國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

重新使用核心驅動程式為使用者模式驅動程式以改善作業系統可信賴度

Improving Operating System Dependability by Reusing Kernel Drivers in User Mode

研 究 生：黃致遠

指導教授：張瑞川　教授

中 華 民 國　九 十 五　年 六 月

重新使用核心驅動程式爲使用者模式驅動程式以改善作業系統可信賴度

學生：黃致遠　　指導教授：張瑞川教授

國立交通大學資訊科學與工程研究所

論 文 摘 要

隨著近年來對於系統可靠度的重視，系統可用度也愈來愈重要，比起硬體，軟體出錯是影響系統可用度較大因素，而軟體通常又依賴底層的作業系統，所以改善作業系統的可信賴度是重要的一環。

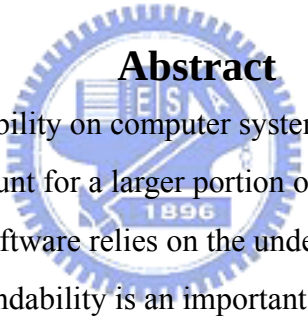
數據顯示出作業系統中最容易出錯的地方是驅動程式，驅動程式出錯通常會影響作業系統可信賴度。因此，我們提出一個使用者模式驅動程式架構，希望利用使用者模式驅動程式去改善作業系統可信賴度。我們主要特色有兩個：(1)我們能夠重用原本核心模式驅動程式，可以直接執行在使用者模式；(2)我們能夠達到無資料遺失的驅動程式，當驅動程式出錯時，我們能夠掩飾其出錯，讓應用程式能夠繼續執行。由實驗結果顯示我們架構中的使用者模式驅動程式負擔並沒有太大，並且在回復過程中具有可接受的效能表現。

Improving Operating System Dependability by Reusing Kernel Drivers in User Mode

Student: Chih-Yuan Huang **Advisor: Prof. Ruei-Chuan Chang**

Computer Science and Engineering College of Computer Science

National Chiao Tung University



Abstract

With the current high reliability on computer system, system availability is increasingly important. Software faults account for a larger portion of system unavailability than hardware failures. Because most of the software relies on the underlying operating systems, how to improve operating system dependability is an important part.

According to previous research, device drivers are the most faulty part of an operating system. Device driver faults usually influence operating system dependability. For the reason, we propose a user mode device driver framework. We hope to improve operating system dependability with user mode device driver. Our characteristics includes:(1)By reusing native kernel mode device driver, device drivers can execute in user mode directly;(2) We can achieve zero-loss device driver. While device driver fault, we can mask device driver faults and make applications continue to execute. According to the performance evaluation, the overhead of the user mode device driver is not high and acceptable performance is achieved during the recovery.

致謝

首先感謝我最尊敬的指導老師 張瑞川教授。這兩年在老師費心的教導下，學生方能順利完成此篇論文。於受業期間，老師耐心的指導我正確的研究態度與研究方法，讓我受益良多。在撰寫論文期間，感謝大緯學長給予的建議。也感謝在實驗室一起努力的同學們，博士班學長以及學弟們的支持和勉勵，讓我不會承受不了撰寫論文的壓力。

最後感謝我的家人，爸爸、媽媽、哥哥、姐姐，在我念研究所的日子裡不斷的關心和鼓勵我，尤其是在碩二撰寫論文時。對於他們的關心，讓我覺得很窩心、感動。最後僅以此論文獻給我最親的家人和朋友，由衷地謝謝他們。

目錄

圖目錄

表目錄

第一章　簡介

1.1　動機

隨著目前系統對於可靠度(Reliability)的重視，系統可用度(Availability)也逐漸愈來愈重要，能讓系統持續保持可用的不再是選擇性的而是義務性的。根據過去的研究顯示，跟硬體出錯比較起來，軟體出錯是造成系統不可用的主要原因[14][24]。因此對於系統可用度，軟體扮演了一個重要的角色。

因爲大部分軟體必須要依賴底層的作業系統，對於一個需要高可用度的系統，改善作業系統的可信賴度是一件重要的事。若作業系統缺少可信賴度，對於許多領域都會造成負面的影響。舉例而言，對於正在蓬勃發展的網路服務，像是一個線上交易的企業網站，如果因爲作業系統執行錯誤發生 crash 而導致服務中斷的話，會對該企業造成許多經濟上的損失[9]。並且對於一般的使用者在使用應用軟體時，也會因爲作業系統出錯造成不悅。所以如何改善作業系統因爲錯誤而 crash，進而去提升作業系統可信賴度是一件相當急迫需要的事。

根據美國 Stanford 大學[4]以及 coverity 公司[5]對於 Linux 核心程式碼的分析顯示，核心程式中最容易發生錯誤的部份是驅動程式。這是因爲驅動程式通常是由硬體供應商所開發，他們對於核心程式的開發比較不熟悉，所以比較容易出錯。由於驅動程式是執行在核心空間，所以驅動程式出錯通常都會影響作業系統運作，嚴重的話甚至造成作業系統 crash。另一由微軟所做的分析[27]也顯示，Windows XP 作業系統 crash 的原因大概有 85%是由驅動程式所造成。因此，能夠避免因爲驅動程式所導致的錯誤便能改善作業系統的可靠度。

過去針對改善驅動程式錯誤的相關研究主要可以分爲兩類，從驅動程式的撰寫做改善以及限制驅動程式執行時的存取權限，如下：(a)一些研究提出新的 type-safe 的描述性程式語言[7][21]去撰寫驅動程式，主要是要減少撰寫驅動程式時容易犯的錯誤；(b)另一方面，限制驅動程式執行時的存取權限可以減少驅動程式出錯時危害到核心程式。首先，利用 Protection Domain 的概念[28][30]，其將驅動程式執行在核心程式以外的 Protection Domains 中，這樣可以限制驅動程式的存取權限，且當驅動程式出錯時不會危害到核心程式與其他 Protection Domains。雖然這樣可以減小驅動程式危害到核心程式，但是卻不能完全的達到錯誤隔離(Fault Isolation)。利用虛擬機器(Virtual Machine)技術把驅動程式跟核心程式切開[12][18]，驅動程式與核心程式分別執行在不同的虛擬機器

中，以避免驅動程式出錯危害到核心程式。另外，一個更安全的作法是將驅動程式跟一般程式一樣，執行在使用者空間，即所謂的『使用者模式驅動程式(User Mode Device Driver)』。早期研究微核心(microkernel)系統的學者就已提出[1][11][19][20][25]。然而在當時，普遍的問題就是因爲需要額外的 IPC 而導致效能不好。但是最近由於作業系統在系統呼叫(system call)與行程切換(context switch)上效能的改進，使用者模式驅動程式漸漸變成一個可行的方法。近幾年在 Linux 平台上有學者提出使用者模式驅動程式架構[6][17][32]。然而，他們的驅動程式需要額外的成本去開發，可能從原本的核心模式驅動程式版本修改，或是完全的重新開發驅動程式。微軟在 Windows 平台上也有提出使用者模式驅動程式架構[2][15][22]，不過目前對於裝置的支援還沒有很完整。另外，雖然使用者模式驅動程式可以改善作業系統可靠度，但是當驅動程式當掉之後，將會影響到其他一般應用程式的執行，造成執行行爲有誤或者是當掉，所以如何去掩飾驅動程式的錯誤，讓一般應用程式能繼續執行，使得作業系統達到可用度也是一件需要考慮的事情。

觀察以上的問題，我們提出一個新的使用者模式驅動程式架構，我們主要是要達到兩個目標：(a)希望能重用(reuse)核心模式驅動程式，因爲核心模式驅動程式已經相當完整，讓核心模式驅動程式能夠不用修改就可以直接執行在使用者空間，這樣可以讓我們減少重新開發使用者模式驅動程式所要額外付出的成本；(b)我們希望這個架構能提供可信賴度。也就是說當驅動程式當掉時，能夠掩飾驅動程式的錯誤，並且自動地去重新啓動驅動程式，讓應用程式能夠繼續執行，而並不會因爲驅動程式錯誤而受到影響，達到作業系統可信賴度。

我們將此架構實作在 Linux 作業系統上。目前已經完成重用 Tulip Ethernet 網路卡驅動程式，一個 RAMDISK 區塊驅動程式，以及一個 PS2 Mouse 字元驅動程式，讓它們可以直接執行在使用者模式，我們的結果顯示：(1)我們的架構能夠與原本驅動程式模型(model)相容，也就是說我們不需要去修改原本的驅動程式模型；(2)沒有造成太大的效能負擔；(3)對於改善作業系統可信賴度，我們能夠達到無資料遺失(Zero-Loss)驅動程式，核心給驅動程式的請求不會遺失，應用程式能夠繼續執行而不會受到驅動程式出錯影響，並且對於驅動程式與應用程式具有透明性，驅動程式與應用程式不需要額外修改。

1.2 論文組織

這篇論文的結構如下：第二章介紹與我們研究有關的相關研究。第三章介紹我們使用者模式驅動程式架構的設計與實作。第四章我們評估該架構的效能。第五章提出結論。

第二章　相關研究

改善驅動程式錯誤的相關研究大致可分成兩類：一類是從驅動程式的撰寫方面做改善，另一分類則是去限制驅動程式的存取權限。在本章中，我們將概述這些相關研究。

2.1 Programming Model

此類研究從驅動程式的撰寫方面著手，傳統驅動程式都是用 C 語言寫的，沒有『type safety』，並且驅動程式有許多程式部分都是操作硬體的程式部分，這種較低階的程式部分被證實是比較容易出錯的，並且需要冗長過程的除錯。於是，有一些研究嘗試開發另一種語言，讓驅動程式開發者可以用比較『安全』的語言去開發驅動程式，降低驅動程式出錯的機率。法國 University of Rennes I 研發出一種 IDL(Interface Definition Language)，稱爲 Devil(A DEVice Interface Language)語言[21]，它是一種較高階的描述性語言，允許去檢查驅動程式的安全性，並且可以讓較低階的操作硬體程式碼能夠被自動化的產生，以減少寫驅動程式時所會犯的錯誤。另外，美國 Columbia 大學所研發的 NDL 語言[7]，它提供裝置資源的抽象化，並且讓驅動程式的操作可以用描述的方式去開發，以減少撰寫程式出錯的機會。

2.2 錯誤隔離(Fault Isolation)

此類研究不是從驅動程式的撰寫做改善，而是去限制驅動程式的存取權限，以達到保護系統的目的，在這一分類中主要可以分爲三個方向:

2.2.1 Protection Domain

早期Palladium[3]利用虛擬記憶體硬體技術去隔離核心擴充套件(extensions)在一個中間特權層級(intermediate privileged level)-Protection Domain，核心擴充套件只有部分權限能夠去存取核心資料，以防止不安全的擴充套件會去危害到核心程式。

美國 Washington 大學所提出的 Nooks[28][30]把驅動程式執行在特權模式(privileged mode)。他們在核心的位址空間切出多個 Protection Domain，讓每一個不同的驅動程式能在自己的 Protection Domain 中執行。所以當任何一個驅動程式出錯時，並不會危害到其他的 Protection Domain 以及核心程式。這樣的好處是幾乎是不用修改原本的驅動程式。另外，爲了避免當驅動程式當掉時會影響到一般到應用程式的執行，nooks 利用 shadow driver[29]去暫時的取代當掉的驅動程式，以防止因爲驅動程式當掉而影響到一

般應用程式的運作。Nooks 的缺點是不能達到完全的錯誤隔離(Fault Isolation)。因爲驅動程式依然是執行在 ring 0，依然可以使用特權(privileged)指令，並且像記憶體漏失(memory leakage)這種會消耗系統資源的錯誤，在核心中危害也比較嚴重。

2.2.2 虛擬機器(Virtual Machine)

德國卡斯魯大學(University of Karlsruhe)[18]與劍橋大學(University of Cambridge Computer Laboratory)所發展的 Xen [12]利用虛擬機器(Virtual Machine)技術把驅動程式跟核心程式切開，驅動程式與核心程式分別執行在不同的虛擬機器中。若核心程式或應用程式需要使用驅動程式時，核心程式就會送請求給另一台虛擬機器，請它的驅動程式幫忙。這主要可達到兩個目的：(1)驅動程式可以重用(reuse)，不必單獨爲每一個作業系統都寫一套驅動程式；(2)驅動程式錯誤隔離(Driver Fault Isolation)，當驅動程式發生錯誤時，因爲是執行在與核心程式不同的虛擬機器中，所以並不會危害到執行核心程式的虛擬機器，可以達到容錯(Fault-Tolerant)的效果。這種方法主要優點是驅動程式不用修改，缺點是虛擬機器的負擔(overhead)比較大，因爲每一個驅動程式就需要一台虛擬機器，效能上也會比較差。

2.2.3 使用者模式驅動程式(User Mode Device Driver)

這類方法早在 1980 年代研究微核心系統的學者就已提出。在這種架構中，驅動程式跟一般使用者程式一樣，都執行在使用者空間，所以不會危害到核心程式，所以也不會造成系統當機。比較著名的有 Liedtke's L3 系統[19]，Mach 3[11]，以及猶他大學利用 OSkit[10]所發展出來的 Fluke kernel[20]，還有許多微核心系統都是利用使用者模式驅動程式[1] [25]。

近幾年有人開始研究把使用者模式驅動程式應用到一般的作業系統中，例如: Linux 與 Windows。印度技術學院(Indian Institute of Technology, Kanpur)[32]提出一個建構在 Linux 作業系統上的一個新的使用者模式驅動程式架構，他們提出類似於原本核心中的 API，希望開發者可以照著他們提出的架構與驅動程式 API 去開發驅動程式，他們並且也改寫了核心模式版本的驅動程式以應用到使用者模式上。另外，澳洲電信設備與系統相關領域協會(National ICT Australia)與新南威爾士大學(University of New South Wales) [6][17]提出了另一個 Linux 作業系統上的使用者模式驅動程式架構。與印度技術學院不同的是，他們強調使用者模式驅動程式的效能已經可以近似於核心模式驅動程式的效能。然而，前面兩者都有共同不足的地方，那就是使用者模式驅動程式都需要額外的人力與時間去撰寫，增加額外重新開發驅動程式的成本。

除了 Linux 作業系統外，Hunt[15]在 Windows NT 上實作使用者模式驅動程式，主要動機是希望驅動程式的開發能夠在使用者模式，這樣可以讓驅動程式的開發比較容易，不足的是它沒有支援需要存取硬體的驅動程式類型。

最近幾年，微軟爲了改善 Windows 作業系統的可靠度，推出 User-Mode Driver Framework(UMDF)[2][22]，希望有一些慢速裝置可以用使用者模式驅動程式。然而，微軟的 UMDF 有一些限制，包括沒有支援中斷處理(interrupt)，DMA 以及不能使用 non-pageable 的記憶體等等，所以支援裝置的類型相當有限。

第三章　設計與實作

這一章將要介紹我們所提出的使用者模式驅動程式架構的設計與實作。3.1 介紹我們如何將核心模式驅動程式搬移到使用者模式，3.2 介紹我們如何達到無資料遺失(Zero-Loss)的驅動程式，3.3 將介紹我們目前已完成的驅動程式架構雛形(prototype)。

3.1　核心模式驅動程式搬移(migration)

過去有關搬移驅動程式的研究通常是將一個作業系統中的驅動程式移植到另外一個作業系統上[13][20]，屬於橫向的搬移。橫向的搬移主要是要解決驅動程式在不同作業系統中的執行差異，例如：不同核心介面的轉換，核心空間記憶體分配，以及同步問題…等。而我們目標是要將核心模式驅動程式搬移到使用者空間，屬於縱向的搬移。縱向的搬移主要是要解決驅動程式在核心模式與使用者模式中的執行差異，例如：驅動程式在使用者模式中如何去存取物理記憶體，在使用者模式中模擬驅動程式的執行環境與核心介面，以及驅動程式優先權變小問題…等。此節我們將探討此架構的設計與實作。

3.1.1　設計

在此節中，我們描述驅動程式搬移相關的議題，首先，我們討論使用者模式驅動程式要如何跟核心子系統溝通。接著，我們討論如何將使用者模式驅動程式模擬爲一個多執行緒的程序。最後，我們說明要重用核心模式驅動程式，如何讓驅動程式能夠存取核心物件。

3.1.1.1　核心程式與驅動程式的溝通

核心程式與驅動程式之間的溝通(Communication)通常都有一個標準的介面，爲了將驅動程式搬移到使用者空間，我們必須將此介面在使用者空間重現。如 Figure 1，此介面主要分爲兩個方向：(a)一是核心程式呼叫驅動程式的介面，即驅動程式必須實作這些介面供核心程式使用。大部分的作業系統對不同類別的驅動程式都訂定了一套標準的介面以讓驅動程式的開發者可以很容易知道他必須實作哪些相關函式。對於此類介面，我們必須把核心程式的呼叫導送到使用者模式驅動程式；(b)第二類的介面是驅動程式呼叫核心程式服務的介面。核心程式爲了管理硬體資源，提供了一套專門的 API 供驅動程式使用。對於此類介面，我們必須要模擬出來以供驅動程式使用。

如 Figure 2 所示，實線表示核心程式呼叫驅動程式的方向, 虛線表示驅動程式呼叫的方向。對於核心程式向驅動程式請求的 Request 我們必須要轉送到使用者空間，我們可以使用 IPC(Inter-Process Communication)方法中的 Shared Memory 去達成。而驅動程式若需要一些核心程式的服務則可以利用系統呼叫(System Call)去向核心程式請求。

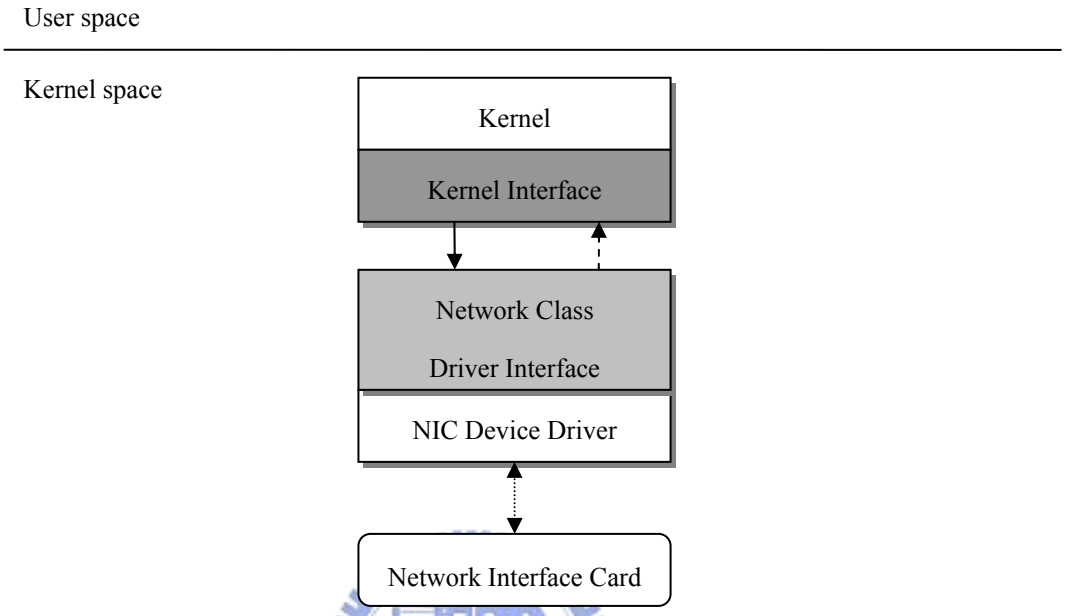


Figure 1：A Sample Kernel Mode Device Driver

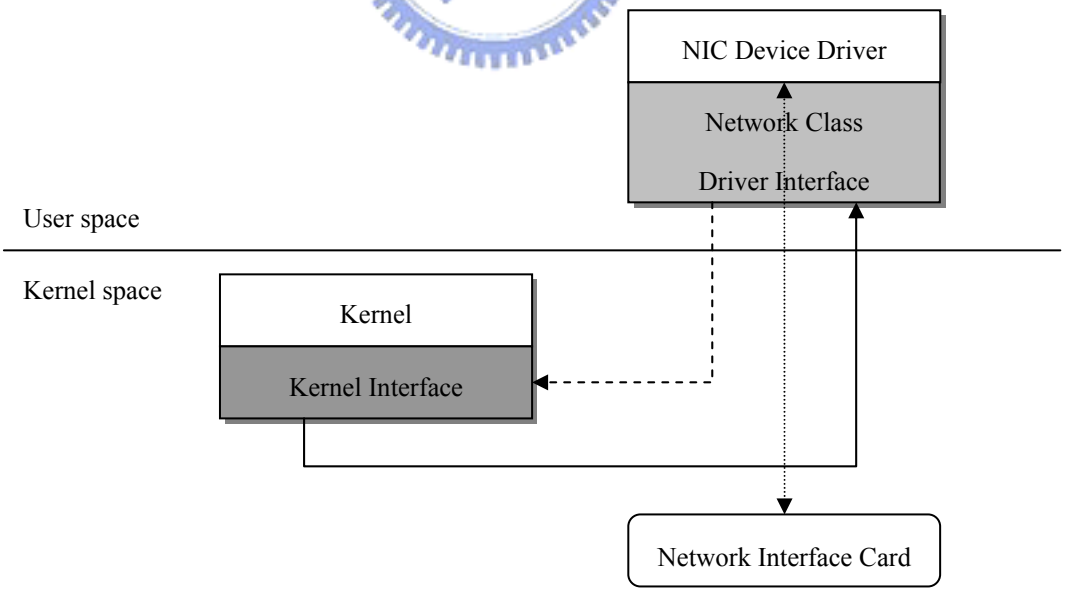


Figure 2：A Sample User Mode Device Driver

3.1.1.2 多執行緒使用者模式驅動程式

傳統的核心模式驅動程式通常都是單執行緒的，這是因爲大部分的作業系統，不管是 Linux，BSD，或 Windows，驅動程式實際上就像函式庫(Library)一樣，提供函式給核心呼叫以存取周邊設備的，所以大部分的驅動程式都是被動的。使用者模式驅動程式跟原本在核心模式一樣也是被動的，平常保持在睡眠(sleep)的狀態，等待有來自核心的請求時，才會起來執行工作以提供服務。

原本驅動程式中的函式可能會有不同的優先權，例如：負責中斷處理的函式應該要比一般函式的執行更需要優先被執行，所以現在我們把原本的核心模式驅動程式搬到使用者模式，也應該要保持這種原則。所以，我們把使用者模式驅動程式模擬成多執行緒的程式，如 Figure 3，我們可以把程式切成幾個不同的執行緒。Main thread 主要功能是接收來自核心的 IO Requests。Interrupt thread 負責執行中斷處理函式，我們可以在驅動程式呼叫 request_irq()去註冊中斷處理時去產生。另外，我們可以在驅動程式的執行環境中產生兩個執行緒，Timer thread 負責自行維護一個計時器佇列(Timer queue)，定期執行一些工作，BH thread 負責執行一些驅動程式延後執行的工作。另外，驅動程式也可以自己產生執行緒，所以一個驅動程式可能會有多個執行不同功能的執行緒存在，我們可以根據執行緒的功能來設定不同的優先權。

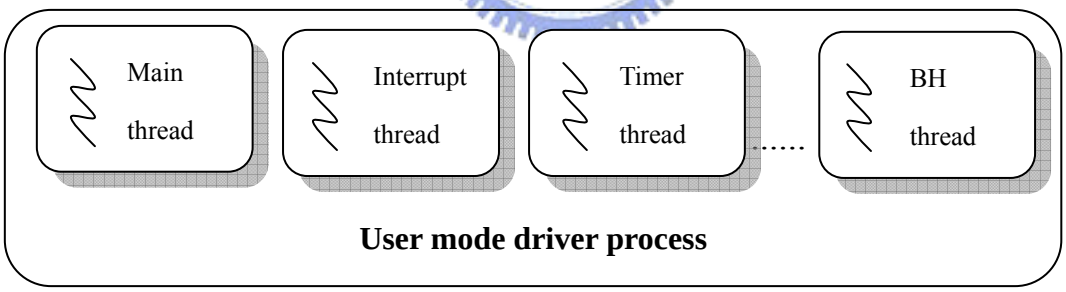


Figure 3：A User Mode Driver Process

3.1.1.3 核心物件同步

爲了能重用核心模式驅動程式，我們必須提供一個類似於核心模式的環境給驅動程式。而其中一個要素便是要允許驅動程式使用它原本需要的核心物件。例如：一個網路卡驅動程式需要使用基座(socket)緩衝區結構(skb_buff)。我們必須讓這些物件在核心空間與使用者空間都有一份複本，這些複本代表同一個的物件，只是它們所在的位址空間不一樣。我們必須適時的去同步這些複本內的資料，因爲核心程式或驅動程式都可以修

改這個物件。

爲了達成上述目的，我們去攔截核心程式與驅動程式之間的介面。不論是核心程式呼叫驅動程式或者是驅動程式呼叫核心程式的介面，以記錄哪些物件是驅動程式會存取到的，並把它們的對應關係記錄起來。並且當這些物件在核心空間與使用者空間傳遞時，我們必須做位址轉換的動作，也就是說把使用者空間的物件位址轉換成它在核心空間中對應的物件位址，反向亦然，這樣才不會執行發生錯誤。

3.1.2 架構元件

爲了實現我們之前所提到的目標，我們必須在核心程式與驅動程式間插入一層新的轉送模組，如 Figure 4，因爲我們的驅動程式現在是執行在使用者空間，所以必須要把原本核心程式的請求轉送到使用者空間，這個模組具有透明性，核心程式並不知道底層驅動程式是執行在核心或使用者模式。接著，我們要重用核心模式驅動程式，所以我們必須提供一個環境去模擬核心所提供的介面，讓驅動程式可以使用。最後，我們將說明我們如何在使用者空間實現驅動程式所要存取的核心物件。我們將在這一小節中介紹我們要達成重用核心模式驅動程式所需要的一些元件(Components)：Proxy Server，Shared Memory，Runtime Environment，及 Translation Table。

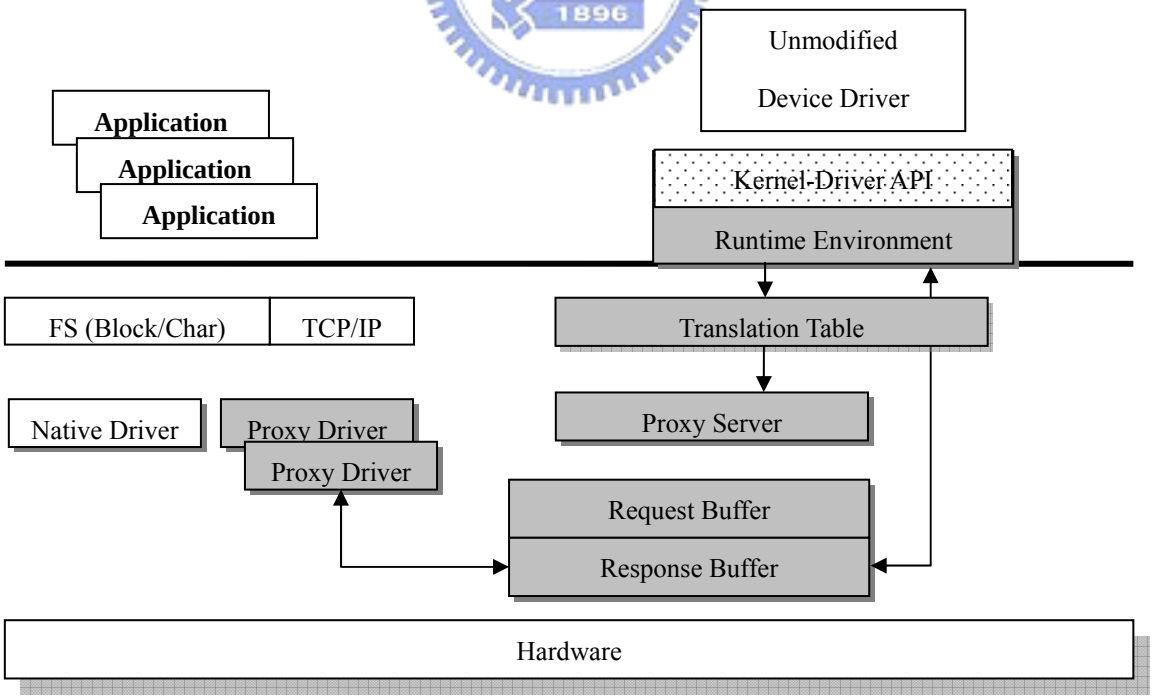


Figure 4：User Mode Device Driver Framework

3.1.2.1 Proxy Server

Proxy Server 位於核心空間中，主要負責以下工作：

1. 如 Figure 5，每當使用者模式驅動程式向核心註冊時，例如：當呼叫函式 register_netdevice()註冊一個新的網路卡驅動程式時，Proxy Server 就會代表使用者模式驅動程式向核心註冊一個代理驅動程式(Proxy Driver)，讓核心知道底層有新的裝置可以使用。代理驅動程式主要目的就是轉送來自核心的 IO Requests 給使用者模式驅動程式。
2. 若使用者模式驅動程式需要一些核心的服務，例如：要求 IRQ 服務，I/O Regions，與 DMA…等，代理驅動程式會代表使用者模式驅動程式去要求這些資源，並且把請求結果傳回給使用者模式驅動程式，例如：驅動程式可能需要去請求一塊 DMA 記憶體空間，所以會傳回 DMA 物理記憶體位址(physical memory address)。另外，Proxy Server 也需要去記錄每一個使用者模式驅動程式曾經要求過哪些核心資源以便在驅動程式執行結束時釋放其佔據的相關資源。

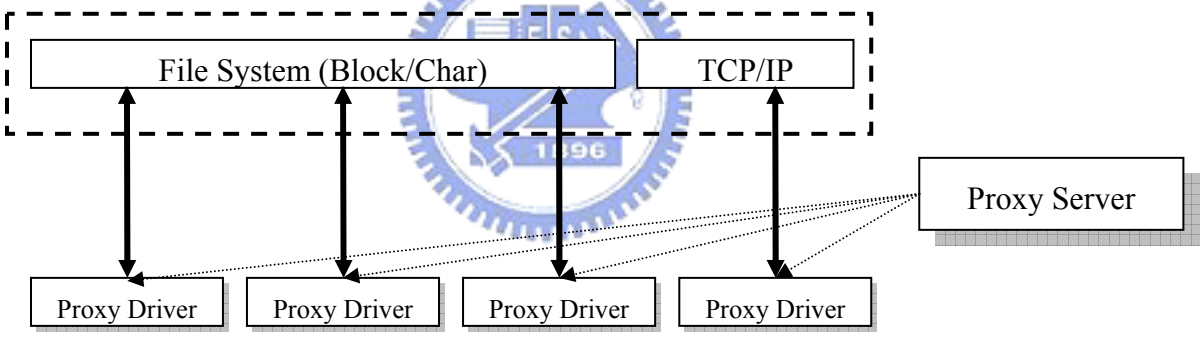


Figure 5：Proxy Server & Proxy Driver

3.1.2.2 Shared Memory

實現核心程式與使用者模式驅動程式之間的溝通有不同的方式，可以用 IPC 方法中的管道(pipe)，讓核心程式與使用者模式驅動程式共同分享一個管道，作爲傳遞資料的方法。然而爲了減少傳遞資料的負擔，我們選擇用 IPC 方法中 Shared Memory 的方式，讓核心與使用者模式驅動程式共同分享一塊記憶體空間。

我們使用二個 Shared Memory 區域：Request Buffer 與 Response Buffer。每一個代理驅動程式與使用者模式驅動程式之間都會有一組 Request Buffer 與 Response Buffer。Request Buffer 與 Response Buffer 的資料佈局(Data layout)雷同，如 Figure 6 所示。我們使用生產者-消費者(Producer- Consumer)的結構。對於 Request Buffer 而言，代理驅動程

式是生產者，使用者模式驅動程式是消費者，對於 Response Buffer 而言則反之。關於 Shared Memory 的資料佈局，分爲三個部分：Metadata，Ring 及 Data。最前面的部份是 Metadata，主要記錄整個 Buffer 的情況。欄位 Ring Producer 與 Ring Consumer 分別表示 Ring 中可以使用以及最後仍然在使用中的 Ring 中元素。欄位 Data Producer 與 Data Consumer 分別表示目前可以使用以及最後仍然在使用中的 Data 區域。接著的部份是 Ring，每一個 Ring 的元素(element)代表一個 IO Request 或一個 IO Response，欄位 Cmd 說明這個 IO Request 是哪一種函式類型，例如：網路驅動程式的 open()與 hard_start_xmit() 函式。欄位 Data start 與 size 記錄傳遞到使用者空間的資料在 Buffer 中的位址及大小，這些資料是函式的參數。並且驅動程式會存取到參數所指到的相關物件，我們都將其放在 Buffer 的最後一個部份--Data Section，使用者模式驅動程式可以直接存取 Shared Memory 中的資料。

Figure 7 爲代理驅動程式與使用者模式驅動程式之間溝通的協定，每當核心程式需要驅動程式服務時，就會呼叫代理驅動程式的函式，代理驅動程式會把要求包裝成 IO Request，然後放到 Request Buffer 中，並且修改 Request Buffer 中 Metadata 的 Ring Producer 與 Data Producer 欄位，並且去喚醒使用者模式驅動程式。當代理驅動程式送 Request 給驅動程式後，驅動程式並不會馬上起來執行，Request 通常會批次(batch)多個之後，驅動程式才會開始處理 Request。接著，使用者模式驅動程式就會去讀取 Metadata，判斷說是否有新的 IO Requests，然後做相對應的工作。若完成之後，則把相對應的 IO Response 放到 Response Buffer，修改 Response Buffer 中 Metadata 的 Ring Producer 與 Data Producer 欄位，並通知代理驅動程式。接著代理驅動程式收到 IO Response 之後，知道哪一個 IO Request 已經完成，則會一起更新 Request Buffer 與 Response Buffer 中 Metadata 的 Ring Producer 與 Data Producer 欄位。所以代理驅動程式能夠修改 Request Buffer 與 Response Buffer 的 Metadata，而使用者模式驅動程式只能修改 Response Buffer 的 Metadata，這樣可以確保每一個 IO Request 是否已經執行完畢。

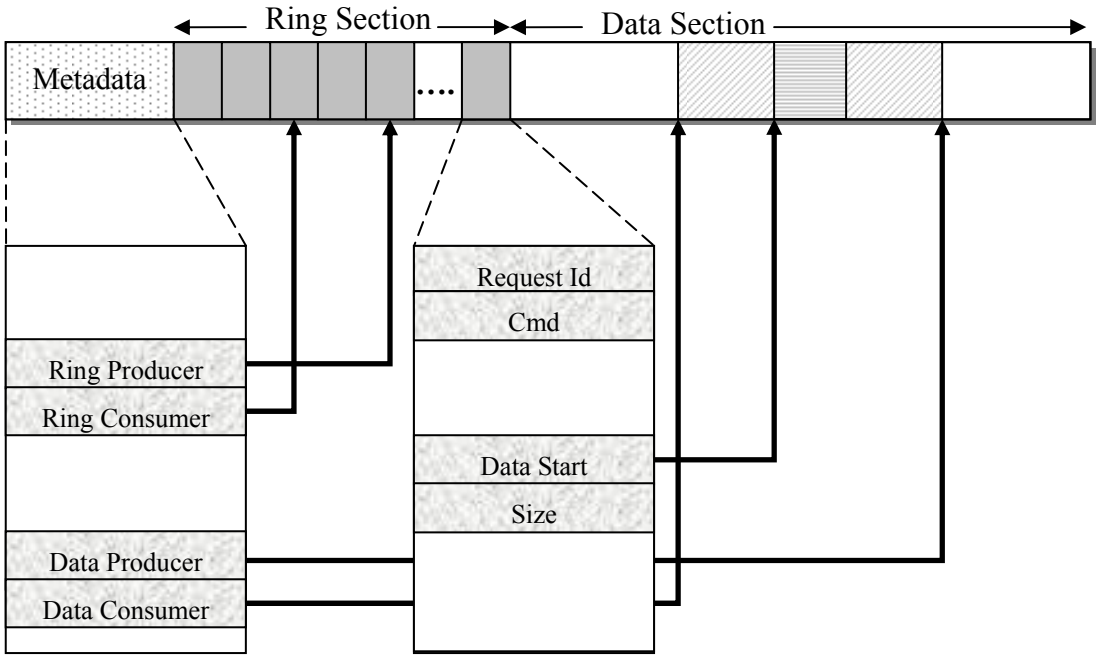


Figure 6：Data Layout in Shared Memory

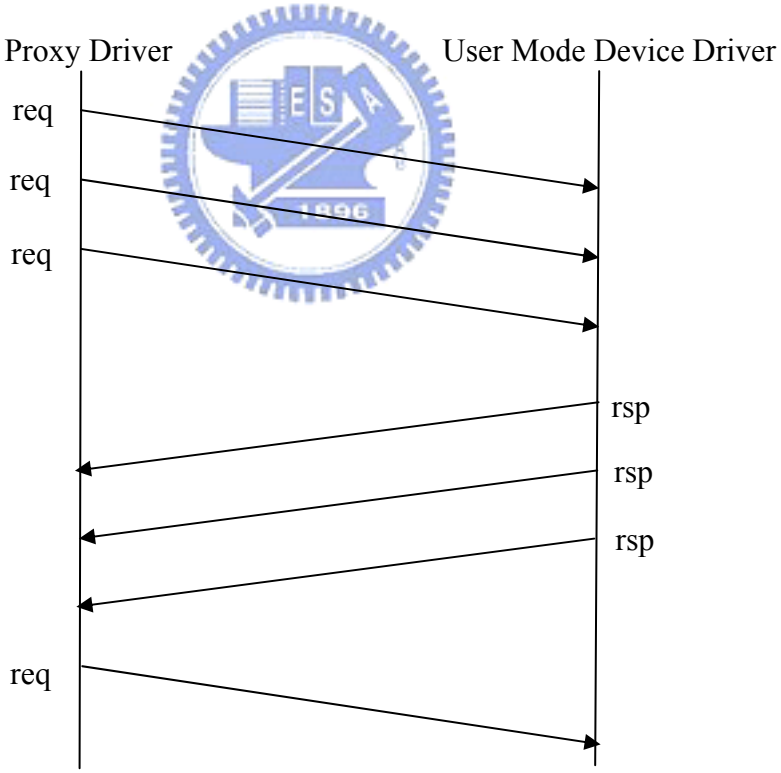


Figure 7：Communication Protocol

3.1.2.3 Runtime Environment

我們將 Runtime Environment 實作爲一個使用者空間的靜態連結函式庫

(Statically-Link Library)。它主要負責兩個工作：

1. 提供驅動程式所需要的 API 以及模擬一些驅動程式需要用到的核心程式變數資料。需要提供的 API 類型根據原本核心分類有 IRQ，I/O Region，DMA，Timer queue…等。這些 API 依據實作的方式可以分爲三類：

(a) 第一類 API 需要核心支援，所以我們利用系統呼叫去向 Proxy Server 要求，Proxy Server 會進一步去向原本核心所提供的服務請求，中斷處理與存取 DMA 都屬於此分類中。我們對於中斷處理與存取 DMA 的方法如下：

- 中斷處理(Interrupt handler)：

  我們使用與澳洲新南威斯大學相同的方法[17]，如 Figure 8，我們一開始在 /proc/irq/n(n 表示 irq 的號碼)目錄下去創造一個檔案 irq，當 Interrupt thread 去打開/proc/irq/n/irq 時，在核心中將有一個我們自己寫的中斷處理函式會被註冊。當 Interrupt thread 讀取這個檔案，如果這個檔案中沒有資料，則 Interrupt thread 將會進入睡眠狀態，等待中斷發生。而當有中斷處理發生時，則核心中的中斷處理函式則會寫入資料到這個檔案，接著 Interrupt thread 則會被喚起去讀取資料，並且去執行驅動程式中的中斷處理函式。

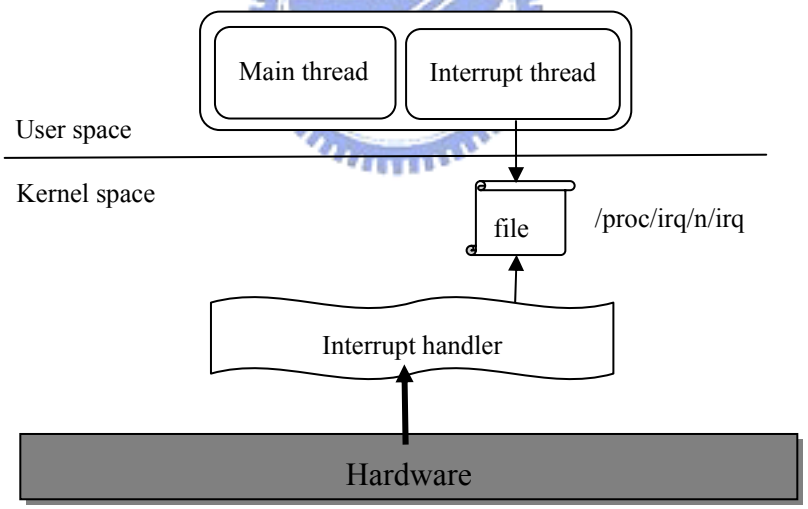


Figure 8：Interrupt Handler Flow

- DMA 存取：

  使用者模式驅動程式所用的緩衝(buffer)是利用虛擬位址(virtual address)做區別，而有 DMA 能力的裝置卻需要把這些位址轉換成 I/O 匯流排位址，I/O 匯流排位址可能是物理記憶體位址或者是 I/O 匯流排映射到物理記憶體的位址。如果說裝置需要使用到 DMA 的話，例如：PCI 裝置去請求 DMA 的 pci_alloc_consistent()函式， 我們會先在核心模式中去分配一塊 DMA 記憶體空

間，然後把這塊記憶體空間映射到驅動程式的位址空間，讓驅動程式可以存取，並且核心也要傳回真實的物理記憶體位址，讓驅動程式可以設定硬體時使用。

(b) 第二類 API 可以直接實作在使用者空間，存取記憶體映射 I/O (Memory-Mapping I/O)與計時器佇列(Timer queue)都屬於此分類。我們對於存取記憶體映射 I/O 與計時器佇列模擬的方法如下：

- 記憶體映射 I/O (Memory-Mapping I/O)：
  有些裝置會映射它們的暫存器到記憶體位址空間，爲了能夠去存取物理記憶體位址，在 Linux 中，/dev/mem 檔案可以提供存取物理記憶體的功能，我們可以藉由對於/dev/mem 檔案用呼叫 mmap 的方式，把驅動程式所需要的位址區段映射到使用者模式驅動程式的位址空間，讓驅動程式可以直接存取。
- 計時器佇列(Timer queue)：
  在核心中，計時器佇列的任務是執行一些驅動程式所設定多久之後要執行的工作，相關函式有 add_timer()-把工作加入計時器佇列中，以及 del_timer()-把工作從計時器佇列中移除等。所以在驅動程式初始化時我們會去創造一個執行緒去模擬計時器佇列的行爲，當這個執行緒被排程到時，則會去檢查計時器佇列中是否有需要執行的工作，如果有的話則去執行，若沒有的話則進入睡眠狀態。

(c) 第三類爲可以直接轉換成使用者空間函式庫中的函式，例如：printk()可以直接轉換成 printf()函式。

而需要提供的核心程式變數資料可能有 jiffies，我們可以利用 gettimeofday()去實現。

2. Runtime Environment 並且負責接收來自 Shared Memory 中的 IO Requests，分辨是哪一種函式類型，然後再呼叫驅動程式中處理此 Request 的對應函式，例如：在驅動程式註冊時，我們會去記錄驅動程式中提供給核心呼叫的函式，若 Request 類型爲 open()，我們則可以去呼叫驅動程式中相對應的 open()函式。當 IO Request 處理完畢之後，會將此 IO Request 對應的 IO Response 填入 Response Buffer 中，表示說這個 IO Request 已經被處理完畢。

3.1.2.4 Translation Table

如前所述，驅動程式會用到的核心物件在核心空間與使用者空間都同時存在一複本，且我們必須記錄這兩複本的位址對應關係。如此，當這個物件在核心空間與使用者空間轉換時，我們才能轉換相對應的位址。我們攔截核心程式與驅動程式之間的一些 API，然後利用 translation table 來記錄這個物件在核心空間以及在使用者空間的位址。對每一個物件，我們都用一個表格元素(table entry)記錄<kernel_addr, user_addr>，kernel_addr 表示說這個物件在核心空間的位址，而 user_addr 表示說它在使用者空間的位址，當核心空間與使用者空間轉換時，我們便可以利用這個表格元素去做轉換。

因爲每一個使用者模式驅動程式都有自己的位址空間，所以有可能會造成兩個不同驅動程式具有相同使用者空間位址，例如：有兩個驅動程式所需要的不同物件可能具有相同的使用者空間位址，所以有可能會造成位址衝突(collision)，因此對於每一個使用者模式驅動程式我們都爲它建立一個表格。

3.2 無資料遺失(Zero-Loss)之驅動程式

過去對於使用者模式驅動程式架構的研究，都只著重於作業系統的可靠度。他們在使用者模式執行驅動程式，使得驅動程式發生錯誤時也不會危害到作業系統。但是如此並不能完全解決問題，因爲當驅動程式一旦發生錯誤而當掉，所有應用程式與核心程式都無法存取該驅動程式所管理的設備。例如：一個 Web 伺服器需要網路裝置，當網路卡驅動程式當掉之後，Web 伺服器也將會無法再提供服務。所以，除了可靠度之外，我們還必須考慮到作業系統的可用度。

對於要實現驅動程式的可用度，意指當驅動程式出錯時，能夠達到無資料遺失(Zero-Loss)，無資料遺失代表著即使當驅動程式出錯，上層的核心程式或應用程式並不知道驅動程式已經出錯，它們依然能夠繼續送請求給驅動程式，並且在驅動程式回復後，驅動程式能夠繼續完成尚未完成的請求，並不會因爲驅動程式出錯而導致原先的請求遺失，進而影響到核心程式或者是應用程式的執行。

對於回復(Recovery)的一般做法是去執行相同的應用程式在不同兩台機器上，一個爲主控(Primary)和一個爲備份(Backup)。所有對於主控的輸入(Input)同時也會有一份轉送到備份，當主控發生錯誤時，備份則會接管並且繼續提供服務。我們也用類似的方式去達到驅動程式的回復。

我們修改原有的使用者模式驅動程式架構以支援可信賴度的驅動程式，修改後架構如 Figure 9 所示，我們增加了一個備份驅動程式(Backup Device Driver)以及兩個新的元

件-Fault Detector 與 Recovery Manager，並且修改 Shared Memory 部份，下面將介紹這些修改。

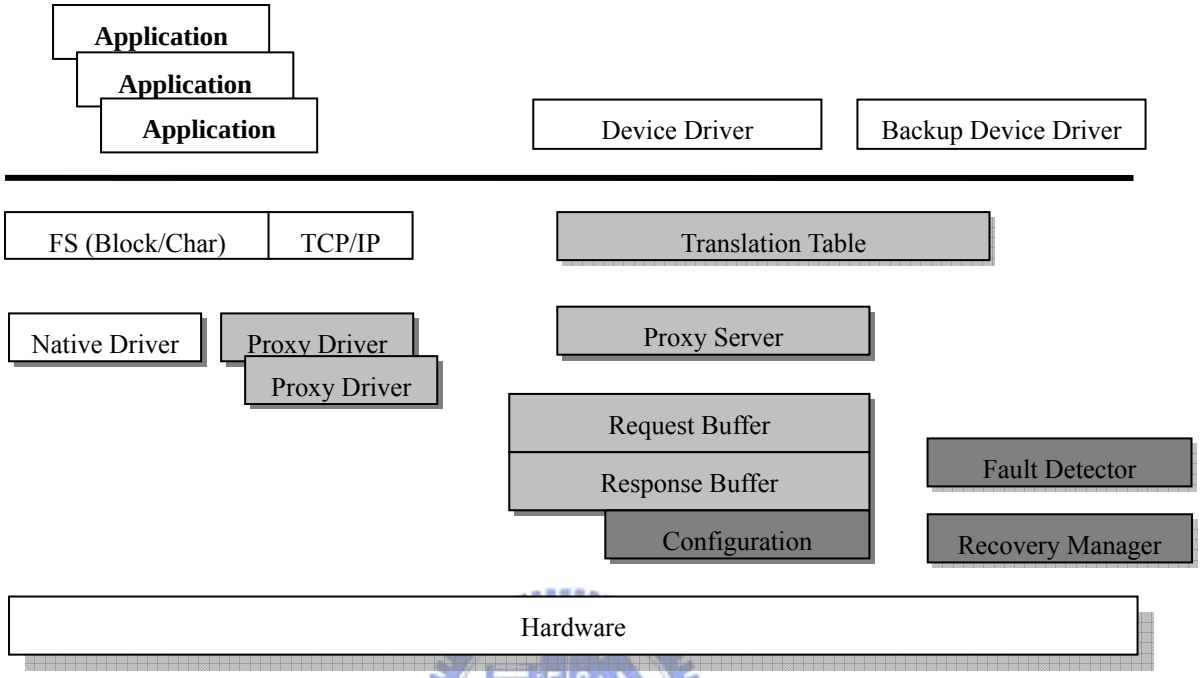


Figure 9：User Mode Device Driver Framework with Fault-Tolerant Mechanism

3.2.1 Backup Device Driver

我們利用主控-備份的回復機制，與過去不同的地方主要有兩點。第一個是我們只針對驅動程式所發生的錯誤，沒有考慮硬體所造成的錯誤，所以我們把驅動程式與備份驅動程式(Backup Device Driver)執行在同一台機器上。第二個是我們的備份驅動程式平時處於睡眠的狀態，所以不會接收與驅動程式相同的輸入。而備份驅動程式可以跟原本的驅動程式一樣版本，也可以是另外一個驅動程式版本。並且這個備份驅動程式並不用做額外修改，在它初始化時會跟核心註冊，Proxy Server 若發現之前已經有相同驅動程式註冊，則備份驅動程式就會被延遲(Blocking)，停止接下來的初始動作而進入睡眠狀態，因此它不會消耗 CPU 時間。

3.2.2 Modified Shared Memory

在 3.1.2.2 節中，我們讓驅動程式可以自由讀寫 Request Buffer，這建立在一個假設之下：驅動程式會永遠正常運作，但事實上，驅動程式可能因爲出錯而污染到 Request Buffer，所以我們必須對 Request Buffer 做記憶體保護(Memory Protection)的機制，只允

許 Request Buffer 能夠讀取而不能寫入，這也意味我們必須重複(replicate)相同資料在驅動程式中，所以這會增加額外記憶體複製(memory copy)的負擔。Response Buffer 並不用做記憶體保護，我們只需要在驅動程式回復後，把 Response Buffer 中的資料忽略掉，因爲這並不會影響到驅動程式的執行。

另外，我們必須新增一個 Configuration Buffer，它主要是要記錄目前驅動程式之前已經作過了哪些 configuration 操作，例如，open()與 ioctl()。Configuration Buffer 一開始並不會映射(map)到原本的驅動程式位址空間，只有當備份驅動程式啓動時，Configuration Buffer 才會映射到備份驅動程式的位址空間，所以當驅動程式出錯時並不會污染到 Configuration Buffer。當重新啓動一個驅動程式程序，必需重新 configure 驅動程式的組態設定，讓它恢復到當掉前的狀態。所以我們可以判斷 IO Requests 是否會改變驅動程式的組態設定，若會的話，我們則把它拷貝一份在 Configuration Buffer 中。

3.2.3 Fault Detector & Recovery Manager

要達到自動的錯誤偵測與啓動驅動程式回復的功能，我們必須要新增另外兩個新的元件，如下：

Fault Detector：

根據 Stanford 大學研究顯示，驅動程式最常出現的錯誤就是 exception 與 blocking 錯誤[4]。前者是驅動程式開發時產生的 bug 造成 CPU 產生 exception，例如：用 C 語言的指標時，就很容易寫出 null pointer 的錯誤，這樣就會造成 CPU 產生 exception。後者則是驅動程式一直停滯，無法繼續執行工作，有可能是程式寫錯造成無窮迴圈，也有可能是因爲死結(deadlock)或活結(livelock)的問題。

對於驅動程式 exception 的產生，我們可以攔截例外處理函式以判斷是否驅動程式已經產生錯誤，然後通知 Recovery Manager。對於 blocking 錯誤，目前研究有許多現成的偵測工具[8]，能夠動態的去偵測是否已經發生了死結的情形，或者我們也可以用簡單的機制，去定期的檢查驅動程式是否已經有一段都沒消耗 IO Requests。若是，表示驅動程式可能已經無法提供服務，必須進行回復的動作。目前我們的 Fault Detector 只支援 exception 的偵測。

Recovery Manager：

Recovery Manager 主要負責啓動驅動程式的回復。回復的流程如 Figure 10 所示。

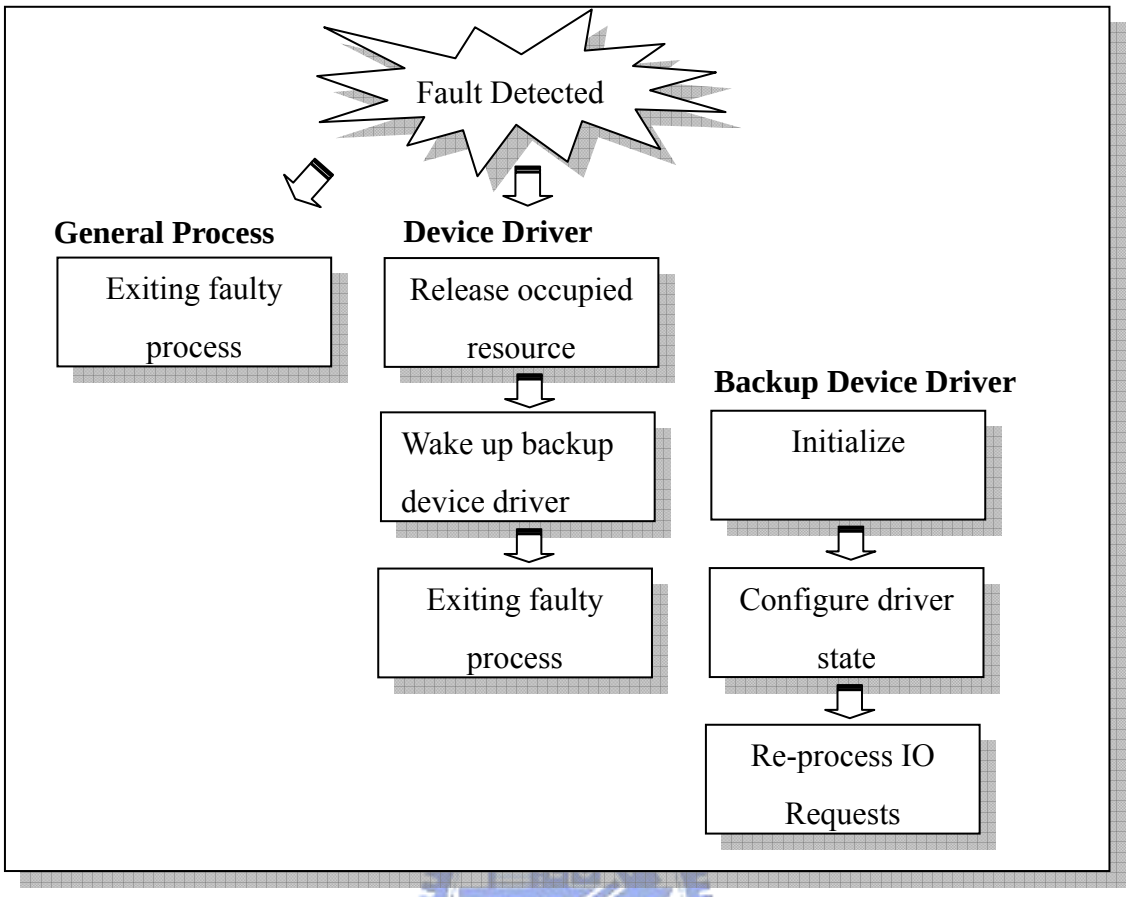


Figure 10：Recovery Flow

當 Fault Detector 偵測到程序發生錯誤時，會先判斷這是一般應用程式或者是驅動程式。若是一般應用程式的話，我們就忽略之，如果是驅動程式的話，我們即啓動回復流程。在回復期間中，核心程式並不知道底層的驅動程式已發生錯誤，所以它可以繼續送 IO Requests 給驅動程式，而這些 IO Requests 都會被暫時佇列在 Request Buffer 中。當然，對於一般的應用程式也不會知道驅動程式已經發生錯誤。

在回復流程中，Recovery Manager 首先釋放驅動程式所佔據的資源，例如：IRQ，DMA…等，因爲這些資源並不是一般應用程式所會用到資源，所以當應用程式結束時，作業系統並不會去釋放，因此我們必需額外去處理。而有一些驅動程式所佔據的資源我們可以保留給新的驅動程式，例如: 在備份驅動程式在初始化時會去請求新的 net_device，但 Proxy Server 會知道之前有相同的 net_device 結構，所以會將 net_device 結構直接傳回。當釋放佔據的資源之後，驅動程式就可以像一般應用程式一樣終止執行。

Recovery Manager 會喚醒備份驅動程式，備份驅動程式會繼續執行初始化的動作，不同的是，新的驅動程式有些資源是沿用之前發生錯誤的驅動程式的，所以可以不用重新請求資源。備份驅動程式爲了能恢復到發生錯誤前的狀態，我們會先將 Configuration Buffer 映射到備份驅動程式的位址空間中，然後先去執行 Configuration Buffer 中的 IO

Requests。接著，我們可以重新映射(remap)之前的 Request Buffer 到備份驅動程式中，驅動程式可以去讀取 Request Buffer 中 Metadata 的相關內容，然後繼續接著執行 Request Buffer 中尚未完成的 IO Requests。

3.2.4 Summary

我們希望能達到無資料遺失驅動程式，也就是說當驅動程式當掉之後，我們不希望影響到其他一般應用程式的執行，一般應用程式並不知道驅動程式已經出錯，仍然能照以往一樣繼續執行，雖然可能會被延遲，但是盡可能讓使用者並不會感受到服務出問題。

我們達到無資料遺失驅動程式的功能藉由：(1)因爲我們是用代理驅動程式，當使用者模式驅動程式當掉之後，對於核心程式而言，它並知道使用者模式驅動程式當掉，核心程式還是能繼續丟 Request 給代理驅動程式，然後代理驅動程式則暫時將 IO Requests 佇列住在 Shared Memory；(2)當備份驅動程式啓動後，它重新接收之前當掉驅動程式的 Shared Memory，它可以根據 Request Buffer 中的 Metadata 去知道那些 IO Requests 尚未被執行，但是可能因爲驅動程式在回覆 IO Response 前就已經當掉，所以驅動程式重新啓動後可能會執行之前已經執行過的 IO Requests，這種重覆執行的行爲對於網路卡或區塊裝置不會有影響，但是對於字元裝置可能會受到影響[29]。

3.3 使用者模式驅動程式架構雛形(prototype)

我們已經將所提出的使用者模式驅動程式架構實作在 Linux-2.6.12 上。此實作分爲兩部份，一個是核心程式部份，我們把它實做成可載入模組(Loadable module)。另一部份是靜態連結函式庫，主要是要提供給驅動程式連結使用。目前 Linux 將驅動程式分成不同類別：網路驅動程式，區塊驅動程式，與字元驅動程式，不同類型提供不同的介面讓核心程式呼叫。目前已經完成三個核心模式驅動程式搬移的工作，一個是 Tulip Ethernet 網路卡驅動程式，一個 RAMDISK 區塊驅動程式，以及一個 PS2 Mouse 字元驅動程式 。

第四章　實驗結果與討論

本章我們對我們所提出的使用者模式驅動程式架構進行效能測試，並且針對測試結果作分析與討論。4.1 節描述我們的實驗環境，4.2 節描述網路卡驅動程式的效能，4.3 節描述 RAMDISK 區塊驅動程式的效能，4.4 節則展示驅動程式之無資料遺失功能的正確性與效能。

4.1　實驗環境

我們選擇 Linux 做爲我們的實驗平台。我們的實驗環境主要包含三台機器，如 Table 4.1 所示，一台 Server 與兩台 Client。這三台機器共同接到一台 100Mbps 的 switch 上，而我們的使用者模式驅動程式執行在 Server 機器上。

我們用 Netperf-2.4.1[23]作爲測試網路卡驅動程式的 Benchmark 以測量網路傳送與接收的效能。用來測試 RAMDISK 區塊驅動程式的 Benchmark 爲 Postmark-1.5[16]與 Dbench-3.03[31]。Postmark 是由美商網域(Network Appliance Inc.)所發展出來的一個檔案系統 Benchmark，它主要是模擬一個郵件伺服器的工作負荷量(workload)。Dbench 則是模擬檔案伺服器(例如：samba 伺服器)的一個 Benchmark。

Table 1：Experimental Environment

	Client 1	**Client 2**	**Server**
Hardware	P4 2.0 GHz 256MB memory Accton EN-1216 100 Mb/s fast NIC	P4 2.0 GHz 256MB memory Accton EN-1216 100 Mb/s fast NIC	P4 3.2 GHz 512MB memory Accton EN-1216 100 Mb/s fast NIC
OS	Linux 2.4.20	Linux 2.4.18	Linux 2.6.12
Benchmark	Netperf-2.4.1	Netperf-2.4.1	Netperf-2.4.1 Postmark-1.5 Dbench-3.03

4.2　網路卡驅動程式的效能測試與分析

這一節中我們將對網路卡驅動程式分析其執行在核心模式與使用者模式的行爲差

異，並且評估網路卡驅動程式的效能以及使用者模式驅動程式架構的負擔。

4.2.1 網路卡驅動程式分析討論

此部份將探討我們將原本的核心模式驅動程式搬移到使用者模式後，驅動程式執行上行爲的差異，並且做進一步的分析討論。

4.2.1.1 網路卡驅動程式傳送行爲改變

一旦我們把驅動程式從核心模式搬到使用者模式，驅動程式的優先權即會降低，因而使核心程式(例如：檔案系統，TCP/IP)的優先權變得比使用者模式驅動程式高(除了驅動程式的中斷處理外，原本優先權應該是一樣的)。如此一來，核心程式就可以一直把驅動程式的執行先佔(preempt)掉，造成傳送封包的行爲改變。

原本驅動程式在核心模式時，TCP/IP 一送封包即會觸發驅動程式的立即執行，所以其屬於同步(synchronous)執行。但是當驅動程式處於使用者模式時，TCP/IP 傳送封包並不會讓驅動程式馬上執行，而是可能會批次多個封包後驅動程式才會被執行，所以其屬於非同步(asynchronous)執行。爲了展示這樣的行爲差異，我們在 Server 上執行 Netperf 以傳送封包給 Client1。在同時我們計算當批次多少個請求後，驅動程式才會開始執行。結果如 Figure 11 所示，核心模式驅動程式因爲是同步執行，所以不會批次任何請求。而使用者模式驅動程式則大約批次 11-25 個請求之後才會開始執行。

Tx ring buffer 是網路卡所需要的一塊記憶體空間，在驅動程式初始化時就會靜態的去分配 Tx ring buffer 的大小。Tx ring buffer 是一個環狀的結構，每個 Tx ring buffer 的元素都代表一個網路卡要傳送的封包，所以之後 Tx ring buffer 的元素個數是固定的。這個現象將造成傳送端的 Tx ring buffer 可能會溢位(overflow)的問題。爲了解釋此一問題，我們假設 Ts 代表硬體層的 Tx ring buffer 的元素個數。

在我們的環境下 Ts 爲 32。當系統需要傳送封包而呼叫網路驅動程式的 hard_start_xmit()這個函式時，此函式會判斷目前已送出但是尚未 ack 的封包數是否已經快超過 Ts，如果快超過，該函式就會呼叫 netif_stop_queue()函式去通知 TCP/IP 暫時停止傳送封包。然而因爲網路卡驅動程式的執行被延遲，所以當 hard_start_xmit()判斷 Tx ring buffer 已經快溢位而呼叫 netif_stop_queue()函式時，早已太遲，因爲之前在 Shared Memory 中已經佇列多個 Request，驅動程式依然會去讀取 Request 並且繼續傳送封包，結果造成 Tx ring buffer 溢位。

爲了解決此一問題而又不更動驅動程式程式碼，我們把此種封包傳送的流量控制 (flow control) 實作在 Runtime Environment 中。我們攔截 hard_start_xmit 與 Tx 中斷處理

函式中的 dev_kfree_skb_irq()函式: 由 hard_start_xmit 我們可以得知已經使用掉多少個 Tx ring buffer 元素，由 Tx 中斷處理函式中的 dev_kfree_skb_irq()函式我們可以知道已經釋放多少個 Tx ring buffer 元素，因此可以知道現在 Tx ring buffer 元素是否已經將用光，而將造成 Tx ring buffer 溢位。

若即將溢位時則呼叫 sys_yield()讓出 CPU 以使得中斷處理函式能夠執行以釋放所佔據的 Tx ring buffer 元素。如此一來，驅動程式執行便不會出問題。

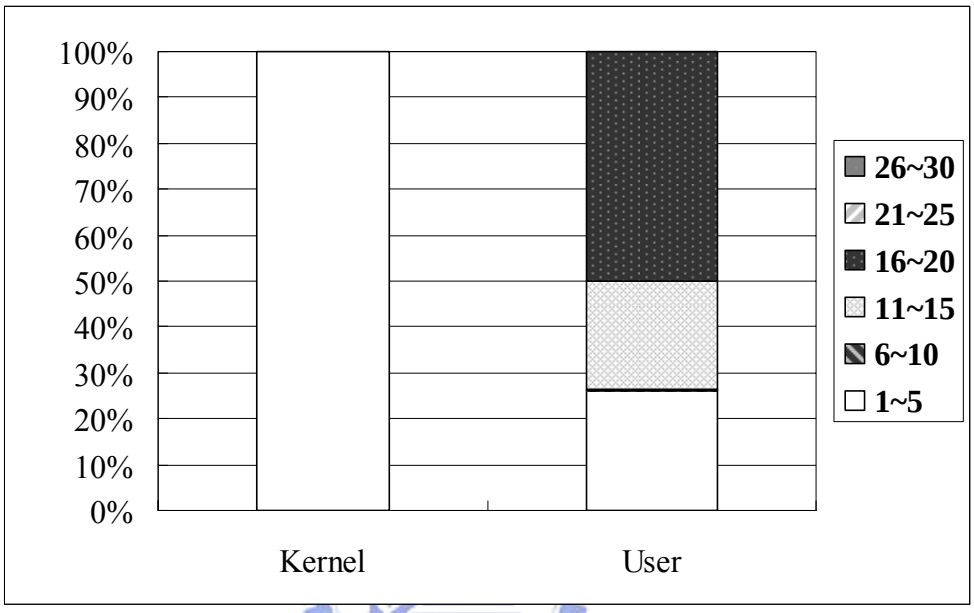


Figure 11：Network Send Behavior

4.2.1.2 驅動程式中執行緒之間優先權關係

在此節我們要討論若使用者驅動程式包含多個執行緒時，各執行緒間的優先權關係對驅動程式效能的影響。以網路卡驅動程式爲例，其包含兩個執行緒。一爲 Main thread，負責傳送封包，另一爲 Interrupt thread，主要負責中斷處理。我們將調整 Main thread 與 Interrupt thread 之間優先權的關係並測量驅動程式傳送與接收方面的效能。

我們在 Server 上執行 Netperf 以傳送封包到 Client1，並且同時接收來自 Client2 的封包。此舉是爲了能夠加重網路卡驅動程式的負載(load)，然後我們去看 Server 端傳送與接收的效能結果。

結果如 Figure 12 所示，此圖分爲兩個區域，左區域代表 Server 傳送封包的效能，右區域則代表 Server 接收封包的效能。在每一區域中，最左邊的直條表示核心模式驅動程式的效能，另外三個直條各表示 Main thread 的優先權大於/等於/與小於 Interrupt thread 時的效能。我們可以看出當 Interrupt thread 大於 Main thread 時最接近核心模式的效能，這是因爲原本在核心模式中，驅動程式的中斷處理函式優先權本來就優先於傳送封包的

函式，在使用者模式也必須保持這個原則，因爲中斷處理通常是比較急迫的，若被延遲的話，可能會影響到驅動程式的執行，造成效能下降。

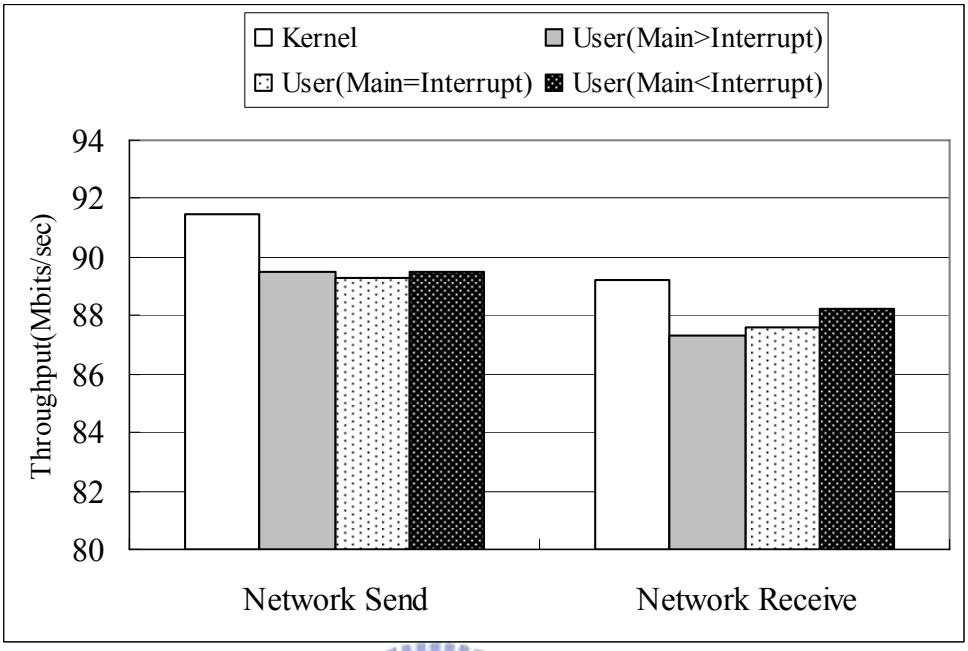


Figure 12：Thread Priority Relationship v.s. Performance

4.2.1.3 Tx Ring Buffer 與網路卡驅動程式效能關係

將驅動程式搬到使用者模式無疑地會增加額外的行程切換(context switch)負擔(Overhead)。此外，如我們在 4.2.1.1 節所提及，爲了能夠重用核心模式驅動程式並使其在使用者模式執行時不發生 Tx ring buffer 溢位的問題，所以當 Tx ring buffer 快要溢位時，我們會呼叫 sys_yield()以讓出 CPU 時間。此舉將又增加行程切換的次數。尤其當 Tx ring buffer 較小時，將會有許多額外行程切換的機會，造成 CPU 使用率的增加。

由上述可知，增加 Tx ring buffer 的容量有助於增加更多批次的優點因而抑制行程切換次數的增加。因此，我們測量在不同的 Tx ring buffer 容量下傳輸封包時的 CPU 使用率。結果如 Figure 13 所示，我們可以看出隨著 Tx ring buffer 容量的增加，CPU 使用率將會隨之變小。

雖然增加 Tx ring buffer 的容量有助於抑制行程切換次數的增加，然而因爲很多驅動程式對於其 Tx ring buffer 的容量是在 compile time 就已經決定，所以若要增加 Tx ring buffer 的容量，我們必須修改驅動程式的程式碼。值得注意的是，這個問題對於大部分核心模式網路卡驅動程式都會出現，這也是我們唯一因爲要改善驅動程式負擔而需要去修改程式碼的地方。

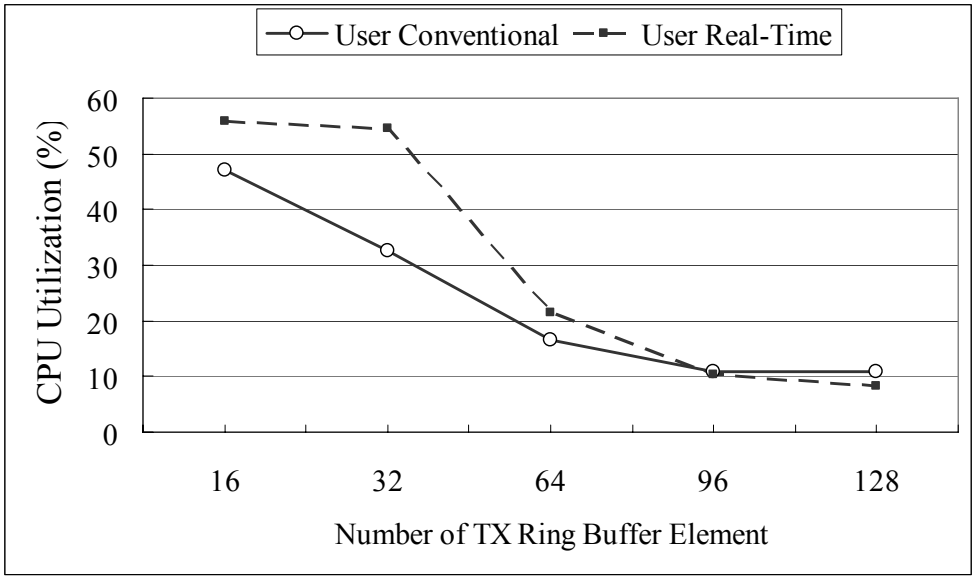


Figure 13：Tx Ring Buffer Size

4.2.2 已修改過的網路卡驅動程式效能表現

根據上一節的討論，我們設定驅動程式的 Tx ring buffer 的元素個數爲 128 個，接著去測試網路卡驅動程式的效能，我們的實驗環境爲 Server 與 Client1 利用 Netperf 去測量其 Server 個別傳送與接收的效能。結果如 Figure 14 所示，在此實驗中，我們利用兩種不同的使用者模式驅動程式設定：一種是將驅動程式設定成一般優先權的程序(User Conventional)，另一種是將驅動程式設定成表示爲即時優先權的程序(User Real-time)並讓驅動程式中的 Interrupt thread 的優先權大於 Main thread。結果顯示出使用者模式驅動程式與核心模式驅動程式的傳送與接收的效能(throughput)並無太大的差異。

接著，我們要量測網路卡驅動程式的 CPU 使用率，如上所提，驅動程式的 Tx ring buffer 的元素個數爲 128 個，並且一個封包的大小最大是 1500bytes。結果如 Figure 15 所示，圖中分別顯示核心模式驅動程式與使用者模式驅動程式的網路傳送與接收時的 CPU 使用率。由圖可知驅動程式的優先權設定成即時優先權時 CPU 使用率較設定成一般優先權的小，這是因爲當設定爲即時優先權時，驅動程式比較不會被其他程序先佔，所以可以執行更久，減少額外環境切換的負擔。

最後，我們測試 CPU 資源不足對於這兩種不同設定的使用者模式驅動程式有何影響。我們執行多個 for-loop 程序以搶奪 CPU 時間，並利用 Netperf 測量封包傳送及接收時的效能。結果如 Figure 16 所示，橫軸表示 for-loop 的程序的個數，縱軸表示網路傳輸的效能，結果顯示隨著 CPU 資源的減少，若使用者模式驅動程式爲一般程序，它的執行的時間將會被其他程序瓜分而造成效能下降。

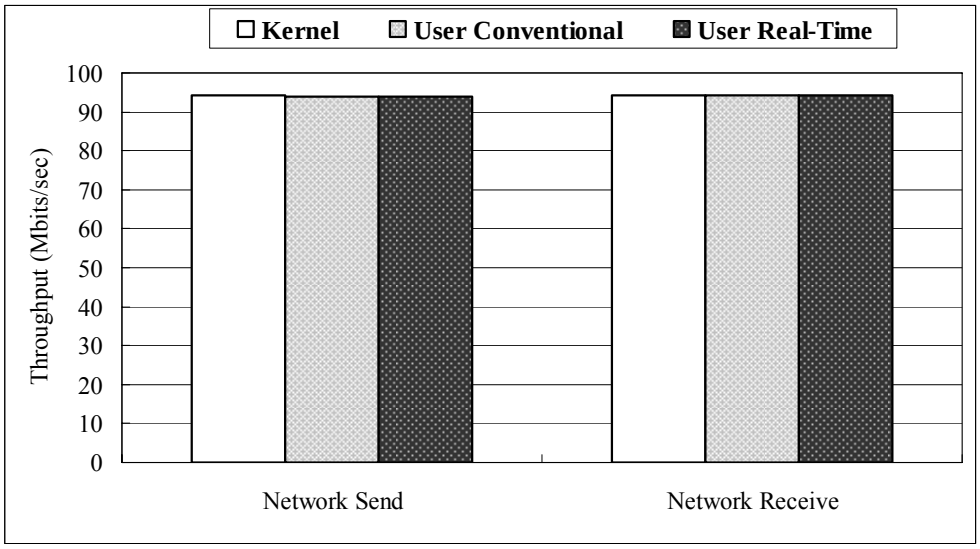


Figure 14：Network Throughput

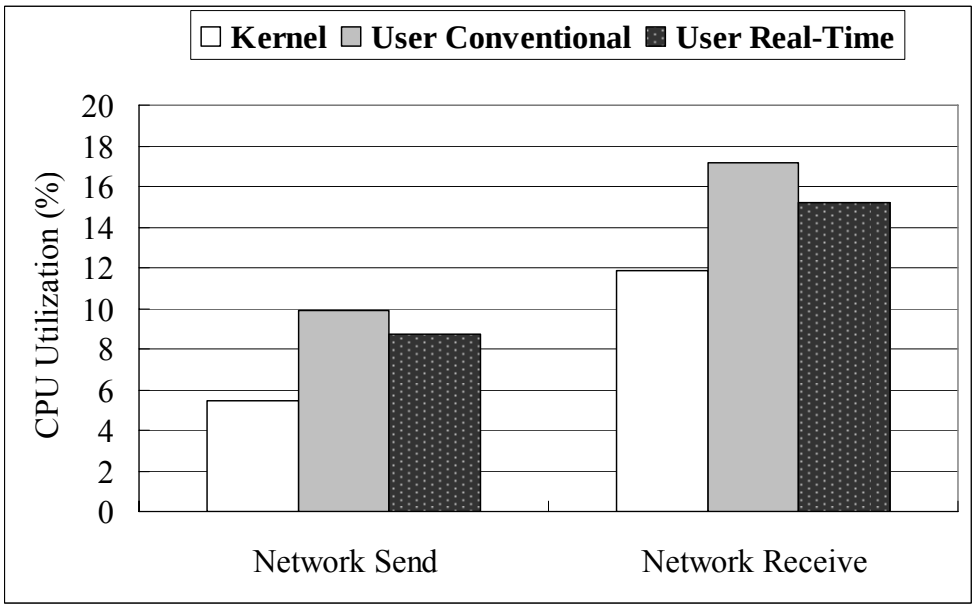


Figure 15：Network CPU Utilization

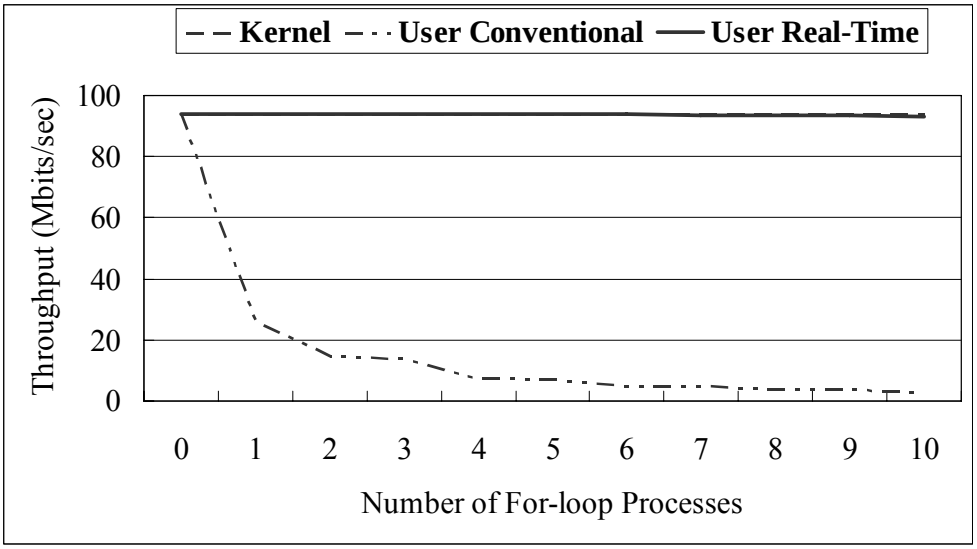


Figure 16：CPU Resource Leak

4.3 RAMDISK 驅動程式的效能測試

此節我們以 Postmark 及 Dbench 這兩個 Benchmarks，以及 tar 程式來評估 RAMDISK 驅動程式的效能。我們將 RAMDISK 的大小設定爲 128MBytes。對於 Postmark，我們設定其一開始產生 750 個大小介於 500Bytes~100KBytes 的檔案，並執行 200000 個 transactions。對於 Dbench，我們設定同時執行 5 個程序，每個程序重複執行 samba 的存取動作，執行時間爲 120 秒。對於 tar，我們利用它解壓縮 MySQL 版本 5.0.19 的原始碼。實驗結果如 Figure 17 所示，由圖可知我們的使用者模式驅動程式跟核心模式驅動程式的效能差異只有 1%以下。

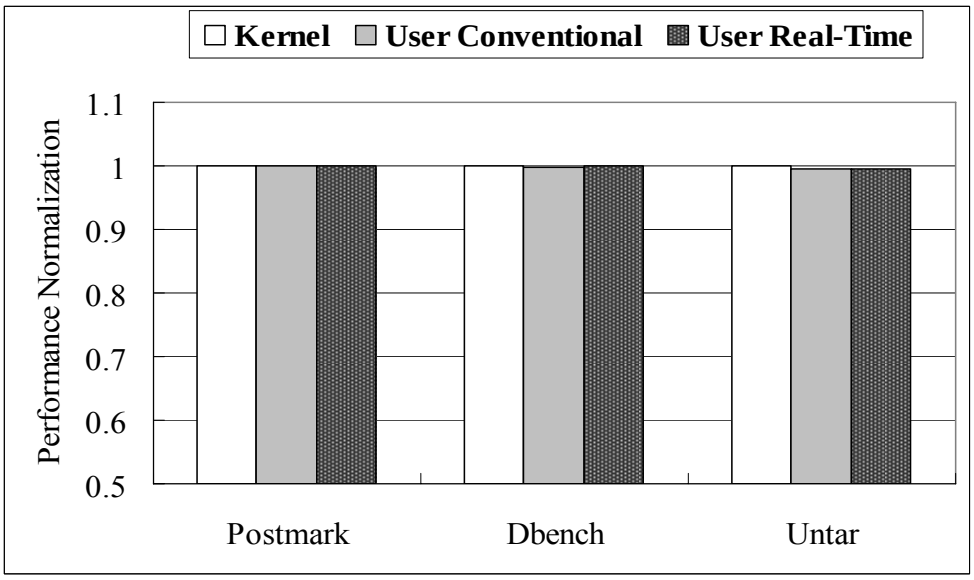


Figure 17：RAMDISK Performance

4.4 多個驅動程式同時執行

此節我們評估網路卡驅動程式與 RAMDISK 驅動程式同時執行時的效能表現，我們讓 Server 利用 ftp 去傳送或接收來自 Client1 一個任意大小的檔案，傳送或接收的檔案儲存在 RAMDISK 中，因此網路卡驅動程式與 RAMDISK 驅動程式是同時執行的，然後我們去比較使用者模式驅動程式與核心模式驅動程式的效能差異。利用 ftp 傳送的實驗結果如 Figure 18，利用 ftp 接收的實驗結果如 Figure 19，結果顯示出與核心模式驅動程式的效能差異大概是 1%~8%。

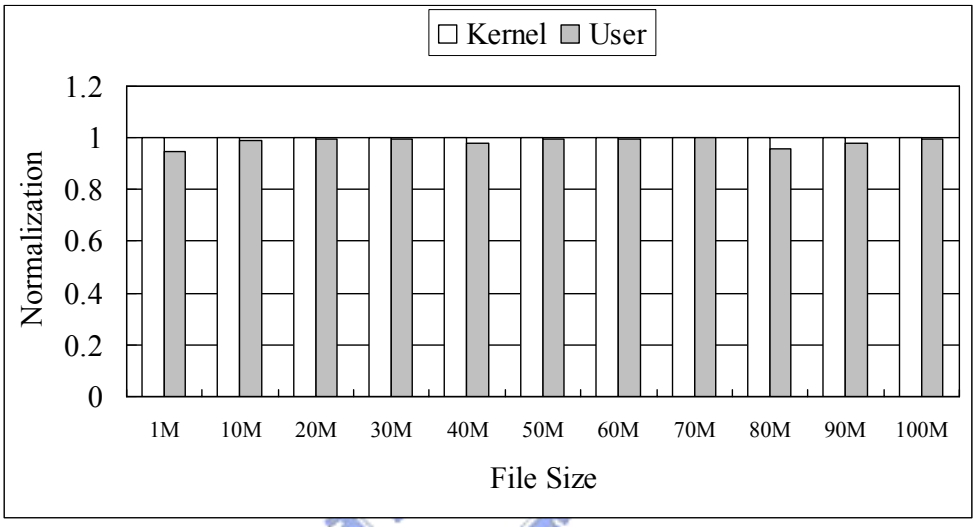


Figure 18：Ftp Send Performance

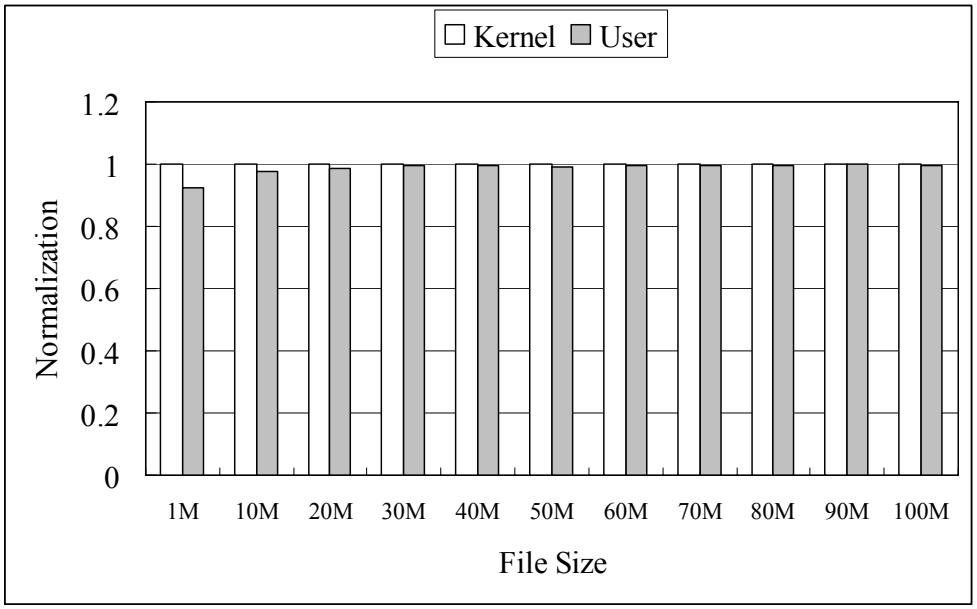


Figure 19：Ftp Receive Performance

4.5 驅動程式容錯

此節我們主要在評估我們要讓使用者模式驅動程式達到容錯能力所需要的負擔。如 3.2.2 節所述，爲了避免驅動程式出錯而污染到 Request Buffer，我們必須要做額外記憶體複製因而會增加額外的 CPU 使用率。我們的實驗環境爲 Server 與 Client1 利用 Netperf 去測量其 Server 個別傳送與接收所需要的 CPU 使用率。實驗結果如 Figure 20 所示，我們發現支援容錯機制大概造成了 0.3%~0.5% 的 CPU 使用率增加。

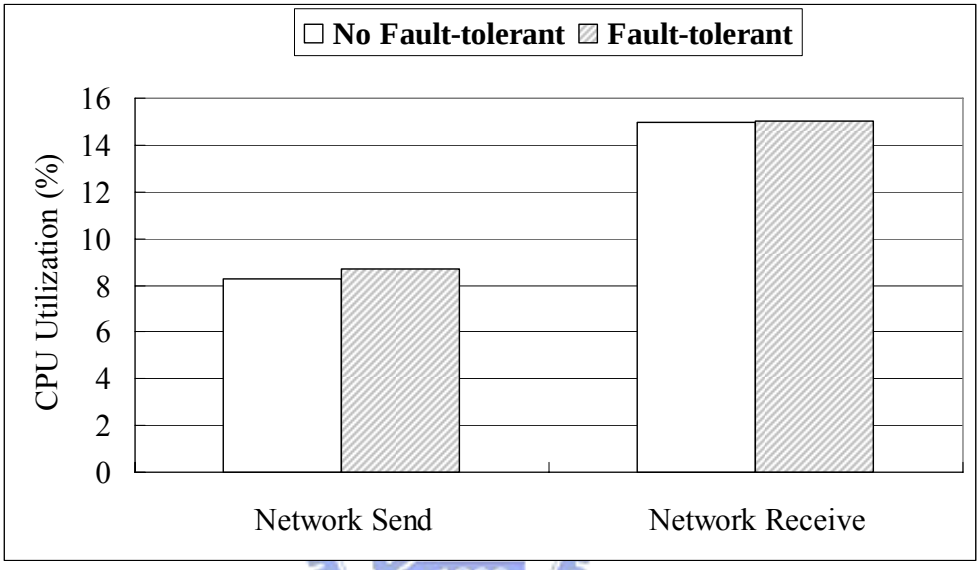


Figure 20：Fault-Tolerant Overhead

接著，我們測量網路卡驅動程式回復所需時間。我們讓 Server 傳送資料到 Client1，在傳送 75000Bytes 後就讓驅動程式發生 divide-by-zero 的錯誤以啓動驅動程式回復流程。結果如 Figure 21 所示，橫軸表示相對時間，縱軸爲 Client1 回覆給 Server 的 ack sequence number，圖中顯示一段時間爲服務停機，結果顯示出服務的停機時間(downtime)大概爲 9ms。

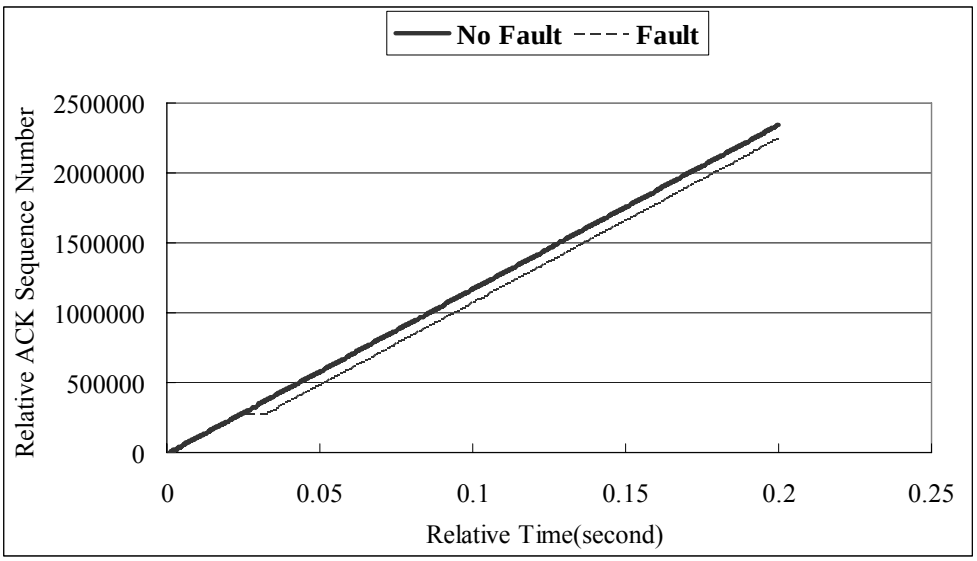


Figure 21：Recovery Time

第五章　結論

在這篇論文我們提出一個使用者模式驅動程式架構，主要目的是要改善作業系統中驅動程式的出錯，以改善其可信賴度。我們論文主要分爲兩部份：第一部份是我們希望能夠直接去重用原本的核心模式驅動程式，讓它能夠直接執行在使用者模式，以減少開發驅動程式的成本。我們目前已完成三個驅動程式的搬移：一個 Tulip Ethernet 網路卡驅動程式，一個 RAMDISK 區塊驅動程式，與一個 PS2 Mouse 字元驅動程式；第二部份是我們希望當驅動程式出錯時，我們能夠掩飾驅動程式的錯誤，去啓動另外一個驅動程式，讓核心與應用程式能夠繼續執行，而不影響其執行，以達到作業系統可用度。由實驗結果顯示我們架構中的使用者模式驅動程式負擔並沒有太大，並且在回復過程中具有可以接收的效能表現

參考文獻

[1] J. Appavoo, M. Auslander, D. DaSilva, D. Edelsohn, O. Krieger, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis, “Utilizing Linux Kernel Components in K42”, available at http://www.research.ibm.com/K42/, Aug. 2002.

[2] S. Arthur, ”Fault Resilient Drivers for Longhorn Server”, Technical Report WinHec 2004 Presentation DW04012, Microsoft Corporation, May 2004.

[3] T. C. Chiueh, G. Venkitachalam, and P. Pradhan, “Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions”, Proceedings of the 17th ACM Symposium on Operating Systems Principles, pp. 140-153, Dec. 1999.

[4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, “An Empirical Study of Operating Systems Errors”, Proceedings of the 18th ACM Symposium on Operating Systems Principles, pp. 73-88, Oct. 2001.

[5] Coverity Inc, “Analysis of the Linux Kernel”, available at http://linuxbugs.coverity.com/linuxbugs.htm, Jan. 2004.

[6] P. Chubb, “Get More Device Drivers out of the Kernel!”, Ottawa Linux Symposium, vol.1, pp. 149-161, Jul. 2004.

[7] C. L. Conway and S. A. Edwards, “NDL: A Domain-specific Language for Device Drivers”, Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems, pp. 30-36, Jun. 2004.

[8] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin, “Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution”, Proceedings of the 2005 USENIX Annual Technical Conference, pp. 31-44, Apr. 2005.

[9] W. Feng, “Making a Case for Efficient Supercomputing”, ACM Queue, vol. 1, no. 7, pp. 54-64, Oct. 2003.

[10] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, “The Flux OSKit: A Substrate for Kernel and Language Research”, Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 38-51, Oct. 1997.

[11] A. Forin, D. Golub, and B. Bershad, “An I/O System for Mach 3.0”, Proceedings of the USENIX Mach Symposium, pp. 163-176, Apr. 1991.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, “Safe

Hardware Access with the Xen Virtual Machine Monitor", Proceedings of the 1th Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure, Oct. 2004.

[13] S. Goel and D. Duchamp, "Linux Device Driver Emulation in Mach", Proceedings of the 1996 USENIX Annual Technical Conference, pp. 65-73, Jan. 1996.

[14] J. Gray and D. P. Siewiorek, "High-availability Computer Systems", Computer, vol. 24, Iss. 9, pp. 39-48, Sep. 1991.

[15] G. C. Hunt, "Creating User-Mode Device Drivers with a Proxy", Proceedings of the 1st USENIX Windows NT Workshop, pp. 55-59, Aug. 1997.

[16] J. Katcher, "Postmark: A New File System Benchmark," Technical Report TR3022, Network Appliance Inc., Oct. 1997.

[17] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Gotz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser, "User-Level Device Drivers: Achieved Performance", Journal of Computer Science and Technology, vol. 20, pp. 654-664, Sep. 2005.

[18] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines", Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 17-30, Dec. 2004.

[19] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay, "Two Years of Experience with a µ-Kernel Based OS", Operating Systems Review, vol. 25, no. 2, pp. 51–62, Apr. 1991.

[20] K. V. Maren, "The Fluke Device Driver Framework", Master's Thesis, University of Utah, available at http://www.cs.utah.edu/flux/papers/vanmaren-thesis-base.html, Dec. 1999.

[21] F. Merillon, L. Reveillere, C. Consel, R. Marlet, and G. Muller, "Devil: An IDL for Hardware Programming", Proceedings of the 4th Symposium on Operating Systems Design and Implementation, pp. 17-30, Oct. 2000.

[22] Microsoft Corporation, "Introduction to the WDF User-Mode Driver Framework", available at http://www.microsoft.com/whdc/driver/wdf/UMDF_Intro.mspx, Apr. 2005.

[23] R. Jones, "Netperf: A Network Performance Benchmark, Version 2.4", available at http://www.netperf.org, 2005.

[24] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why Do Internet Services Fail, and What Can be Done about It?", Proceedings of the 4th USENIX Symposium on

Internet Technologies and Systems, Mar. 2003.

[25] D. S. Ritchie and G. W. Neufeld, “User Level IPC and Device Management in the Raven Kernel”, Proceedings of USENIX Association Symposium on Micro Kernels and Other Kernel Architectures, pp. 111-125, Sep. 1993.

[26] A. Rubini and J. Corbet, “Linux Device Drivers, 3rd Edition”, O’reilly, 2005.

[27] R. Short, “Vice President of Windows Core Technology”, Microsoft Corp. Private Communication, 2003.

[28] M. M. Swift, B. N. Bershad, and H. M. Levy, “Improving the Reliability of Commodity Operating Systems”, Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 207-222, Oct. 2003.

[29] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, “Recovering Device Drivers”, Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation, pp. 1-16, Dec. 2004.

[30] M. M. Swift, S. Martin, H. M. Leyand, and S.J. Eggers, “Nooks: An Architecture for Reliable Device Drivers”, Proceedings of the 10th ACM SIGOPS European Workshop, pp. 101-107, Sep. 2002.

[31] A. Tridgell, “Dbench – Filesystem Benchmark, Version 3”, available at http://samba.org/ftp/tridge/dbench/, 2001.

[32] H. Vemuri., D. Gupta, and R. Moona, “Userdev: A Framework for User Level Device Drivers in Linux”, Proceedings of the 5th NordU/USENIX Conference, Feb. 2003.