深入淺出話「核心」

作者：范維如、廖素貞

前言

「核心」，英文稱做 Kernel，是一個在各個場合都常見到的字眼，而套在不同的領域中又各代表不同的意義。在軟體的世界中，「核心」幾乎與「作業系統（Operating System）」劃上等號，它代表了一個軟體作業系統一切能力的來源，對上層程式設計者而言，「核心」提供了應用程式的可執行環境，對下層則掌控硬體的邏輯行爲，並依應用程式的需要對週邊執行輸出入動作。基本上「核心」是軟體系統的靈魂，一個好的「核心」可以充分提供軟體執行時所需要的各種資源（resource），同時也能將硬體的效能（performance）發揮到淋漓盡致，因此軟體設計者在開發應用程式時，慎選一個優異的「核心」是不可或缺最重要的第一件工作。

隨著軟體技術的不斷演變，「核心」作業系統的技術也從早期的「單工」進步到「多工」，時到今日一提到「核心」，絕大多數的人都會與「多工核心（Multitasking Kernel）」聯想到一起。在 Windows 作業環境普及化之後，具有多工能力的作業系統核心早已成爲最基本的功能，而相對的程式設計者若無多工化程式的觀念，是無法開發真正具有價競爭力的好的應用程式。

筆者因爲工作的機會，在 MS-DOS 大行其道的時代即開始接觸多工核心應用程式開發的訓練，曾經使用與自行設計過多個核心作業環境或系統，因此對各式的「核心」並不陌生，也希望經由這篇文章，在儘可能不使用艱澀的軟體專有名詞，同時不涉及複雜的設計流程，能讓讀者對「多工核心」這個將要成爲 common sense 的字眼，有更深一層的了解。當然筆者也期望這篇文章的發表，對所有工研院機械所曾經參與多工核心設計與應用的同仁，致上最深的敬意。

單工與多工的世界

單工與多工究竟有什麼不同？在 MS-DOS 作業系統盛行的時代，這個問題一直爲許多的軟體工程師所爭論著，持單工觀念者認爲以最終主處理器（CPU）執行時，任一瞬間也只能執行一個機械指令（machine instruction），因此扣除多個 CPU 的平行處理系統之外，在單一 CPU 平台上，跟本就沒有多工可言，畢竟到了硬體一次還是只能處理一個指令，所以多工只是一些似是而非的花俏理論而已。然而在現實的世界中如果真的把觀點侷限在這麼小的地方，似乎就有點小題大作，想想看，如果在一個應用程式中，執行不同功能的模組都由不同的人設計，同時又可有多個應用程式都在執行狀態下，由一個 "藏鏡人" 在幕後主動偷偷的來分配與協調不同應用程式或是模組之間的執行，就可大幅的擴充了模組的整合性與程式執行的便利性，這該是一件多麼美好的事情。如果在把這些幕後分配與協調的功能一致化與介面化，就形成了一個

「多工核心」。核心存在的價值只是讓我們寫程式時更容易模組化、單純化，不同的工作單元（Task）之間完全由核心負責同步與協調的工作，有了它只是節省軟體程式設計者的負擔，架構在一個多工核心之上來發展程式，何樂而不爲？

單工的觀念其實很簡單，運作原理就是讓 CPU 依我們寫好並且 compile、link 完成的二進位（binary）machine instruction 一個接一個的執行，直到最後一個 instruction 完成，再將這個應用程式 unload，結束它的執行。以 C 的眼光來看，就是從 main() function 的下一個大括號開始執行，直到最後一個大括號執行完畢，就完成這個應用程式簡單又直接。但是絕大多數的應用程式都不只是這麼單純，尤其是應用在工業控制上龐大的系統程式，爲了配合某些硬體訊號發生時，必需快速的執行一段程式碼以達成必要的控制功能，因此有了中斷服務程式（Interrupt Service Routine，簡稱 ISR）的機制產生。當中斷發生時 CPU 將目前正在執行的指令指標（instruction pointer）值推入堆疊（push stack），再將程式碼跳躍到中斷服務程式的進入點（entry point）開始執行，ISR 執行完畢 CPU 再將先前的置入堆疊的指令指標取出（pop），再回復中斷之前的軟體執行狀態繼續向下執行。

一個單工的應用程式加上中斷的功能，其實就是一個最簡單的多工程式，如果再把中斷功能定義爲「固定時間、循環發生」，就已經具備了多工核心的雛形。多工核心的定義其實可以很簡單，以最白話的字眼來說可以解釋爲：「有某一個機制固定時間去執行某些固定的工作」，這些固定的工作以軟體的術語來說就是「排程（scheduling）」，而固定的時間則稱之爲一個 time tick。

多工機制的運作原理

既然多工核心最主要的任務是執行「排程」，那麼排程到底是什麼？而可以作爲排程的基本單位又是什麼？在軟體的領域中排程的基本單位稱之爲工作單元或是叫做 Task。依據一定的法則在眾多的 Task 之間，選定一個最適當的 Task 作爲下一個分割時間（time sliced）CPU 可執行的 Task，就叫做「排程」，而執行這個核心控制功能的機制就叫作「排程器（Scheduler）」。在現有絕大部份的多工核心機制均是採用所謂「priority based」的排程法則，每一個工作單元都依其重要性而被附予不同相對權值（relative weight）的 priority。排程器在每次 time tick 由中斷觸發，在眾多等待執行的 Task 之中（通常這些等待的 Task 會形成一個串列叫做 ready list）選擇 priority 最高的 Task 設定成爲「Running Task」，在下一個 time tick 來臨前擁有 CPU 的執行權利，並在 Task 內部以單工的方式依序執行，直到 ready list 中有 priority 更高的 Task 取的執行權利，則先前執行的 Task 又重新被置入 ready list 之下，等待下一次的競逐 CPU 的執行權利。這樣的 Task 彼此之間交換執行的動作稱之爲「context switch」，多工核心的機能說穿了就是不斷的進行不同 Task 之間的 context switch 動作，以達成許多 Task 可以輪流的執行，priority 高的工作單元執行完成，即切換到次高 priority 的工作單元執行；即使某個 Task 尚未執行完成，有 priority 更高的 Task 加入 CPU 執行權競爭的行列，這個 Running Task 也得讓出執行權；這就是多工核心最基本的機制（如圖一）。

應用多工的使命就是要使軟體更模組化，因此核心使每一個工作單元都配置有本身完整可使用的資源，且各工作單元執行時彼此之間並無相互依存的高度獨立性，亦即任何一個工作單元在需要的時刻，可以隨時建立（create）並進入待命的等待串列，而且當某個工作單元完成其任務後，可以即刻中止（terminate）並歸還其對核心所要求而取得的所有資源。既然每個工作單元之間具有高度的獨立性是核心所附予的天性，倘若不同工作單元之間必須一定要交換某些非常重要的資料的話，或是一個工作單元必須要等待某個事件發生後才可繼續向下執行後續的程式碼，而這個事件又是由另一個工作單元所觸發而產生的，如此就衍生了許多工作單元間的「同步協調（synchronization）」的需求。

當然這些橫跨多個工作單元間溝通協調的任務，核心是責無旁貸的必需挺身而出來解決的，所以核心除了基本的排程工作外，還必需提供許多的核心服務機制，使各工作單元之間得以完成彼此溝通同步協調的需求，這些機制如號誌（Semaphore）、事件（Event）、訊息信箱（Message Mailbox）、計時器（Timer），乃至於嵌入式（Embedded）環境最常用的中斷（Interrupt）、裝置管理（Device Management）等。每當執行中的工作單元（Running Task）呼叫這些核心服務的時後，即會觸發核心服務中心開始執行，如果是必需等待某個同步協調的條件發生時，而此時核心又判斷該條件此時並不存在或成立，核心會將 Running Task 置入等待串列（waiting list）之中，並自 ready list 取出 priority 最高的 Task，並執行一次 context switch 更換 Running Task。此外若 Running Task 呼叫核心服務是產生某個條件，核心服務中心會檢查是否有其它的工作單元在等待這個條件的發生，如果滿足核心會將等待這個條件發生的 Task 自 waiting list 中取出並置入 ready list，核心再啓動排程機制選出此時 priority 最高的 Task 並設定成爲 Running Task。

多工核心的機制即是如此生生不息的運作著，除了有固定時基觸發使核心控制中心在固定時間執行一次之外，還有眾多的核心服務也會引發核心控制中心起來運作，進行必須的排程行爲。雖然一 般而言核心控制中心運作的機制相當複雜，但對程式設計者而言則完全不需要碰觸這些相當底層的機制。在多工核心的環境中開發程式時，設計者只需獨立包裝其工作單元的功能即可，若需要與其它工作單元進行同步協調的工作，只要呼叫核心服務函式，剩下的事核心會解決一切。在多工環境中來發展應用程式，是一件單純又很模組化的工作，開發效率較單工至少提升了一倍以上，並且日後系統維護的工作也因高度的模組化而變的簡單，聰明的使用者，您會選擇那一個？

即時多工核心的訴求

在各式控制用軟體系統之中，「即時（Real-Time）」性能是一個相當重要的功能，它直接的決定了當一個訊號發生時，系統要花多少時間才能接收到這個訊號同時完成後續的控制行爲。依照「即時」這個功能的定義，只要是系統中任何與「時間」相關的訴求，都是即時功能應用的領域，一般而言它有三個非常重要因素來決定一個 Real-Time System 的適用性：

（1）Dead Time：即時系統的時基（time base）；簡單來說就是有訊號（可能是軟體或是硬體所產生的）產生後，系統最多要花多少時間才能發現這個訊號，基本上最大值就是系統時基的週期時間 time tick 值，10 ms、100 ms、1 sec 都是不同等級的即時系統 Dead Time。

（2）Hard / Soft real-time：強弱即時；在 DOS 此類以單工機能爲主的作業系統，當外部中斷產生時，DOS 幾乎立即將 CPU 執行權移轉到中斷向量表（Interrupt Vector Table）所規劃的中斷服務程式（Interrupt Service Routine，ISR）來執行應用程式必要的中斷控制動作，由硬體訊號產生到軟體 ISR 開始執行，這中間中斷延遲（Interrupt Latency）的時間幾乎是微乎其微，也許在現今快速如 Pentium 等級的 CPU，可能只會造成 us 等級的時間延遲。但在多工核心作業系統之中，因爲內部應用中斷機制來達成許多的核心服務功能，而在如 scheduling、context-switch 過程中，又是不予許受到太多外部的干擾，因此一般中斷發生經由核心處理後所造成的延遲時間，其等級就無法忽視了。一個具有較短的中斷延遲時間的核心稱之爲「Hard Real-Time System」，相對來說如 Windows NT 雖然具備可達 1ms 系統時基的能力，但本身也只能宣稱其爲 10 ms 等級的「Soft Real-Time System」。

（3）Determinism：準確性；即時多工核心運作時，它的每一項功能在執行時所花費的時間是必需可以準確預期的，無法明確告知每個工作單元在執行多工同步協調時最多會耗費多少時間的核心，是會使程式設計者所擔心的不定時炸彈。

綜合以上特性的描述，可以大略得知「即時」性能幾乎決定了一個多工核心系統的適用性。更進一步來說，在多工核心控制的軟體領域中，並沒有「好的核心」或是「壞的核心」的區別，只有適不適用的核心。這是什麼意思？難道 1ms 時基的核心沒有比只能達到 10 ms 時基的核心來的好嗎？其實我們不能只從表面的呈現的時間數據來評斷核心。舉例來說，兩個具有相同中斷延遲時間與準確性的核心稱之爲「甲核心」與「乙核心」，而它們的系統時基分別爲 1 ms 與 10 ms，假如中斷延遲時間爲 200 us，則這個時間數據在甲核心造成了 20% 的時間延遲，相同的數據在乙核心只有 2% 的 Delay；另外假設兩個核心服務中心觸發執行時都需花費最多 200 us 的時間，同樣的我們可以說甲核心的系統負擔（overhead）是 20%，也就是甲核心只剩下 80% 的時間供一般工作單元來執行，然而相同的數據在乙核心卻只有 2% 的 overhead，使用者的工作單元擁有 98% 絕大部份的執行權。有了以上兩個數據，我們還能說甲核心的 1 ms 系統時基一定優於乙核心的 10 ms 系統時基嗎？相信大部份的人都不會在有這個印象。那麼究竟選擇怎樣的核心才會適用呢？其實很簡單，完全從應用的環境來決定，如果 10 ms 系統時基的乙核心可以適用的話，就不需要 1ms 的甲核心經常執行一些額外的負擔，選擇的標準就是這麼簡單。

設計一個簡單的多工核心－－x86真實模式

如果多工機制真的可以爲程式設計師帶來這麼多的便利，那麼進入一個多工的軟體環境它的障礙高不高？如果一個軟體工程師慣用的是單工的系統，那

麼在單工的作業系統之上設計一個外掛式的多工核心，使得工程師仍然保有其原環境使用的習性，附加了這個多工核心機制又使得可以增加多工的選擇，這樣的困難度究竟高不高？倘若真的要著手來設計開發一個多工核心環境，又該從何下手？什麼樣才是開始的第一步？一連串的問號相信一直深深的困擾著大多的軟體工程師。

所謂「工欲善其事，必先利其器」，要選擇一個常用的單工作業系統來設計多工核心，最熱門的候選者莫過於 MS-DOS 了，在 Windows 作業系統盛行之前，MS-DOS 曾是絕大部份 PC 的作業系統，時至今日仍有相當多的應用程式`以此爲執行環境，而在工業界來說甚至仍有許多新的系統開發仍將 MS-DOS 作爲最優先的選擇。爲什麼 MS-DOS 如此熱門？原因無他只是它夠簡單，而且它所運用的 CPU 處理模式爲最簡單且最開放的 x86 真實模式（Real Mode），在此一執行模式之下，所有 PC 的資源包括輸出入埠（I/O Port）、記憶體空間（Memory Space，一般來說 MS-DOS 可直接定址到 1M 以下的位址空間）、中斷（Interrupt）等都可以透過非常容易的方式來存取，同時因爲所需執行空間小於 1 MBytes，麻雀雖小但五臟俱全，因此 MS-DOS 在工業界的應用仍是最高呼聲的不二人選。

既然 MS-DOS 是單工作業系統多工化的最佳選擇，那麼接下來就得對 x86 真實模式做更詳盡的了解，才能有機會來設計一個多工核心作業環境。首先必須要了解在真實模式下 CPU 執行時常用的「暫存器（register）」暫存器（register），通常來說重要的有以下數種：

（a）一般暫存器：AX、BX、CX、DX；作爲 CPU 執行時一般資料暫存搬移的緩衝區，一般來說每個暫存器無特定用途。

（b）節區（segment）暫存器：CS（Code Segment）指令節區；DS（Data Segment）資料節區；SS（Stack Segment）堆疊節區；ES（Extra Segment）額外節區。

（c）指標（pointer）暫存器：BP（Base Pointer）基本指標，SS:BP 代表目前程式所使用 stack 的起始位址；SP（Stack Pointer）堆疊指標，SS:SP 指向目前 stack 可使用的位址；IP（Instruction Pointer）指令指標，CS:IP 所指向的位址爲 CPU 目前正在執行的指令。

了解 x86 真實模式 CPU 執行時重要的暫存器意義之後，接下來就是要剖析一般單工程式執行時的特性，以作爲設計附加多工功能時的參考依據。以大記憶體模式（Large Memory Mode）而言，任一個應用程式執行時四個節區暫存器都會指向一個大小空間爲 64K Bytes 的節區，CS:IP 構成目前 CPU 執行的指令；SS:BP & SS:SP 組成堆疊的使用；DS:offset 則提供全域變數（Global Variable）的絕對位址，因此只要掌握上述暫存器的值，就可以有效的監控這個應用程式的執行狀況，而這又是 x86 真實模式執行時的重要特性。在 MS-DOS 的作業系統之下，一次只能執行一個 .exe 的可執行檔是無法改變的事實，但若以另一個角度來看，一個應用程式是由許多的函式（function）所組成，如果能夠把一個函式包裝成一個工作單元，應用程式執行後可以允許許多函式所獨立組成的工作單元以多工的模式執行著，如此雖然是單工的作業系統確因附加了某些東西，使得具有多工作業環境的能力，就達到了多工能力的需求。

由上衍生，一個工作單元倘若擁有自己的 CS、SS、DS、ES 等節區暫存器值，在不同工作單元交換時的 context switch 流程中，只要將至少上述的的值存入一個多工核心所掌管的記憶體中（姑且稱之爲 Task Control Block，簡稱 TCB），然後交換不同工作單元之間的重要暫存器的值，就完成了 context switch 的主要工作，而此也是設計一個多工核心的重要精神所在。因此在每一個工作單元的 TCB 之中，都將會記錄著這個工作單元執行時所有的重要資料，在多工核心執行 context switch 時，也都將會自 Running Task TCB 中取出許多執行時重要的資料，所以 TCB 是一個工作單元的靈魂，一個工作單元產生建立之後就會跟隨著一個 TCB，而當這個工作單元消失的時刻，它的 TCB 才會隨之消滅，所謂「生死榮辱共存亡」，這正是工作單元與它的 TCB 最好的寫照。

接下來還必須了解的就是程式語言的特性，現今最普遍使用的程式語言莫過於 C 語言了，而不論使用的是那一種程式語言，函式呼叫是絕對免不了的，一個稍具結構化與模組化的程式，必然是各個函式層層呼叫，而在 C 語言中呼叫另一個函式時，編譯產生的機械碼（machine code）必定依循下列程序：（a）將函式呼叫時所傳遞的引數（argument）值推入堆疊中（push stack）；（b）將函式執行完成後的返回位址（return address）置入堆疊中；（c）計算函式中局部變數（local variables）所佔的位址空間，將 SP 值減去該值指向可用的堆疊位址。所以當 C 語言的程式呼叫函式時，堆疊的變化如圖二，堆疊的使用在 C 語言中扮演相當重要的地位，了解堆疊機制的運作將會對於我們使用 C 語言設計一個多工核心有相當大的助益。

如前所言 x86 真實模式是一個典型的單工執行環境，而 MS-DOS 又是標準的單工作業系統，因此在同一時刻 MS-DOS 只能執行一個 .exe 的可執行檔，那麼當我們已經全盤了解了如「CPU 的暫存器」、「TCB 的觀念」、「C 語言的特性」等，我們又應該如何著手來設計一個多工核心，並且把一個函式包裝成一個工作單元呢？首先我們知道堆疊在一個應用程式中扮演非常重要的腳色，多工作業環境設計的主要目的就是在單工中執行一個應程式時，好像有多個應用程式在一齊執行，因此每一個工作單元都應擁有私有的堆疊，同時代表堆疊位址的 SS 與 SP 暫存器的值都應記錄在 TCB 之中，在 context switch 交換工作單元時時才能自 TCB 中取出或寫入。完成了堆疊的運用之後，接下來另一個對工作單元相當重要的資料就是 CS 與 IP 暫存器的值，由於 CS:IP 指向目前 CPU 應該執行的指令碼，既然每一個工作單元都是由獨立的函式所包裝而成，所以每個函式都應有不同的位址空間，而在一個工作單元經由 context switch 交換出去之前，這個工作單元目前的 CS:IP 值應儲存起來，在下一次該工作單元又再獲的執行的權利時，就必需將切換之前所儲存的 CS:IP 執讀出，並寫入 CPU 的暫存器之中，這個工作單元才會自上一次被交換而中止直行的指令繼續向下執行。因此 CS 與 IP 這兩個暫存器值的重要性並不小於堆疊，那麼在多工交換時，每一個工作單元的 CS 與 IP 值應該存放在那裏？是堆疊？還是 TCB？

在要找到適當的地方來存放 CS 與 IP 的值之前，我們必須要對形成工作單元交出執行權的狀況作進一步的分析。如圖二所言，多工的交換機制會產生

作用，基本上是核心服務中心被觸發，而造成觸發的條件有兩個：（1）系統時基中斷；（2）呼叫核心服務程式群。基本上有權利呼叫系統核心服務程式的只有目前擁有執行權的 Running Task，在呼叫任一個系統核心服務程式之後，Running Task 的 CP:IP 值都被推入堆疊中存放，同時更新 CS:IP 值爲系統核心服務程式的程式碼起始位址，倘若在系統核心服務程式呼叫核心控制中心之後排程法則認爲不須要進行工作單元的交換，CS:IP 值也隨著系統核心服務程式的結束而自堆疊中 POP 出來，繼續執行下一個 Running Task 的指令。但是如果在核心控制中心排程法則認爲須要進行工作單元的交換，因此核心控制中心會先進行 TCB 與堆疊的更換，假如下一個成爲 Running Task 的工作單元是一個尚未執行過的工作單元，而其初始堆疊的規劃如果如圖三表示，那麼在系統核心服務程式結束的時後，自堆疊中 POP 出來的 CS 與 IP 值就是新的工作單元函數的起始位址，透過工作單元堆疊初始值的巧妙安排，就完成了工作單元 CS 與 IP 暫存器值的交換。

另一個會呼叫核心控制中心執行的機會是經由固定時間產生的系統時基中斷，而在許多談論 x86 真實模式與 MS-DOS 的書籍中都會提到，當系統任一個中斷發生時，首先目前的 CS:IP 值會被 PUSH 進入堆疊，CPU 再自中斷向量表中，找出該中斷對應的中斷服務程式（ISR）並予以執行，在 ISR 執行結束後，IRET 指令會自堆疊中 POP 出 CS:IP 值，並回復到中斷之前的程式碼繼續向下執行。以此分析，不論是何種原因造成核心控制中心的被呼叫，Running Task 目前的 CS:IP 值都會完整的被儲存在堆疊中，而如果一個工作單元的堆疊初始值如圖三的安排來設定，則不論這個工作單元被切換多少次，都可以自堆疊的頂端得到其被切換的 CS:IP 位址值，若該工作單元尚未執行過則自堆疊頂端得到的就式函式起始位址，在這樣的設計安排之下我們永遠可以自堆疊中取的該工作單元的 CS:IP 值，所以 CS 與 IP 暫存器值不須要存放在 TCB 中，堆疊就是它們的最佳住所。

堆疊的使用在多工機制是相當重要的，如同單工環境下的應用程式，所有執行過程中的資料都存放在堆疊中，既然我們所設計的多工機制將要模擬許多應用程式同時執行，因此堆疊中除了儲存工作單元最重要的 CS:IP 暫存器值之外，其餘的 CPU 暫存器的值也應一併寫入堆疊之中，在多工排程 context switch 時才可與 CS:IP 值一併交換，維持一個應用程式執行時的正確性。圖三的堆疊的底端存放了一個 TASK_exit() 系統核心服務程式呼叫的函式位址，表示任一個工作單元執行到函式的最後一個大括號 “}”，即爲該工作單元執行完畢，經由堆疊 POP 函式 TASK_exit() 的位址，核心服務中心會自動將該工作單元移除，並釋放其所擁有的資源，所以堆疊內的資料代表了一個工作單元建立、執行移除時的所有訊息，因此建立一個工作單元時配置一個空間大小適當的堆疊區是非常重要的。

對一個工作單元而言，TCB 的重要性並不亞於它的堆疊，如前所述，工作單元堆疊的 SS 與 SP 值都必須記錄於 TCB 中，工作單元交換時才可至新的 Running Task TCB 中得到它的堆疊訊息，接下來再自堆疊中 POP 出其它暫存器的值。圖四是一個基本的多工工作單元的 TCB 資料結構宣告，當然 TCB 中除了 SS 與 SP 值之外，還有很多串列指標用來連繫所有具有相關性質的工作單元，例如當一個工作單元的狀態是 READY 時，則自 TCB 中的 READY

list 可以 trace 到所有狀態皆爲 READY 的工作單元，而這些串列搭配系統核心服務程式，組構成爲堅強的的多工核心系統服務功能。

「Context Switch」是一個多工核心的大腦，所有工作單元交換的決策均由其來執行，圖五是一個 Context Switch 的流程，基本上 context switch 機制每一次被呼叫觸發，進行排程時都會至 READY list 的 HEAD TCB 尋找是否即爲 Running TASK，如果不是則進行工作單元的交換。相同的當一個工作單元進入 READY list 的時後，priority-based 的排程法則會依工作單元的 priority 值大小，插入 READY list 之中，所以 READY list 的 HEAD TCB 所表示的工作單元永遠是 priority 最高的，自 HEAD TCB 向下 serach 則可得到依 priority 排序的所有狀態爲 READY 的工作單元。

了解了一個多工核心運作時的重要機制，那麼應該如何來啓動這個多工核心呢？我們如何在 MS-DOS 的單工環境下重新規劃計時功能，以達到即時性要求的固定時間觸發的系統時基呢？在進入描述多工核心的執行主流程之前，我們必須要知到一件相當重要的事，只要使用 C 語言所設計的應用程式，其執行時唯一的程式進入點必定爲 main() 函式，而執行權在交到使用者所設計的程式碼之前，核心必需完成所有初始化的設定，並且包裝一個最基本的工作單元（姑且稱之爲 USERMAIN Task），使用者的程式即從 USEMAIN Task 的進入點開始執行，第二個以上的工作單元即從這個 USERMAIN Task 來建立與執行，直到所有的工作單元都已經被移除不再執行之後，多工核心最後接管將系統設定值回復 MS-DOS 的單工模式下後，結束核心的執行也結束這個單工應用程式的執行，重新回到 MS-DOS 的控制模式之下。多工核心執行時的主流程如圖六所示，原則上這種多工核心的設計精神是在於執行應用程式時，增加多工程式設計與執行的便利性，並不是改變作業系統使之成爲具有多工能力，作業系統一次仍是只能執行一個應用程式，如果要使作業業統能夠同時執行多個應用程式，而每一個應用程式中又須具備如多工的工作單元的執行能力，要滿足這個訴求，只有直接選擇一個具多工能力的作業系統，才能解決這個問題。

應用至不同 CPU 平台－－DSP C32 Kernel

前節所介紹的多工核心設計原理，雖然只是針對 x86 真實模式，但相同的開發理念可以應用到各種與 x86 真實模式極爲類似的各種 CPU 平台上，特別是許多的嵌入式（Embedded）平台。一般而言，許多的嵌入式 CPU 的應用發展工具，都會支援 C 語言編譯器，對一個嵌入式應用程式開發者而言，通常都必須借助 PC 的環境撰寫特定適於嵌入式平台執行的 C 語言程式，再經由下載（Download）方式將程式植入嵌入式系統之中。由於嵌入式平台的使用者均爲特定對象，因此對程式開發展而言相對可取得的資源就少了許多，因此在 PC 環境下甚至有許多具多工能力的作業系統可供選擇，但在嵌入式平台就幾乎只有原廠所提供的開發與使用環境一個選擇，應用程式開發者也就只能選擇不得不用的系統作爲執行環境。所以如果將多工的觀念導入這些嵌入式平台，使得執行在這些平台上執行的應用程式都具有多工的能力，相信必可拓廣嵌入式平台的應用。

不同的嵌入式 CPU 都有其獨特的執行模式與暫存器的定義，同時支援的 C 編譯器產生 Machine Code 的機制也都會有所不同。因此要開發一個適用於嵌入式平台的多工核心，第一步就必須得先了解 CPU 暫存器的的意義，接下來特定的 C 編譯器產生 Machine Code 時是否因應 CPU 的暫存器而有特殊的功能，最後在來尋找與 x86 真實模式的相似與不同的地方，包括重新設定堆疊的使用以及 TCB 的資料結構，只要能將這些巨細而微的差異點找出來，我們就可以著手來設計一個嵌入式的多工核心了。

在本文中我們使用了 TI 公司的 TMS320C32 的數位訊號處理器（Digital Signal Processor，簡稱 DSP）作爲嵌入式平台的 CPU，同時選用搭配這個 CPU 的 C 語言編譯器作爲發展多工核心程式的軟體工具。接下來的工作就是研究這個 CPU 的所有暫存器的定義，在這個支援浮點運算的 DSP 的暫存器中，首先我們發現了 PC（Program Counter）這個暫存器的功用類似 x86 真實模式的 CS:IP，都是指向目前正在執行指令的位址。其次我們發現 SP （Stack Pointer）暫存器值指向目前所使用的堆疊起始位址，但是經過檢查後發現這個 DSP 暫存器並沒有功能定義如同 x86 真實模式下的 SP 與 BP 指標值，來指明目前堆疊的使用狀況，在 DSP 的手冊中提到程式設計者必須組合語言來控制堆疊的使用狀態。缺少了這些代表堆疊使用狀態的暫存器，似乎對於我們移植多工核心至這個嵌入式 DSP 環境遇到了若干阻礙，幸而我們在 C 編譯器使用的手冊中發現了這個 C 編譯器借用 AR3 這個暫存器，定義其爲 Frame Pointer 指向下一個可使用堆疊的位址，因此只要是多工核心以外的程式均由 C 語言撰寫而成，則 AR3 功能定義就極爲類似 x86 真實模式的 SP 指標。經過了這些研讀分析的步驟，構成 x86 真實模式多工核心最重要的幾個暫存器：CS、IP、SS、SP 等，在 TMS320C32 DSP 的暫存器中均有相對功能的暫存器存在，因此我們就非常有把握的可以繼續從事設計開發的步驟了。

一個設計開發完成的 TMS320C32 DSP 嵌入式多工核心作業系統的工作單元 TCB 資料結構規劃，以及堆疊的使用初始化規劃分別如圖七與圖八所示。以此來看，扣除不同 CPU 的暫存器之外，其運作的基本原理與在 x86 真實模式之下的多工核心並無兩樣。所以只要能有效的掌握不同 CPU 的某些特性，就可以應用多工的原理，設計開發特定的多工核心系統，執行平台的轉換對一個熟悉多工機制的設計者而並，並不是一個很大的障礙。

Windows 作業系統的多工

現今 PC 最熱門的作業系統統莫過於 Microsoft Windows 作業系統了，Windows 作業系統的家族成員包括了 Windows3.x、Windows95/98、Windows NT、Windows CE 等各式產品，並且普遍的使用在商業以及工業自動化控制系統之中。基本上所有的 Windows-Based 作業系統均爲標準的多工作業系統，除了視窗化的作業環境功能之外，內建的 Kernel 本身即已具備多工排程能力，原則上不需要再外掛額外的核心，因此使用 Window-Based 作業系統即可規劃具多工控制能力的應用程式。

Window-Based 作業系統之中多工排程基本的工作單元稱之爲 Thread，應用程式則稱之爲「行程（Process）」，一個 Process 由至少一個 Thread 所組成，而 Windows-Based 作業系統本身即已規劃執行在保護模式（protected mode）之中，因此作業系統予許很多的 process 同時執行（也就是可以讓很多的 .exe 檔案同時執行中），而多工的排程仍以很多 process 中的 thread 爲基本單位，倘若 context-switch 只發生於同一 process 之中的兩個 thread，其排程交換的速度遠較進行不同 process 之間的 thread 來的快許多，因爲 context switch 的動作並不牽涉 process space & resource 的交換。Windows 3.x/95 的作業系統架構如圖九，多工核心的機制設計在 VMM（Virtual Machine Manager）之中，負責進行眾多 Virtual Machine 之中 thread 的排程。Windows NT v4.0 的作業系統則如圖十所示，多工的排程機制是由底層 Kernel Mode 的 Micro-kernel 模組來執行。原則上 Windows-Based 的多工排程所採用的仍爲 priority-based 的法則，priority 較高的 thread 理應享有較優先的執行權利，但 Windows NT 所使用的 Dynamic priority 方式則爲平衡前景（foreground）行程與背景（background）行程的執行時間。

基本上 Windows NT 將 priority 分爲 0 ~ 31 等共 32 level，其中 0 ~ 15 稱爲「Dynamic Range」，而 16 ~ 31 則稱之爲「Real-Time Range」，設定在「Dynamic Priority range」的 thread，在執行一個時間單位（稱之爲 Quantum）之中，若無發生 context switch 交出執行權，則在這個時間之後會由 Kernel 將這個 thread 的 priority 自動減一，然後觸發排程機制執行 context switch 再選出 priority 最高的 thread 成爲「running thread」。圖十一則表示了這樣的「動態優先權式」的排程法則，thread A 在 priority 被變化之後即成爲 ready thread，等待兩個 Quantum 時間後，thread B & C 的 priority 也被動態調整減一，因此 thread A 又再獲得至少 Quantum 時間的執行權利。但是被設定在「Real-Time Range」的 thread 排程方式又有所不同，Thread X 執行完 Quantum 時間之後，它的 priority 並沒有被減少，而是被置於相同 priority 串列的最後面，等待所有相同 priority 的 thread 都執行一次之後，thread X 又再取得執行權。簡單來說，「Dynamic Range」的 priority 值有相當濃厚的「相對」意味，而「Real-Time Range」的 priority 則定義爲「絕對」的。

以前述的即時觀念來檢視所有的 Window-Based 作業系統，它們到底是不是一個好的「即時作業系統」？許多的文章一直在討論這個話題，見仁見智、各抒己見。雖然討論的觀點相當多，但逐漸都有一個共識，在所有的Window-Based 作業系統之中扣除尚未正式發表的 Windows CE 3.0 版之外，較適合應用於工業即時控制的作業系統應爲 Windows NT。然而在許多文件之中僅宣稱 NT 爲 10 ms 等級的「Soft Real-Time O.S.」，雖然 Win32 API 可規劃計時器的等級到 1 ms，但當許多行程共同執行時，單一計時器的誤差將會高達 5 ~ 10 ms，也許這個數據等級在很多的系統之中已足夠，但在講求高即時的工業控制系統之中，這個數字似乎就有點不盡理想了。

即時功能的延伸

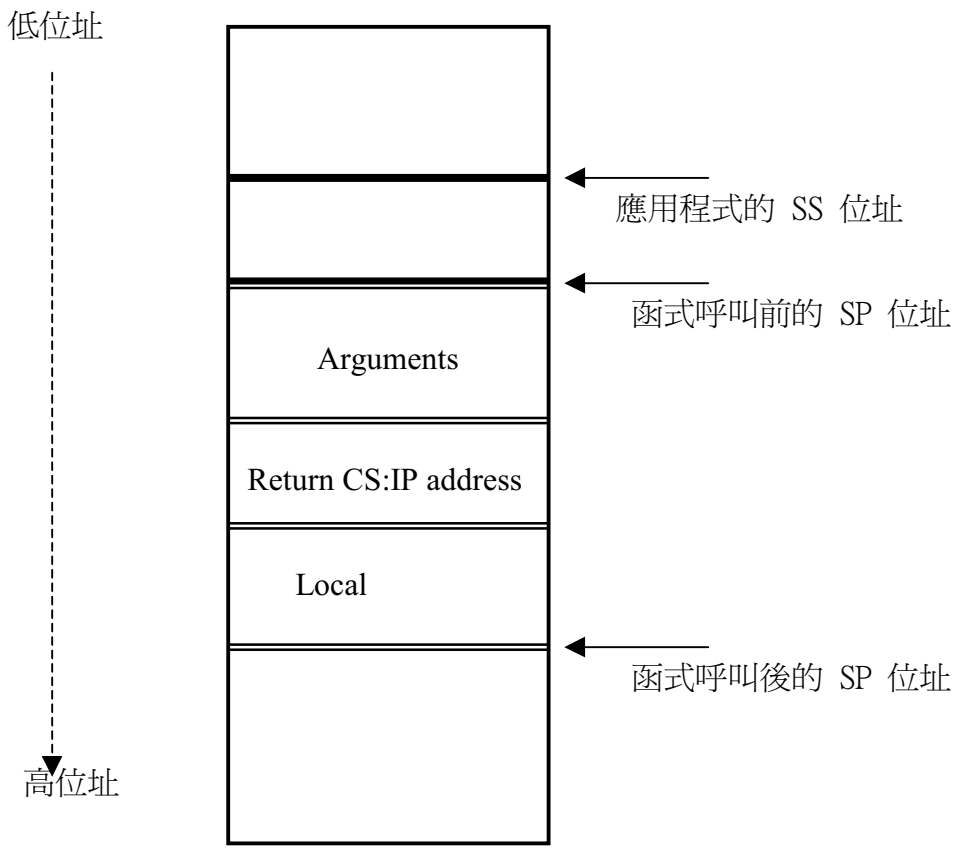
走向圖形化、視窗化的 Windows-Based 作業系統，將成爲不論是在商業及

工業控制的潮流，畢竟 Win32 提供豐富的應用程式執行環境包括網路等，是讓人無法抗拒的，然而即時能力的不足，又讓許多工業即時控制應用者爲之卻步。爲了提供這個「強即時（Hard Real-Time）」的需求，許多軟體公司紛紛與 Microsoft 合作，以 third-party 型式開發附加於 Windows NT 環境下的延伸模組（Extension Module），以提升作業系統的即時能力。當今世界上較常用的即時延伸模組有 RTX、InTime、HyperKernel 等幾種，其附加於 Windows NT 的方式也如圖十二~十四所示。

不論各種軟體產品以何種架構來提升 Windows NT 的即時控制能力，其基本出發點都是希望保有原先 Windows 作業系統的優點，同時透過增加的即時子系統（Real-Time SubSystem），使的執行其中的行程具有至少低於 1ms 等級的「Hard Real-Time」能力。以 RTX 爲例，提供了一組豐富的 Real-Time API可以運用於 Win32 子系統與其設計的即時子系統之中，在即時子系統之中執行的行程，其即時響應能力可達 50 us，較原先 Windows NT 所宣稱的數據提升了 100 倍以上，有了這些加強即時能力的軟體工具的輔助，將可在 Windows NT 作業系統之中開發具有高即時性的多工應用程式，充份的運用 Windows 作業系統所帶來的各種好處，加快我們開發工業即時控制系統的腳步。

結論

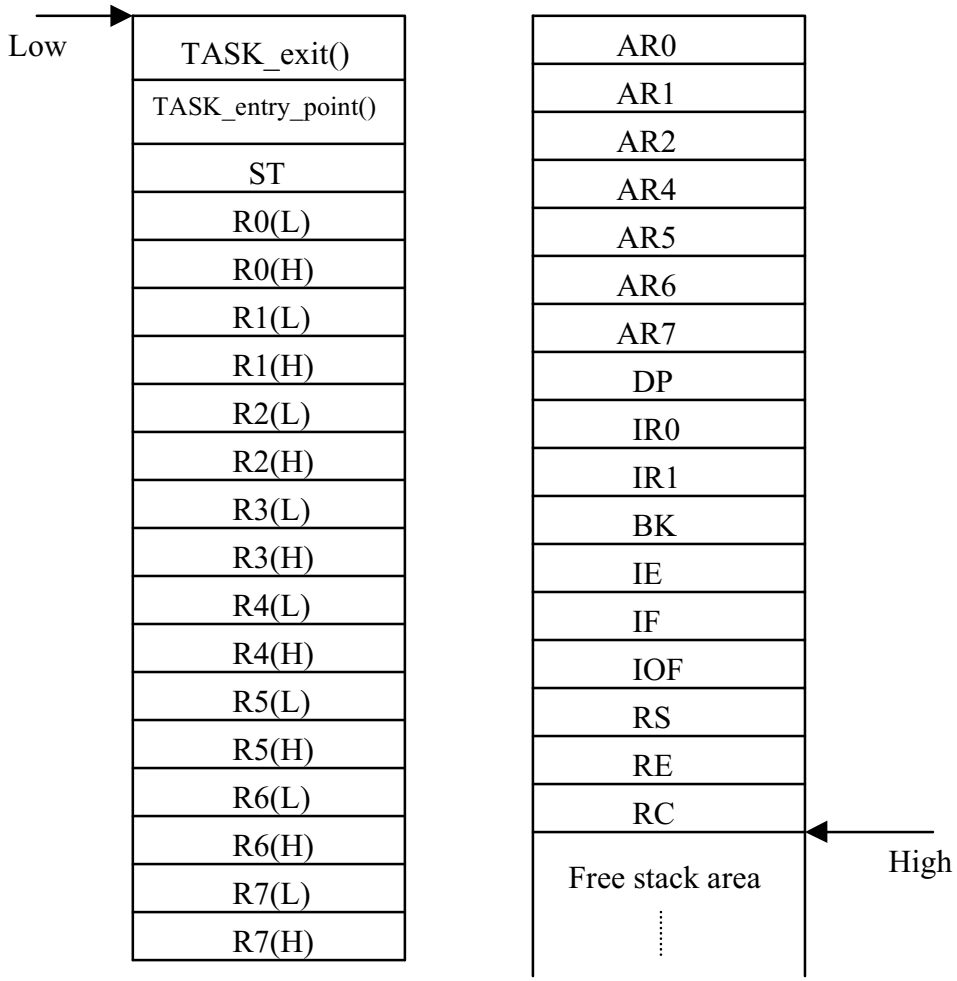
多工核心機制的影子是無所不在的，小到 CD-ROM 或是數位相機的控制晶片軔體，向上延伸到專用的嵌入式系統或是開放式的個人電腦軟體，乃至於功能龐大且複雜的系統工作站，都有許多的產品選用了某個多工核心作業系統作爲其應用發展的軟體平台。可以預見的未來世界將逐漸走向數位化的時代，系統中的關鍵軟體模組也會由許多 third-party 應用廠商分工合作來開發，彼此享用與交流互相的成果，如此新產品的開發期才能有效的縮短，加強產品的競爭力。要達成這個理想，單工作業系統上的軟體是無法獨立有效分工來設計的，單工的系統必須由系統設計者全盤掌握每一個環節，同時應用程式環環相扣，無法達成模組化資源共享的要求。也只有認識「核心」、了解「核心」，才能大幅提升系統的開發速度，縮短產品的開發時程，在核心的協助下有效率的跨入事倍功半的最高境界。



圖二：x86 真實模式 C 語言函式呼叫之堆疊變化

```
Struct tcb{
        struct tcb              *plink,*nlink;
        struct tcb              *rdy_wait_plink,*rdy_wait_nlink;
        struct tcb              *count_plink, *count_nlink;
        word                    priority;
        word                    status;
        word                    sp_value;
        word                    AR3_value;
        word                    *stack_ptr;
        TASK_proc_ptr   task_entry_ptr;
        word                count;
        word                    event_or_flag;
        word                    event_and_flag;
        word                    result;
        } TCB, *TCB_PTR, *TCB_ID;
```

圖七：DSP 多工核心 TCB 資料結構



圖八：DSP 多工核心堆疊安排