的支援；又由於它是屬於 Open Source 的軟體，所以我們可以輕易的拿到他的原始碼，正好可以讓我們一探協定驅動程式的奧秘。也借此希望能夠找到一個很好的方法來了解 Linux 核心程式的寫作。

二、協定驅動程式與 Linux 網路實作

在此我們先了解一下什麼是協定驅動程式，並對 Linux 在網路方面的實作有點概念性的認識，以便配合下一節介紹的核心追蹤方法，進行網路協定處理流程之追蹤。

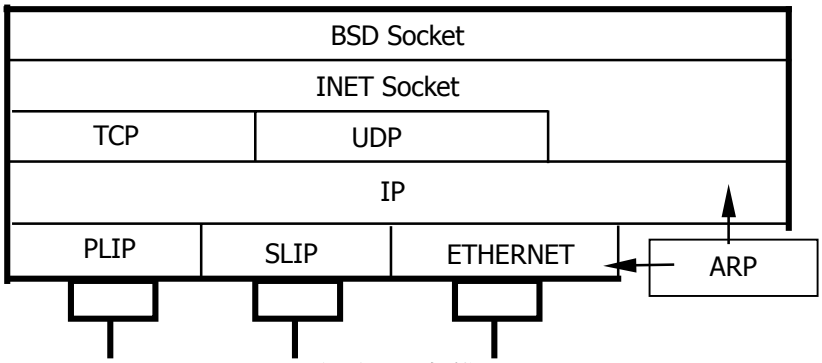
什麼是協定驅動程式

在一部個人電腦中，有關網路的部分大概可以分爲硬體與軟體的部分。硬體方面如網

路卡、數據機等，是負責將網路封包轉換成可以在網路介質中傳送的型態(如：將封包轉成電壓訊號在同軸電纜中傳送，轉成電波在空氣中傳送，轉成光束在光纖中)；軟體的部分又可細分爲三層，網路卡驅動程式(Adapter Driver)，協定驅動程式(Protocol Driver)，應用程式(Application)。應用程式就是我們常常使用的軟體部分，如 IE，NetTerm 等，通常應用程式會把要傳遞的資料交由下面的協定驅動程式來做進一步的處理。現行的網際網路所使用的通訊協定爲 TCP/IP，而協定驅動程式就是實作這個部分的軟體，當收到應用程式的資料後協定驅動程式就會把這些資料包裝成封包傳給網路卡驅動程式，然後再由硬體傳送出去。

Linux 在 TCP/IP 實作上的架構[1,2]

Linux 在網路上的實作採取分層的概念，如圖一所示，並以檔案系統來實現。整個網路協定的操作都是由經 BSD Socket 這一層來管理，而使用者透過檔案系統來連接 BSD Socket，並以類似操作檔案系統一般的操作網路通訊。BSD Socket 並不只支援 Internet socket，它支援了許多其他的 Socket 比如，Unix Socket，IPX，AX25 等。而我們常見的 TCP/IP 就是 BSD Socket 下的 INET Socket 的部分。所以在 TCP/IP 的實作上，第一層是 BSD Socket，再來是 INET Socket 層，其下還有 TCP，UDP 二層，也可以直接接觸更下面的 IP 層，最後就是網路通訊設備，如，乙太網路卡，數據機等。表一條例原始碼所在位置。



圖一：網路分層架構

層次	原始碼位置
BSD Socket	/linux/net/socket.c
INET Socket	/linux/net/ipv4/af_inet.c
TCP	/linux/net/ipv4/tcp_*.c
UDP	/linux/net/ipv4/udp_*.c
IP	/linux/net/ipv4/ip_*.c
ETHERNET	/linux/net/Ethernet
ARP	/linux/net/ipv4/arp.c

表一：TCP/IP 各層原始碼位置

三、追蹤 Linux 核心的方法

想要了解 Linux 核心原始碼是一件很難下手的事，尤其在 Linux 新版的核心不斷的推出，支援的功能一個一個的增加，想要看得懂原始碼更是難上加難。當然了，如果想要了解原始碼爲什麼要這樣寫，那一行要實現什麼樣的功能，直接動態追蹤程式碼執行過程是最好不過的了；至少比用編輯器來“硬看”(靜態)來得好懂多了。在這裡提供三個方法來追蹤核心程式碼，分述如後。

3.1 使用核心函式 PrintK

程式設計師最簡單的除錯工具大概就是 printf，很多寫過 c 語言的人，常常會用這個函式來傾印系統上的變數值以便除錯；不過在 Kernel Space 中是無法使用 printf 把訊息列印出來的。不過在 Linux 核心系統中有另外一個函式 – printk，可以傾印訊息，但其與 printf 最大的不同點在於 printk 並沒有辨法處理浮點數。另外，printk 還有所謂的 log level 可以幫訊息做分級的動作，如警告訊息，除錯訊息，嚴重訊息等，可說是這些訊息的 priority。在此我們使用 printk，配合 klogd、syslogd，將 printk 印出來的訊息存在 log 檔中；利用這些資訊來追蹤核心程式碼流程。更進一步的做法可以參考 [3]。

使用 printk 就像使用 printf 一樣，先在要印出訊息的地方加上 printk 的敘述，並指定它的 log level，之後再重新編譯核心。在這我們主要的是追蹤 Linux 在 TCP/IP 上的實作，其原始碼在/linux/net/ipv4 中。舉例如下：

```
（例）static void icmp_reply(struct icmp_bxm *icmp_param, struct sk_buff *skb)
                {
                        struct sock sk=icmp_socket->sk;
                        printk(KERN_INFO “icmp.c: icmp_reply()\n”);
                / *log level 爲「KERN_INFO」，印出 “icmp.c: icmp_reply()”，當然我們也可以印出一些核心變數，其用法和 printf 相似*/
                        if (ip_options_echo(&icmp_param->replyopts, skb))
```

再來就是設定 syslogd，編輯/etc/syslogd.conf，加入 kern.=info /var/log/kern_info 後，存檔重新開機，就可以在/var/log/kern_info 這個檔案中看到 printk 所印出來的訊息。

我們可以使用這樣的方式爲每一個函式加上 printk 的敘述，這樣我們就可以由 log 檔得知在網路進行溝通時，核心做了那些事！不過這還不是一個很好的方法，因爲我們只能得知整個函式的呼叫關係，不能在執行時一步步的追蹤程式碼。更不能設定中斷點或是修改其中的數值。

3.2 使用 KDB 追蹤核心原始碼[4]

由於 printk 無法讓我們設定中斷點，也不能使用單步執行來追蹤程式碼，所以在了解原始碼方面還是有點缺憾。接下來介紹一個可以讓我們設定中斷點，並能單步追蹤核心的工具 --- KDB。KDB (Built-in Kernel Debugger)是 Linux 核心的一部分，它支援在執行期間的核心記憶體的分析，也可以設定中斷點與單步執行。

安裝 KDB

KDB 是內嵌於核心之中的，所以安裝它的方式就是先要 patch 核心，所以在 patch 時，要注意核心的版本是不是和 KDB 所支援的核心版本是否一樣。我們可以在 KDB 的網頁 (http://oss.sgi.com/projects/kdb/)上下載 KDB，它的檔名會包含 Linux 核心的版本號，如：kdb-v0.6-2.2.13 指的就是 KDB 0.6 版，適合於 Linux2.2.13 的核心，將 KDB 下載之後存於 /usr/src/linux 中，就可以照下列步驟安裝：

STEP 1. 下指令「cd /usr/src/linux」切換到 Linux 核心的目錄
STEP 2. 下指令「patch –p1 < kdb-xxxxxx」執行 patch
STEP 3. 下指令「make *config」(make menuconfig or make xconfig)設定核心，把 CONFIG_KDB，CONFIG_FRAME_POINTER 這二個選項打開，然後重新編譯核心。
STEP 4. 用重新編譯過的核心開機，就可以使用 KDB 了。

如何使用 KDB

有二種方法可以啓動 KDB，一是在開機時經由 boot prompt 下達 kdb 這個參數給核心，如：LILO: linux kdb。或是在開完機之後，任何時候都可以按 Pause 鍵來啓動 KDB。啓動 KDB 之後，我們就進入了 Debug 模式了。鍵入 help 可以看看線上使用說明，鍵入指令”go ” 就可以回到系統中繼續原來的工作。現在我們可以執行一個網路程式，然後快速的按下 Pause 鍵進入 KDB 的 Debug 模式，然後可以下”ss ”來單步執行，或是用”ssb ”來執行程式直到下一個 branch/call。這樣我們就可以看到核心是怎麼樣執行程式的。另一個就是先設好中斷點，比如，我們可以把中斷點設在 sys_socketcall，指令爲”bp sys_socketcall ”這樣只要呼叫 sys_socketcall 這個函式，就會自動進入 KDB 的 Debug 模式，我們可以用 “ bt ” 做 BackTrace，再用 ss，ssb 等去追蹤。詳細的指令列表於表二.。

Command	Description
md	Display memory contents
mds	Display memory contents
mm	Modify memory contents
id	Instructions
go	Continue Execution
rd	Display Registers
rm	Modify Registers
ef	Display exception frame
bt	Stack traceback
btp	Display stack for process <pid>
bp	Set/Display breakpoints
bl	Display breakpoints
bpa	Set/Display global breakpoints

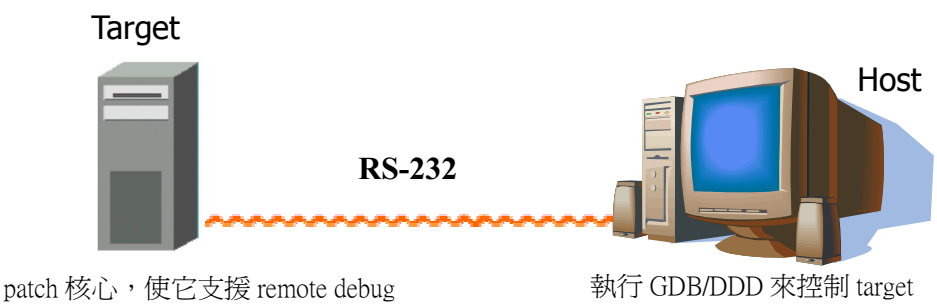
Command	Description
bc	Clear Breakpoint
be	Enable Breakpoint
bd	Disable Breakpoint
ss	Single Step
ssb	Single step to branch/call
ll	Execute cmd for each element in linked
env	Show environment variables
set	Set environment variables
help	Display Help Message
?	Display Help Message
ps	Display active task list
reboot	Reboot the machine immediately

表二：KDB 指令速覽

3.3 使用 KGDB 遠端除錯[5]

雖然 KDB 擁有了很完全的功能能讓我們追蹤核心程式碼，不過我們一般看到的 Linux 原始碼是由 C 所寫成的，而在 KDB 模式中，我們只能看到組合語言碼而已。組合語言對於想要了解整個核心程式碼這件事來說是非常可怕且困難的。所以我們就得借助更強而有力的除錯程式來追蹤核心程式碼。

GDB 是 Richard Stallman 所寫的一個強而有力的除錯程式，它本身是一隻程式，所以在系統核心除錯方面會有所限制，比如設中斷點，或是單步執行，都必需要把作業系統給停下來，但是 GDB 本身也需要作業系統的執行，所以在單機時，我們只能讀取核心的資料，不能設中斷點或是單步執行。幸運的是 GDB 還支援所謂的遠端除錯模式，它是以一台電腦當 Host，執行 GDB 透過 RS-232 來控制另一台當做 Target 的電腦，如圖二所示；另一台電腦則執行要除錯的系統核心，這樣就可以在作業系統上做中斷點與單步執行了。這樣的功能



圖二：遠端除錯示意圖

在 Alpha，Sparc 上的 Linux 早有支援，可是在 x86 上卻遲遲沒有支援。不過現在已經改觀了，有人爲 Linux x86 版本的核心寫出了支援 GDB Remote Debug 的 patch，名爲 KGDB。我們可以在它的網址(http://oss.sgi.com/projects/kgdb/)下載這個 Target 的 patch。

安裝 KGDB

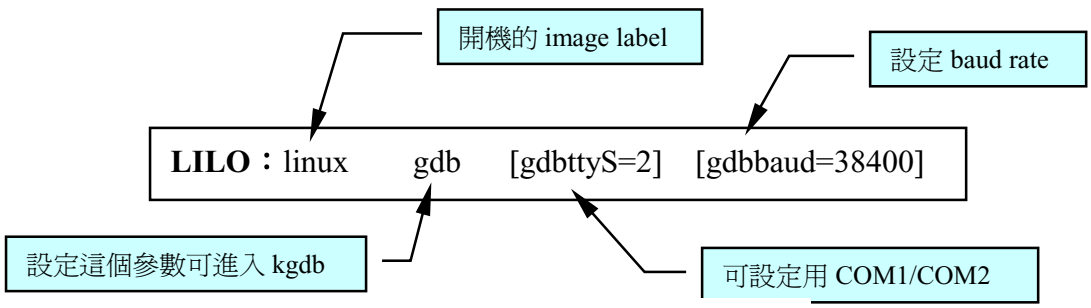
KGDB 和 KDB 一樣是一個核心的 patch 檔，下載時必需要注意版本的問題，必需針對核心版本下載適合的 patch。然後重新設定核心，重新編譯，然後用新的核心重開機後就能使用了。這些步驟和安裝 KDB 很像。在這我們以核心 2.2.12 爲例，先在網站上下載 kgdb0.2-2.2.12 並把檔案存於/usr/src/linux 中，並照下列步驟安裝：

STEP 1. 下指令「cd /usr/src/linux」切換到 Linux 核心的目錄
STEP 2. 下指令「patch –p0 < kgdb0.2-2.2.12」執行 patch
STEP 3. 下指令「make *config」(make menuconfig　or　make xconfig)設定核心，把 CONFIG_GDB 這個選項打開，然後重新編譯核心。
STEP 4. 改寫 lilo 並將重新編譯過的核心加入。

啟動 KGDB

要使用遠端除錯，必需要先把二台電腦用 RS-232(Null Modem)的線來連接二台電腦的 COM Port，然後在 Host 端執行 GDB，並把核心程式載入，其檔案就在/usr/src/linux/vmlinux 這個檔是剛才重新編譯出來的核心，不過是沒有壓縮的版本，並且含有除錯訊息的。可開機版一般在/usr/src/linux/arch/i386/boot/bzImage，我們可以把這個 bzImage 烤貝到 Target 端的電腦，並用這個 image 開機進行除錯。整個設定流程分敘如下：

首先在 Target 用剛才編譯過核心開機，在 boot prompt 下參數 gdb 之後，開機到一半就會出現“waiting for remote debug…”並中斷開機，等待遠端的連線。參數設定如圖三。

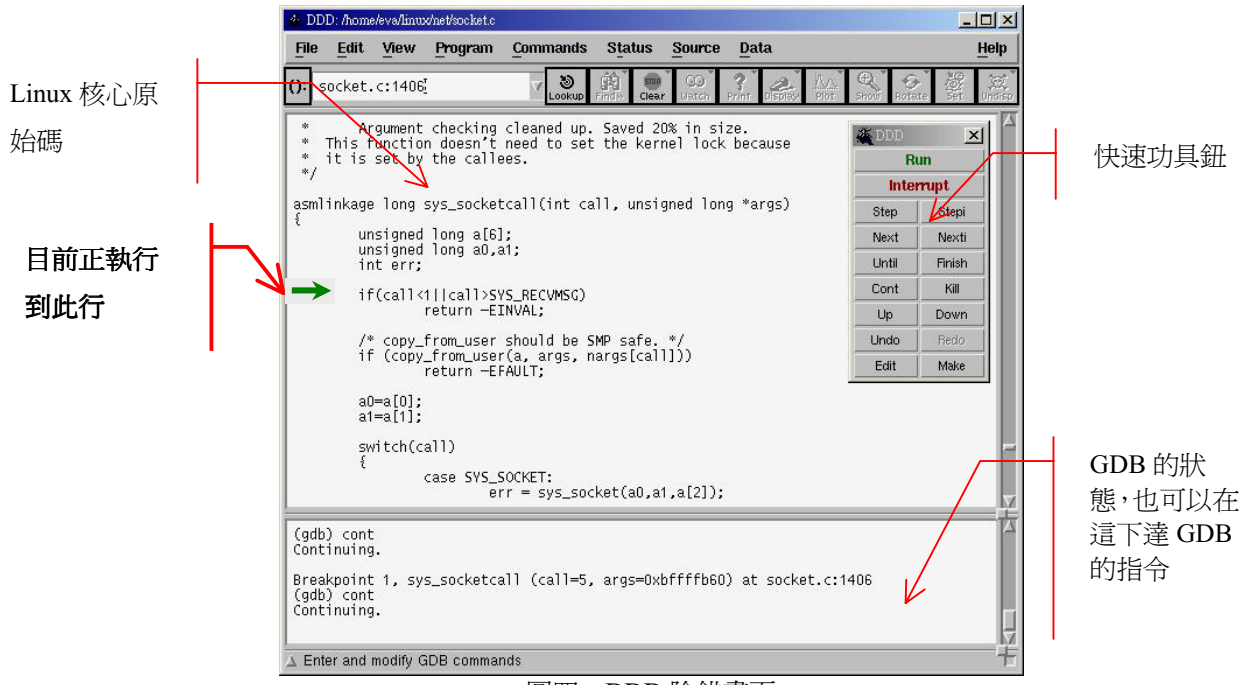


圖三：KGDB 開機設定

接著設定 Host，先執行 GDB 並將 vmlinux 載入，接著鍵入下面指令來連接：

Step 1.	set remotebaud 38400	//設定 baud rate
Step 2.	target remote /dev/ttyS1	//設定本機要用那個 COM Port 和遠端連線
		//在 Step 2.之後就可以看到訊息，一個中斷點在 xxx 位置
		//如果用的是 DDD 更可以直接看到在程式的某一行上有個
		//STOP 的符號以及一個指向目前所在行號的前頭。
Step 3.	cont	//這個指令可以讓遠端繼續完成開機的程序。

這樣整個環境算是 Setup 好了，接下來如果要啓動除錯模式的話，可以在 Target 按“Ctrl+c”或是在 Host 下“interrupt”來中斷執行。然後可以用“step”來做單步的執行，或是用“break sys_socketcall”設中斷點於 sys_socketcall 這個函式上。其許多的用法參考 GDB 的使用手冊或是線上說明。不過在這我們還是建議用有 GUI 介面的 DDD，如圖四所示，因爲這樣我們可以很清楚的看到原始碼與目前執行到的程式敘述，這樣一步步的追蹤核心，我們就能很容易地了解核心程式碼的寫作。



圖四：DDD 除錯畫面

3.4 追蹤核心記錄 – 使用 KernProf 對程式碼初步了解[6]

現在我們有了很強悍的工具來對核心程式碼做單步的追蹤，可是如果我們要設中斷點，我們得設在那？對核心一竅不通的話跟本不知道從何下手！不過我們可以用 KernProf

來了解核心呼叫過那些函式。KernProf 和 gProf、Prof 一樣，都是用來做效能測式用的，他能夠爲我們找出那個函式用了多少的時間，那個函式呼叫了那些子函式，它的父親又是誰？又某函式一供執行了幾次。不過 KernProf 和 gProf 不同的是，KernProf 是用於 Linux 的核心，而 gProf 及 Prof 是用於一般 User Space 的程式的。在此，我們主要是利用 KernProf 裡 Call Graph 的功能，它會詳盡的告訴我們核心呼叫過那些函式，以及他們之間的關係；知道這些之後，我們就能知道要在那些函式上下中斷點來追蹤核心。

安裝 KernProf

安裝 KernProf 有點複雜，它和 KDB，KGDB 一樣都是內建於核心的，一樣都得要注意核心版本的問題。此外，因爲 gcc 2.95.2 版的 bug，我們得要重新 patch 過 gcc 才能正確無誤的把 KernProf 給編譯出來[7]。安裝過程條列如下：

Step 1. patch gcc

1. 先到 http://oss.sgi.com/projects/kernprof/ 下載"GCC-PATCH"，以及在就近的網站下載"gcc-2.95.2"的原始碼。
2. 將 GCC-PATCH 移至 gcc-2.95.2 原始碼的目錄中，下指令「patch –p1 < GCC-PATCH」來 patch gcc
3. 下指令「configure」讓 gcc 自動設定一些參數
4. 下指令「make bootstrap」開始編譯 gcc
5. 下指令「make install」來安裝 gcc

Step 2. 安裝 KernProf

1. 在 http://oss.sgi.com/projects/kernprof/ 下載" PROFILE-06-240-2.PATCH"，以及在就近的網站下載 Linux 核心 2.4.0.Test2。
2. 將核心解開後，把" PROFILE-06-240-2.PATCH"移到核心目錄中，下指令「patch –p1 < PROFILE-06-240-2.PATCH」來 patch 核心。
3. 「make menuconfig」或「make xconfig」來設定核心參數，將
"Character devices Kernel Profiling Support "，
" Kernel hacking Compile with frame pointers"
" Kernel hacking Instrument kernel with calls to mcount()"這三個選項 enable。
4. 「make bzImage」編譯核心，用這個核心重開機就能使用 KernProf。

Step 3. 安裝 KernProf 的工具

1. 在 http://oss.sgi.com/projects/kernprof/ 下載"KERNPROF-08-2I386.RPM"
2. 安裝這個工具「rpm –i KERNPROF-08-2I386.RPM」，這個工具是日後啓動，設定 KernProf 的程式。
3. 另外要執行 KernProf 還得要自已建一個字元裝置，只要下「mknod /dev/profile c 192 0」的指令就可以了。

使用 KernProf

KernProf 一共有五種模式，在這裡，我們所用的是第二種模式 — cg(call graph)，這個模式可以讓我們了解函式之間的呼叫的關係。使用 KernProf，首先使用剛才編譯好的核心開機，然後在 User Mode 下用 kernprof 來控制核心啓動，關閉，設定 profing 的動作。以下提個簡單的例子：

Step 1. 在使用新的核心開機之後，在 profing 之前我們必需要讓 KernProf enable，並且選擇我們所要的模式爲何。下指令「kernprof –b –t cg 」表示我們要啓動 KernProf，並使用 call graph 這個模式。

Step 2. 試試下個指令，讓核心做點事吧，比如「ping –c 1 140.113.128.45」。

Step 3. 結束 KernProf 「kernprof –e 」

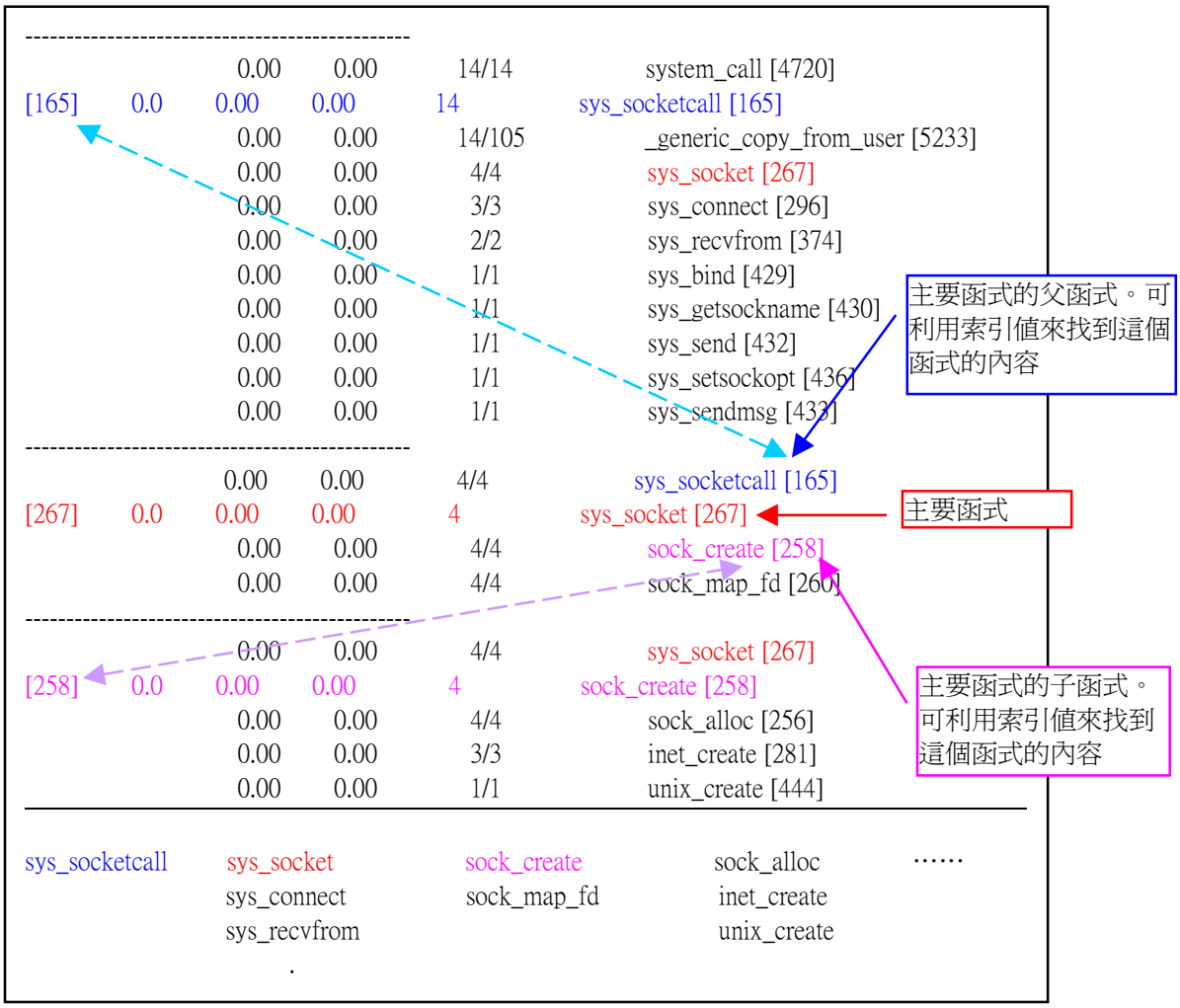
Step 4. 產生給 gProf 用的報告檔「kernprof –g 」輸出的檔案爲 gnom.out 也可以加參數「- o」來指定輸出的檔名，如：「kernprof –g –o output.txt」

Step 5. 使用 gProf 產生給人看的報告。在用 gProf 時必須要有核心的未壓縮檔，其位置在原始碼的目錄下，如/usr/src/linux/vmlinux。gProf 的第一個參數是執行檔，第二個參數是由 Step 4 做出來的檔案，爲了方便我們把 gProf 的結果重導向到一個檔案。

「gprof /usr/src/linux/vmlinux gnom.out > out.txt」現在整個結果就在 out.txt 這個檔案裡了，我們只要用文字編輯器來看就好了，先深呼吸一下，呼叫的函式多的要命。

怎麼讀 KernProf 的結果

如果您在上一節已先看過了檔案，也許您已經昏到了……，一個小小的 ping 居然叫了那麼多函式，加上複雜的函式呼叫圖，整個檔案印出來也要 60 多頁。不過，這些函式的罪魁禍首並不一定是 ping，裡面還有很多是由 cpu 排程及其他函式所造成的。現在簡單的講解一下這個檔案要如何閱讀。這個檔案分成三個部分，第一個部分是列出所有被呼叫函式的名字，以及被呼叫過幾次以及所花費的時間。第二個部分就是 call graph，在這個部分每一個區



圖五：KernProf 報告檔

間(每二個虛線的中間，我稱之爲一個區間)都有一個主的函式，在主要的函式的那一行前面會有一個數字，如[33]這是這個函式的索引值。在這行之上是這個函式的父親，也就是呼叫主要函式的函式，這個父函式其後也會有個索引值，我們可以照這個值去找到這個父函式是由誰呼叫的，相對的主要函式的下面就是由主要函式呼叫的兒子們，其後也有索引值。舉例如圖五。第三個部分比較單純，它條列出這份報告用的函式與相對應的索引值。

3.5 三種方法的比較

現在我們再來檢視一下這三種方法，其中 printk 的方法是最簡單的，不過也是功能最少的，而且，如果每次你想看看別的資訊，你必需要再重新編譯一次核心，這是很累人的一件事。KDB，和 KGDB 在功能上大致相同，KDB 所有的優勢是，它只需要單一的電腦就能執行；但雖然 Remote Debug 需要二台電腦，不過它卻可以看到核心的原始碼，這是 KDB 只能看到組合碼所不及的地方。所以如果真的想要了解核心程式碼的寫作，Remote Debug 的

確是一個很棒的方法。而且配合像 DDD 這樣的 GUI 工具，更是一大助力；目前在 Linux 下也有虛擬機器的軟體 — VMWare[8]，可以讓我們在 Linux 上開出一個 bare Machine，我們就可以在同一台電腦上做 Remote Debug 了，並不一定要二台電腦，如果你的主機夠強的話。表三是三種方法的比較。

項目 \ 方法		PrintK	KDB	KGDB
安裝難易度		易	不易	難
所需電腦數		1	1	2
除錯功能	列印變數	✓	✓	✓
	單步執行		✓	✓
	設定中斷點		✓	✓
	看 Machine code		✓	✓
	看 Source code			✓
	GUI 介面			✓

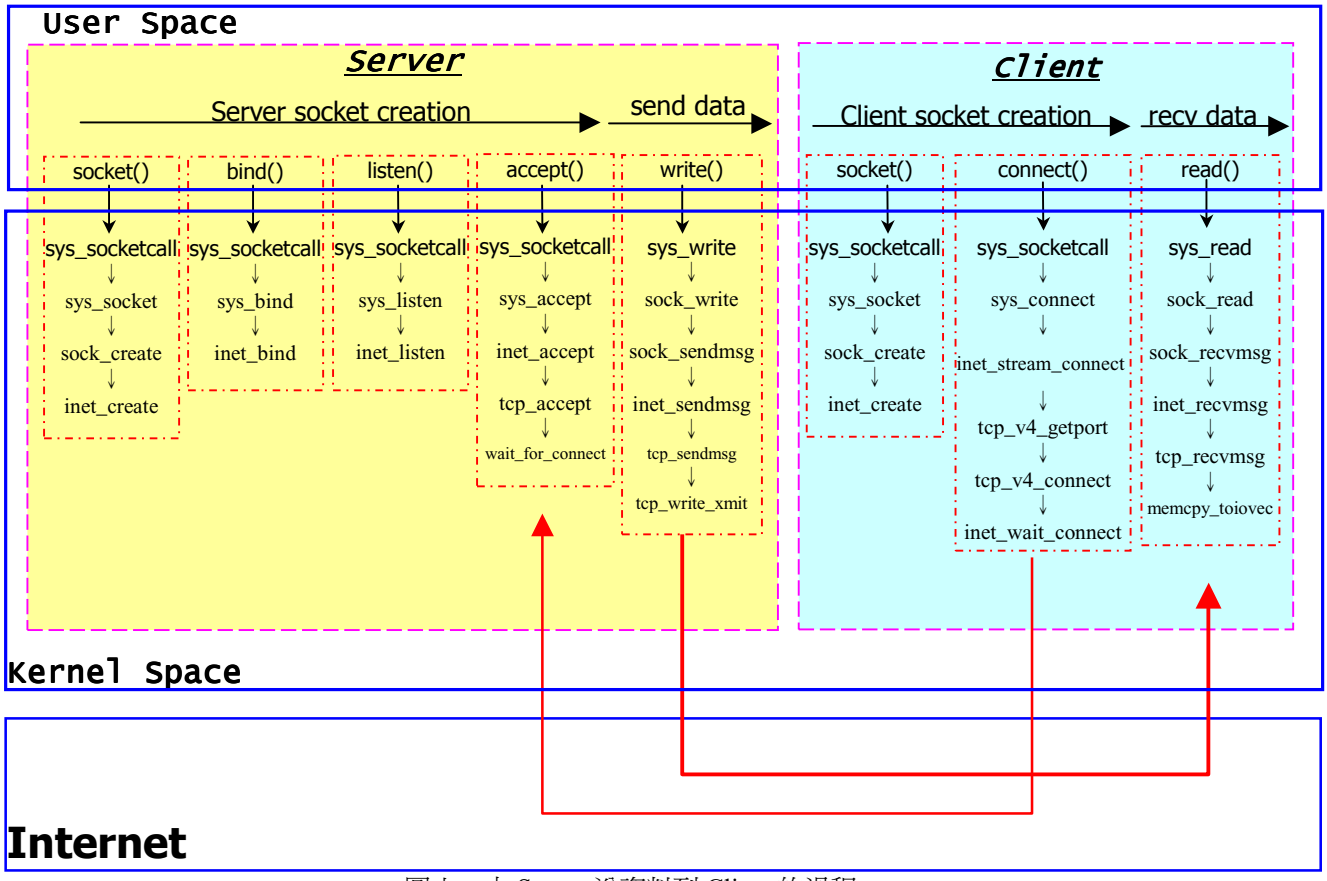
表三：三種核心追蹤方法之比較

四、使用 KGDB 追蹤 Linux 核心

介紹完了工具之後，我們回到文章的主題，追蹤協定驅動程式。一般的網路封包大概可以分爲 User Plane 和 Control Plane 這二個部分；User Plane 大概就是使用者應用程式所發出來的網路封包，如 WWW、 BBS 等，Control Plane 就像是 router 與 router 間更新路徑時所發出的封包，如 RIP、BGP、ICMP 等。在此，我們在每一個部分選出一個例子，使用 KGDB 來追蹤。在 User Plane 我們寫一個簡單的 clinet/server 的程式，由 Server 送出一個封包給 Client 接收。在 Control 的部分，我們以常用的 ping 爲主，試看看 Linux 如何回應一個 icmp 的要求。

4.1 Client/Server 的核心過程

爲了讓追蹤過程單純化，不受到其他的影響，我們寫了最簡單的 Client/Server 的程式。Server 端在 Create，…Accecp 之後就一直等待連線，如果有 Client 連線過來，Server 就會送出一個字元給 Client，並繼續等待下一個連線，而 Client 一啓動就會向 Server 要求連線，然後接收一個字元，並印在螢幕上後就結束。圖七爲 Server 建立 socket，到傳送資料給 Client 接收之過程。追蹤的方法是在將中斷點設在 sys_socketcall 上，然後用 step 這個指令來單步執行，其中會有很多細節，我們只選出比較重要的部分說明如下。



圖六：由 Server 送資料到 Client 的過程

核心過程 I：Server 端建立 socket

現在，我們先從 Server 端開始追蹤，首先我們先用 KernProf 來看看 Server 端執行過程中呼叫過什麼函式，KernProf 的使用方法請參考 3.4 的說明。可以分析一下 KernProf 的報告，我們可以發現，呼叫 sock_create()的函式是 sys_socketcall，我們可以在這個函式上下中斷點，來看看 Server socket 整個建立的過程。

1. 啓動 KGDB，在 Target 開完機後，可以在 Host 端下 “interrupt”讓遠端中斷，並使 Host 取得控制權。
2. 下 “break sys_socketcall”指令來設定中斷點在 sys_socketcall 這個函式上。
3. 下 “cont”釋放控制權，讓遠端繼續執行。
4. 在 Target 端執行 Server 程式，之後就可以看到控制權回到 Host 端，並中斷在 sys_socketcall
5. 現在我們可以用 “step” “next” 來一步步追蹤核心。

首先，第一個中斷點是由 Server.c 中的 “socket(AF_INET, SOCK_STREAM, 0)” 這個函式呼叫所引起的。我們可以看到 sys_socketcall 會先呼叫 “copy_from_user” 從 user space 複製一些參數過來，如圖七所示。之後我們可以看到一個 switch，它會跟據 call 的值來選擇

要呼叫的函式，在這呼叫的是 sys_socket，它會爲我們建一個 socket，記得使用“step”的指令來看看核心在 sys_socket 中做了什麼事，當然我們也可以在 sys_socket 下中斷點，這樣比較不會錯過它。

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
{
                        ……
    /* copy_from_user should be SMP safe. */
    if (copy_from_user(a, args, nargs[call]))
        return -EFAULT;
                        ……
    switch(call)
    {
        case SYS_SOCKET:
            err = sys_socket(a0,a1,a[2]);
            break;
        case SYS_BIND:
            err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
            break;
        case SYS_CONNECT:
            err = sys_connect(a0, (struct sockaddr *)a1, a[2]);
            break;
        case SYS_LISTEN:
            err = sys_listen(a0,a1);
            break;
        case SYS_ACCEPT:
            err = sys_accept(a0,(struct sockaddr *)a1, (int *)a[2]);
            break;
```

圖七：sys_socketcall 部分程式碼

sys_socket 會呼叫 sock_create，如圖八所示。我們可以 看到傳進去的參數 family=2 指的是 INET sock ，type=1 指的是 Stream 也就是 TCP 協定。

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
        ……
    retval = sock_create(family, type, protocol, &sock); //據 family 來建立不同的 socket 比如，
    if (retval < 0)                                      // INET,UNIX…等。
        goto out;
    retval = sock_map_fd(sock);
        ……
}
```

圖八：sys_socket 部份程式碼

sock_create 會先檢查 family 是不是正確，如圖九所示。之後還有一些相容性的檢查。然後會呼叫 sock_alloc 來配置 socket。

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
                                ……
    if(family<0 || family>=NPROTO)   //檢查是不是正確的 family
        return -EINVAL;
                                ……
    if (!(sock = sock_alloc()))          //建一個 socket
    {
        printk(KERN_WARNING "socket: no more sockets\n");
                                ……
    if ((i = net_families[family]->create(sock, protocol)) < 0)     //在這 family 是 INET 所以呼叫
    {                                                                // inet_create 來建立 sock
        sock_release(sock);
        goto out;
                                ……
```

圖九： sock_create 部份程式碼

sock_alloc 會先找一個空的 inode 來建 socket，如圖十所示。然後再爲 socket 填入一些初始值，然後把 socket 的指標回傳。這個指標由 sock_create 的 sock 這個變數值接受。

```
struct socket *sock_alloc(void)
{   inode = get_empty_inode();                  //找一個空的 inode 來建 sock
    if (!inode)                                 //如果找不到就發生錯誤……
        return NULL;
    sock = socki_lookup(inode);
                ……
    inode->i_mode = S_IFSOCK|S_IRWXUGO;
    inode->i_sock = 1;
                ……
    sock->flags = 0;
    sock->ops = NULL;
                ……
    sockets_in_use++;                           //把這個 socket 的使用者加 1
    return sock;                    }           //傳回這個 sock 的指標
```

圖十： sock_alloc 部份程式碼

sock_alloc 結束後，程式的流程回到了 sock_create，如圖九所示。接下來 if ((i = net_families[family]->create(sock, protocol)) < 0)會去呼叫 inet_create 來建立 sock，inet_create 呼叫 sk_alloc 來向 kernel memory 要求記憶體空間，如圖十一所示。其呼叫函式 kmem_cache_alloc 來完成工作。之後的 switch 會根據 socket 的 type 來執行，在這我們的程式是 Stream，所以跑的是 case SOCK_STREAM。值得注意的是最後一行，核心把 &inet_stream_ops 這個指標給 sock_ops，這個指標指向許多操作 inet sock 的函式，如：accept，listen…等。之後 inet_create 會對 sock 做一些初始設定。

```
static int inet_create(struct socket *sock, int protocol)
{
                ……
    sk = sk_alloc(PF_INET, GFP_KERNEL, 1);
                ……
    switch (sock->type) {
    case SOCK_STREAM:
        if (protocol && protocol != IPPROTO_TCP)
            goto free_and_noproto;
        protocol = IPPROTO_TCP;
        if (ipv4_config.no_pmtu_disc)
            sk->protinfo.af_inet.pmtudisc = IP_PMTUDISC_DON
        else
            sk->protinfo.af_inet.pmtudisc = IP_PMTUDISC_WAN
        prot = &tcp_prot;
        sock->ops = &inet_stream_ops;

                ……
```

inet_stream_op

family=2
release=inet_release
bind=inet_bind
connect=inet_stream_connect
socket_pair=socket_no_pair
accept=inet_accept
getname=inet_getname
poll=inet_poll
ioctl=inet_ioctl
listen=inet_listen
shutdown=inet_shutdown
setsocketop=inet_setsocketop
fcntl=sock_no_fcntl
sendmsg=inet_sendmsg
rcvmsg=inet_rcvmsg
mmap=sock_no_mmap

圖十一： inet_create 部份程式碼

核心過程Ⅱ：Server 端 socket bind

之後會遇很多次的 return，然後就會結束 sys_socketcall 這個函式呼叫，進入排程，不久又會中斷在 sys_socketcall 這是因爲 Server.c 呼叫 bind()這個函式。參考圖七，sys_socketcall 會呼叫 sys_bind，並把參數傳入。sys_bind 先用 sockfd_lookup，由 file descript 找到指向 socket 的指標，如圖十二所示。然後呼叫 move_addr_to_kernel，將目地的 inet address 考貝到核心，其中呼叫的函式爲 copy_from_user()。接著呼叫的是 sock->ops->bind()，由圖十一得知，sock->ops 的指標所指的結構中，bind 的這一項是指向函式 inet_bind，所以啦，下一個執行的函式就是 inet_bind。

```
asmlinkage long sys_bind(int fd, struct sockaddr *umyaddr, int addrlen)
{
                ……
    if((sock = sockfd_lookup(fd,&err))!=NULL)
    {
        if((err=move_addr_to_kernel(umyaddr,addrlen,address))>=0)
            err = sock->ops->bind(sock, (struct sockaddr *)address, addrlen);
        sockfd_put(sock);
                ……
```

圖十二：sys_bind 部份程式碼

inet_bind 會先檢查使用者是不是有自已寫 bind 的程式，如圖十三所示。如果有的話就會用使用者自已寫的 bind 程式。然後會檢查位址長度是否正確，位址是不是符合 inet 位址格式……在一連串的檢查後，會呼叫 get port 的函式，在這裡所連接到的是 tcp_v4_get_port 這個

函式，它的工作取得一個 port number 並且和 sock 連接在一起，如果我們給的 port number 是 0 的話，這個函式會自行選一個空的 number 連接。

```
static int inet_bind(struct socket *sock, struct sockaddr *uaddr, int addr_len)
{
                ……
    if(sk->prot->bind)
        return sk->prot->bind(sk, uaddr, addr_len);
    if (addr_len < sizeof(struct sockaddr_in))
        return -EINVAL;
                ……
    chk_addr_ret = inet_addr_type(addr->sin_addr.s_addr);
                ……
    if (sk->prot->get_port(sk, snum) != 0) {
        sk->saddr = sk->rcv_saddr = 0;
                ……
```

圖十三：inet_bind 部份程式碼

核心過程Ⅲ：Server 端 socket listen

接著整個程序又結束了，進入排程，下一個中斷點出現，sys_socketcall 呼叫 sys_listen，參考圖七。sys_listen 跟據 sock->ops->listen，呼叫 inet_listen，可參考圖十一的 inet_stream_op。inet_listen 會將目前的 state 移到 TCP_LISTEN 這個 state，如圖十四所示。

```
int inet_listen(struct socket *sock, int backlog)
{
                ……
    if (old_state != TCP_LISTEN) {
        sk->state = TCP_LISTEN;
        sk->ack_backlog = 0;
                ……
```

圖十四：inet_listen 部份程式碼

核心過程Ⅳ：Server 端等待連線(socket accept)

接下來 bind 程序也結束了，現在 Server將會進入等待連線的狀態。系統排程後進入中斷點，這次是由 Server.c 中的 accept 引起的，sys_socketcall 呼叫 sys_accept，這個函式主要的工作是建立連線，它會先自已建一個新的 socket，如圖十五所示。然後等待建立連線，一但成功了，經過一些處理後它會回傳一個 file descript，這個 file descript 是指向新建的 socket，也因爲新建了 socket 所以 Server 才能繼執行 listen 的工作。由圖十一我們可以後知程式碼 sock->ops->accept 所呼叫的函式是 inet_accept。

```
asmlinkage long sys_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
{
                ……
        if (!(newsock = sock_alloc()))                          //建立一個新的 socket
                goto out_put;
                ……
        err = sock->ops->accept(sock, newsock, sock->file->f_flags);      //呼叫 inet_accept
                ……
                if(newsock->ops->getname(newsock, (struct sockaddr *)address, &len, 1)<0) {
                ……
                err = move_addr_to_user(address, len, upeer_sockaddr, upeer_addrlen);
                ……
        if ((err = sock_map_fd(newsock)) < 0)
                ……
        return err;
```

圖十五：inet_listen 部份程式碼

inet_accept 中的 sk1->prot->accept 會呼叫 tcp_accept，由它來接受 connection。

```
int inet_accept(struct socket *sock, struct socket *newsock, int flags)
{
        struct sock *sk1 = sock->sk;
        struct sock *sk2;

        if((sk2 = sk1->prot->accept(sk1,flags,&err)) == NULL)
                ……
```

圖十六：inet_accept 部份程式碼

tcp_accept 會先呼叫 tcp_find_established 去找看看現在有那一個 connection 可以 accept 的，如圖十七所示。如果現在沒有的話，它還會呼叫 wait_for_connect 來等待 connect。

```
struct sock *tcp_accept(struct sock *sk, int flags, int *err)
{
                ……
        req = tcp_find_established(tp, &prev);
        if (!req) {
                error = -EAGAIN;
                if (flags & O_NONBLOCK)
                        goto out;
                error = -ERESTARTSYS;
                req = wait_for_connect(sk, &prev);
```

圖十七：tcp_accept 部份程式碼

整個 Server 的程式到這理，就會一直等待遠方的 connect。只要遠方一有 connect 的話，系統就會被 wake up 繼續執行 wait_for_connect 後面的程式碼，如圖十八所示。這時如果在另一台主機執行 client 端程式的話，我們就可以看到中斷點停在 wait_for_connect，並執行下面的 tcp_find_established 來建立連線。

```
static struct open_request * wait_for_connect(struct sock * sk,
                                        struct open_request **pprev)
{
        DECLARE_WAITQUEUE(wait, current);
        struct open_request *req;

        add_wait_queue_exclusive(sk->sleep, &wait);
        for (;;) {
                current->state = TASK_EXCLUSIVE | TASK_INTERRUPTIBLE;
                release_sock(sk);
                schedule();
                lock_sock(sk);
                req = tcp_find_established(&(sk->tp_pinfo.af_tcp), pprev);
                        ……
```

圖十八：wait_for_connect 部份程式碼

核心過程Ⅴ：Server 傳送資料到 Client

當連線建好了，二端就可以開始傳送資料了，在傳送資料時系統會呼叫 sys_write，這個函式會先做一些檢查，比如，是不是有足夠的權限來做寫入的動作，其 file descript 是不是含有運算向量且有寫入的函式等，如圖十九所示。然後執行 write，在這裡，write 函式會指向 sock_write。

```
asmlinkage ssize_t sys_write(unsigned int fd, const char * buf, size_t count)
{
                        ……
        file = fget(fd);
        if (file) {
                if (file->f_mode & FMODE_WRITE) {
                        struct inode *inode = file->f_dentry->d_inode;
                        ret = locks_verify_area(FLOCK_VERIFY_WRITE, inode, file,
                                file->f_pos, count);
                        if (!ret) {
                                ssize_t (*write)(struct file *, const char *, size_t, loff_t *);
                                ret = -EINVAL;
                                if (file->f_op && (write = file->f_op->write) != NULL)
                                        ret = write(file, buf, count, &file->f_pos);
                        ……
```

圖十九：sys_write 部份程式碼

sock_write 只做一些轉換的動作，首先制利用 socki_lookup 把 inode 所屬的 socket 找出來，然後再把一些寫入時所需要的資料編成一個 mssage，再來就呼叫 sock_sendmsg 來傳送 mssage，如圖二十。sock_sendmsg 根據 sock->op->sendmsg 呼叫 inet_sendmsg。inet_sendmsg 根據 sk->prot->sendmsg 呼叫 tcp_v4_sendmsg。在經過一些檢查後，它會呼叫 tcp_do_sendmsg。

```
static ssize_t sock_write(struct file *file, const char *ubuf,
                          size_t size, loff_t *ppos)
{
                        ……
        sock = socki_lookup(file->f_dentry->d_inode);
```

```
        msg.msg_name=NULL;
        msg.msg_namelen=0;
        msg.msg_iov=&iov;
        msg.msg_iovlen=1;
                        ……
        return sock_sendmsg(sock, &msg, size);
}
```

圖二十：sock_write 部份程式碼

到目前爲止，只是一些設定值，參數在各層之間傳遞而已，真正的資料傳遞在 tcp_do_sendmsg 開始。這個函式會從使用者空間烤貝資料到 socket 中，並開始傳送。tcp_do_sendmsg 會先呼叫 sock_wmalloc 來向記憶體要求空間建立 sock buffer，接著爲 sock buffer 作初始設定完後系統會呼叫 tcp_send_skb 來傳送 sock buffer，tcp_send_skb 呼叫 tcp_write_xmit 和 tcp_transmit_skb 來完成工作。

核心過程 VI：Client 接收到封包

接下來我們不再詳細說明核心程式碼與過程。當 Clinet 端收到封包時，乙太網路卡會發出一個中斷，依照網路卡驅動程式的不同會有不一樣的函式負責，可以在/linux/driver/net 下找到您網路卡對應的驅動程式，筆者使用的是 AMD PCnet32 晶片的網卡，所以處理這個中斷的函式是 pcnet32_interrupt，我們也可以在 KernProf 產生出來的報告中找到一點蛛絲馬跡，我們知道會發生中斷事件，我們可以察看 handle_IRQ_event 下所呼叫的函式，就可以知道它呼叫什麼函式來處理中斷了。然後呼叫 pcnet32_rx 這個函式來處理接收的動作。這個函式會建立新的 sock buffer 並爲它作初始化的設定，之後再呼叫 netif_rx()這個函式，把這個新的 sock buffer 加入 backlog 串列中，這個串列記錄了系統所收到的所有封包。再經過一些處理，我們已經收到一個 ip 封包，接著系統呼叫 ip_rcv 來檢查標頭的正確性，及其他的檢查，之後呼叫 ip_local_deliver，把 ip 封包往上一層送。再來就是由 tcp_v4_rcv 接手處理，它會再呼叫 tcp_v4_do_rcv 來完成工作，並呼叫 tcp_rcv_established，tcp_rcv_state_process 作適當的回應給 source 端。到目前爲止都是核心收到一個封包所做的事，之後 Client 端讀取資料還會觸動 sys_read、sock_read、inet_rcvmsg、tcp_rcvmsg 等函式以完成讀取資料的動作。接下來就由您來追蹤吧。

4.2 追蹤 Ping 的流程

事實上核心收到 ping 的封包時的反應和收到一般封包時大致上相同，不同點是從 ip_local_deliver 開始的，當它發現這個封包的 protocol 是 icmp 時，就會呼叫 icmp_rcv 來處理

這個函式，在經過檢查後，他會依照 icmp->type 去查 icmp_pointers 這個資料結構中的 handler，並交給這個 handler 處理，在這呼叫的 handler 是 icmp_echo。icmp_echo 在設定完一些參數之後，交由 icmp_reply 處理回應的工作。icmp_reply 在建完 icmp 的標頭後，呼叫 ip_build_xmit 來建立 ip 標頭並傳送封包。整個核心從接收，到回覆一個 icmp 的封包大致是這個樣子，細部的過程，讀者可以一步步的用“step”指令來追蹤。

五、結論

最後，我們給想要閱讀 Linux 核心原始碼者幾個建議：

I. 關於整個追蹤的環境，我們建議使用遠端除錯的方法來了解 Linux 核心。建議在 Host 端使用有 GUI 介面的 DDD，並在同一台電腦執行 VMWare 模擬一台 Linux 電腦來當 Target。這樣就不用二台電腦，也不用在操作過程中換鍵盤換滑鼠的。但當 Target 是一台資源受限(如無太多的記憶體執行除錯程式，無顯示器等)的嵌入式系統時，Target 與 Host 就必定是二台不同的機器。

II. 關於追蹤的方法，我們建議先作出 KernProf 的報告，先對函式的呼叫流程有一點認識，也方便對特定功能的函式下中斷點來追蹤。

III. 核心的行爲是事件驅動的，即有 user space 程式呼叫 system call 或有硬體中斷時，所以我們可以對想了解的部分自已創造事件。這樣有二個好處，一是比較單純，不會有其他事件影響；一是我們可以了解到 API(即 system call)執行時，核心的動作爲何。

六、參考資料

1. M Beck, H Böhme, M Dziadzka, U Kunitz, R Magnus, D Verworner, "Linux Kernel Internals ", Second Edition, Addison Wesley Longman, 1997.
2. David Rusling, "The Linux Kernel", http://metalab.unc.edu/LDP/LDP/tlk/tlk.html
3. 林盈達, "計算機網路實驗", 維科出版社, 1999。
4. KDB homepage http://oss.sgi.com/projects/kdb/
5. KGDB homepage http://oss.sgi.com/projects/kgdb/
6. KernProf homepage http://oss.sgi.com/projects/kernprof/
7. gcc homepage http://www.gnu.org/software/gcc/gcc.html
8. VMWare http://www.vmware.com