目錄

8

CHAPTER

多工核心預覽

8.1 前言

預覽多工核心，快速地走過多工核心的重點。之後，再摸進多工核心，是筆者的想法。多工核心的重心是 "多工" 二字，內容牽涉多工方式、多工原理、多工管理和多工運作。

8.2 多工方式

多工方式是指任務是以怎樣的方式交替執行。這一層次的內容最貼近使用者，使用者可以親身感受到多工正在運作著。在 WINDOWS 和 LINUX 上，使用者很容易感受到作業系統的執行速度不一樣。即使是微軟自家的 WINDOWS 系列作業系統，從早期的 WINDOWS 95、98、2000、ME、XP，我們也可以明顯感受到各個 WINDOWS 平台上所呈現的不一樣的執行慣性。對多媒體的任務，或對文字編輯器的任務，對不同性質的任務有不一樣的處理方式，呈現出來的就是不一樣的工作習性。

8.2.1 什麼是多工

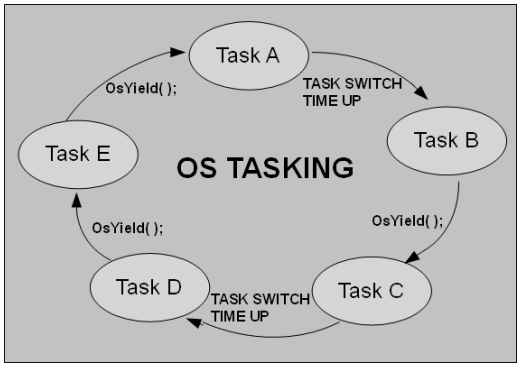
多工一詞譯自英文 MULTI TASK，意思是多個任務。一個可以執行多個任務的作業系統就叫做多工作業系統。80 年代和 90 年代，WINDOWS 95 作業系統發表之前，IBM 相容個人電腦幾乎都是 MS-DOS 作業系統的天下。MS-DOS 是單工的作業系統，就算是跑 WINDOW 3.1，它還是單工的作業系統。從 WINDOWS 95 開始，多工作業系統開始進入 IBM 相容個人電腦的市場。在多工的環境中，作業系統可以同時執行不同而彼此獨立的任務，任務間的通訊則是靠事件或訊息來傳遞。舉例來說，一個任務負責使用者介面，一個任務負責 COM1，一個任務負責 LPT，他們之間是完全獨立的，各自處理各自的事件。由外面看起來，他們是同時執行的任務。事實

上，它們是執行時間很短，並且快速地此起彼落交替執行的任務。使用者本身察覺不到這樣任務交替過程，因為任務交替的過程都是在作業系統的多工核心執行，而使用者只是看到正在運作的使用者介面而已。

想像一個畫面，使用者正在使用電腦打字，打字速度是每秒一字。對作業系統和 CPU 來說，這一秒鐘的時間都是在等待這一個字的輸入，而一秒鐘對中央處理器來說已經可以執行上百萬個指令了，實在不該浪費在等待一個字的輸入上面，除非真的無事可做。再想像一個畫面，使用者正在上網，也正在編輯多份文件，也正在聽音樂，而這一切的動作都在同一部電腦上執行，這就是多工的優點。在多工作業系統中，可以執行簡單的任務，也可以執行複雜的工作，把 CPU 的效能發揮到最高點。

8.2.2 分時多工

任務的排程是跟著時間的間隔進行，當任務交換的時間到的時候，排程器就會被執行，任務交換的動作就會被啟動，這樣的任務運作方式叫做分時多工。分時多工的作業系統執行效率最差的狀況就是所有的任務都只跑分時多工。如果所有的任務都佔據相同的時間週期，作業系統的運作也會變得鈍鈍的、呆呆的，此時需要一個機制來活絡多工核心的運作，加速多工的運行，而這個機制就是任務級任務交換，在本作業系統中是呼叫 OsYield 函式。當任務無事可做時，任務本身就等於是在消耗 CPU 運算資源，應該要把 CPU 的運算資源讓給別的任務使用，這時可以呼叫 OsYield，也應該要呼叫 OsYield。



另外一種多工方式是先佔式多工，先佔式多工作業系統強調把最多的 CPU 資源讓給目前使用者正在執行的任務，其他也正在執行的任務則分配到剩下的 CPU 資源。先佔式多工的好處是可以讓使用者偏愛的任務儘快被執行完畢，壞處是有些需要很多 CPU 運算資源的任務會被嚴重拖慢、甚至停住。先佔式多工作業系統早期的代表是 WINDOWS 95，也就是微軟第一代的多工作業系統。當 CPU 運算能力還不夠強時，先佔式多工可以防止 CPU 資源因分散而導致的效能不彰。當 CPU 運算資源夠強時，先佔式多工會導致 CPU 的運算資源被鎖住，而無法執行更多的任務，這也是另一種

效能不彰。如何在作業系統的效能上取得最佳平衡，必須衡量作業系統所處的環境，選擇合適的排程器，以得到最佳的效能。

限制完成時間的多工方式是即時多工，支援即時的作業系統叫做即時多工作業系統。即時的意義是指當某事件發生時，必須在一定的時間內，將事件處理完成。即時多工作業系統又分軟即時多工作業系統與硬即時多工作業系統，他們的差異在於對即時的要求程度不一樣，並不是每個即時多工作業系統都可以達到最高的即時水準。

大的作業系統所面對的環境比較複雜，往往做不到即時性，把每份工作穩當的完成是它的主要工作，如 WINDOWS、LINUX。小的作業系統所面對的環境比較單純，即時性較佳，所以市面上大多數的即時作業系統通常都比較小，如 ECOS、UC/OS-II、VXWORKS。每個作業系統都有它瞄準的戰場，不能期望它的每個方面都完美。

8.3 多工原理

任務交換 (TASK SWITCH) 是多工作業系統運作的基本原理。每一個任務，基本上，都是一個無窮迴圈，在停止之前會一直不斷的執行。如何從一個無窮迴圈跳到另一個無窮迴圈就是任務交換的功能。多工作業系統中有很多個無窮迴圈(任務)，在這些無窮迴圈之間跳躍，進行多工作業系統的運作。

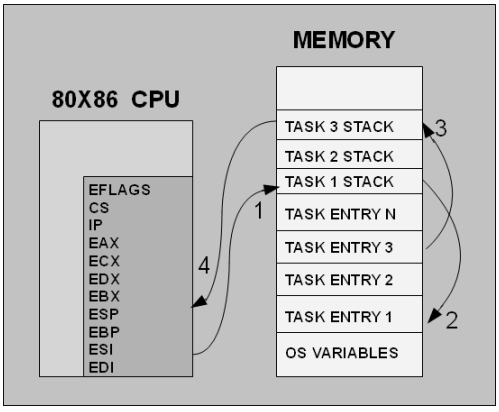
本作業系統只用到保護模式的 GDT，沒有用到 LDT 和 TSS，採取最簡單的方式進行任務交換。任務交換的過程中，僅會使用到 8 個工作暫存器，EAX、ECX、EDX、EBX、ESP、EBP、ESI、和 EDI。任務交換的重點在內文交換 (CONTEXT SWITCH)，內文包含 EFLAGS、CS、IP 和 8 個暫存器。當中斷發生時，CPU 會主動把 EFLAGS、CS、IP 堆入堆疊。之後，在中斷服務函式中，使用 PUSHAD 指令將 8 個工作暫存器一次堆進該任務的堆疊中，如此便完成該任務的內文儲存。等執行完中斷處理函式後，再把內文從堆疊中取回，回復 8 個暫存器值，並藉著中斷回返指令 IRETD 將堆疊中的值回復到 EFLAGS、CS、IP 中，以完成任務交換的動作。

本作業系統支援任務級任務交換和中斷級任務交換。任務級任務交換是由任務本身引發，目的是為了加強 CPU 資源的使用。本作業系統使用 INT 0x30 中斷呼叫實現任務級任務交換。中斷級任務交換是由外部中斷引發，外部中斷的來源是可程式中斷控制器 8259，特別是計時器中斷，用於實現分時多工。其他的外部中斷也可以用來引發任務交換，用於實現先佔式多工和即時多工。

任務交換步驟分解：

步驟	說明
1	將 TASK1 任務內文堆入堆疊，包含 EFLAGS、CS、EIP 和 8 個工作暫存器。
2	將目前的堆疊指標存入 TASK1 的任務控制結構中。

步驟	說明
3	從 TASK3 的任務控制結構中取出堆疊指標值。
4	將 TASK3 任務內文從堆疊回復到 CPU 中。



8.3.1 任務級任務交換

任務級任務交換是任務本身主動提出任務交換的行為，這樣的行為可以幫助作業系統運行得更有效率。任務級任務交換發生時，任務會呼叫 OsYield，主動把 CPU 讓給別的任務使用。別的任務也包含這個任務本身，因為作業系統有可能只有一個任務在跑。

任務級任務交換的動作分成三個步驟。

8.3.1.1 步驟一

這裡以 OsYield()說明任務交換的動作細節。當某任務主動要把 CPU 讓給別的任務使用時，呼叫 OsYield。

```
01 Task(){
02     ...
03     OsYield();
04     ...
05 }
```

8.3.1.2 步驟二

OsYield 呼叫函式 OsTaskSwitchOut。

FORMOSA_V1\OSKERNEL\OS\OS_CORE.C

```
01 void OsYield(void){
02     OsTaskSwitchOut();
03 }
```

8.3.1.3 步驟三

函式 OsTaskSwitchOut 會使用 INT 0x30 執行任務交換的動作。

FORMOSA_V1\OSKERNEL\SYSTEM\ENTRY.ASM

```
01 _OsTaskSwitchOut:
02     int 0x30
03     ret
```

8.3.1.4 中斷 0x30

中斷 0x30 是筆者為了執行任務級任務切換而設計的中斷號碼。中斷 0x30 本身也是一個軟體中斷，是一個由任務本身主動引發的中斷，目的是為了執行任務切換。當中斷 0x30 執行時，CPU 第一時間會將 EFLAGS、CS、IP 等暫存器值堆到堆疊中。之後，CPU 就會根據中斷描述子表中第 0x30 的中斷描述子的設定值，跳到相對應的中斷執行器執行中斷。

中斷 0x30 的中斷執行器的部份程式碼:

FORMOSA_V1\OSKERNEL\SYSTEM\ENTRY.ASM

```
01 SoftwareInt48Executor:
02     ....
03     mov esp,KernelStackTop
04     call dword [SoftwareInt48Handler]
05     mov eax,[_OsTaskNext]
06     mov [_OsTaskCurrent],eax
07     mov esp,[eax]
08     dec dword [_OsIntNesting]
09     popad
10     iretd
11     ....
```

行號	說明
01	SoftwareInt48Executor 是 0x30 的中斷執行器的符號名稱。
03	取得核心堆疊，做為中斷函式執行時的堆疊。
04	呼叫 INT 0x30 的中斷處理函式。
05	取得下一個任務的任務控制結構位址值。
06	更新目前的任務結構指標值。
07	取得新的任務的堆疊位址值。
08	巢狀式中斷計數器減 1。
09	從堆疊回復 CPU 的暫存器值。
10	中斷回返。回返後，就會在新的任務中執行。

執行中斷處理函式之後，下一個任務就已經被放在任務結構指標中了。再過幾行程式碼後，任務交換的動作就完成了，只是好像還沒有看到下一個任務是怎麼被找到的。這個答案是在 INT 0x30 的中斷處理函式 InterruptSoftwareInt48Handler。函式 InterruptSoftwareInt48Handler 是 INT 0x30 的中斷處理函式，負責執行

OsSchedulerFindNextIsr。OsSchedulerFindNextIsr 會找到下一個要執行的任務，並且將該任務的結構位址值放入 OsTaskNext 中。OsTaskNext 就是下一個任務的控制結構指標。函式 OsSchedulerFindNextIsr 隸屬於排程器，會在排程器中說明。

FORMOSA_V1\OSKERNEL\SYSTEM\INTERRUPT.C

```
01 void InterruptSoftwareInt48Handler(void){
02     OsSchedulerFindNextIsr();
03 }
```

8.3.2 中斷級任務交換

中斷級任務交換是由硬體引發的任務交換。中斷的來源是可程式中斷控制器 8259。早期的電腦中，只有一個 8259，可以提供 8 個中斷源。現在的電腦中，都會有兩個 8259 串接，以提供 15 個外部中斷源。目前本作業系統使用計時器中斷，實現分時多工。

這裏使用計時器中斷的例子來說明中斷級任務切換的動作細節。當計時器中斷發生時，CPU 會主動讀取 8259 可程式中斷控制器的中斷號碼，並執行該中斷的中斷執行器。中斷執行器的執行過程中，會呼叫安裝在中斷處理器的中斷處理函式 TimerHandler，而 TimerHandler 就是計時器的中斷處理函式。

中斷級任務交換的動作分三個步驟。

8.3.2.1 步驟一

CPU 讀取 8259 中斷號碼，執行計時器的中斷執行器。

FORMOSA_V1\OSKERNEL\SYSTEM\ENTRY.ASM

```
01 Irq0Executor:
02     ....
03     call dword [Irq0Handler]
04     ....
05     ret
```

行號	說明
01	CPU 讀取 8259 中斷號碼，執行計時器的中斷執行器。
03	中斷執行器呼叫計時器的中斷處理函式 TimerHandler。
05	中斷執行器使用 RET 跳躍到中斷中斷出口點，執行中斷結束程序。

這裡將 RET 解釋成程式跳躍而不是函式回返，因為程式使用 RET 做變相的跳躍行為。外表行為看起來是函式回返的動作，但事實上是跳躍到外部中斷出口點，執行中斷結束程序。

8.3.2.2 步驟二

中斷執行器呼叫計時器的中斷處理函式 TimerHandler。

```
   FORMOSA_V1\OSKERNEL\DRIVERS\TIMER.C
01 void TimerHandler(void){
02     ....
03     if(OsSchedulerCtrl.TaskSwitchTimer==0){
04         OsSchedulerFindNextIsr();
05     }
06     ....
07 }
```

行號	說明
03	如果任務交換時間已到，執行排程器。
04	排程器會去尋找下一個可執行的任務，有找到 下一個可以執行的任務時，排程器會設定任務交換旗號 OsTaskSwitch 的值為 1，表示要執行任務交換。

8.3.2.3 步驟三

外部中斷的中斷執行器使用 RET 跳躍至外部中斷出口點。任務交換的動作會在外部中斷出口點完成。關於外部中斷出口點的說明，請參考 9.2.7 外部中斷結束出口點一節。

```
   FORMOSA_V1\OSKERNEL\SYSTEM\ENTRY.ASM
01 IrqExit:
02     mov eax,[_OsTaskSwitch]
03     cmp eax,1
04     jne L_IrqExitNoSwitchTask
05 L_IrqExitSwitchTask:
06     ....
07     iretd
08
09 L_IrqExitNoSwitchTask:
10     ....
11     iretd
12
13 IrqExitForIrqReenter:
14     ....
15     iretd
```

行號	說明
01~11	一般中斷出口點。
02	取得任務交換旗號值。
03	比較任務交換旗號值與 1。
04	如果任務交換旗號值不為一時，跳至不執行任務交換出口點。
05~07	執行任務交換出口點。
09~11	不執行任務交換出口點。
13~15	巢狀式中斷出口點。

8.4 多工管理

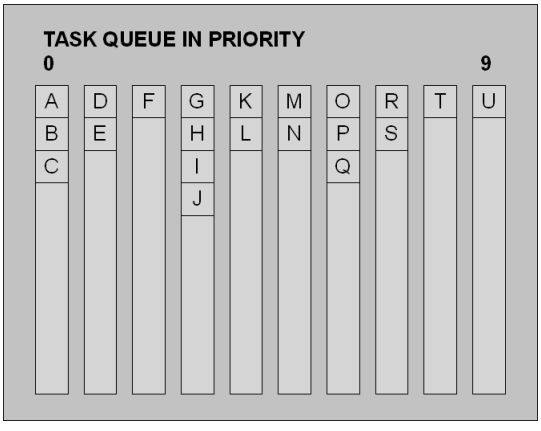
多工管理是多工核心運作的心臟，目的是要讓 CPU 資源可以得到妥善的利用，以滿足作業系統工作上的需要。筆者把多工管理分三個部分，任務管理、事件管理和資源管理。

8.4.1 任務管理

任務管理把任務的狀態分成四個狀態，分別是準備態、執行態、懸置態和閒置態。

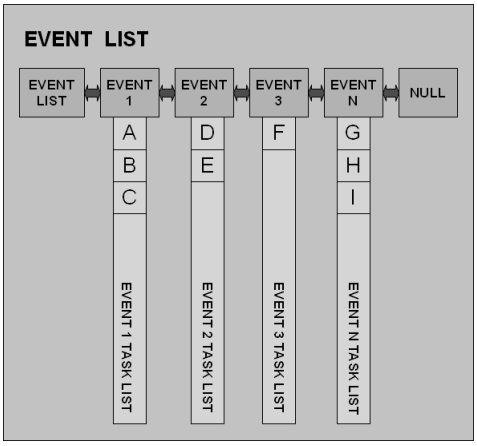
任務狀態	說明
準備態	表示任務已經在優先權佇列中，等待執行。
執行態	表示任務正在執行中。
懸置態	表示任務正在等待事件的發生或資源的取得。只要事件發生或資源取得後，該任務就會回到準備態，等待執行。
閒置態	閒置態表示任務不在執行佇列中，但也沒有被刪除。只要作業系統喚醒它，它就可以立即回到準備態，等待執行。

本作業系統把準備態的任務分成十個優先權佇列，排程器會照著優先權，尋找下一個準備要執行的任務並執行之。在執行下一個任務之前，會將目前執行的任務設定為準備態，並安置到所屬的優先權佇列中，等待下一次的執行。



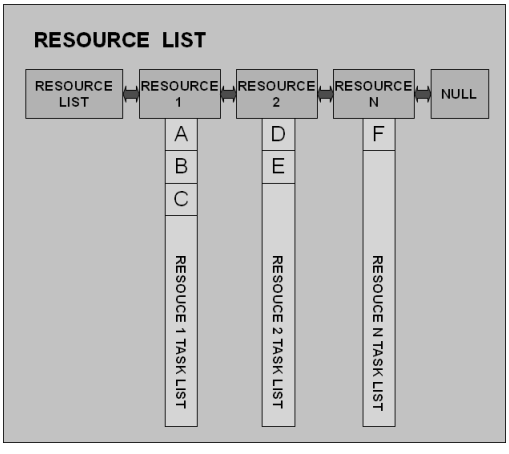
8.4.2 事件管理

事件管理是作業系統的神經網路，事件的管理是要讓事件能以最快的速度通知到相關的任務，使任務可以在最短的時間內做出回應。每個事件都要註冊到作業系統中，如此可以讓所有的事件集中管理。事件集中管理的好處就是可以知道有哪些任務在事件等待佇列中，也可以讓時間事件監視任務的等待時間是否耗盡。



8.4.3 資源管理

想像一下，如果有兩個檔案總管正在做複製檔案的動作，而檔案系統只有一個。這時它們必須互相等待對方的動作完成。之後才能取得檔案系統的資源，並繼續進行自己的工作。一個檔案總管不能永遠獨占檔案系統，否則另一個檔案總管就會形同當掉一樣，無法工作。當一個檔案總管在工作時，另一個檔案總管的複製工作就會稍微暫停，只是使用者察覺不到，以為是兩個檔案總管同時在複製檔案一樣。這就是資源管理的目的，讓所有的任務都能夠順利的取得資源並歸還資源。等待資源的任務會被懸置在該資源的等待佇列中，除非等待時間耗盡，否則該任務會一直等下去，直到取得該資源為止。這樣的方式也許會造成死結，不過因為有時間做為等待條件，所以應該還好。



8.5 多工運作

多工的運作是多工管理的應用，筆者用四個範例說明。第一個範例是排程器，第二個範例是計時器，第三個範例是事件管理器，第四個範例是資源管理器。這四個範例都有它們所代表的意義，了解它們的運作原理很重要。

8.5.1 排程器

排程器是用來尋找下一個準備態的任務。在任務管理中，任務被安置在不同的優先權佇列中，以先進先出的秩序排列。排程器使用優先權演算法得到下一個要執行的任務的優先權，並到該優先權佇列中，取出下一個要執行的任務控制結構。如果有找到下一個可以執行的任務時，就要把目前執行的任務安置到所屬的優先權佇列中，等待下一次的執行。如果排程器沒有找到下一個可以執行的任務時，就不做任務交換。當作業系統中只有一個任務在執行時，就是這樣的狀況。

目前使用到排程器的地方有二個，一個是計時器中斷，另一個是任務級任務交換。

8.5.1.1 計時器中斷呼叫排程器

在計時器中斷處理函式中，呼叫排程器。

FORMOSA_V1\OSKERNEL\DRIVERS\TIMER.C

```
01 void TimerHandler(void){
02     ....
03     if(OsSchedulerCtrl.TaskSwitchTimer==0){
04        OsSchedulerNextTaskFind();
05     }
06     ....
07 }
```

行號	說明
03	排程器的任務交換時間到，執行排程器。
04	排程器找出下一個可執行的任務，在離開中斷的時候，會做任務切換的動作。

8.5.1.2 任務級任務交換呼叫排程器

在任務級任務交換的中斷處理函式中，呼叫排程器。

FORMOSA_V1\OSKERNEL\SYSTEM\INTERRUPT.C

```
01 void InterruptSoftwareInt48Handler(void){
02     OsSchedulerNextTaskFind();
03 }
```

行號	說明
1	任務級任務交換 INT 0x30 的中斷處理函式。
2	排程器找出下一個可執行的任務，在離開中斷的時候，會做任務切換的動作。

8.5.2 計時器

本作業系統有一個計時器。這個計時器的計時功能可以用兩種方式達到，一個方式是使用多工核心的時間延遲函式 OsTimeDelay，另一個方式是使用作業系統的時間計數器。

8.5.2.1 使用時間延遲函式

OsTimeDelay(500) 的意思是說，讓這個任務懸置 500 毫秒。之後，再繼續執行這個任務。在時間未到之前，任務都會懸置在計時器的事件等待佇列中。在等待時間耗盡之後，任務就會重新回到優先權佇列中，等待執行。

FORMOSA_V1\OSKERNEL\TASKS\TASKB.C

```
01 void TaskB(void){
02     ....
03     while(1){
04         OsTimeDelay(500);
05         ....
06     }
07 }
```

8.5.2.2 使用時間計數器

這裡以 TASKC 做為時間計數器的程式範例。時間計數器的使用方法分成二個步驟。

8.5.2.2.1 安裝時間計數器

將任務的計時器安裝到多工核心的時間管理器。

FORMOSA_V1\OSKERNEL\TASKS\TASKC.C

```
01 void TaskCInit(TASKC_CONTROL *pTaskc){
02     pTaskc->MyTimer.Timer=1000;
03     OsTimeTimerPut(&pTaskc->MyTimer);
04 }
```

行號	說明
02	設定 TASK C 的時間計數器的時間計數值為 1000 毫秒。
03	安裝時間計數器。

8.5.2.2.2 應用時間計數器

在任務中實現時間計數器的應用程式碼。

FORMOSA_V1\OSKERNEL\TASKS\TASKC.C

```
01 void TaskC(void){
02     TASKC_CONTROL *pTaskc;
03
04     pTaskc=(TASKC_CONTROL *)OsMemoryAllocate(sizeof(TASKC_CONTROL));
05     if(pTaskc==(TASKC_CONTROL *)NULL) OsTaskFinish();
```

```
06      TaskCInit(pTaskc);
07      while(1){
08          if(pTaskc->MyTimer.Timer==0){
09              pTaskc->MyTimer.Timer=1000;
10              GuiStringPrint("#");
11          }
12          else OsYield();
13      }
14  }
```

行號	說明
02	宣告任務的應用程式結構指標。
04	向記憶體管理器取得任務的應用程式結構記憶體，用以執行任務。
05	如果記憶體取得失敗，結束 TASKC 任務的執行。
06	TASKC 任務初始化。
07~13	TASKC 任務執行迴圈。
08~11	當 TASKC 的計時器為 0 時，重新設定計時值並顯示 # 字號的任務訊息。
12	當 TASKC 的計時器時間未到之前，執行任務級任務切換，把 CPU 運算資源讓給別的任務使用。

任務中的程式碼的意思是當 MyTimer.Timer 不等於 0 的時候，就把呼叫 OsYield，把自己切換掉，讓別的任務執行。等到 MyTimer.Timer 等於 0，任務也再次被執行時，任務就重新設定 MyTimer.Timer 的值，並顯示信息於螢幕上。這個方式和 OsTimeDelay(TimeInMs) 不一樣的地方在於任務本身還會繼續待在優先權佇列中，被執行。只是每次執行的時候，就又會呼叫 OsYield，把自己切換掉。這樣的動作會浪費 CPU 的運算資源。如果只是要做單純的時間等待，使用 OsTimeDelay 的方式會比較好。一樣的目的有不一樣的做法，了解動作的原理，有助於寫出更有效率的程式碼，這一點很重要。

時間計數器的功能比較適合於複雜的條件環境，此時使用 OsTimeDelay 並不適合。面對這樣的狀況，作業系統可以用更高明的處理，例如利用事件管理器做事件等待條件並設定等待時間。在事件與時間等待條件都未滿足之前，把任務置於事件的懸置佇列中，其原理和 OsTimeDelay 的做法類似。時間本身也是事件的一種，每種事件都有其特殊性質，使用方式也許不一樣，但本質上都是事件，可以使用事件管理器處理之。

8.5.3 事件管理器

事件管理就像是人體的神經系統，如果體積太過龐大，事件傳遞太慢，就會影響作業系統運作的敏捷度。事件可以觸發任務的執行。有些任務是事件導向的任務，事件發生時，任務就從事件懸置佇列中被喚醒並執行。處理事件完成後，任務就又回到懸置佇列中，等待下一次的事件發生。

8.5.3.1 事件管理器的使用方法

以鍵盤事件做為事件管理的範例。使用鍵盤的中介軟體是 CONSOLE，使用 CONSOLE 的應用程式是 SHELLTASK。當沒有鍵盤的輸入時，SHELLTASK 會因為使用 CONSOLE 的關係而被移至鍵盤事件的任務等待佇列中，當使用者使用鍵盤而產生按鍵信號時，SHELLTASK 就會從鍵盤事件的任務等待佇列，被移到優先權佇列中，等待執行。

8.5.3.1.1 註冊事件

將鍵盤事件註冊到多工核心的事件管理器。

FORMOSA_V1\OSKERNEL\DRIVERS\KEYBOARD.C

```
01 void KeyboardInit(void){
02     ....
03     OsEventAdd(&KeyboardCtrl.KeyboardEvent,
04                &KeyboardName,
05                OS_EVENT_KEYBOARD);
06     ....
07 }
```

行號	說明
03	事件註冊函式 OsEventAdd，第一個參數是事件控制結構位址值。
04	第一個參數是事件名稱字串。
05	KEYBOARD 事件代號。

8.5.3.1.2 使用事件

SHELLTASK 中執行 CONSOLE，提供一個命令列的使用者介面。

FORMOSA_V1\OSKERNEL\TASKS\SHELLTASK.C

```
01 void ShellTask(void){
02     ....
03     while(1){
04         ConsoleSvc(&pShell->Console);
05     }
06 }
```

行號	說明
03~05	SHELLTASK 任務執行迴圈。
04	呼叫 CONSOLE，而 CONSOLE 的資料結 構來自於 SHELLTASK 本身的資料結構，這樣的方式可以允許多個 SHELLTASK 同時執行，並共同使用同一個中介軟體 CONSOLE。在視窗作業系統中，這樣的情形很普遍，但這個任務不是給視窗作業系統使用的，所以只允許執行一個 SHELLTASK。

在中介軟體 CONSOLE 的命令列參數接受器中，呼叫事件等待函式 OsEventWait。在這裡 OsEventWait 需要二個參數，鍵盤事件和時間等待條件。這表示任務要等待

的事件是鍵盤輸入，如果太久都沒有鍵盤輸入的話，時間等待條件還是會讓任務回到優先權佇列，等待執行，不會永遠懸置在事件等待佇列中。

FORMOSA_V1\OSKERNEL\MIDDLEWARE\CONSOLE.C

```
01 void ConsoleArgumentSvc(CONSOLE *pConsole){
02     while(KeyboardKeyBufferCheck()==BUFFER_EMPTY){
03           OsEventWait(&pOsDriverKeyboard->KeyboardEvent,1000);
04            ....
05     }
06     ....
07 }
```

行號	說明
02	當鍵盤資料緩衝區為空時，呼叫鍵盤事件等待。
03	呼叫事件等待函式，第一個參數是鍵盤事件位址值，第二個參數是時間條件 1000 毫秒。

8.5.3.1.3 處理事件

當有鍵盤輸入時，任務要如何回到優先權佇列呢？答案是 ROOTTASK。ROOTTASK 會一直監聽鍵盤事件的狀況，一但有鍵盤輸入時，就會呼叫 OsEventUp，將所有懸置在該事件的任務轉移至優先權佇列中，等待執行。

FORMOSA_V1\OSKERNEL\OS\ROOTTASK.C

```
01 void RootTask(void){
02     ....
03     if(KeyboardKeyBufferCheck()!=BUFFER_EMPTY){
04         OsEventUp(&KeyboardCtrl.KeyboardEvent);
05     }
06     ....
07 }
```

行號	說明
03	檢查鍵盤資料緩衝區不為空時，呼叫 OsEventUp。
04	OsEventUp 釋放在鍵盤事件等待佇列中的所有任務，放回到優先權佇列中，等待執行。

8.5.3.2 不使用事件的對照組

當使用到鍵盤事件的時候，SHELLTASK 都會待在鍵盤事件等待佇列中，除非有鍵盤輸入，否則 SHELLTASK 很少執行，也因此不會浪費 CPU 的運算資源。透過事件管理，CPU 的資源得到更有效率的利用。筆者使用另外兩種寫法來當對照組，大家就更能夠了解事件管理的重要性了。

8.5.3.2.1 對照組一

沒有使用事件管理的時候，使用 OsYield 函式也可以有節省 CPU 運算資源的效果，只是使用事件管理的方式會節省更多 CPU 運算資源。因為使用 OsYield 函式時，任務還是會繼續待在優先權佇列中，等待執行。只是當任務被執行的時候，就又馬上

被切換掉，而這種被執行後又馬上被切換掉的行為本身就是浪費 CPU 運算資源的行為。

FORMOSA_V1\OSKERNEL\MIDDLEWARE\CONSOLE.C

```
01 void ConsoleArgumentSvc(CONSOLE *pConsole){
02     ....
03     while(KeyboardKeyBufferCheck()==BUFFER_EMPTY) OsYield();
04     ....
05 }
```

8.5.3.2.2 對照組二

既沒有使用事件管理，也沒有使用 OsYield 函式的話，SHELLTASK 就會一直執行下去，直到被排程器換掉為止。這樣的方式是最簡單的方式，也是最消耗 CPU 運算資源的方式。

FORMOSA_V1\OSKERNEL\MIDDLEWARE\CONSOLE.C

```
01 void ConsoleArgumentSvc(CONSOLE *pConsole){
02     ....
03     if(KeyboardKeyBufferCheck()==BUFFER_EMPTY) return;
04     ....
05 }
```

筆者觀察 WINDOWS XP 的行為反應，打開幾個文字編輯器，然後檢查 CPU 使用率，居然不到 1%。筆者猜想，文字編輯器的任務應該就是用事件管理器的方法處理。無論使用者開了幾個文字管理器，它們都在桌面的任務等待佇列中。當使用者選擇了某個文字編輯器的時候，該文字編輯器就從任務等待佇列被提出執行。之後，該任務就會處於等待使用者按鍵輸入的狀態中，進入按鍵事件等待佇列。此時，作業系統的多工核心幾乎沒有任務在跑，所以 CPU 的使用率極低，甚至逼近於 0%。當作業系統察覺到這樣的情形時，也可以進入省電模式，讓 CPU 進入休眠狀態，以降低電腦的功率消耗。這就是多工作業系統中事件管理器的妙用。

8.5.4 資源管理器

資源管理的目的是要讓作業系統的資源可以得到妥善的運用。當一個資源要被很多個任務使用時，所有任務都必須經過獲取資源、使用資源和歸還資源這三個過程。沒有立即取得資源的任務並不需要立即認定取得資源失敗，因為可能有別的任務正在使用該資源，只要經過一個時間的等待之後，應該就可以取得資源，並使用資源。

8.5.4.1 資源管理器的使用方法

筆者使用 GUI 做為資源管理的範例，所有要在螢幕上顯示訊息的任務，都要先獲得 GUI 資源，沒有獲得資源的任務就會被移至 GUI 的懸置佇列中。當使用 GUI 的任務將 GUI 資源歸還後，下一個在 GUI 懸置佇列中等待的任務就會獲得 GUI 資源並被移至優先權佇列中，準備執行。

8.5.4.1.1 註冊資源

將 GUI 資源註冊到多工核心的資源管理器中。

FORMOSA_V1\OSKERNEL\MIDDLEWARE\GUI\GUI.C

```
01 void GuiInit(void){
02     OsResourceAdd(&GuiCtrl.GuiResource,&GuiName,OS_RESOURCE_GUI);
03 }
```

行號	說明
02	呼叫資源加入函式，將資源註冊到資源管理器。

8.5.4.1.2 使用資源

使用資源之前，要先取得資源，取得資源成功之後，才能使用資源。使用資源完畢之後，必須歸還資源，好讓下一個任務可以取得並使用該資源。這是資源管理器的使用方法。

舉例來說，在 GuiPrintChar 中，呼叫 OsResourceAllocate 取得資源，如果不能立即取得資源的話，該任務就會懸置在該資源的等待佇列中。懸置的任務最終如果有取得資源，就可以繼續使用該資源的功能，否則就要直接回返。使用資源完畢，必須呼叫 OsResourceRelease 將該資源釋放。如果沒有做釋放資源的動作，該資源就不會再被別的任務使用。這也是必須有時間等待條件的原因，因為任務不應該被資源綁住而無法執行其他工作。

FORMOSA_V1\OSKERNEL\MIDDLEWARE\GUI\GUI.C

```
01 void GuiCharPrint(U8 CharValue){
02     if(OsResourceAllocate(&GuiCtrl.GuiResource,1000)==OS_FALSE) return;
03     GuiTextCharPut(CharValue);
04     OsResourceRelease(&GuiCtrl.GuiResource);
05 }
```

行號	說明
02	取得 GUI 資源。
03	GUI 呼叫字元顯示函式，將該字元顯示於螢幕上。
04	釋放 GUI 資源。