第10章 作業系統

在本章中，我們將介紹作業系統的理論基礎，由於作業系統的目的是為程式打造一個方便的執行環境，讓程式設計師可以很方便的撰寫程式，並且讓使用者可以很方便的使用電腦，因此，不論是程式師或一般使用者，通常都相當依賴作業系統。

在本章中，我們將介紹作業系統的基本模組，包含行程管理 (10.2 節)、記憶體管理 (10.3 節)、檔案與輸出入 (10.4 節) 等，並且在實務案例 (10.5 節) 中，說明 Linux 作業系統的使用與設計方式，以實際體驗作業系統的用途。

10.1 簡介

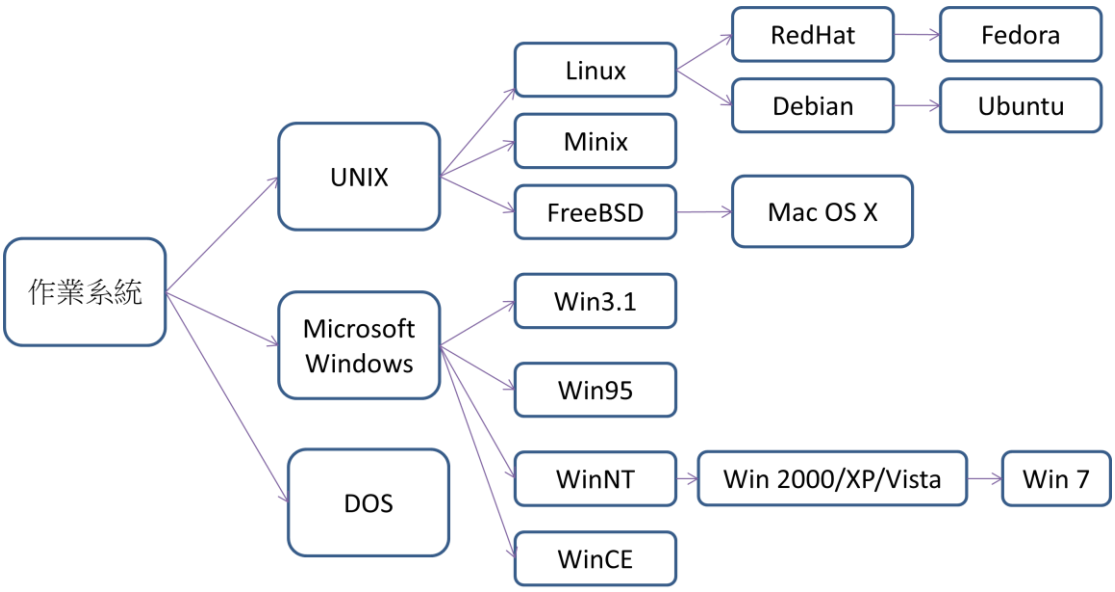

圖 10.1 今日常見的作業系統家族圖

對於學習過作業系統課程的學生而言，作業系統往往仍是很神祕與抽象的，這是因為作業系統的設計太過複雜，因此，作業系統的課程往往流於傳授理論，但卻無法與實務配合，這是作業系統課程相當難以跨越的障礙。

然而，在現實生活中，程式設計師往往又太過依賴作業系統而不自知。雖然程式師常常使用電腦，但是因為一開機後就進入作業系統，因此，通常無法想像沒有

作業系統會是甚麼樣的狀況。這就好像人們生活在充滿空氣的環境中，反而感覺不到空氣的存在一樣，甚至完全無法察覺其中還有氧氣、氮氣和二氧化碳的區分。

目前，許多電腦的使用者，可能一開啟電腦之後，就進入視窗模式。然後，可能一邊聽音樂一邊上網、上網時同時開啟許多網頁，甚至其中有些網頁 (例如 Youtube) 正在播放影片。要達成這種使用狀況，必須要有一個強力的作業系統，像是 Windows、Linux 等作業系統就能支應此種使用狀況，但是像傳統的 DOS 就很難支應，因為，DOS 並不是一個多工作業系統，無法同時執行多個程式。

作業系統的目的，其實就是要讓程式師很方便的寫程式，而不會感覺到任何困難，然後讓使用者很方便的使用電腦，能盡情的發揮電腦的功能。現今的多工作業系統，幾乎都很成功的達成了這些功能。但是，也正因為如此，程式師與使用者都幾乎感覺不到『它』（也就是作業系統）的存在。

現今的作業系統，通常透過『行程管理』、『記憶體管理』、『輸出入系統』、『檔案系統』、『使用者介面』等五大功能模組，打造出方便的程式與使用環境。以下，我們將先簡短的說明這五大模組的用途，以便建立作業系統的整體概念。

- 『行程管理系統』的目的，就是要打造出一個環境，讓任何程式都能輕易的執行，而不會受到其他程式的干擾，就好像整台電腦完全接受該程式的指揮，彷彿沒有其他程式存在一般。

- 『記憶體管理系統』的角色也具有類似的功能，其目的是在打造出一個方便的記憶體配置環境，當程式需要記憶體時，只要透過系統呼叫提出請求，就可以獲得所要的記憶空間，完全不用去考慮其他程式是否存在，或者應該用哪一個區域的記憶體等問題，就好像整台電腦的記憶體都可以被該程式使用一般。

- 『輸出入系統』的功能是將輸出入裝置包裝成系統函數，讓程式師不用直接面對複雜且多樣的裝置。作業系統的設計者會定義出通用的介面，將這些硬體的控制包裝成系統函數，讓輸出入作業變得簡單且容易使用。

- 『檔案系統』則是輸出入系統的進一步延伸，主要是針對永久儲存裝置而設計的，其目的是讓程式師與使用者能輕易的存取所想要的資料，而不需要考慮各種不同的儲存裝置的技術細節。程式設計師只要透過作業系統所提供的『檔案輸出入函數』，就能輕易的存取這些檔案。而一般使用者也只要透過『命令列』或『視窗介面』，就可以輕易的取得或儲存檔案，這是作業系統

當中設計得非常成功的一個模組。

- 最後，『使用者介面』則是提供程式師與一般使用者一個方便的操作環境，讓使用者感覺整台電腦都在其掌控之下，毫無障礙的運行著。當使用者想要某個功能時，能夠很輕鬆的找到該功能以執行之。在早期，使用者通常透過命令列介面以指令的方式使用電腦，但是，這種方式並不容易使用。當視窗介面被發明後，逐漸取代命令列介面，成為主要的使用者介面，視窗介面無疑是使用者介面的一個重要里程碑。

10.2 行程管理

對程式設計師而言，行程是個很難理解的概念，這並不是因為行程的概念很難學習，而是因為太容易使用，反而難以感覺到行程的存在。就像物理學當中的電磁波一樣，看不到又摸不著。要理解電磁波，只能依靠想像力。

當程式設計師寫完一個程式，編譯後會產生可執行檔 (像是 MS Windows 中的 .exe 檔，例如：test.exe)，此時，只要透過使用者介面執行，原本在硬碟中的執行檔就會被載入到記憶體執行，而這個正在執行中的程式，就是所謂的行程。

正因為行程的使用是如此的自然，反而使程式設計師難以理解，對程式師而言，『程式』、『執行檔』與『行程』這三個概念，往往難以區分，然而，對於系統程式的學習者而言，這三者的區分是相當重要的。

簡單來說，程式是撰寫者用編輯器所撰寫出來的文字型檔案，在程式寫完後，程式師會用組譯器或編譯器將程式轉換成可執行檔，作業系統可以將執行檔載入到記憶體後開始執行，這個執行中的程式，就稱為行程。

在現今的電腦與作業系統當中，通常具備同時執行多個程式的能力，這種能力稱為『多工』(Multtasking)，多工能力是行程管理系統的重點，現今的電腦系統非常仰賴多工機制，像是 MS. Windows、Linux 等系統都具備 Multitasking 的能力，這種具備多工機制的電腦系統通常能力較強，而且比較好用。

在 Windows 或 Linux 等現今的作業系統中，你可以不斷的啟動並執行新程式。因此常會有數十個程式同時在系統中執行的情況，這種能力就是 Multitasking。

最簡單的行程管理系統是一個載入器，該載入器會從硬碟中載入程式並執行之，這種系統通常稱為『單工』 (Single Task) 系統，因為每次只能執行一個『工作』

(Task)，等到該工作完成後才可以載入下一個程式執行。

DOS是一個典型的單工系統,是一個由載入器與檔案系統組合而成的作業系統，因此才稱為 DOS (Disk Operation System)。在 DOS 系統當中，一次只能執行一個程式，因此使用者無法一邊用 DOS 聽 mp3，然後一邊上網。

圖 10.2 顯示了多工與單工系統的對照狀況，圖 10.2 (a) 是一個完全沒有作業系統的電腦系統，整個系統就只有是一個行程，在這種系統當中，通常不能載入外部程式，因為整個系統是『燒死』在記憶體當中的。一般而言，簡易型的嵌入式系統通常採用此種模式，在此種模式當中，整個系統在製造時就將程式燒入到唯讀記憶體 (ROM) 當中，當開機鍵被按下，或者電源被插上時，就直接進入該程式，整個系統都在該程式的控制當中，這是一種連作業系統都沒有的單行程系統。

圖 10.2 (b) 的系統雖然有作業系統，但是只能執行單一行程。此種作業系統稱為單工作業系統。由於每次只能執行一個行程，因此，這樣的行程管理系統其實就只是一個載入器而已。單工的行程管理系統，負責將程式載入，然後將 CPU 控制權交由該行程執行。在該行程執行完畢後，會將控制權交還給作業系統，如此，作業系統才能再度載入下一個行程以便執行。

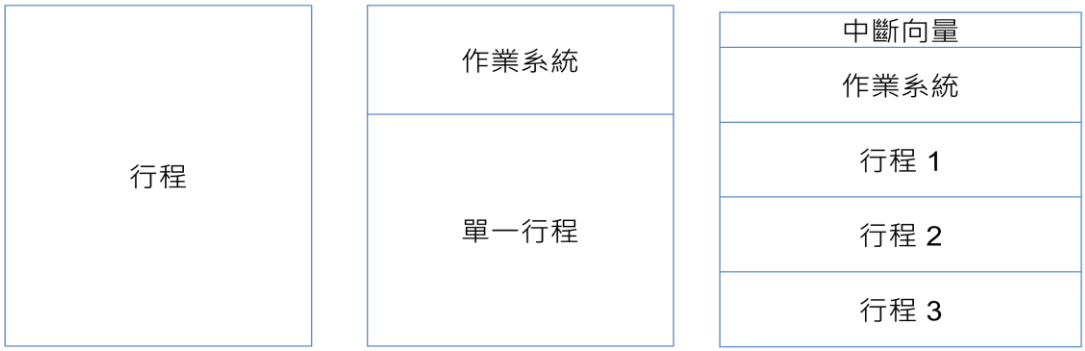

圖 10.2 單行程系統與多工系統之比較

圖 10.2 (c) 的系統則是多工作業系統，因為記憶體當中同時容納了許多行程，這些行程在作業系統的管控之下同時執行，每個行程都可以使用到 CPU、記憶體、硬碟等資源。在這種系統當中，中斷向量扮演了一個重要的角色，因為要讓作業系統在適當的時候切換行程，使得每個行程都可以被公平的執行到，就必須依賴

中斷機制有效的運作，在適當的時候中斷某些行程的執行，以便讓作業系統取回控制權，以便安排其他行程的執行。

但是，請讀者不要誤解，雖然『多工系統』當中有很多程式在電腦當中執行，但對於單一 CPU 的電腦而言，每一個時間點只有一個行程真正被執行。只是，在多工系統當中，這些行程同時存在記憶體內，由於 CPU 的速度很快，作業系統可以透過『行程切換』的方式，讓每個行程都可以被執行到，而且可以隨意使用所想要的資源，包含記憶體、硬碟等等。

在此種『多工』的環境之下，雖然系統當中有許多行程存在，但每個行程都不會互相干擾，而且都『感覺到』好像擁有整個系統一般。

在多工系統當中，當一個行程因為讀取鍵盤的動作而進入等待狀態時，作業系統就會立即將 CPU 分配給其他行程，但是，由於現代 CPU 的速度非常非常的快，在使用者還沒有來的即按下鍵盤之前，就可以進行數百萬到數千萬次的計算，因此，多工系統可以同時服務數百個行程，而不會讓這些行程等候太久，甚至完全不會感覺到任何的延遲，這就是多工系統的好處。

抽象來看，程式通常只做兩件事情，一個是使用 CPU，一個是使用輸出入裝置 (Input/Output 簡寫為 I/O)。在整個程式的生命周期中，不斷的在這兩個過程之間交替輪流，直到程式結束為止。圖 10.3 顯示了程式的這種交替的行為模式，在該程式當中，雖然開檔、讀檔、關檔的程式碼只佔據了三行，但實際上卻耗費了絕大部分的時間。

使用輸出入的程式	說明

int wc(const char *fname) {
  int words=0;
  FILE *fp=fopen(fname, "r");
  while((ch=getc(fp))!=EOF) {
  if(isspace(ch)) sp=1;
  else if(sp) {
  words++;
  sp=0;
  }
  if(ch=='\n') ++lines;
  }
  printf("共有 %d 個英文詞彙\n", words);
  fclose(fp);
  return words;
}
	計算檔案詞彙數的程式
	
	開檔
	讀取字元
	
	關檔

圖 10.3 程式的行為模式 – CPU 與 I/O 交替運行

對作業系統而言，程式的需求不過就是對 CPU 與輸出入裝置的需求而已。然而，由於 CPU 速度極快，相對而言，輸出入所占用的時間區段很長，於是，作業系統就可以利用某行程在進行輸出入的『空檔』，將 CPU 在神不知鬼不覺的狀況之下，挪給其他行程使用。圖 10.4 顯示了三個行程與作業系統間的執行順序關係，當行程透過系統呼叫進行輸出入 (I/O) 時，作業系統就趁機切換行程，讓另一個行程得以使用 CPU，如此就可以充分利用 CPU，以支援每一個行程的執行。

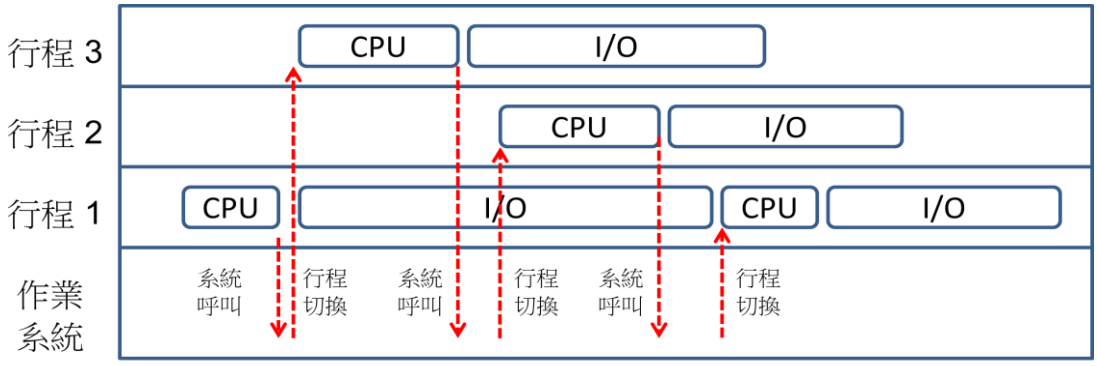

圖 10.4 作業系統利用輸出入的空檔切換行程

有些行程可能會有不正常的執行模式，舉例而言，假如有一個程式撰寫錯誤，不小心寫出如範例 10.1 的無窮迴圈，那麼，這個行程將會霸佔整個 CPU，而且不會進行系統呼叫，此時，作業系統可能會苦等不到行程切換的機會，因而導致整

個系統當機。

範例 10.1 不正常的行程 – 無窮迴圈導致當機

```
int sum=0, i=0;
while (i < 100) {
   sum += i;
}
```

要能處理這種不正常的狀況，必須依賴中斷機制，在作業系統將 CPU 交給一個行程之前，先設定中斷時間點，以便當行程霸佔 CPU 時，作業系統能透過中斷機制取回 CPU 控制權，如此，就能避免行程佔據 CPU 不放的行為。

在現代的作業系統當中，中斷機制是相當重要的硬體配套措施，除了能防止行程霸佔 CPU 之外，還能在硬體輸出入完成之後，中斷目前的行程，以便處理此輸出入，這樣的方式比起另一種『輪詢』輸出入機制[1]而言，會更有效率。

行程的狀態

在多工系統中，一個行程通常有三種主要的狀態，也就是『執行』、『就緒』與『等待』三種狀態，我們可以將『新建』與『結束』等兩個狀態也納入，形成五個狀態的轉換圖，如圖 10.5 所示。

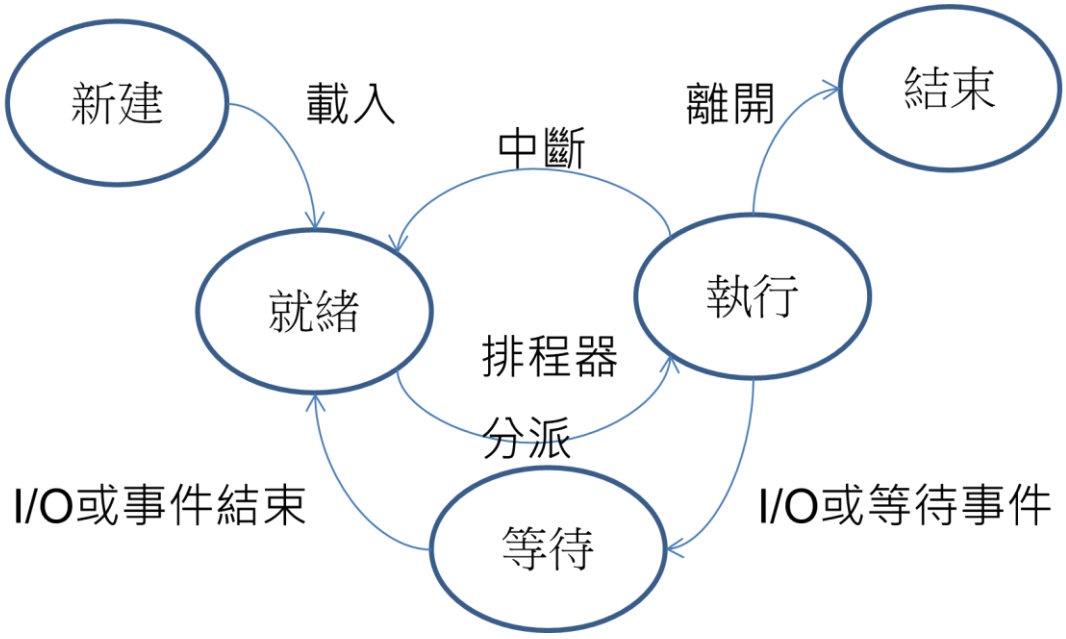

圖 10.5 行程的狀態轉換圖

[1] 有關輪詢機制的原理與實作方式，請參考第 11 章。

當使用者透過命令列或視窗介面啟動程式時，該行程會進入『就緒』狀態，然後被放入『就緒佇列』中，以等待『排程器』的挑選，一旦排程器挑選了一個行程之後，會將狀態改為『執行』，然後真正執行該行程，一旦該行程進行輸出入動作時，會進入『等待』的狀態，直到輸出入工作完成時，又被作業系統放回『就緒佇列』中，等待作業系統的選取。

排程問題

當『執行』狀態的行程因為輸出入而暫停時，假如系統當中有許多『就緒』的行程等待被執行。那麼，到底哪一個行程應該被挑選出來執行呢？這個問題，就是行程管理當中著名的排程問題，這個問題是作業系統效能的關鍵，所以有許多方法被提出以解決此問題。常見的排程演算法有先到先做排程 FCFS (First-Come, First Served)、最短工作優先排程 SJF (Shortest Job First)、最短剩餘優先排程 SRF (Shortest Remaining First)、優先權排程 PS (Priority Scheduling)、循環分時排程 RR (Round-Robin Scheduling) 等等。除此之外，還有綜合性的排程方法，像是多層佇列排程 (Multilevel Queue Scheduling) 與多層反饋佇列排程 (Multilevel Feedback Queue Scheduling) 等。

在實務上最常使用的排程方法是循環分時排程 (Round-Robin Scheduling)，該方法會為行程事先分配一個時間片段 T (Time Slice)，然後才切換到該行程中。在切換到某行程之前，排程系統先設定 T 時間後應發生時間中斷，然後才將 CPU 的控制權交給該行程。

一但時間片段 T 被用盡之後，中斷就會發生，於是排程系統就能透過中斷取回 CPU 的控制權，然後切換到下一個行程，以防止某行程霸佔 CPU 過久而導至其它行程無法執行。

當排程器選定下一個行程之後，必須進行『內文切換』的動作，將 CPU 交給該行程執行，內文切換是將一個行程從 CPU 中取出，換成另一個行程進入 CPU 執行的動作。

由於內文切換的動作經常發生，而且該動作進行時必須進行大量暫存器的存取，所以會以組合語言撰寫，這使得內文切換成為作業系統當中相當神祕的一個動作。內文切換的動作與 CPU 的設計密切相關，往往因平台的不同，內文切換的程式碼也就完全不同，當作業系統被移植 (porting) 到另一個平台之時，內文切換的程式通常必須完全重寫。

當行程完全結束之後，會進入『結束狀態』，接著行程將被釋放，所有該行程所佔據的記憶體與輸出入資源將會被釋放，以便讓新的行程有記憶空間與輸出入資源可用。如此，作業系統得以不斷運行，而不至於產生資源不足的窘境。

通常每一個程式都是一個行程，但是也有可能一個程式會分裂出許多個行程。但即便如此，各個行程之間通常是獨立執行的，互相之間並不會共享資料。如果我們希望讓行程之間能共享某些資料，通常有兩種方式，第一種是讓行程之間能透過作業系統的通訊機制互相聯絡，第二種則是使用執行緒 (Thread) 的機制取代行程，這些執行緒之間由於共用所有記憶空間的緣故，因此可以共用全域變數。

執行緒

執行緒又被稱為輕量級的行程 (Light Weight Process)，但與行程不同的地方是，執行緒會完全共用記憶體空間與相關資源，這使得執行緒在切換時不需要儲存這些資源，所以執行緒的切換相當快速，因為只需要保存暫存器即可。

傳統的兩個行程通常擁有不同的記憶空間，在具有記憶體管理單元 (Memory Management Unit: MMU) 的作業系統中，兩個行程的記憶空間是完全獨立的，各自擁有自己的記憶體映射表。所以在行程切換的同時也必須更換映射表，這樣的動作會消耗許多時間。但是執行緒就沒有這個問題，因此可以進行快速的切換。

執行緒由於共用記憶體空間，因此很容易進行變數共享的動作，這使得執行緒之間很容易互相合作，以便完成某一件共同的任務。舉例而言，假如我們希望撰寫一個網頁伺服器 (Web Server) 時，就可以為每個連線建立一個執行緒以便分別服務，但這些執行緒卻可以共享某些變數，像是『目前連線人數』、『歷史記錄檔』等資源，這是在設計網路程式時常用的一種程式設計手法。

有時程式師為了加快程式的執行速度，會採用多執行緒的設計方式。舉例而言，如果我們想要撰寫一個網路爬蟲 (Crawler) 程式，以便抓取網頁後儲存，就可以建立十個執行緒，分別抓取不同的網頁回來儲存，一但某執行緒抓完一頁後就再度從共享的網址列表中，取得下一個待抓取的網址再度進行抓取，透過這種執行緒的平行機制，可以讓程式的執行速度增快達 10 倍之多。

行程同步機制

當多個行程或執行緒共用某些變數時，如果兩個執行緒 P1, P2 同時修改某個變數 V1 的值為 X1, X2，那麼在修改完畢之後 V1 的值應該是多少呢？這個問題的答案有可能是 X1、也有可能是 X2、甚至還有可能是其他值，這種不確定的情

形被稱為『競爭狀況』，而這些修改共用變數的程式區段則被稱為『臨界區間』。

我們可以利用『禁止中斷』、『支援同步的硬體』或『號誌』等『鎖定機制』，排除兩個執行緒同時進入臨界區間的可能性，以便防止競爭狀況的發生。

最容易實作的鎖定機制是採用『禁止中斷』的方法，該方法是利用組合語言程式，將狀態暫存器的中斷旗標設定為禁止狀態，以防止行程在臨界區間內被中斷，因而避免競爭情況的方法。

範例 10.2 就使用了禁止中斷的方式，鎖定處理器以避免競爭情況的發生。該程式利用 LD R1, MaskLock 指令，將鎖定遮罩載入後，再執行 AND R12,R2, R1 指令，以設定狀態暫存器 R12 的 I, T 旗標為 0，完成鎖定功能。如此，就可以安心的進入臨界區間，修改共用變數，而不需要擔心競爭情況的問題了。

範例 10.2 實作 CPU0 中的鎖定與解鎖機制

```
; 使用禁止中斷的方式進行鎖定
LD R1, MaskLock;將鎖定遮罩載入 R1
AND R12, R12, R1;將狀態暫存器 (SW=R12) 的兩個中斷旗標 I,T 設定為 0

; 臨界區間，可修改共用變數

; 取消禁止中斷的方式進行解鎖
LD R2, MaskUnlock;將解鎖遮罩載入 R2
OR R12, R12, R2; 將狀態暫存器 (SW=R12) 的兩個中斷旗標 I,T 設定為 1
…
MaskLock: RESW 0xFFFFFFF3    ; 鎖定遮罩
MaskUnlock: RESW 0x0000000C ; 解鎖遮罩
```

在範例 10.2 當中，由於已經設定了狀態暫存器的 T, I 位元為 0，因此 CPU 將不會接受任何的中斷，行程也就不可能被強制中斷後換出，當然也就不需要擔心競爭情況的問題了。

當行程離開臨界區間之後，必須盡快的解除鎖定，否則該行程將會佔用 CPU，使得其他行程無法運行。因此在範例 10.2 離開臨界區間之後，必須立刻用 LD R2, MaskUnlock 與 OR R12, R12, R2 等指令，將狀態暫存器的 T, I 位元設定為 1，以允許中斷的發生，因而解除了鎖定機制，讓作業系統得以再度透過中斷切換行程。

由於設定程式透過設定中斷旗標就可以鎖定整個 CPU，造成作業系統無法取回控制權，因此這種指令通常是作業系統專用的特權指令，一般程式只能呼叫作業系統的函數去執行鎖定動作，而不能自行以組合語言進行鎖定。

但是即便鎖定機制可以防止競爭情況的發生，卻無法避免一個執行緒 (例如 P1) 鎖住時把持某些資源，讓另一個執行緒 (P2) 無法取得的情況。更糟糕的是，若此時 P2 也把持了 P1 所需的某些資源，就會導致兩者互相等待，卻又永遠都無法完成的窘境。這種情況就被稱為『死結』。

在作業系統的設計中，死結是相當難以處理的，於是有很多作業系統根本就不處理死結問題，而將問題留給應用程式自行處理。因此，能夠理解死結問題，並且在設計程式時能防止死結的發生，也是程式設計師的責任，雖然大部分的程式並不會遭遇到這樣的問題，但是使用多執行緒模式的程式就可能會遭遇死結問題，這是使用執行緒時必須特別小心的部分。

10.3 記憶體管理

記憶體管理是作業系統的責任之一，有效的管理記憶體除了能提高電腦的效率之外，還可以保護電腦不受到駭客或惡意程式的入侵。當程式提出記憶體的需求時，例如載入器要載入一個程式到記憶體當中，或者是 C 語言程式利用 malloc() 函數要求分配記憶體時，作業系統就必須從可用的記憶體空間中取得足夠大的記憶體空間傳回給需求程式。

當這些記憶體被使用完畢後，像是程式執行完畢將控制權交回給作業系統時，或者是 C 語言程式利用 free() 函數釋放記憶體時，作業系統必須回收這些記憶體，以便分配給其他的程式使用。

在這個過程當中，如何分配記憶體空間將會是一個重要的策略。採取較好的分配策略，可以讓記憶體保持在足夠的狀態，發揮整個系統的效能。因此，作業系統需要有一個好的記憶體分配策略。

記憶體分配策略

分配記憶體時，必須找出足夠大的可用區塊，以便分配給程式。通常記憶體管理系統會利用鏈結串列 (Linked List) 結構記載各個可用區塊的大小，然後遵循特定的配置策略，以尋找足夠大的可用區塊。一個好的記憶體分配策略，應該能有效的管理這些記憶體空洞，快速的分配記憶體，並且延後錯誤的發生時間。在有新的記憶體需求出現時，記憶體分配系統必須決定要將哪一段記憶空間分配給程式，

這就是記憶體管理當中頗具關鍵地位的分配問題。常見的記憶體分配策略有最先符合法 (First-Fit)、最佳符合法 (Best-Fit) 、最差符合法 (Worst-Fit) 與下一個符合法 (Next-Fit) 等等。

1. First Fit (最先符合法)：從串列開頭開始尋找，然後將所找到的第一個足夠大的區塊分配給該程式。
2. Next-Fit (下一個符合法)：使用環狀串列的結構，每次都從上一次搜尋停止的點開始搜尋，然後將所找到的第一個足夠大的區塊分配給該程式。
3. Best-Fit (最佳符合法)：從頭到尾搜尋整個串列一遍，然後將大小最接近的可用區塊分配給該程式。
4. Worst-Fit (最差符合法)：則是將大小最大的區塊分配給程式，以便留下較大的剩餘區塊給其他程式。

研究顯示 First-Fit 的記憶體使用率比 Next-Fit 與 Best-Fit 好，而 Worst-Fit 是四種當中最差的。但是仍然有一些更複雜的配置策略，能得到更快且更好的記憶體使用率，以上所列只是記憶體分配策略中最簡單的四種而已。

堆積空間不足時的處理方式

當 C 語言使用 malloc 分配記憶體時，如果無法找到足夠大的可用區塊，就會產生記憶空間不足的情況，此時系統可以直接回報錯誤，或者試圖處理記憶體不足的狀況。

有兩種方法可以處理記憶體不足的狀況，一種稱為記憶體聚集法，另一種稱為垃圾蒐集法。

記憶體聚集法 (Memory Compaction) 乃是將記憶體重新搬動，以便將分散的小型可用區塊聚集為大型可用區塊，然後再試圖分配給使用程式的方法。但是記憶體聚集的代價非常的高，需要耗費大量的時間搬移記憶體，因此在現代的系統中很少被使用到。

垃圾蒐集法 (Garbge Collection Algorithm) 則是利用程式自動回收記憶體。在使用垃圾收集法的程式中，通常不需要由程式主動釋放記憶體，因為垃圾蒐集系統會在記憶體不足時被啟動，以蒐集記憶體中已經沒有被任何程式變數指到的記憶區塊，然後再將這些區塊標示為可用區塊，以便回收使用。

C 語言當中通常不支援垃圾蒐集演算法，但是在 Java, C# 等語言中則內建了垃圾蒐集機制，該機制會在發現記憶體不足時，自動回收記憶體。因此 Java 與 C# 的

程式通常不需要釋放記憶體，這對程式師而言是很好用的一項功能，可以減輕不少負擔。

記憶體管理單元 (MMU)

記憶體除了可以用來儲存資料之外，還可以用來儲存程式，在程式被啟動之前，必須先被載入到記憶體當中。作業系統必須決定要將程式載入到哪裡？特別是針對多工系統而言，作業系統必須有效的分配記憶體給各個行程，才能將更多的行程同時放入記憶體當中執行，提升多工的能力。

許多高階的處理器會以硬體的方式，支援記憶體管理機制，這種硬體被稱為記憶體管理單元 (Memory Management Unit: MMU)。MMU 可以增進存取的效率並且加強保護機制。

透過 MMU 的硬體，作業系統能夠扮演安全控管的角色，MMU 可以防止木馬程式竊取其他程式記憶體中的資料，也可以防止惡意程式寫入跳躍指令碼到他人的記憶空間中，甚至寫到作業系統的空間中以接管系統控制權等行為。MMU 對系統的安全性有相當大的幫助，像是 x86 系列處理器在 80386 之後就開始採用分段與分頁的硬體機制，以支援記憶體管理與保護的功能。

具有 MMU 的處理器，有能力檢查記憶體位址的合法性，在確定合法後才將指令中的邏輯位址 (Logical Address，又稱虛擬位址 Virtual Address) 映射到真實的記憶體位址 (稱為實體位址 Physical Address)。這樣的能力通常透過某些暫存器達成，像是『重定位暫存器』、『基底暫存器』、『界限暫存器』、『分段表暫存器』、『分頁表暫存器』等等。

重定位暫存器

最簡單的 MMU 是利用一個重定位暫存器 (relocation register)，在記憶體存取指令發出前進行重定位的動作，因而讓載入器不需要負責重定位的功能，以節省載入程式的時間。

圖 10.6 顯示了一個擁有重定位暫存器的 CPU，當載入器將某目的檔載入到記憶體位址 0x1030 之後，載入器不需要進行重定位的動作，只要將重定位暫存器 Base 設定為 0x1030，然後將程式計數器 PC 設定為起始位址就可以開始執行程式。

當指令 LD R1, [R2+0x0100] 被執行時，假如 R2 的值為 0x0212，則 LD 指令存取的記憶體位址應為 R2+0x0100 = 0x0212+0x0100 = 0x0312，這個位址稱為邏輯

位址。邏輯位址 0x0312 與重定位暫存器 Base=0x1030 相加的結果為 0x1342。於是存取的真實位址將會是 0x1342，而非 0x0312 這個邏輯位址。

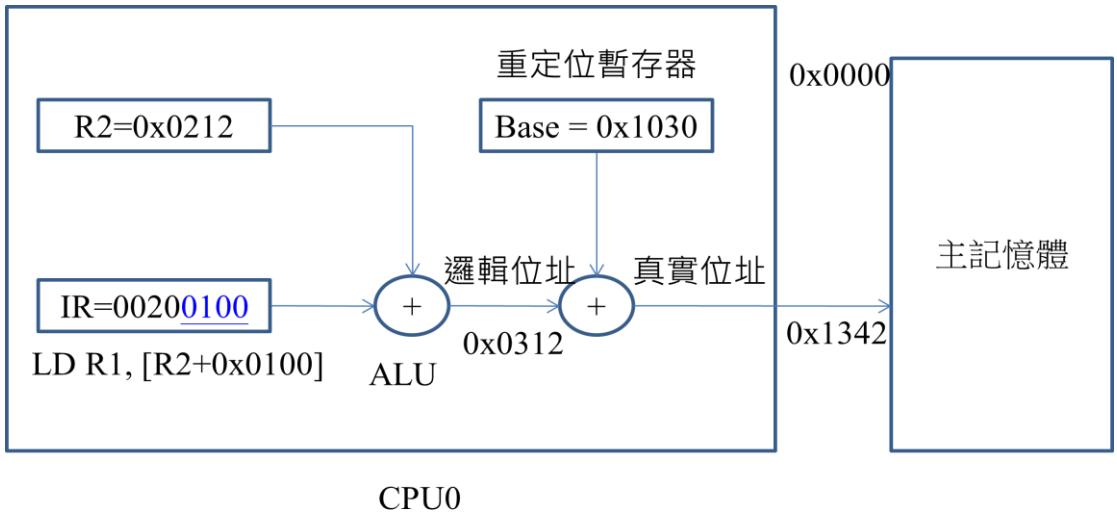

圖 10.6 最簡單的 MMU - 使用『重定位暫存器』

基底界限暫存器

如果我們希望為 CPU 加上安全性的保護的功能，可以使用『基底暫存器』(Base Register)與『界限暫存器』(Limit Register)，以防止程式存取其他程式的記憶體。基底暫存器的功能與前述的重定位暫存器相同，而界限暫存器則可用來防止程式存取範圍超過『界限』。

圖 10.7 顯示了『基底-界限暫存器』架構的示意圖，假如我們擴充 CPU0 以便納入『基底-界限暫存器』架構的 MMU 單元，則當 CPU0 執行記憶體存取指令，例如：ST R1, [R2+100] 時，邏輯位址 R2+100 會被拿來與界限暫存器進行比較，若位於合法範圍才能與基底暫存器相加形成真實位址，然後再利用真實位址進行記憶體存取動作。

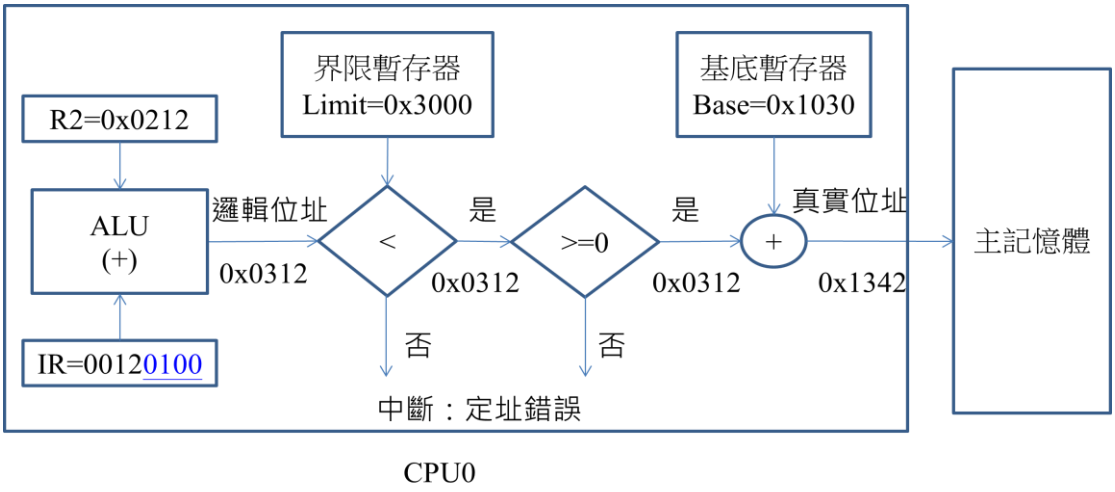

圖 10.7 使用『基底-界限暫存器』提供硬體保護措施

當作業系統透過載入器載入一個程式時，可以先設定好『基底-界限暫存器』的值，以限制該程式的記憶體存取空間。一但該程式中有任何指令存取了範圍以外的記憶體，則電腦硬體會發生記憶體存取範圍錯誤的中斷訊號，這使得作業系統所預先佈下的錯誤處理程式可以被硬體直接啟動，達成記憶體安全控管的功能，防止惡意程式的存取動作。

舉例而言，假如在圖 10.7 的『基底-界限暫存器』環境中，有一個存取指令為 STR R1, [R2+0x3000]，那麼這個指令就會因為超越合法範圍而引發中斷，因為邏輯位址 R2+0x3000 = 0x3212 大於界限暫存器的範圍。當中斷發生後，作業系統就會趁機取回控制權，中止該行程並進行錯誤報告，然後再利用排程系統取得下一個行程，並利用行程切換機制將 CPU 的控制權交給該行程。

『重定位暫存器』與『基底-界限暫存器』都是非常簡易的記憶體管理單元。『重定位暫存器』可以簡化載入過程並加快載入時間，而『基底-界限暫存器』則進一步執行了安全防護的動作，避免惡意的程式存取非法的位址。但是在較為高階的處理器中，通常會支援更複雜的 MMU 功能，其中較重要的 MMU 機制可分為兩類，第一類是分段機制，第二類是分頁機制。

分段機制

分段機制的方法是為每個行程分配一個分段表 (Segment Table)，然後利用硬體的分段表暫存器 (Segment Table Register: STR) 指向該分段表，以便在存取時透過分段表將邏輯位址轉換成實體位址。

使用分段機制的作業系統，在載入器執行程式前，會先將各分段的基底與界限填入到分段表中，然後才開始執行程式。

假設分段表中第 s 個元素由基底 B[s] 與界限 L[s] 組成，當記憶體存取指令執行時，CPU 會先將邏輯位址 [s, d] 透過分段表轉換成真實位址 B[s]+d，然後才進行記憶體存取。在轉換過程中，會先利用分段界限 L[s] 檢查位址範圍是否合法，如果有不合法的情形就會觸發中斷。

舉例而言，在指令 LD R1, [data1+100] 中， data1 代表區段，假如 data1 的分段代碼為 7。那麼，該指令可能被編為 16 進位碼 00170100，其中的第 4 個數字 7 即為分段代碼，而 0100 則是分段位移。圖 10.8 顯示了該指令的轉換過程，指令暫存器 IR 中的 00170100 被分解為 IR=(OP:00) + (R1:1) + (s:7) + (d:0100) 等四個部分，於是 CPU 透過分段表暫存器 (STR) 與分段代碼 7 取得基底 B[s] = 0x1030 與界限 L[s] = 0x3000，接著再將基底 B[s]=0x1030 與位移 d=0x0100 相加後，成為真實位址 0x1130。然後才將 0x1130 傳出到匯流排，以進行記憶體存取動作。

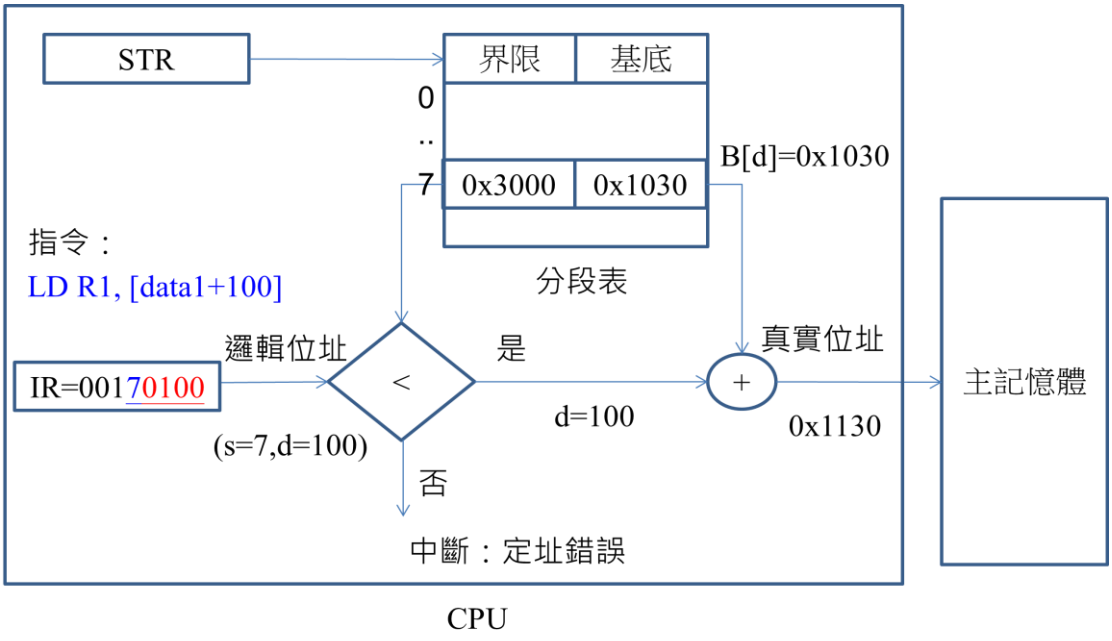

圖 10.8 使用分段表進行記憶體位址轉換的一個範例

如果使用主記憶體儲存分段表，那麼 LD 等指令就需要先取得記憶體中的分段表內容，然後才能存取真實位址的記憶體，這會造成兩次的記憶體存取，嚴重影響系統效能。為了改善此種情況，處理器會利用『位址查找緩衝區』 (Translation Look-aside Buffers:TLB) 這樣的特殊記憶元件 (或稱相關暫存器 associative registers)，以儲存分段表的 (key, value) 配對，加快 TLB 的存取速度。

假如在 TLB 存取時，發現分段代號 s 位於 TLB 中，此時 TLB 會直接輸出 B[s] 與

L[s]，於是就不需要從記憶體中查找出『基底-限制』值，因而加快了存取速度。

分頁機制

目前最常用的 MMU 單元通常採用分頁機制，分頁機制將記憶體分成大小相同的頁，然後利用分頁表將指令中的邏輯位址 (邏輯頁碼 p, 位移 d) 轉換成真實位址 (真實頁碼 f, 位移 d)，其方法較分段機制更為簡單，只要利用分頁表將邏輯頁碼 p 轉換成真實頁碼 f，然後與位移 d 一起送到匯流排上即可。

分頁表的使用同樣減慢了記憶體的存取速度，所以也會使用 TLB 進行輔助，以增快存取速度，圖 10.9 顯示了利用 TLB 輔助分頁機制的過程。

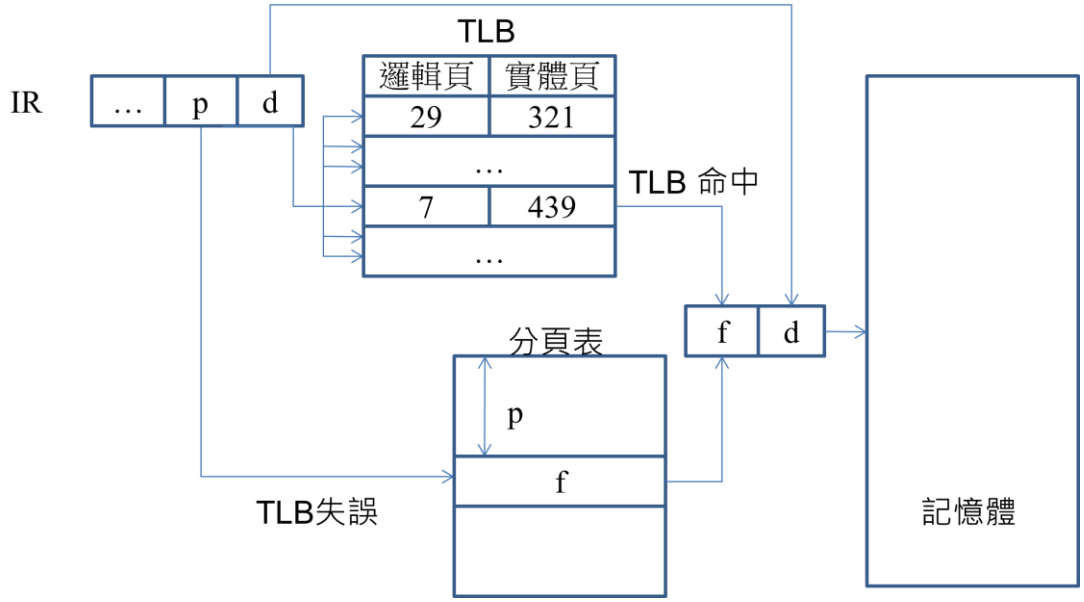

圖 10.9 使用分頁表搭配 TLB 進行位址轉換的過程

使用分頁機制的好處是可以實作出『需求分頁』技術，也就是在程式載入時並不需要將整個執行檔全部載入，而是先載入第一頁的程式，就可以開始執行，當成是執行到第二個分頁時，才將第二個分頁載入，這種技術稱為需求分頁。

在需求分頁的機制中，當某個指令企圖存取尚未被載入的分頁時，會觸發分頁中斷，此時作業系統會先發出載入分頁的請求後，就將原行程暫停，讓排程系統選取另一個行程繼續執行。

當分頁載入完成之後，又會引發輸出入完成的中斷，通知作業系統該分頁已經載入完畢。於是作業系統就可以安排該行程繼續執行。

透過這樣的機制，就不需將所有行程全部載入，於是記憶體所能容納的分頁數可

以超過真實的記憶體容量，因為大部分的分頁都被儲存在硬碟當中，這種技術讓硬碟成為記憶體的延伸，因而被稱為『虛擬記憶體』(Virtual Memory) 技術。

分頁技術雖然具有很好的特性，但是分頁表通常會很龐大，難以放入 TLB 中，因而會產生很多分頁失誤。

為了解決這個問題，作業系統的設計者提出的更先進的 MMU 技術，像是『分段式分頁』、『反轉式分頁』、『雙層分頁』等，圖 10.10 顯示了這些機制的轉換結構。

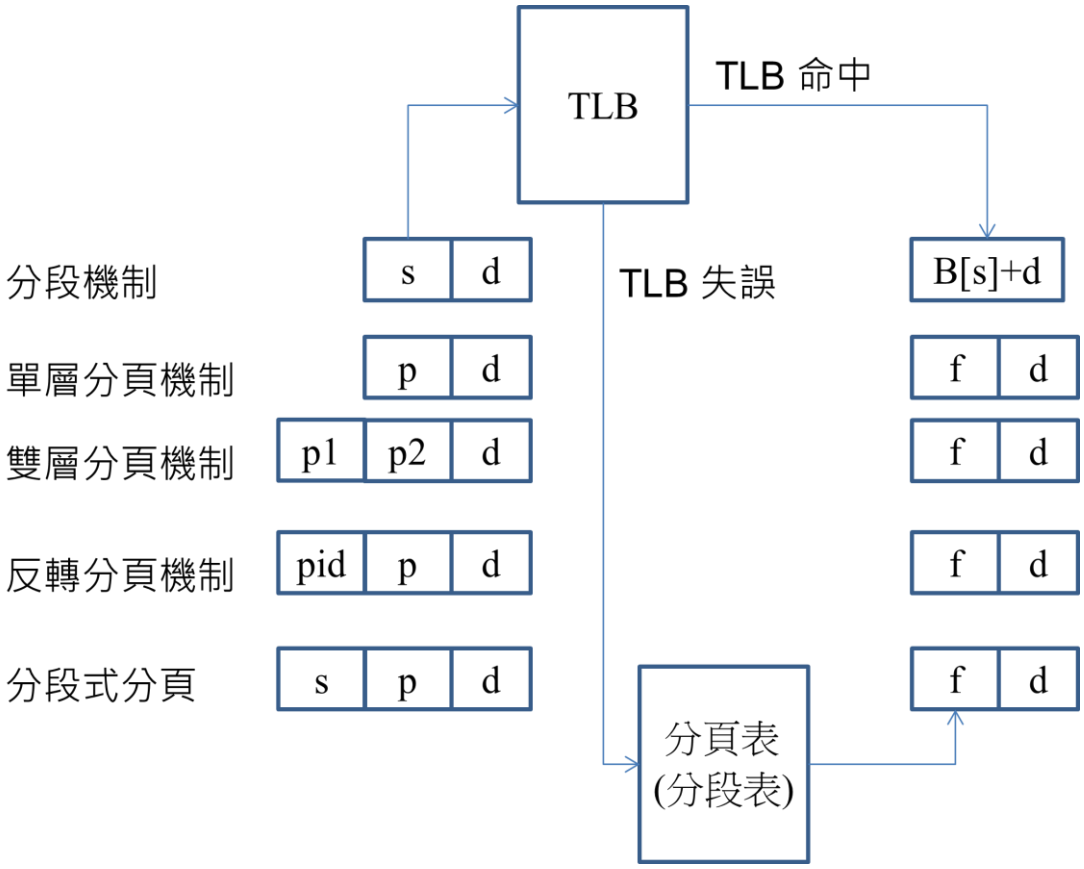

圖 10.10 各種分段與分頁機制的組合

在『分段式分頁』的結構中，邏輯位址 (s, p, d) 中的 s 是分段代號、p 是分段中的分頁代號、而 d 則是分頁內的位移。透過這種方式可以對每個分段再進行分頁，同時具有分段與分頁的好處。利用分段機制可以精確的防止行程存取非法範圍的位址，而利用分頁機制則可實作出需求分頁，並使用虛擬記憶體技術，因此同時具備了分段與分頁的優點。

『雙層分頁』機制 (Two Level Paging) 被提出的原因，是由於分頁表可能過於龐大，以致無法放入記憶體當中，因此使用兩層的機制讓分頁表也儲存在某些分頁當中，形成分頁表的需求分頁機制。而『反轉式分頁』 (Inverted Page)，則是將

行程代號 pid 也視為邏輯位址的一部分，所發展出來的分頁機制，通常會搭配雜湊表格 (Hash Table) 實作，反轉式分頁表曾經被用於 IBM RISC 6000、IBM RT 與 HP 的 Spectrum 工作站中。有關分頁機制的更進一步資訊，已經超出本書的範圍，敬請參考作業系統的相關書籍。

10.4 檔案與輸出入

作業系統的另一個重要任務，是利用輸出入驅動程式建構出檔案系統，由於檔案系統簡化了輸出入的動作，讓程式與使用者都能很方便的使用輸出入裝置，因此是輸出入系統的一個優秀的組織呈現方式。

檔案系統當中的主要核心概念為『目錄』與『檔案』，這種概念成功的讓程式師與使用者組織並使用輸出入裝置，而不需要真的理解輸出入的原理。在圖形化的視窗介面被發明後，檔案系統更以極為自然的方式，融入了視窗介面當中，成為視窗系統中不可分割的一部分。

檔案系統

對使用者而言，檔案 (File) 是一個資料的儲存單元，而資料夾 (Folder) (或稱為目錄 Directory)，則是可用來儲存許多檔案的容器，使用者也可以在其中建立子資料夾，以形成樹狀檔案結構。

在過去，曾經有單層的檔案系統，所有檔案都被放在單一層次中，這種結構被稱為檔案清單，這種單層的檔案系統很難使用，因為我們無法建立資料夾，也很難對檔案進行整理與分類，目前已經很少看到此類檔案系統了。

現代的檔案系統通常具有資料夾，因此可以容納樹狀的結構，甚至是非循環性的格狀結構。

MS. Windows 是一個採用樹狀結構的檔案系統，兩個資料夾當中不可以共用子資料夾，但是您可以利用捷徑 (Shortcut) (或稱符號連結, Symbolic Link) 的概念，連結其他資料夾中的檔案或子資料夾。

UNIX/Linux 是格狀的檔案系統，您可以利用硬式連結 (Hard Link) 讓兩個資料夾共用一個檔案 (或子資料夾)，形成格狀結構。

檔案系統通常會為每個檔案附加一些屬性資訊，像是檔名、附檔名、存取權限、

檔案類型、檔案大小、修改時間、備註欄位等，使用者可以透過這些資訊對檔案進行排序搜尋等功能，以便管理這些檔案。

作業系統對檔案名稱的長度通常有些限制，在早期的 DOS 當中，檔案名稱最多只能有 8 個字元，最多再加上 3 個字元的附檔名，所以 DOS 中檔案名稱不能超過 11 個字元。目前我們所使用的 MS. Windows 與 Linux 等作業系統，都支援所謂的長檔名機制，檔名可長達 255 個字元。

由於檔案的實際儲存體通常是像硬碟、光碟這樣的區塊儲存裝置，所以實際儲存時是以區塊為單位的。但是邏輯上的檔案系統卻是以檔案與資料夾的方式呈現的，這導致檔案系統的兩端有相當大的落差，作業系統必須試圖彌補這個落差，將一個變動大小的檔案儲存到許多固定大小的區塊當中。

如果作業系統將檔案儲存在連續的磁區當中，那麼當檔案變大時可能會因為後續的磁區被占用了而無法完成擴展的動作。因此，檔案通常可被分散儲存在不連續的磁區當中，然後再用特殊的資料結構進行串接。舉例而言，我們可以使用鏈結串列將不連續的磁區群串接起來，以便儲存整個檔案。

同樣的，目錄結構 (資料夾) 也必須被儲存在磁區當中，目錄結構中除了檔案清單之外，還必須儲存檔案名稱、屬性等資訊，這些資訊通常被儲存在稱為檔案描述器 (File Descriptor) 的基本單元中。

有效的組織磁區以便表達目錄結構，是檔案系統的重要任務，舉例而言，MS. Windows 的 NTFS 檔案系統使用了一種稱為 B+ 樹的著名資料結構，以便有效的組織檔案，而在 UNIX/Linux 中，傳統上都是使用一種稱為 i-node 的樹狀結構以組織檔案。

在檔案的建立與成長過程中，會需要取得可用區塊，以便容納新的資料，就好像程式在請求記憶體時必須取得可用記憶體區塊一樣，因此，作業系統必須管理可用區塊，並且配置給檔案系統使用。

檔案系統在分配磁碟空間時，通常以一種固定大小的磁區為單位，進行區塊分配動作。要管理這些區塊，必須使用某種資料結構，以下是兩種常見的區塊管理結構。

- 第一種結構是以鏈結串列 (Linked List) 記錄可用區塊，鏈結法乃是在可用磁區當中，記錄下一個可用磁區的代號，將可用磁區一個一個串接起來。但是這種結構的效率很差，較好的方法是將相鄰的磁區組成群組 (Group)，而非

單一的區塊，這有助於縮短鏈結串列的長度，並藉由一次分配數個磁區而提升效率。因此鏈結串列的組織方式通常會採用磁區群組模式，而非單一磁區的鏈結方法。

- 第二種結構是以位元映射法 (Bit mapped) 記錄可用區塊，將整個磁碟的映射位元儲存在數個磁區中。舉例而言，如果該磁碟的大小為 1G，而每個磁區大小為 1K，那麼總共就會有 1G/1K = 1M 個磁區，於是我們可以用 1M/8=0.125MB 的磁碟空間，儲存整個磁碟的位元映射地圖。由於每個磁區大小為 1K，因此整個映射圖可以被放在 0~124 號磁區中，於是我們就可以用 125 個磁區記錄整顆硬碟的一百萬個磁區之使用狀況，其效用非常高，是相當經濟且快速的一種實作方式。

程式可以透過開檔、讀取、寫入與關閉等函數，輕易的存取檔案系統所管轄的輸出入裝置，而一般使用者能透過介面，輕易的查看與編輯檔案的內容，這對電腦資料的維護而言，是一種很有效的方式。

檔案系統甚至可以將遠端電腦上的裝置視為目錄結構的一部分，讓使用者感覺到檔案資料就在手邊，像是 MS. Windows 上的網路芳鄰機制，就可以讓同一區域的電腦互相分享檔案，感覺就好像是自己電腦中的檔案或資料夾一樣。

如果更進一步將資料分散儲存在網路上的各個電腦中，還可以提升檔案的安全性，利用特殊的備份機制，可以避免單一儲存裝置的損毀造成的資料損失，像是 Google 的雲端運算 (Cloud Computing) 就是分散式檔案系統的一個大型範例。

檔案系統的設計有時會與儲存體的特性有關，舉例而言，『固態硬碟』的出現就導致了檔案系統的變更，讓『日誌式檔案系統』開始盛行，其原因與『快閃記憶體』的特性有關。

固態硬碟是利用快閃記憶體所組合而成的硬碟，由於快閃記憶體的磁區通常在寫入十萬次之後就會損毀，因此作業系統必須盡可能的讓每個磁區的寫入次數都相同，以避免過度的使用某些磁區而導致損毀的情況。於是就發展出了『日誌式快閃檔案系統』(Journaling Flash File System, JFFS)。

要能理解 JFFS 的設計理念，必須先理解日誌式檔案系統 (Journaling File System, JFS) 的原理，JFS 系統在改變資料的內容之前，會先將其即將要執行的動作記錄起來，然後再執行該動作。如果遇到不正常的關機或電力中斷時，因為系統尚有日誌的紀錄，因此當你重新開機後，系統會參考日誌的記錄來完成未完成的工作，或取消動作而回復原有的資料，可確保未完成的動作之資料不致遺失或損毀，可

將資料保持其完整性。

而 JFFS 比 JFS 更進一步，將整個 Flash 裝置視為一個環狀的日誌結構，寫入實永遠從尾部開始，因此保證了整個系統的寫入次數是均勻的，這使得許多使用固態硬碟的 Linux 系統採用這類檔案系統 (像是 JFFS2) 作為固態硬碟的檔案系統。

輸出入系統

輸出入系統的主要目的，乃是將複雜且多樣的輸出入裝置，透過函數包裝後，提供給程式設計師使用。這可以讓程式設計師很方便的使用這些輸出入函數，而不需要詳細瞭解輸出入裝置的運作方式與細節。對於程式師而言，這是相當重要的一件事。

電腦的輸出入裝置控制，往往相當的複雜，每一種裝置都有不同的特性。例如硬碟有磁軌、磁區的區分。光碟讀寫時則必須先控制步進馬達的轉動，等待步進馬達轉到特定位置後才能開始讀取資料，甚至必須瞭解光碟片上特殊的編碼法則，才能順利的解碼出二進位序列。另外，程式設計師還必須理解 CPU 的架構，例如是採用記憶體映射輸出入，或者是用特殊的組合語言指令進行輸出入動作，再決定選擇何種方式 (組合語言或 C 語言) 撰寫輸出入程式。

在沒有輸出入系統時，面對輸出入裝置，程式設計師必須研究該裝置的線路配置方式。假如該裝置採用記憶體映射機制連接到電腦上，程式設計師就必須知道記憶體映射的方式，包含每一個位元或位元組在此映射機制下所代表的意義，而這也正是嵌入式系統與驅動程式開發人員最常做的事情。然後在理解了每一個位元的意義後，才開始撰寫輸出入程式。

然而，如果每次使用到輸出入裝置，程式師都得要使用記憶體映射方式進行輸出入控制，那將會是程式師的一大噩夢，也會是軟體專案管理災難的開始。因為每一位程式師都將被迫瞭解每一個輸出入裝置的細節，而且不熟練的程式師往往會在撰寫這些輸出入程式的時候，會浪費掉許多時間，甚至犯下致命的錯誤。例如使用錯誤的參數設定、錯認記憶體映射的位址、或者以不正確的順序使用裝置等等。

驅動程式

即使是在嵌入式系統當中，專案管理者也不會讓每個程式設計師都直接用組合語言控制輸出入裝置，而是會由專人負責撰寫輸出入函數以控制該裝置，然後其他

程式只要透過該函數對裝置進行存取即可。這些函數，可以被視為是嵌入式系統的裝置驅動程式。但是通常我們所說的驅動程式，是專指針對某種作業系統，像是 MS. Windows 或 Linux，所撰寫的輸出入函數，必須符合作業系統的規範。

在作業系統中，輸出入裝置的控制，也是透過一組函數代為進行的。但不同的是，作業系統會規定這些函數的寫法，必須要符合該作業系統預設的方式，以便連接上作業系統，讓作業系統可以在適當的時候呼叫這些函數。

驅動程式會將一些函數指標傳遞給作業系統，讓作業系統記住這些函數，作業系統會在有該裝置的輸出入需求時，呼叫這些函數。這種『註冊-呼叫』機制是驅動程式常用的方式。而這些函數通常被稱為反向呼叫函數 (Call Back Function)。

驅動程式必須符合作業系統的規範，以便與作業系統互動，像是函數的參數型態與傳回值，都必須符合作業系統的規範。對於不同類型的裝置，作業系統會有不同的函數原型與規格，驅動程式必須滿足這些函數原型與規格，才能順利的與作業系統互動。

舉例而言， Linux 當中就以 register_chrdev() 註冊字元裝置的處理函數，而用 unregister_chrdev() 清除該註冊的函數。當然，還有許多其他反向呼叫函數，驅動程式的設計師必須實作出這些函數，才能順利將驅動程式『掛載』到作業系統之下。

透過這種規範，作業系統可以封裝輸出入裝置，但是對高階應用程式隱藏實作細節，以便讓使用者能很方便的呼叫，避免程式設計師直接存取該裝置而導致錯誤。除此之外，由於作業系統可以讓許多行程共用輸出入裝置，因此可以提升裝置的使用效能，這是輸出入系統好壞的重要指標。

除了『行程管理』、『記憶體管理』、『輸出入系統』、『檔案系統』之外，作業系統當中還有許多其他主題，像是『使用者介面』、『行程通訊』、『死結處理』、『分散式系統』等主題，請有興趣的讀者自行參考進一步的相關書籍。

10.5 實務案例(一)：Linux 作業系統

在本章中，我們已經簡單介紹了作業系統的設計原理，但是，對於系統程式的設計師而言，要深入理解作業系統，最好能透過程式實作的方式。在本節中，我們將學習 Linux 作業系統的原理 [2] 與程式設計方式，以實務的方式理解作業系統。

[2] 您可從 http://www.kernel.org/ 網站中下載最新的 Linux 原始碼

Linux 作業系統 [3][4][5] 是 Linus Torvalds 於芬蘭赫爾辛基大學當學生時，希望在 IBM PC 個人電腦上實作出類似 UNIX 系統的一個專案。在 Linux 剛發展時主要參考的對象是荷蘭阿姆斯特丹大學教授 Andrew S. Tanenbaum 的 Minix 系統，後來 Torvalds 決定利用 GNU 工具全面改寫，於是發展出一個全新的作業系統，後來該作業系統被稱為 Linux。

Linux 的系統架構大致分為『硬體』、『核心』、『函式庫』、『使用者程式』等四層，硬體層主要包含許多硬體裝置的驅動程式、核心層乃是由 Linus Torvalds 所維護的 Linux 作業系統，而函式庫層則對作業系統的功能進行封裝後，提供給使用者程式呼叫使用。

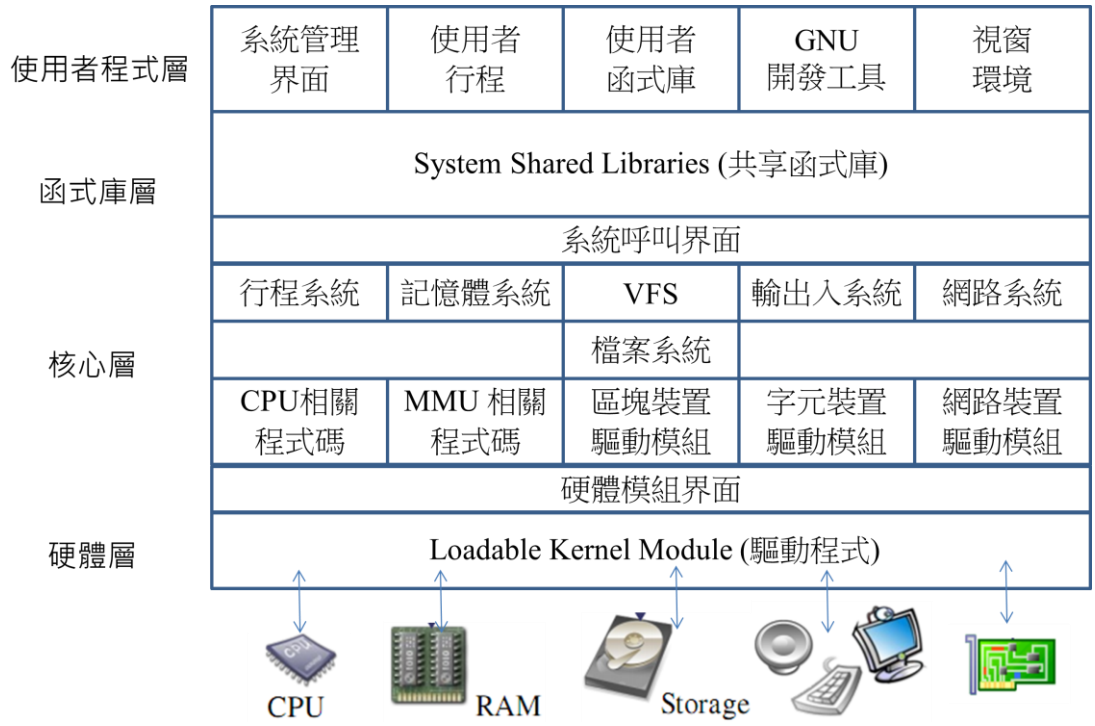

圖 10.11 Linux 的基本架構

當行程需要作業系統服務 (例如讀取檔案) 時，可以利用『系統呼叫』請求作業系統介入，此時處理器會由使用者模式 (User Mode) 切換到核心模式 (Kernel Mode)，核心模式具有最高的權限，可以執行任何的動作。圖 10.11 中的系統呼叫界面所扮演的，正是這樣一個中介的角色。

Linux 所支援的硬體模組眾多，這些模組必須被掛載到作業系統當中，當然不可

[3] http://www.tldp.org/
[4] http://en.tldp.org/LDP/tlk/tlk.html
[5] The Linux Kernel HOWTO http://24.221.230.253/HOWTO/kernel-howto/linux_boot_process.html

能由 Torvalds 一個人包辦寫出所有的驅動程式，所以 Linux 定訂了一整套輸出入介面規格，透過註冊機制與反向呼叫函數，讓驅動程式得以掛載到作業系統中。作業系統會在適當的時機呼叫這些驅動函數，以便取得輸出入資料。而這正是硬體模組介面的功能，這個介面可以載入驅動程式 (Loadable Kernel Module)，以進行裝置輸出入的動作。

Linux 2.6 版的核心包含『行程』、『記憶體』、『檔案』、『輸出入』、『網路』等五大子系統。行程系統支援行程與執行緒等功能，實作了排程、切換等機制。記憶體系統可利用硬體的 MMU 單元支援分段式分頁與虛擬記憶體等機制。檔案系統的最上層稱為虛擬檔案系統 (Virtual File System: VFS) ，VFS 是一組檔案操作的抽象介面，我們可以將任何的真實檔案系統，像是 FAT32, EXT2, JFS 等，透過 VFS 掛載到 Linux 中。真實檔案系統則是利用區塊裝置驅動模組所建構而成的。網路系統也是透過網路裝置驅動模組所建構出來的。輸出入系統則統合了『區塊、字元、網路』等三類裝置，以支援檔案、網路與虛擬記憶體等子系統。

Linux 是一個注重速度與實用性的系統，因此沒有採用微核心[6]的技術，以避免因為行程切換次數過多而減慢執行速度。目前圍繞著 Linux 作業系統已經形成了一個龐大的產業，幾乎沒有任何一家公司能主導 Linux 的發展方向，因為 Linux 是開放原始碼社群的集體開發成果，而且已經被迅速的納入到產業當中，成為整個工業體系的一部分。

由於開放原始碼的影響，Linux 擁有眾多的版本，像是 Red Hat、Ubuntu、Fedora、Debian 等，但是這些版本幾乎都利用 Tovarlds 所維護的核心，整合其他開放原始碼軟體後所形成的，因此雖然版本眾多卻有統一的特性。

雖然 Tovarlds 最早是利用 IBM PC 開發 Linux 作業系統的，但是目前 Linux 已經被移值到各種平台上。因此 Linux 所支援的處理器非常多，包含 IA32、MIPS、ARM、Power PC 等。當您想要將 Linux 移植到新的處理器上時，必須重新編譯 Linux 核心，您可以利用 GNU 的 gcc, make 等工具編譯 Linux 核心與大部分的 Linux 程式。

在本節中，我們將就 Linux 中的行程、記憶體、檔案與輸出入等子系統，分別進行說明，以便讓讀者能更進一步的理解 Linux 作業系統。

[6] 所謂的微核心 (Micro Kernel) 系統是將作業系統的大部分功能都切分到核心之外，只留下行程管理系統。然後再利用載入器將這些切分出去的功能，像是檔案系統、輸出入系統等，以掛載的方式掛入到微核心系統上，如此可以讓作業系統的核心盡可能的微小，因此稱為微核心作業系統。

Linux 的行程管理

在 10.2 節當中，我們已經介紹過行程 (Process) 與執行緒 (Thread) 的概念，但是這個概念或許對許多人而言仍然是很抽象而神秘的。在本節中，我們將利用程式實際建立行程，以便讓讀者能親自感受到行程與執行緒的奧妙之處，讓理論變為實務。

必須注意的是，本節中的範例只能在支援 POSIX 標準的環境 (像是 UNIX / Linux / Cygwin) 下執行，但是不能在 Dev C++ 的命令列環境當中編譯，因為該環境不支援 POSIX 標準，因此沒有 fork() 與 pthread 等函數，您必須改用 Linux 或者是 Cygwin 環境，才能順利的執行以下範例。

使用 fork 建立新行程

在 UNIX/Linux 中，我們可以利用 fork() 函數從舊行程中『分叉』(fork) 出新行程的，範例 10.3 顯示了一個利用 fork() 機制，從舊行程中產生新的行程 child，然後再利用 execvp() 函數，讓新行程執行命令列中的 ls -l /etc 指令，於是主行程繼續執行的結果，先印出 The end of program，然後子行程再列出 /etc 資料夾中的檔案與屬性，其結果顯示在範例 10.3 程式的下方。

範例 10.3 利用行程分叉(fork)函數產生多行程的範例

檔案 ch10/fork.c	說明
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <sys/types.h>	引用函式庫

int spawn(char *prog, char **arg_list) {
  pid_t child;
  child = fork();
  if (child != 0) {
  return child;
  } else {
  execvp(prog, arg_list);
  fprintf(stderr, "spawn error\n");
  return -1;
  }
	函數 Spawn 為生育的意思
	
	用 fork()函數分叉出子行程
	如果不成功
	傳回失敗的行程代碼
	否則
	將 prog 參數所指定的
	程式載入到子行程中

}
 	
int main() {
  char *arg_list[] = { "ls", "-l", "/etc", NULL };
  spawn("ls", arg_list);
  printf("The end of program.\n");
  return 0;
}
	
	主程式開始
	設定分叉行程的指令
	開始分叉
	印出主程式結束訊息

執行過程與結果

```
$ gcc fork.c -o fork

$ ./fork
The end of program.

$ total 94
-rwxr-x---    1 ccc Users    2810 Jun 13    2008 DIR_COLORS
drwxrwx---+ 2 ccc Users          0 Oct    7    2008 alternatives
-rwxr-x---    1 ccc Users        28 Jun 13    2008 bash.bashrc
drwxrwx---+ 4 ccc Users          0 Oct    7    2008 defaults
-rw-rw-rw-    1 ccc Users      716 Oct    7    2008 group
....
```

使用 pthread 建立執行緒

POSIX 標準中支援的執行緒函式庫稱為 pthread，我們可以透過 pthread 結構與 pthread_create() 函數執行某個函數指標，以建立新的執行緒。範例 10.4 是利用 pthread 建立兩個執行緒的程式，這兩個執行緒會重複的印出 George 與 Mary，因而導致 George 與 Mary 交錯被印出的情況。

在該程式中總共有三個執行緒，第一個執行緒是 print_george()、第二個執行緒是 print_mary()、第三個執行緒則是主程式 main()，由於每隔 1 秒印出一次 George ，但是每隔 2 秒才印一次 Mary，因此執行結果會以 George, Mary, George, George,Mary 的形式印出。

範例 10.4 利用 pthread 函式庫建立執行緒的範例

檔案 ch10/thread.c	說明
#include <pthread.h> #include <stdio.h>	引用 pthread 函式庫

```
void *print_george(void *argu) {
   while (1) {
      printf("George\n");
      sleep(1);
   }
   return NULL;
}

void *print_mary(void *argu) {
   while (1) {
      printf("Mary\n");
      sleep(2);
   }
   return NULL;
}

int main() {
   pthread_t thread1, thread2;
   pthread_create(&thread1, NULL, &print_george, NULL);
   pthread_create(&thread2, NULL, &print_mary, NULL);
   while (1) {
      printf("------------------------\n");
      sleep(1);
   }
   return 0;
}
```

每隔一秒鐘印出一次 George 的函數

每隔 2 秒鐘印出一次 Mary 的函數

主程式開始
宣告兩個執行緒
建立執行緒 1
建立執行緒 2
主程式每隔一秒鐘就印出分隔行

執行過程與結果

```
$ gcc thread.c -o thread

$ ./thread
George
Mary
------------------------
George
------------------------
George
Mary
```

George

…

Linux 的記憶體管理

Linux 作業系統原本是在 IA32 (x86) 處理器上設計的，由於 IA32 具有 MMU 單元，因此大部分的 Linux 都支援虛擬記憶體機制。然而，在許多的嵌入式處理器中，並沒有 MMU 單元，於是 Jeff Dionne 等人於 1998 年開始將 Linux 中的 MMU 機制去除並改寫，後來釋出了不具有 MMU 的 uClinux 版本。

曾經有一段時間，嵌入式的系統開發者必須決定應使用具有 MMU 的 Linux 或不具 MMU 的 uClinux，但是在 2.5.46 版的 Linux 中，Tovards 決定將 uClinux 納入核心當中，所以後來的 Linux 核心內包含了 uClinux 的功能，可以選擇是否要支援 MMU 單元。

由於 Tovarlds 最早是在 IA32 (x86) 中發展出 Linux 作業系統的，因此 Linux 的記憶體管理機制深受 x86 處理器的影響。要瞭解 Linux 的記憶體管理機制[7][8]，首先必須先理解 x86 的 MMU 記憶體管理單元。

IA32 (x86) 的記憶體管理單元

Intel 的 IA32 (Pentium 處理器) 採用了 GDT 與 LDT 兩種表格，其中的 LDT 分段表 (Local Descriptor Table) [9] 是給一般行程使用的，而 GDT 分段表 (Global Descriptor Table) 則包含各行程共享的分段，通常由作業系統使用。

IA32 同時具有分段與分頁單元，可以支援『純粹分段』、『單層分頁式分段』與『雙層分頁式分段』等三種組合。『邏輯位址』 (Logical Address) 經過分段單位轉換後，稱為『線性位址』 (Linear Address)，再經過分頁單位轉換後，稱為真實位址 (Physical Address)，其轉換過程如圖 10.12 所示。

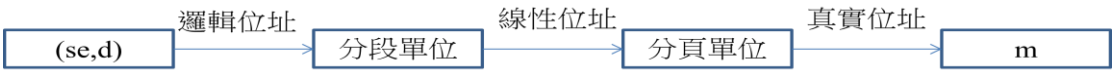

圖 10.12 IA32 的兩階段位址轉換過程

[7] http://linux-mm.org/
[8] 探索 Linux Memory Model (上), http://blog.linux.org.tw/~jserv/archives/001461.html
[9] 在 Pentium 中分段表被稱為描述表 (Descriptor Table)，分段表中的一個項目被稱為分段描述器 (Descriptor) ，分段碼 s 被稱為選擇器 (Selector)。

IA32 邏輯位址 (虛擬位址) 的長度是 48 位元，分為選擇器 S (selector : 16 bits) 與偏移量 D (offset : 32bits) 兩部分。其中的選擇器欄位中的 pr 兩個位元用來記錄保護屬性，g 位元記錄表格代碼[10] (可指定目標表格為 GDT 或 LDT)，另外 13 個位元則記錄分段碼 (s)。

IA32 的分段表 LDT 與 GDT 各自包含 4096 個項目，每個項目都含有 『分段起始位址』 (base)、『分段長度』 (limit) 與數個『輔助位元』 (aux) 等三種欄位。其中 base 與 limit 的功能與一般分段表相同，而輔助位元則用來記錄分段屬性。這兩個表格都可以用來將邏輯位址轉換成線性位址，是 IA32 中的分段單元。詳細的轉換過程如圖 10.13 所示。

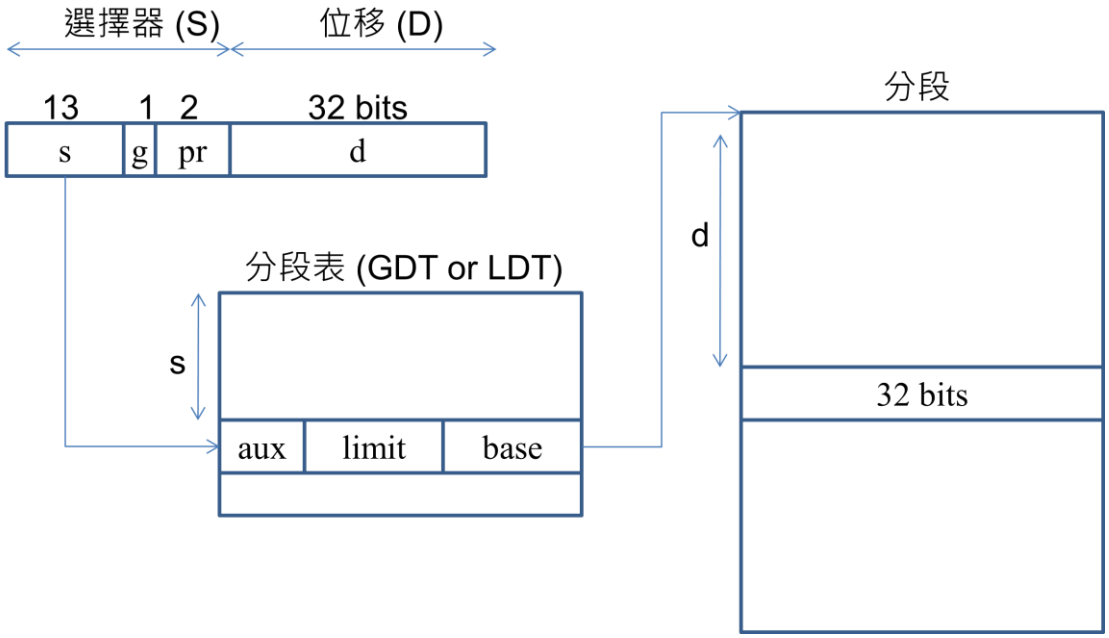

圖 10.13 IA32 分段模式

除了選擇器 (S) 的分段轉換過程之外，位移 (D) 部分會經過分頁表轉換成真實位址。位移 (D) 可再細分為三段 (p1, p2, d)，其中的 p1 是分頁目錄代號，p2 是分頁表代號，而 d 則是分頁內位移。IA32 可以不採用分頁機制，或採用單層分頁機制，甚至可以採用雙層分頁機制。其中的分頁目錄 PD 是第一層分頁，分頁表 PT 則是第二層分頁。當採用單層分頁時，每頁的大小為 4-MB。而採用雙層分頁時，每頁的大小為 4-KB。圖 10.14 顯示了 IA32 的分頁機制之轉換過程。

[10] 表格代碼 g 為 1 代表 GDT, 0 代表 LDT。

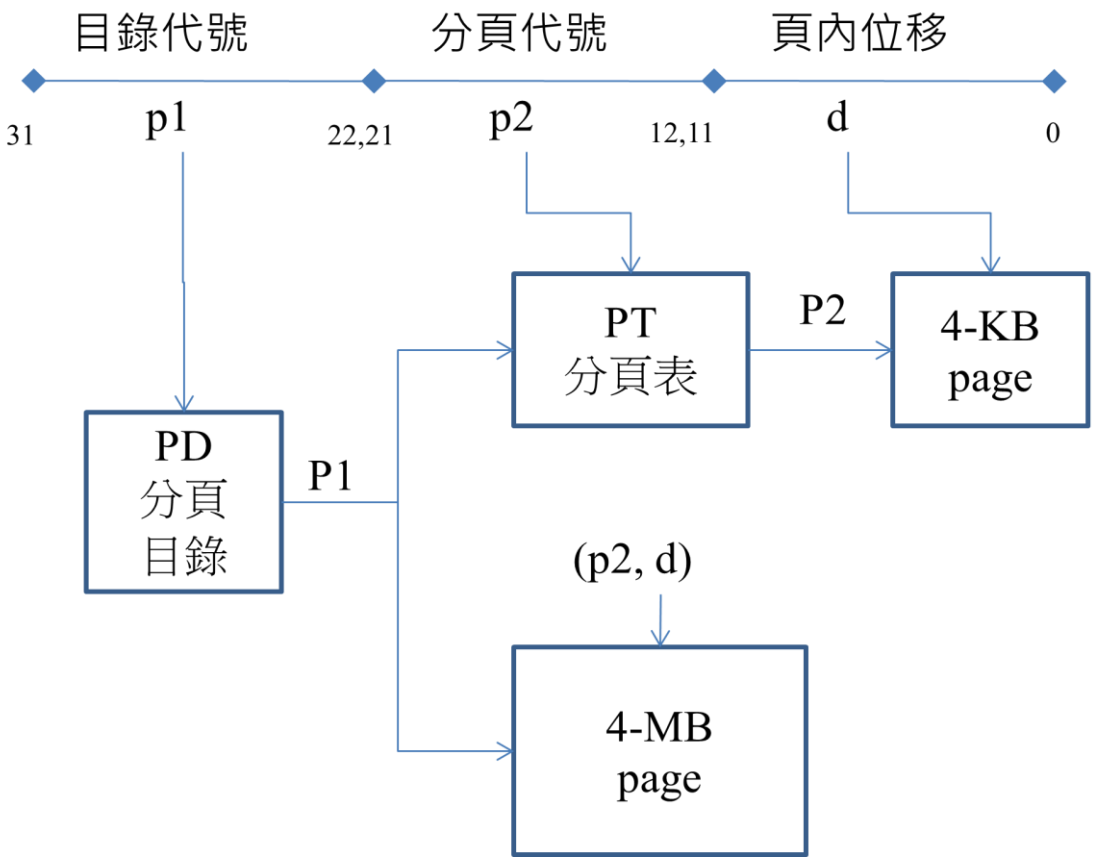

圖 10.14 IA32 的分頁模式

IA32 可以不使用分頁機制，直接將分段後的線性位址輸出，此時相當於一般的分段模式，其輸出位址如圖 10.15 中的 L 所示。如果使用單層分頁機制，則相當於一般的分段式分頁，其輸出位址如圖 10.15 中的 M1 所示。如果採用兩層分頁機制，則形成雙層分頁式分段體系，其輸出位址如圖 10.15 中的 M2 所示。圖 10.15 顯示了 IA32 MMU 單元的轉換過程。

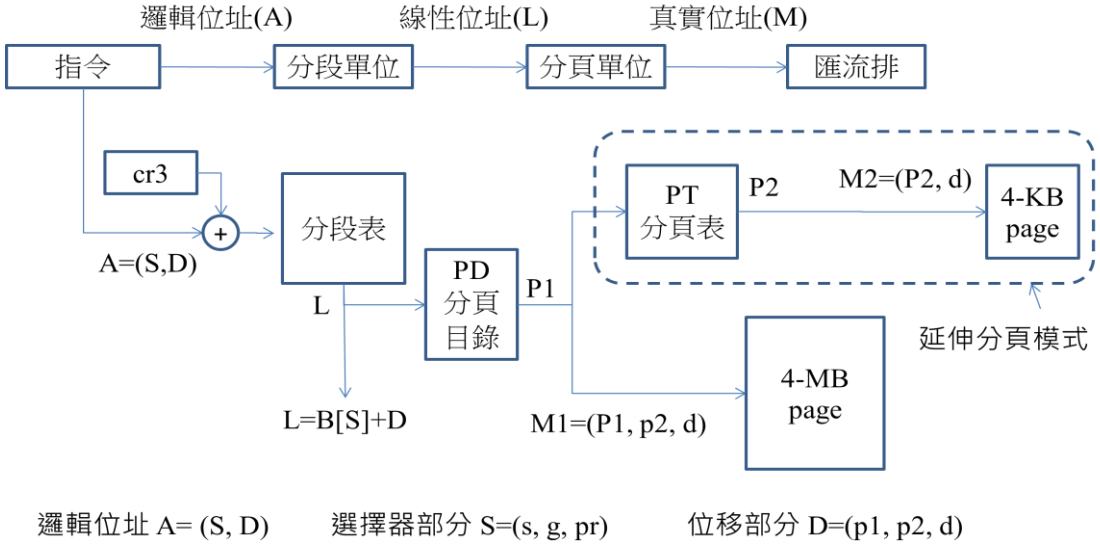

圖 10.15 IA32 的分段分頁模式

從 IA32 的 MMU 單元設計中，我們可以看到 IA32 處理器留下了相當大的選擇空間給作業系統，作業系統可以自行選擇使用哪一種分段分頁機制，這與作業系統的設計有密切的關係。

Linux 的記憶體管理機制

X86 版本 Linux 利用 GDT 指向核心的分頁，然後用 LDT 指向使用者行程的分頁。LDT 中所記載的是各個行程的分段表，以及行程的狀態段 (Task State Segment : TSS)。而 GDT 中則會記載這些分段表的起始點，TSS 起始點，以及核心的各分段起點。圖 10.16 顯示了 x86 版的 Linux 的分段記憶體管理機制。

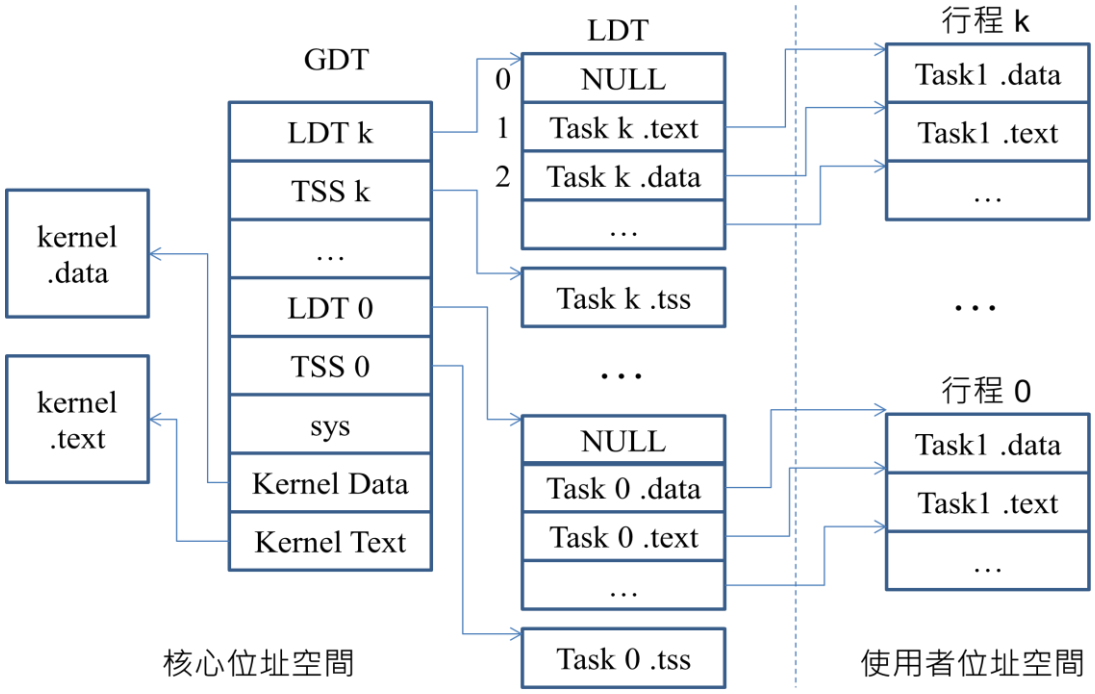

圖 10.16 Linux 的記憶體管理機制

透過圖 10.16 的分段機制，Linux 可為每一個行程分配一塊大小不等的區段。其中每一個區段又可能佔據許多個分頁，x86 版本的 Linux 採用 IA32 的延伸分頁模式，利用 IA32『分段+雙層分頁』的延伸記憶體管理模式，因此每個頁框的大小為 4KB。

Buddy 頁框分配系統

當需要進行分段配置 (例如載入行程) 時，Linux 會使用對偶式記憶體管理演算法 (Buddy System Algorithm) 配置分頁，在 Buddy 系統中，一個頁框代表一段連續的分頁，該演算法將頁框區分為十種區塊大小，分別包含了 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 個連續的頁框，其中每個區塊的第一個頁框位置一定是區塊大小的倍數。舉例而言，一個包含 32 個頁框的區塊之起始位址一定是 32 * 4KB 的倍數。

Buddy 系統的運作方法，乃是利用一個名為 free_area[10] 的陣列，該陣列中儲存了對應大小的位元映像圖 (bitmap)，以記錄區塊的配置狀況。當有分頁配置需求時，Linux 會尋找大小足夠的最小區塊，舉例而言，如果需要 13 個頁框，則 Linux 會從大小為 16 的頁框區中取出一個可用頁框，分配給需求者。但是如果大小為 16 的頁框區沒有可用頁框，則會從大小為 32 的頁框區取得，然後分成兩半，一半分配給需求者，另一半則放入大小為 16 的可用頁框區中。

當某頁框被釋放時，Buddy 系統會試圖檢查其兄弟頁框是否也處於可用狀態，若是則將兩個頁框合併以形成一個更大的可用頁框，放入可用頁框串列中。

Slab 記憶體配置器

當 Linux 需要配置的是小量的記憶體 (像是 malloc 函數所需的記憶體) 時，採用的是一種稱為 Slab Allocator 的配置器，其中被配置的資料稱為物件 (Object)。Slab 中的物件會被儲存在 Buddy 系統所分配的頁框中，假如要分配一個大小為 30 bytes 的物件時，Slab 會先向 Buddy 系統要求取得一個最小的分頁 (大小為 4KB)。然後 Slab 配置器會保留一些位元以記錄配置資訊，接著將剩下的空間均分為大小 30 的物件。於是當未來再有類似的配置請求時，就可以直接將這些空的物件配置出去。

Linux 的檔案與輸出入

檔案與目錄

在 UNIX/Linux 的使用者的腦海中，檔案系統是一種邏輯概念，而非實體的裝置。這種邏輯概念包含『檔案』、『目錄』、『路徑』、『檔案屬性』等等。我們可以用物件導向的方式將這些邏輯概念視為物件，表格 10.1 就顯示了這些物件的範例與意義。

表格 10.1 檔案系統中的基本邏輯概念

概念 (物件)	範例	說明
路徑	/home/ccc/hello.txt	檔案在目錄結構中的位置
目錄	/home/ccc/	資料夾中所容納的項目索引 (包含子目錄或檔案之屬性與連結)
檔案	Hello World !\n …..	檔案的內容
屬性	-rwxr-xr-- 1 ccc None 61 Jun 25 12:17 README.txt	檔案的名稱、權限、擁有者、修改日期等資訊

Linux 檔案系統的第一層目錄如表格 10.2 所示，認識這些目錄才能有效的運用 Linux 的檔案系統，通常使用者從命令列登入後會到達個人用戶的主目錄，像是使用者 ccc 登入後就會到 /home/ccc 目錄當中。目錄 /dev 所代表的是裝置 (device)，其中每一個子目錄通常代表一個裝置，這些裝置可以被掛載到 /mount 資料夾下，形成一棵邏輯目錄樹。

表格 10.2Linux 檔案系統的第一層目錄

目錄	全名	說明

/bin	Binary	存放二進位的可執行檔案
/dev	Device	代表設備，存放裝置相關檔案
/etc	Etc…	存放系統管理與配置檔案，像是服務程式 httpd 與 host.conf 等檔案。
/home	Home	用戶的主目錄，每個使用者在其中都會有一個子資料夾，例如用戶 ccc 的資料夾為 /home/ccc/
/lib	Library	包含系統函式庫與動態連結函式庫
/sbin	System binary	系統管理程式，通常由系統管理員使用
/tmp	Temp	暫存檔案
/root	Root directory	系統的根目錄，通常由系統管理員使用
/mnt	Mount	用戶所掛載上去的檔案系統，通常放在此目錄下
/proc	Process	一個虛擬的目錄，代表整個記憶體空間的映射區，可以透過存取此目錄取得系統資訊。
/var	Variable	存放各種服務的日誌等檔案
/usr	User	龐大的目錄，所有的使用者程式與檔案都放在底下，像是 /usr/src 中就存放了 Linux 核心的原始碼，而 /usr/bin 則存放所有的開發工具環境，像是 javac, java, gcc, perl 等。(若類比到 MS. Windows，此資料夾就像是 C:\Program Files)

磁碟分割

然而，檔案畢竟是儲存在區塊裝置中的資料，要如何將這些概念化為區塊資料的組合，必須依賴某些資料結構。為了能將目錄、檔案、屬性、路徑這些物件儲存在區塊當中。這些區塊必須被進一步組織成更巨大的單元，這種巨型單元稱為分割 (Partition)。

在 MS. Windows 中，分割是以 A: B: C: D: 這樣的概念形式呈現的。一個分割概念在 Windows 中通常稱為『槽』。由於歷史的因素，通常 A: B: 槽代表軟碟機，而 C:槽代表第一顆硬碟，D: E: …. 槽則可能是光碟、硬碟、或隨身碟等等。

但是並非一個槽就代表單一的裝置，有時一個裝置會包含好幾個分割，像是許多人都會將主硬碟進一步分割成兩個 Partition，形成 C: D: 兩個槽，但實際上卻儲存在同一個硬碟裝置中，Linux 中的 Partition 的概念也與 Windows 類似，但是命名方式卻有所不同。

在 Linux 中並沒有槽的概念，而是直接將裝置映射到 /dev 資料夾的某個檔案路徑中。舉例而言，第一顆硬碟的第一個分割通常稱為 /dev/hda1，第二個分割則

為 /dev/hda2, …。而第二顆硬碟的第一個分割區則為 /dev/hdb1，…。軟碟則被映射到 /dev/sda1, /dev/sda2, …./dev/sdb1, …. 等路徑當中。

在 Linux 中，我們可以利用 mount 這個指令，將某個分割 (槽) 掛載到檔案系統的某個節點中，這樣就不需要知道某個資料夾 (像是 /mnt) 到底是何種檔案系統，整個檔案系統形成一棵與硬體無關的樹狀結構。舉例而言，mount -t ext2 /dev/hda3 /mnt 這個指令可以將 Ext2 格式的硬碟分割區 /dev/hda3 [11] 掛載到 /mnt 目錄下。而 mount -t iso9600 -o ro /dev/cdrom /mnt/cdrom 這樣的指令則可將 iso9600 格式的光碟 /dev/cdrom 以唯讀的方式掛載到 /mnt/cdrom 路徑當中，當然，我們也可以用 unmount 指令將這些掛載上去的裝置移除。

目錄與屬性

當我們使用 ls -all 這樣的指令以列出資料夾中的目錄結構時，看到的就是這些概念所反映出的資訊。圖 10.17 就顯示了 ls 指令所呈現出的資訊與所對應的概念[12]。

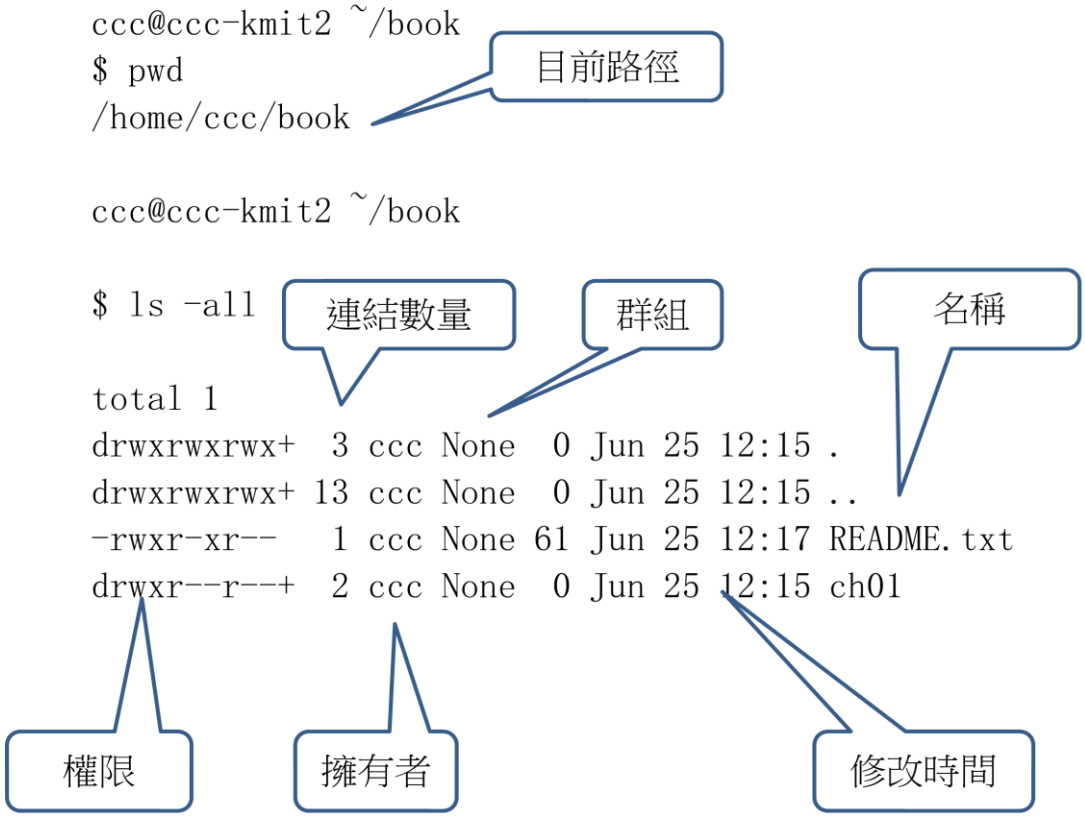

圖 10.17 UNIX/Linux 中的檔案與目錄概念

[11] 在 UNIX/Linux 中，硬碟的分割為 hda1, hda2, hda3, … 這樣的順序，
[12] 請參考 鳥哥的 Linux 私房菜網站 第六章、Linux 的檔案權限與目錄配置 - http://linux.vbird.org/linux_basic/0210filepermission.php

圖 10.17 中的權限欄位，第一個字元代表項目標記，目錄會以 d 標記，而檔案則會以橫線 - 標記，後面九個字元分為三群，分別是檔案擁有者權限、群組成員權限、一般人權限等等。例如，檔案 README.txt 的權限欄為 -rwxr-x-r--，可以被分為三個一組，改寫為 owner(rwx)+group(r-x)+others(r--)。這代表該檔案的使用者具有讀取 Read (r)、寫入 Write (w) 與執行 eXecute (x) 的權限，群組成員則只能讀取與執行，而其他人則只能讀取。

擁有權限者可以修改檔案的屬性，像是 chown 指令可以改變檔案的擁有者，chgrp 指令可以改變檔案的所屬群組，chmod 可以改變檔案的存取權限等。

對於 Linux 程式設計師而言，必須關心的是如何用程式對這些『物件』進行操作，像是開檔、讀檔、關檔、改變屬性等動作。表格 10.3 顯示了如何利用程式對這些物件進行操作的一些系統函式庫。

表格 10.3 程式對檔案系統的基本操作

物件	範例	說明
檔案	fd = open(“/myfile”…)	開關檔案
檔案	write, read, lseek	讀寫檔案
屬性	stat(“/myfile”, &mybuf)	修改屬性
目錄	DIR *dh = opendir(“/mydir”)	開啟目錄
目錄	struct dirent *ent = readdir(dh)	讀取目錄

作業系統必須支援程式設計師對這些物件的操作，程式可以透過系統呼叫 (像是 open(), read(), write(), lseek(), stat(), opendir(), readdir(), 等函數) 操控檔案系統。而檔案系統的主要任務也就是支持這些操作，讓應用程式得以不需要處理區塊的問題，而是處理『路徑』、『目錄』與『檔案』等較為容易使用的物件。

Linux 的輸出入系統

Linux 的輸出入系統會透過硬體模組介面，以管理各式各樣的驅動程式。Linux 將硬體裝置分為『區塊、字元、網路』等三種類型，這三種類型的驅動程式都必須支援檔案存取的介面，因為在 Linux 當中裝置是以檔案的方式呈現的。像是 /dev/hda1, /dev/sda1, /dev/tty1 等，程式可以透過開檔 open()、讀檔 read()、寫檔 write() 的方式存取裝置，就像存取一個檔案一樣。

因此，所有的驅動程式必須支援檔案 (file) 的操作 (file_operations)，以便將裝置偽裝成檔案，供作業系統與應用程式進行呼叫。這種將裝置偽裝成檔案的方式，

是從 UNIX 所承襲下來的一種相當成功的模式。

字元類的裝置 (Character Device) 是較為簡單的，常見的字元裝置有鍵盤、滑鼠、印表機等，這些裝置所傳遞的並非一定要是字元訊息，只要可以用串流型式表示即可，因此字元裝置又被稱為串流裝置 (Stream Device)，必須支援基本的檔案操作，像是 open(), read(), ioctl() 等。

Linux 的驅動程式必需將裝置包裝成一組支援檔案存取形式的函數，讓裝置的存取就像檔案的存取一般。在驅動程式的內部，仍然必須採用『註冊-反向呼叫』機制，將這些存取函數掛載到檔案系統當中，然後就可以透過檔案操作的方式讀取 (read) 或寫入 (write) 這些裝置，或者透過 ioctl() 函數操控這些裝置。

目前，已經有些書籍專門講解 Linux 的驅動程式的撰寫方式，這是一個較為複雜的主題，我們將不在此講解，有興趣的人可以參考相關書籍。[13]

走筆至此，我們已經介紹完 Linux 中的行程管理、記憶體管理、檔案與輸出入系統等主題，完成了我們對 Linux 作業系統的介紹。但是 Linux 是個龐大的系統，筆者無法進行太詳細與深入的介紹，有興趣的讀者請進一步參考本書網站與相關書籍。

習題

10.1 請說明何謂行程？何謂執行緒？行程與執行緒有何不同？
10.2 請說明排程系統中的行程切換機制是如何進行的？當行程切換時作業系統需要保存哪些資料？
10.3 請說明何謂 MMU 單元，具有 MMU 單元處理器的定址方式與沒有 MMU 者有何不同？
10.4 請說明何謂驅動程式？驅動程式與輸出入系統有何關係？
10.5 請說明何謂檔案系統？為何在 Linux 當中可以將裝置當作檔案進行讀寫呢？
10.6 請說明 Linux 中的行程管理系統採用哪些策略？
10.7 請說明 Linux 中的記憶體管理系統採用哪些策略？
10.8 請說明 Linux 中的輸出入管理系統採用哪些策略？
10.9 請說明 Linux 中的檔案管理系統採用哪些策略？
10.10 請安裝 Ubuntu Linux 的最新的版本，然後使用看看。

[13] Linux Device Drivers, Third Edition, 網址位於 http://lwn.net/Kernel/LDD3/，筆者存取時間為 2010/3/25。

10.11 請於 Cygwin 環境下編譯 ch10/fork.c 檔案，並執行看看。

10.12 請於 Cygwin 環境下編譯 ch10/thread.c 檔案，並執行看看。

10.13 請於 http://www.kernel.org/ 網站中下載 Linux 核心原始碼的最新版本，然後看看其中的 include/linux/sched.h、arch/x86/include/asm/processor.h、arch/x86/include/asm/system.h 等原始碼，試著理解其運作邏輯。