視窗作業系統之 Process 虛擬記憶體管理
Management of Virtual Memory for Processes of Windows operating systems

董呈煌	林威男
國立屏東商業技術學院資訊科技系	國立屏東商業技術學院資訊管理研究所
chdong@npic.edu.tw	lin-wn@yahoo.com.tw

摘要

本文系針對視窗作業系統之虛擬記憶體架構進行研究，我們首先探討 Windows 2000 整個虛擬空間佈局，並呈現了不同於參考文獻及技術文件對虛擬記憶體配置的觀點，除了以 Process 來實現虛擬記憶體位址轉換，更以多個 Process 的觀點來解釋 user space 私有、system space 共享的原理，作業系統經由相關 PDE 及 PTE 結構的維護，來完成這些虛擬記憶體空間的管理，我們也透過實驗證明，PTE 在需求分頁記憶體管理機制中所扮演重要的角色。由於多個 Process 在作業系統中運作，就可能發生 Context Switch 的問題，本研究除了提出解決方法，並以實驗來驗證該方法確實能夠解決由 Context Switch 所產生的問題。另外，我們也準確的完成了不同 Process 之間，user space 虛擬位址共享及 system space 虛擬位址私有的實驗，實驗結果顯示了本研究對視窗作業系統虛擬記憶體的操控能力。

關鍵詞：視窗作業系統 2000，虛擬記憶體。

Abstract

This paper provides an in-depth survey of the memory management of Windows operation system. First, we explore the virtual memory in Windows 2000 in detail, and present the viewpoint different from other official documents. We not only realize virtual address can be translate into physical address by process, but use perspective of processes to explain the principles that user space is private and that system space is public. Windows maintain PDE, PTE and PF to govern the virtual memory. We also prove that PTE plays a critical role at mechanism of demand paging memory through experiment. Windows needs to switch the memory context to a certain process for accessing its own address space. Therefore, we propose a method to solve the problems resulting from context switch. Experiments reveal the method could work efficiently than others. In addition, we precisely finish the experiment that two processes can share their private memory space and that the public memory space belong to one of them. These experiments demonstrate that we have more capability to control and operate the virtual memory in Windows Operation System.

Keywords: Windows 2000, Virtual Memory.

1. 緒論

作業系統的演進，由早期單人單工的簡單運作模式，一路發展到 Multiprocessing、Multithreading 等複雜架構。在這演進的過程中，記憶體不足一直都是作業系統必須面對的問題，早期作業系統限制被執行的程式段，必須小於實際的記憶體空間，當時的程式設計人員受此限制，必須撰寫精簡的程式碼。但隨著作業系統的演進，相當耗用記憶體的圖形介面(GUI)，由於操作簡單、介面友善，成為作業系統發展的趨勢，為了讓新一代的作業系統免除實體記憶體不足的問題，因此提出了虛擬記憶體技術[6]。

虛擬記憶體技術利用需求分頁的概念，將實體記憶體分割成許多大小相同的 Page，在實體記憶體不足的情況下，作業系統利用了 swap 的技巧，讓實體記憶體的 Page 被充分利用。另外，作業系統中的每一個 Process 都有一份自己的虛擬位址空間，因此，當 CPU 控制權由一個 Process 換成了另一個 Process 時，虛擬位址空間與實體位址空間的對應就會發生變化，這轉變就稱為 Context Switch[5,7]，以應用程式的角度來看，由於僅只是向作業系統提出記憶體使用的要求，在作業系統合宜的資源調度策略下，Context Switch 都能在正確的時刻被執行。但實際上，一旦進入了作業系統的核心，Context Switch 就不像以上所描述的這麼單純了[7]。以下我們配合圖 1 所標示的數字順序，說明在 Context Switch 時所可能會發生的問題：

(1) 當應用程式與作業系統中的驅動程式建立了連結關係，Context Switch 仍依原有的方式運作，由於驅動程式可以在任意的 Process 中執行，因此當 Context 切換成虛擬記憶體空間 2 時，而驅動程式正好完成相關 I/O 工作，並將資料“12345678”寫回虛擬記憶體，於是，這些資料寫入了虛擬記憶體空間 2。

(2) 接下來，驅動程式通知原本的應用程式來讀取資料，因此，作業系統將 Context 切換回應用程式原本所使用的虛擬記憶體空間 1。

(3) 應用程式讀取驅動程式所指定的虛擬記憶體位址，但資料卻是寫入了虛擬記憶體空間 2，所以應用程式仍讀取不到正確的資料。

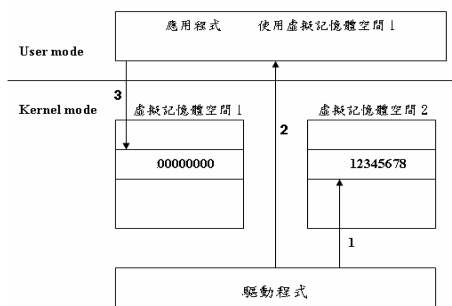


圖 1.Context Switch

相關研究[1]，及一些驅動程式研發論壇的 developer[11]，也提出無法將資料寫入正確虛擬位址空間的問題。由於現今的作業系統都採用虛擬記憶體管理架構，對撰寫作業系統核心的 developer 來說，虛擬記憶體的使用，伴隨而來的將是 Context Switch 的操作問題，故是否能經由對虛擬記憶體管理架構的深入了解，尋求一個不影響作業系統效能且能解決上述問題的方法，進而主動操控作業系統虛擬記憶體的使用，便是本研究的主要動機。

本研究主要是探討有關視窗作業系統虛擬記憶體管理的使用問題，為了擁有較高的執行權限，本研究將採用 NTDDK[10]中的驅動程式 Packet.sys，並以 Windows 2000 為主要的實驗平台。在以下的第 2 節中，本文將深入了解視窗作業系統的虛擬記憶體架構，並對虛擬記憶體的運作及系統中 Process 的結構進行觀察，以提出不同於文獻對虛擬記憶體使用的觀點，第 3 節除了以多個 Process 的角度探討虛擬記憶體共享機制，也對需求分頁管理及 Page Fault 進行研究，第 4 節分別描述了三個現行 Context Switch 操控問題的解決方法及限制，針對這些限制，本研究將提出另一個方法，並在第 5 節進行相關實驗，最後，在第 6 節提出結論及後續之研究方向。

2. Windows 虛擬記憶體運作之剖析

長久以來，所有相關文獻[4,5,7,8]對視窗作業系統虛擬位址轉換的描述，皆依循微軟所發佈的文件，以虛擬位址空間的觀點來解釋虛擬位址轉換的過程，但這樣的解釋並無法清楚說明在虛擬記憶體空間中，某些位址由不同 Process 共享、私有的情況，以及 Context Switch 的運作原理，我們將提出以 Process 及實體記憶體的觀點，來詮釋虛擬位址的轉換過程。本節分為三個小節，分別探討虛擬記憶體位址空間之佈局、Process 結構、及虛擬位址轉換的方法。

2.1 虛擬位址空間之佈局

Windows NT/2000 及其後的版本，皆利用 32 bits 的指標來存取資料，因此，定址能力高達 4GB [5]。基於系統安全性的考量，視窗作業系統將這 4GB 的位址空間分為兩個主要部分(見圖 2)，分別是 user space (虛擬位址由 00000000 至 7FFFFFFF)及 system space (也稱 kernel space，虛擬位址由 80000000 到 FFFFFFFF)，各別佔用 2GB 的位址空間，user space 所儲存的是各 Process 的程式碼及私有的資料，system space 則存放作業系統的核心及驅動程式。這樣的分割是為了讓應用程式限制

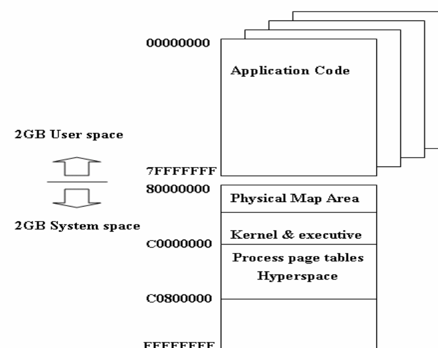


圖 2.視窗作業系統虛擬空間配置

在自己 2GB 的位址空間中活動，而無法存取到作業系統的核心，因為作業系統的核心一旦可以任意由應用程式修改的話，必然會破壞作業系統原有運作，導致不可預期的後果。相反的，作業系統的核心及驅動程式就有比較高的權限，有能力存取整個虛擬位址空間。值得一提的是，實體記憶體將由虛擬位址 80000000 之處，產生一段連續的對應，也就是圖 2 中間標示為 Physical Map Area 的這塊區域。另外，圖 2 上半部顯示了多個 user space，這是因為系統在任何時刻都可能執行多個應用程式而產生數個 Process，為了 Process 之間不被彼此干擾，所以每個 Process 都有一份自己的 user space，相對的 system space 只有一份，由所有 Process 共享[5,7]。

2.2 Process 結構

Process 可以視為在作業系統中用來描述特定應用程式的資料結構，由於每個 Process 都各有一份自己的 user space，必然會在 Process 的結構中放存相關的資訊，本研究利用文獻[5]及 Debug 工具 WinDbg[9]導出 Process 的資料結構(見圖 3)，發現兩個重要欄位，分別是 Directory Table Base 及 NextProcess，個別位在 Process 起始位址 18 及 a0 之處，而 Directory Table Base 欄位所指向的實體位址，就是用來做為虛擬位址轉換的起點，稱為 Page Directory Entry，本文將在下一節說明虛擬位址轉換的流程。NextProcess 欄位則指向了下一個 Process，因此所有的 Process 將形成環狀串列的結構，另外，起始位址位移 1fc 之處，存放了 Process 的名稱，可以來辨識或指定特定的 Process。

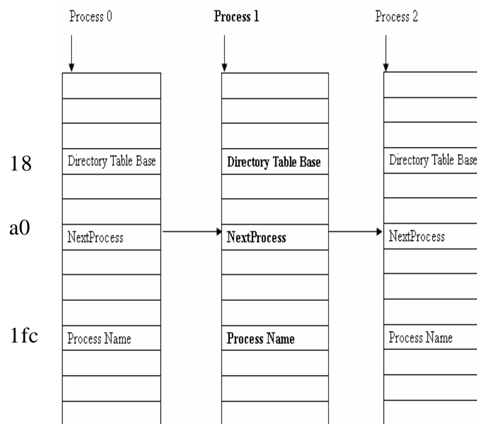


圖 3.視窗作業系統 Process 結構

2.3 Process 虛擬位址轉換

視窗作業系統將虛擬位址經由一連串的步驟轉換為相對應的實體位址。實體位址空間以 4KB 為分割單位，稱之為 Page Frame(PF)，虛擬位址轉換的目地就是要去找到正確的 PF，才能取存實體記憶體上的資料，要執行虛擬位址轉換之前，必須將虛擬位址依序以 10、10、12 個位元為單位，分割成不同轉換階段所需要的索引值，接著利用這三份索引值，由 Page Directory Entry (PDE)開始，去尋找正確的 Page Table Entry (PTE)、PF 及 Page 的位移處(byte index)。接下來，本文將以虛擬位址"00A00020"配合圖 4 所標示的 A、B、C，解說虛擬位址轉換的流程。

A. 首先將 00A00020 轉換成二進位的表示方式，並利用先前描述的方法對這個虛擬位址做適當的分割，得到 2、800、20 三個分別代表 PDE、PTE 及 byte index 的索引值，隨後由 Process 獲得 PDE 的位址(見圖 4 之 A)，再將這個位址加上 PDE索引值 "2"，就取得了指向 PTE 位址的值。

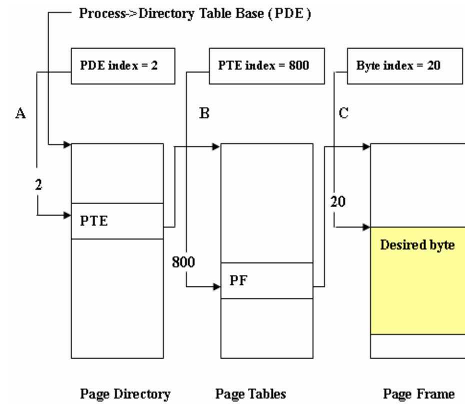


圖 4.虛擬位址轉換流程

B. 獲得 PTE 的位址後，將這個位址加上 PTE 索引值"800"(見圖 4 之 B)，便能取得真正存放資料的 PF。

C. 因為 PF 的位址空間為 4096 bytes，我們利用 byte index "20"(見圖 4 之 C)，來指定存取這個 PF 第 20 個 byte 的資料，至此，便完成了本次虛擬位址轉換的流程。

3. 虛擬記憶體管理

本節將分為三個小節，分別以 Process 的觀點來探討虛擬記憶體的管理，除了深入了解 Process 的 system space 虛擬記憶體共享機制，並探討 Page Fault 及需求分頁記憶體管理機制。

3.1 Process 虛擬空間切換

在多個 Process 運行的情況下，除了虛擬位址轉換的問題之外，作業系統還必須在適當的時機，選擇一份正確 Process 的 PDE，做為虛擬位址轉換的起點，圖 5 呈現了多個 Process 的情況下，虛擬、實體記憶體對應的重要概念，可以見到圖 5 中，3 份 Process 分別佔用了實體記憶體不同的區段，我們利用前一小節所描述的位址轉換觀念，可以從圖 5 右半部任一 Process 的 Directory Page Base 開始，依循著箭頭的方向，經過 PDE、PTE 而找到存放資料的 PF，圖 5 清楚呈現了不同 Process 的 PF，並不佔用相同的實體記憶體區段，另外，這些 PF 被個別地對應成不同 Process 4GB 的虛擬記憶體位址空間。

依據微軟及相關文件[5,7]的說法，取得 CPU 控制權的 Process，其 PDE 將存放在 system space c0300000 的位址上(見圖 2)，作業系統採用 c0300000 為 PDE的做法，便可以營造出雖然存在數個 user space，但任何時刻

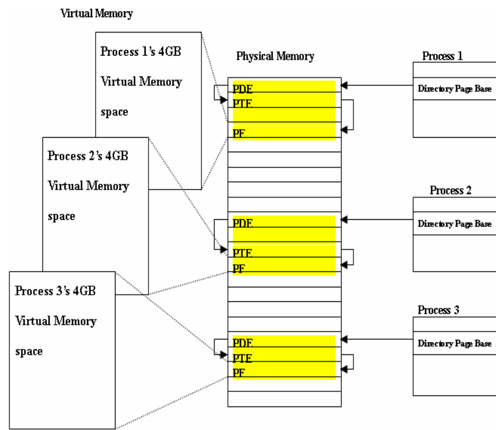


圖 5.不同 Process 實體位址 PDE、PTE、PF 的配置與虛擬位址空間之對應關係。

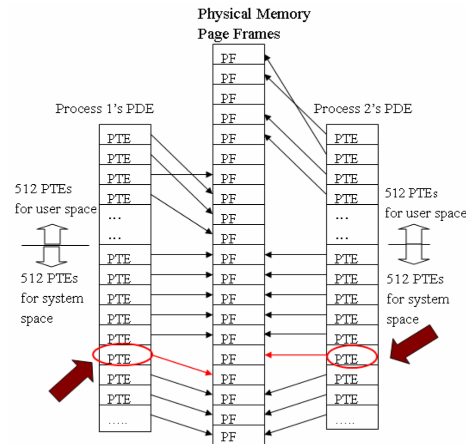


圖 6.不同 Process PTE、PF 之對應方式

只會顯示一份 user space 的假象。但本文利用 Process 的結構，直接在實體記憶體中取得所有 Process 的 PDE、PTE、PF(見圖 5)，而每一份虛擬位址空間都以 80000000 的位址，做為實體記憶體對應的起點，因此我們可以在任何時刻存取不同 Process 的虛擬位址空間，不再受限於 c0300000 虛擬位址只能顯示一份 Process 的虛擬位址空間，對本研究來說，已克服不同 Process 之間，虛擬空間切換的問題。另外，由於切換的虛擬位址空間範圍為 4GB，本文認為 system space 實際上也應隨著 Context Switch 而產生變化，但這與相關文件[2,5,7,9]的說法有著明顯的差別，針對這個問題，本研究將對不同 Process 共享同一份 system space 的概念及作法，在下一節提出說明。

3.2 Process 系統虛擬記憶體共享機制

文件[2,5,7,9]指出，system space 由系統中所有的 Process 共用，但這個說法與前文所描述"取得 CPU 控制權的 Process，其 PDE 將存放在 system space c0300000 位址上"的說法相互矛盾，因為 c0300000 的位址屬於 system space(見圖 2)，但卻又隨 Process 而變動，這些文件的描述並不能讓人明白 system space 共享的具體做法。本研究對視窗作業系統虛擬記憶體架構進行仔細地觀察後發現，system space 的共享，實際上與 Process PDE 中 PTE 的配置有著密切的關係，以下將以圖 6 來說明 system space 共享的作法。PDE 實際包含了 1024 個 PTE，其中前半部 512 個 PTE 描述了 2GB 的 user space，可以看到圖 6 有兩個 Process 的 PDE，它們前半部 512 個 PTE 對應不同的 PF，但是後半部用來描述 system space 的 PTE，除了指定 c0000000 至 c0800000 這段虛擬位址的 PTE 之外(即圖 6 的圈選處)，皆使用相同 PTE 與 PF 的對應方式，讓不同的 Process 共享 system space。由於這項觀察的結果顯示，Process 之間並不共用 system space 的 PTE，這與我們在第 3.1 節所提出的推論相符，因此，本研究針對這項觀察結果進行了以下的實驗。

實驗在 Packet.sys 中，配置一塊 system space 的虛擬記憶體，由於共享的原因，所有 Process 都會產生相對應的 PDE、PTE 及 PF 來描述這塊記憶體，本研究利用第 2 節所提出的方式，在取得系統中特定 Process 的結構之後，找出這個 Process 用來對應 Packet.sys 所配置虛擬記憶體的 PTE，並將其值修改為 0，實驗結果顯示，該 PTE 所描述的 system space 不再共享，我們以圖 7 說明實驗結果。圖 7 之 1 指向的"e23c6000"是 Packet.sys 配置的 system space address，圖 7 之 2 分別指向不同 Process 的 PDE，計算出"e23c6000"的 PTE index 為 e20，於是本文利用 WinDbg 列印了兩個 Process 存放對應於"e23c6000"虛擬位址的 PTE，圖 7 之 3 圈選之處則清楚的顯示了兩

```
linaddr = e23c6000                                              1
:dd 30000+e20
00030E20 041EE163 082B5163 00000000 00000000 00000000
00030F30 00000000 00000000 00000000 00000000 00000000
00030F40 00000000 00000000 00000000 00000000 00000000
00030F50 00000000 00000000 00000000 00000000 00000000
2 00030F60 00000000 00000000 00000000 00000000 00000000
00030F70 00000000 00000000 00000000 00000000 00000000
00030F80 00000000 00000000 00000000 00000000 00000000
00030F90 00000000 00000000 00000000 00000000 00000000
:dd 936000+e20                        3
00936E20 00000000 0[illegible]2B5163 00000000 00000000 00000000
00936E30 00000000 00000000 00000000 00000000 00000000
00936E40 00000000 00000000 00000000 00000000 00000000
00936E50 00000000 00000000 00000000 00000000 00000000
00936E60 00000000 00000000 00000000 00000000 00000000
00936E70 00000000 00000000 00000000 00000000 00000000
00936E80 00000000 00000000 00000000 00000000 00000000
00936E90 00000000 00000000 00000000 00000000 00000000
```

圖 7.實驗結果

個 Process 指向"e23c6000"虛擬位址的 PTE 已經不同，而圖 7 下半部所顯示的就是之前遭 Packet.sys 修改的 PDE，這個被修改 PTE 的 Process 將無法使用這塊共享的 system space 虛

擬記憶體，因為在虛擬位址轉換的過程中，作業系統將找不到對應的 PTE，亦無法存取其對應之 PF。實驗成功地突破了 system space 的共享機制，讓 system space 也能讓特定 Process 獨自擁有。

3.3 需求分頁記憶體管理

視窗作業系統使用了需求分頁記憶體管理，主要是利用 swap 的技巧來處理實體記憶體不足的情況，這表示 PTE 必須要保留相關 swap 的資訊，才能讓作業系統確切地掌控 PF 的使用情形，另外，因為資料特性的關係，PF 的取存屬性也有不同的變化，例如，若存放了要執行的程式碼，PF 就有唯讀的屬性，這樣的資訊也記錄在 PTE 中，對視窗作業系統來說，PTE 就成了需求分頁記憶體管理的重要介面，圖 8 為一個典型的 PTE 格式[7]，圖中三個屬性欄位 Protection、Pagefile 及 State，各使用了 5、4、3 個 bits 來描述這個 PTE 所對應 PF 的相關屬性，為了驗證 PTE 在需求分頁管理機制中的重要性，本文將實際對 PTE 的 State 欄位進行修改，並觀察結果。

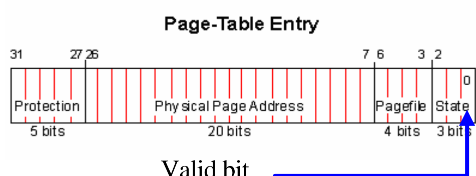


圖 8.PTE 格式

Valid 屬性位於 PTE 中 State 欄位的最後一個 bit(見圖 8)，當這個 bit 被設定為 1 時，代表所指向的 PF 是合法的，反之則代表一個不合法或不存在的 PF，本文將修改此 bit 來觀察作業系統在存取該 PTE 所指向的 PF 時，所產生的反應。我們利用 TESTCPP.exe 與 Packet.sys 這一組程式來進行相關實驗。

TESTCPP.exe 為應用程式，它會發出訊息要求 Packet.sys 將資料寫入 TESTCPP.exe 所指定的位址中。實驗步驟是利用第 2.2 節的虛擬位址轉換方式，在 Packet.sys 找出 TESTCPP.exe 所指定虛擬位址的 PTE 後，修改 Valid bit，讓作業系統誤認為這個 PTE 指向了一個非法的 PF，隨後 Packet.sys 將資料寫入所指定的虛擬位址中。實驗結果反應了修改 Valid bit 對作業系統所產生的影響，圖 9 是由 WinDbg 所截取的訊息畫面，其中所圈選出的兩個數值都是同一份 PTE，上面那一份是 PTE 原本的數值，下面那一份是經 Packet.sys 修改過後的數值，圖 9 顯示了修改過 PTE 的 Valid bit 後，再對 PTE 所指向的 PF 執行存取動作，將造成 Page Fault(即圖 9 中 “Break due to Page Fault(0Eh)” 的警告訊息)。另外，作業系統也因為這個原因而 crash ，因此，由此次實驗得知 PTE 在需求分頁記憶體管理中，確實扮演了一個重要的角色。

```
TESTCPP.exe
pde offset = 0
pte offset = 4bc
PDE        = 976b000
*PTE       = 9779067
*PTE       = 9779066
Break due to Page Fault(0Eh)

eax=0000f002 ebx=c2920074 ecx=0
eip=c001ff87 esp=cfc71e50 ebp=cf
cs=0028 ss=0030 ds=0030 es=0030
```

圖 9.非法分頁失敗訊息

4. Process Context Switch

本節將分別描述現行三個解決 Context Switch 問題的方法，特別要說明的是，在此，本文仍以相關文件對虛擬記憶體原有的描述來說明這些方法及原理，以突顯它們原本的限制。最後，本研究將以先前對虛擬記憶體運作的觀察，提出另一個解決辦法。

4.1 I/O Request Packet

IRP 是視窗作業系統中，應用程式與驅動程式用來相互傳送訊息的資料結構[2,7,4]，由於使用虛擬記憶體的關係，應用程式將使用私有的 user space 來存放資料，因此，應用程式與驅動程式若要正確無誤的溝通，驅動程式就必須讓 IRP 傳回正確的虛擬位址空間。在驅動程式中使用 IoMarkIrpPending 函式可以決解這樣的問題。驅動程式在使用 IRP 之前，利用 IoMarkIrpPending 函式先將 IRP 儲存起來，而當驅動程式完成指定的 I/O 工作後，便將所要傳送的訊息或資料寫入 IRP 結構中的特定欄位，並回傳 IRP 給應用程式，此時作業系統會先保留 IRP，直到虛擬位址空間切換回原本應用程式的 Context 時，才將 IRP 回傳。這方法可以輕易地完成應用程式與驅動程式的溝通，但 IRP 的資料結構並不具太大的延伸性，使得這個方法所能傳送的資料量非常少，只適合於訊息及微量資料的傳遞。

4.2 系統虛擬記憶體

由圖 2 可以得知，system space 2GB 的虛擬記憶體並不會隨 Process 而改變，因此這個方式是利用 MmGetSystemAddressForMdl()函式，由 system space 配置一塊虛擬記憶體，而這塊虛擬記憶體將與特定應用程式所使用的虛擬記憶體共用同一份 PF，由於 PF 共享的原因，驅動程式無須考慮 Context Switch 的問

題，也能讓應用程式讀到正確的資料。但這個方法仍有以下兩個缺陷。

(1) 因為 system space 將由所有的 Process 共享，若有一個以上的應用程式使用這個方法，Process 之間就可能產生互相干擾的問題。
(2) 要求配置的系統虛擬記憶體空間有容量上限，在本研究的測試下，所得到的最大的配置量約為 120 Kbytes，這個限制讓程式無法配置較大且連續的記憶體區塊。

4.3 KeAttachProcess

KeAttachProcess()及 KeDetachProcess()是微軟所保留的一組函式[13,14]，可以用來解決 Context Switch 的操作問題，主要的使用步驟如下:

(1) 利用 IoGetCurrentProcess()函式取出程式想要切換的 Process 結構，保留這個 Process 結構的位址。
(2) 將先前所得到的 Process 結構為參數，呼叫 KeAttachProcess()函式，呼叫之後，虛擬位址空間就切換為所傳入參數 Process 的 Context，此時便可以與相關的應用程式溝通，傳輸資料。
(3) 當相關操作結束時，呼叫 KeDetachProcess()函式，就切換回原本的 Context。

這組函數最主要的限制是，僅適用於特定版本的視窗作業系統，且無法在作業系統發生中斷時使用。

4.4 Context Switch 之修正做法

本文在先前對虛擬記憶體所做的觀察及相關實驗中，已跳脫了原本虛擬記憶體的使用觀念，在第 2 節，由於可以得到所有 Process 虛擬記憶體的配置方式，因此已不存 Context Switch 的問題，我們所要面對的只是如何簡單地操作虛擬記憶體。雖然已經可以在任何 Context 中去取存任意 Process 的虛擬位址，但仍然必須經過一連串的轉換步驟才能得到實體記憶體位址，過程仍嫌複雜，經由 Intel 技術手冊查證[12]，視窗作業系統採用 CR3 暫存器的內容為 PDE，因此，本研究的作法是，選定某個 Process 的 PDE，存入 CR3 暫存器中，就可以對該 Process 的虛擬位址進行存取的操作。除了不受虛擬記憶體空間切換的限制外，也不必透過煩雜的轉換程序才能使用，亦不受中斷發生時之限制，只須確定所要存取的虛擬記憶體已有實體記憶體對應即可。我們將在下一節，利用本文所提出的方法，進行相關的實驗來驗證這個方法的可行性及效度。

5. 實驗

本節將利用第 4.4 節所提出的方法，分別進行 Context 操控實驗及 Process user space 共享實驗，其中所使用的程式為 Packet.sys、TESTCPP.exe、linwn.exe 及 linwnexp2.exe。

5.1 Context 操控

本節實驗的主要目地是測試本文在 4.4 節所提出的方法有選擇、控制不同 Process Context 的能力，我們利用 linwn.exe 及 Packet.sys 兩支程式進行實驗，主要步驟如下:

(1) 在應用程式 linwn.exe 中，宣告一個整數變數，變數值預設為 0，隨後，程式就依固定的時間間隔，不斷重覆列印這個變數。
(2) Packet.sys 啟動之後，利用驅動程式較高的權限，取得 linwn.exe PDE的內容後，填入 CR3 暫存器中，並保留原本 CR3 暫存器的內容。
(3) 由於 linwn.exe 所宣告的變數都固定在 12ff7c 的虛擬位址上，因此，Packet.sys 便將“11111111”寫入虛擬位址 12ff7c，隨後，再將 CR3 暫存器恢復成原來的內容。

實驗執行以上的步驟後發現，linwn.exe 所宣告的變數，果然在 Packet.sys 啟動之後被修改，接著，我們提升了實驗難度，除了依循原有的步驟之外，並讓 linwn.exe 同時開啟 3 份，且修改 Packet.sys，讓它可以針對不同的 linwn.exe，將變數修改為不同的值，圖 10 為實驗結果，可以看到圖中三份 linwn.exe，在列印了 3、4 次後，變數值就遭 Packet.sys 分別修改為“11111111”、“55555555”及“ffffffff”，實驗結果顯示了本文所提出的方法，確實有能力選擇、控制 Process Context，進而修改特定

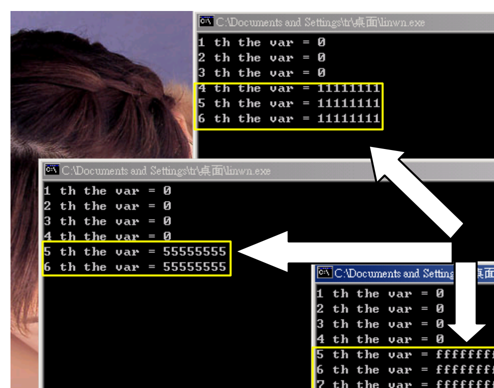
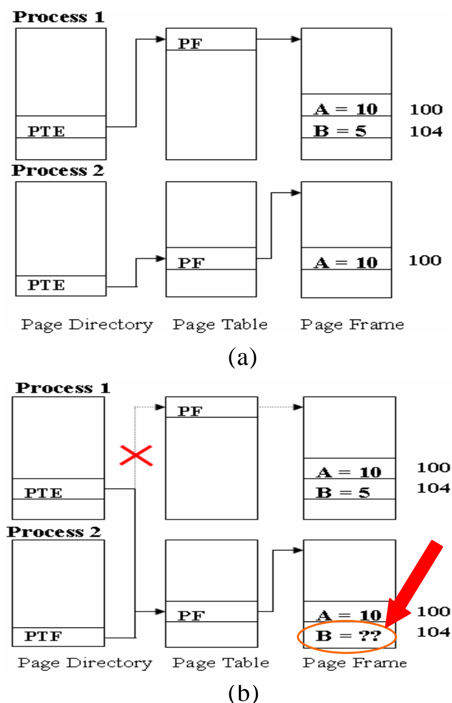


圖 10.Context 操控實驗

Process 虛擬位址的內容。

5.2 Process user space 共享

根據文件[3]指出，視窗作業系統 user space 的共享是透過 PTE 達成的，主要的作法是讓不同 Process 的 PTE 指向同一份 PF，但這有可能造成相關虛擬位址的 side effect，我們利用圖 11 來解釋發生 side effect 的情況。圖 11 包含了兩個部份，分別呈現了不同 Process 之間，區域變數共享前、後的情形。首先在圖 11 之(a)中，可以見到 Process1 及 Process2 利用自己的 PTE 轉出存放變數的 PF，而 Process1 所宣告的兩個區域變數(A 及 B)，連續地被配置在 100 及 104 的虛擬位址上，初始值分別為 10 與 5，而 Process2 也在自己 PF 100 的虛擬位址上宣告了一個區域變數 A，初始值為 10，



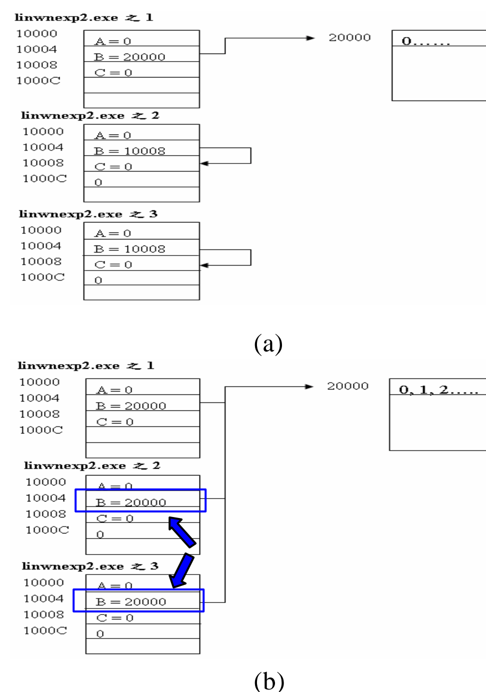
(a)

(b)

圖 11.修改 PTE之 side effect

為了要讓兩個 Process 共享同變數 A，所以我們修改了 Process1 的 PTE，讓它與 Process2 的 PTE 一起指向相同的 PF(見圖 11 之(b))，加上變數 A 在兩個 Process 的位址皆為 100，因此成功地達成了 Process1 及 Process2 共享同一份區域變數 A，但仔細觀察原本在 Process1 中宣告的區域變數 B，仍存放在原先的 PF 中，因此，在共享機制達成後，若 Process1 要存取區域變數 B 時，將存取到另一份 PF 在 104 位址上的內容，也就是本文在圖 11 之(b) 之圈選處，由於只為了共享某些變數結構而修改 PTE，反而造成相關實體記憶體對應的變化，這樣的現象就稱為 side effect。

分析過 side effect 發生的原因之後，本節將示範 user space 共享實驗，實驗將不使用微軟所提供相關 user space 的共享函式，此外，我們將配置三個位址連續的變數，準確地讓其中一個變數與其它的 Process 所共享，並避免對其它的私有的區域變數造成 side effect。實驗利用了 linwnexp2.exe 及 Packet.sys 這一組程式，以下將配合圖 12 來說明實驗步驟。圖 12 分為兩個部份，首先在圖 12 之(a)中，我們將 linwnexp2.exe 開啟三份，而這三份都是獨立的 Process，因此，都各擁有獨立的 user space，程式配置了三個整數變數，並以連續的位址存放，其中，位於變數 A、C 之間的變數 B，在程式中被宣告為指標變數，而三份 linwnexp2.exe 主要的差異在於，我們針對圖 12 中 linwnexp2.exe 之一，另外配置一個虛擬位址為 20000 的變數 P，除了讓指標變數 B 指向變數 P 外，程式將不斷增加變數 P 的內容，以方便後續實驗的觀察。如同圖 12 之(a)所描繪的情形一樣，對於 linwnexp2.exe 之二及 linwnexp2.exe 之三，程式並不另外配置記憶體，只是讓變數 B 指向變數 C。實驗的目地，就是要讓 linwnexp2.exe 之一所建立的變數 P，讓其它兩份 Process 共享。



(a)

(b)

圖 12.單一區域變數共享實驗概念

linwnexp2.exe 之二及 linwnexp2.exe 之三並未宣告變數 P，因此，並沒有對應這個變數位址的 PTE 及 PF，本文使用 Packet.sys 找出 linwnexp2.exe 之一建立變數 P 時產生的 PTE、PF，並新增到 linwnexp2.exe 之二及 linwnexp2.exe 之三的相關結構中，另外，Packet.sys 也將 linwnexp2.exe 之二及 linwnexp2.exe 之三變數 B 的內容修改為 20000，也就是變數 P 的位址，如圖 12 之(b)

圈選處所示，linwnexp2.exe 之二及 linwnexp2.exe 之三的變數 B 都指向變數 P，對 linwnexp2.exe 之二及 linwnexp2.exe 之三來說，雖然共享了變數 P，但變數 A、C 仍使用原本的 PTE 及 PF，故不產生 side effect 的問題。實驗讓 linwnexp2.exe 不斷地列印變數 A、B、C 的內容，以呈現實驗過程中的變化，圖 13 為實驗結果，圖中最下方的應用程式就是實驗說明中的 linwnexp2.exe 之一，其它兩份是 linwnexp2.exe 之二及 linwnexp2.exe 之三，可以看到圖 13 中，三份 linwnexp2.exe 原本互不干擾地列印自己的區域變數，在 linwnexp2.exe 之一列印第五次時，實驗讓 Packet.sys 開始修改 linwnexp2.exe 之二及 linwnexp2.exe 之三相關 PTE、PF 等資料結構，圖 13 圈選處顯示了第六次列印之後，linwnexp2.exe 之二及 linwnexp2.exe 之三的變數 B，已隨著 linwnexp2.exe 之一的變數 B 一起遞增，這表示 linwnexp2.exe 之一的變數 P 已被共享，另外 linwnexp2.exe 之二及 linwnexp2.exe 之三的變數 A 及變數 C，其內容仍舊為 0，side effect 並未發生，因此，本研究成功地完成了此次實驗。

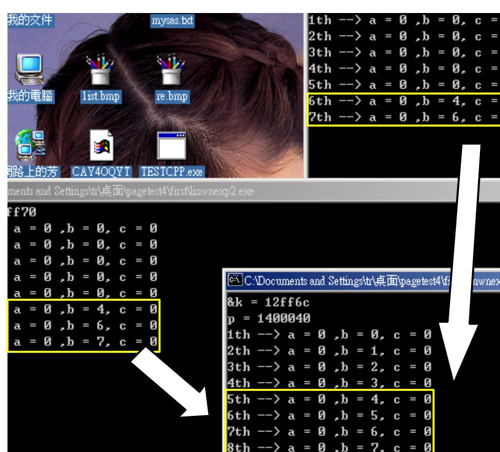


圖 13.區數變數共享實驗結果

6. 結論

本文探討了視窗作業系統的虛擬記憶體管理架構，主要是因為大部份文件對於視窗作業系統虛擬記憶體管理等相關問題的描述並不清楚，像是 system space 由所有的 Process 共享，而任何時刻 user space 都只顯示一份的類似說法，一旦依循這種思考模式之後，便落入了單一虛擬位址空間的邏輯陷阱，當 developer 僅能以單一 Process 的觀點來看待視窗作業系統的內部運作，許多虛擬記憶體管理及操作的問題就非得依循微軟所設計的方式來處理，除此之外，相關文件無法說服本研究對虛擬空間定址的疑問，32 bits 的虛擬位址能定址 4 GB 的記憶體空間，但其中卻有 2 GB 共用，而共用的虛擬位址空間，有些部份卻又隨 Process 變動，所以本文不採用這些文獻的說法，而以 Process 及實體記憶體的觀點來詮釋虛擬位址的配置及轉換。

本文除了由實驗得知 PTE 在需求分頁記憶體管理機制的重要性，還提出了一個解決相關 Context Switch 操作問題的方法，實驗結果顯示了該方法對虛擬記憶體控制的能力確實產生了效用。另外，本文也準確的完成 Process 之間，user space 虛擬位址共享及 system space 虛擬位址私有的實驗，這都仰賴對虛擬記憶體運作的仔細觀察。

7. 參考文獻

[1] 董呈煌(2003，6 月)。"視窗作業系統間網路驅動程式移植可行性之研究"。國立屏東商業技術學院主辦，*2003 資訊技術應用與發展研討會*，屏東市。

[2] Chris Cant(2000), WDM 驅動程式設計手冊(Writing Windows WDM Device rivers)(葛湘達譯)。台北市:碁峰資訊。

[3] Randy Kath(1992,December 21), Microsoft MSDN Library.Message posted to http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dngenlibhtml/-msdn_ntvmm.asp

[4] Walter Oney(1999), Programming the Microsoft Windows Driver Model. Washington : Microsoft Press.

[5] David A. Solomon(1998),Inside Windows NT. Washington : Microsoft Press.

[6] Andrew S. Tanenbalim and Tlbert S. Woodhull(1998),作業系統：設計與實作（OPERATING SYSTEMS Design and Implementation）(蔡明志譯)。台北市：東華書局。

[7] Peter G. Viscarola and W. Anthony Mason (2001), WINDOWS NT 驅動程式設計手冊(WINDOWS NT Device Driver Development)(葛湘達譯)。台北市:碁峰資訊。

[8] Microsoft(2000), Windows 2000 Driver Design Guide. Washington : Microsoft Press.

[9] Microsoft.(2002), WinDbg 6.1 [Computer Program].Washington : Microsoft Press.

[10] Microsoft.(2000), Windows NT Driver Development Kits[Computer Program]. Microsoft. Washington.

[11] http://www.driverdevelop.com

[12] http://www.Intel.com

[13] http://www.microsoft.com/DDK/IFSkit/ntupto2k.asp

[14] http://webcrazy.yeah.net.