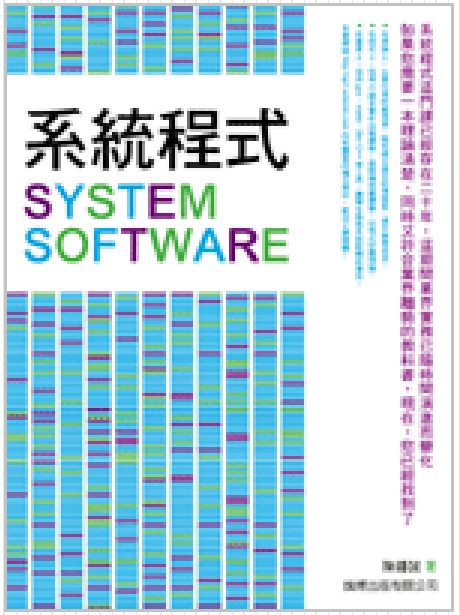
第 10 章、作業系統

作者：陳鍾誠

旗標出版社



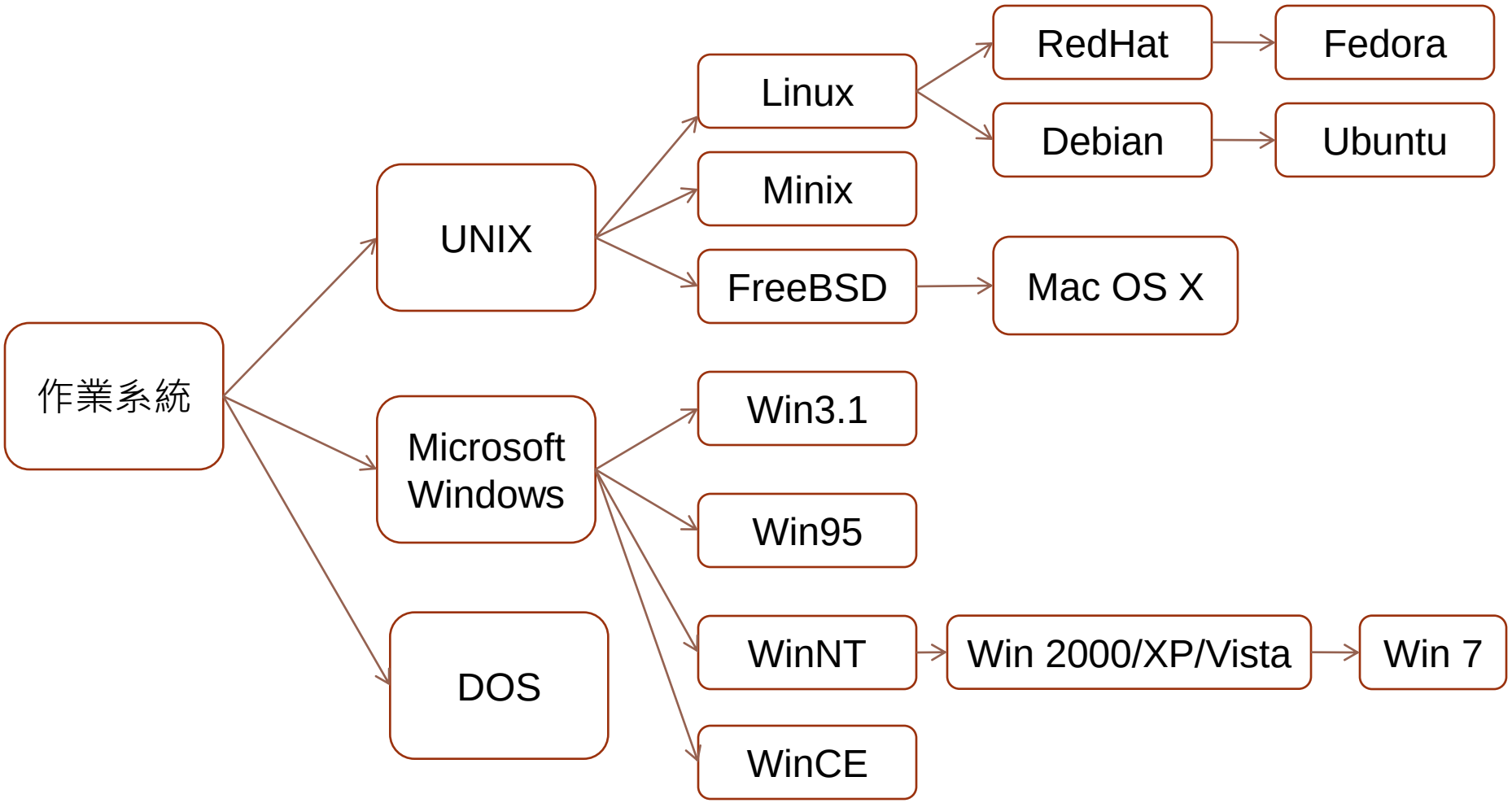
第 10 章、作業系統

- 10.1　簡介
- 10.2　行程管理
- 10.3　記憶體管理
- 10.4　檔案與輸出入
- 10.5　實務案例：LINUX 系統

10.1 簡介

- 作業系統的功能
  - 對使用者而言
    - 讓使用者方便的使用電腦, 盡情發揮電腦的功能。
  - 對程式設計師而言
    - 讓程式師很方便的寫程式, 而不會感到任何困難。

圖 10.1 今日常見的作業系統家族圖



作業系統的五大功能模組

- 行程管理
- 記憶體管理
- 輸出入系統
- 檔案系統
- 使用者介面

行程管理系統

- 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

- 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

- 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

- 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。
- 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。

使用者介面

- 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。

10.2 行程管理

- 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」。

程式、執行檔與行程

- 程式
  - 程式是撰寫者用編輯器所撰寫出來的文字型檔案
- 執行檔
  - 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
- 行程
  - 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。

多工 (Multtasking)

- 何謂多工？
  - 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)

- 範例
  - 具備多工能力的作業系統
    - MS. Windows、UNIX/Linux、Mac OSX 等系統。
  - 不具備多工能力的作業系統
    - MS. DOS

圖 10.2單行程系統與多工系統之比較

(a) 無作業系統的情況

例如：簡單型嵌入式系統

(b) 單一行程的作業系統

例如：DOS 作業系統

(c) 多工作業系統

例如：Linux 作業系統

協同式多工系統

- 何謂協同式多工？
  - 感覺好像是多工系統，但實際上任何一個程式當機都會導致系統失效，並非真正的多工系統。
  - 因此協同式多工系統可以說是一種「偽多工系統」。

- 範例
  - Windows 3.1 就是一種協同式多工系統，所有程式都必須適時的透過系統呼叫釋放 CPU 的控制權，否則將導致整個系統鎖死。

行程的行為模式

- 行程的動作通常可分為兩種
  - 一個是使用 CPU
    - 進行計算動作
  - 一個是使用輸出入裝置 (Input/Output 簡寫為 I/O)。
    - 進行輸出入動作

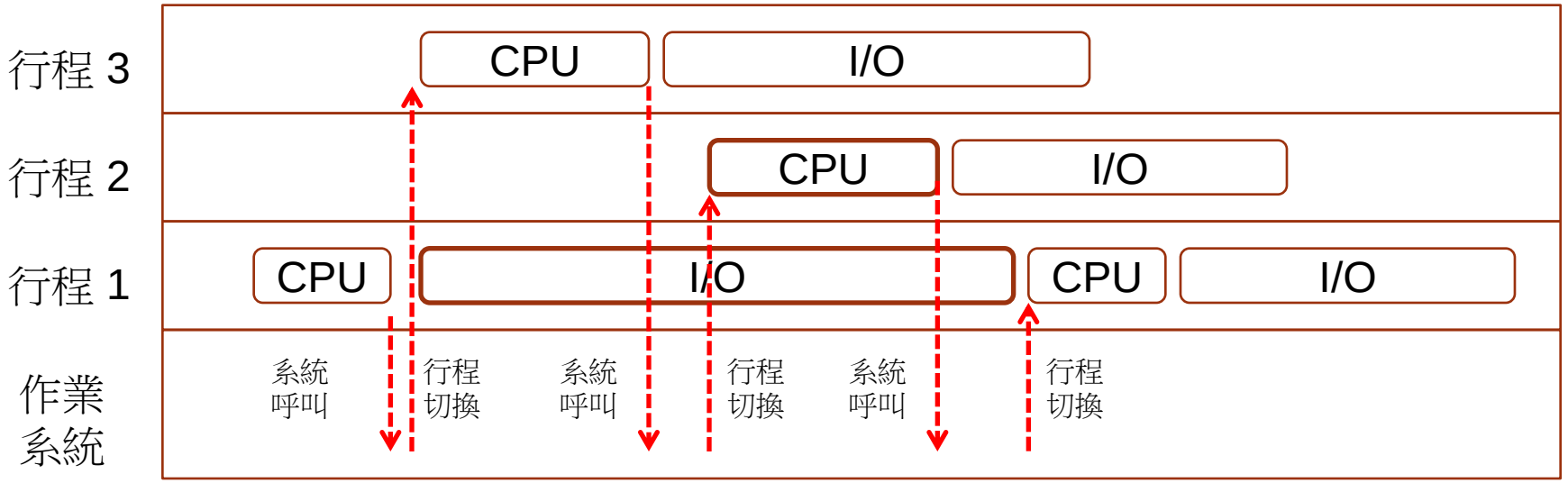
行為模式的範例

▶圖 10.3 程式的行為模式 - CPU 與 I/O 交替運行

使用輸出入的程式	說明
`int wc(const char *fname) {`	計算檔案詞彙數的程式
`  int words=0;`	
`  FILE *fp=fopen(fname, "r");`	開檔
`  while((ch=getc(fp))!=EOF) {`	讀取字元
`    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);`	關檔
`  return words;`	
`}`	

圖 10.4作業系統利用輸出入的空檔切換行程

- 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。



範例 10.1 不正常的行程 – 無窮迴圈導致當機

- 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;
}
```

利用中斷機制避免當機

- 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。

行程的狀態

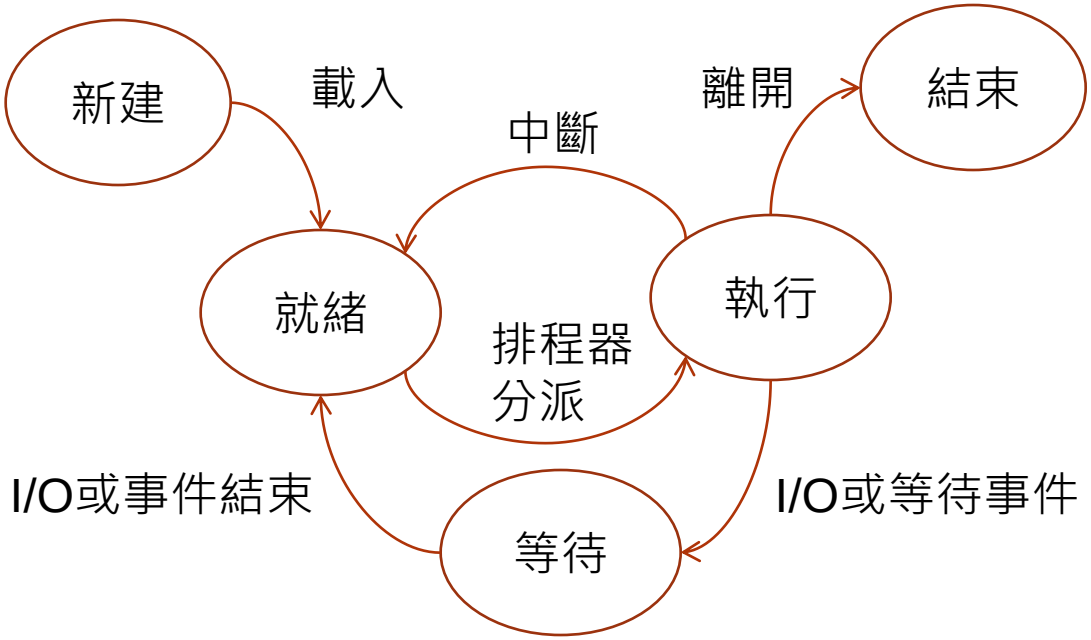


圖 10.5行程的狀態轉換圖

排程問題

- 排程問題的定義
  - 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。

- 排程問題的重要性
  - 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。

排程的方法

- 基本排程方法
  - 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)

- 綜合性的排程
  - 多層佇列排程 (Multilevel Queue Scheduling)
  - 多層反饋佇列排程 (Multilevel Feedback Queue Scheduling)

循環分時排程 (Round-Robin Scheduling)

- 重要性
  - 循環分時排程是實務上最常使用的排程方法。
- 方法
  - 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。
  - 一旦時間片段 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。

內文切換

- 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
- 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。

行程的資料共享方法

- 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 (Thread)

- 執行緒的定義
  - 又被稱為輕量級的行程 (Light Weight Process)
  - 執行緒之間會共用記憶體空間與相關資源
  - 在切換時只需保存暫存器，切換動作相當快速。

行程 v.s. 執行緒

- 行程
  - 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: MU) 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。

- 執行緒
  - 執行緒之間共用記憶體空間
  - 切換時不需更換映射表
  - 切換動作非常快速。

競爭狀況 v.s 臨界區間

- 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』。

行程同步機制

- 行程同步機制的用途
  - 利用鎖定等方式，避免兩個行程同時進入臨界區間的可能性, 以便防止競爭狀況的發生。

- 行程同步機制的方法
  - 方法 1：禁止中斷
  - 方法 2：支援同步的硬體
  - 方法 3：利用「號誌」等『鎖定機制』

禁止中斷的機制

- 在 CPU0 中，設定 I、T 旗標為 0，可以禁止中斷行為，如此就能防止兩個行程同時修改同一變數。

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
```

死結問題

- 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』。

死結問題的處理

- 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

- 記憶體管理的用途
  - 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。

- C 語言中的記憶體分配與回收
  - 分配：malloc()
  - 回收：free()

記憶體分配策略

- 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。

記憶體不足的問題

- 問題
  - 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

- 處理方法
  - 方法 1. 直接回報錯誤

  - 方法 2. 試圖處理記憶體不足的狀況
    - 記憶體聚集法 (Memory Compaction)
    - 垃圾蒐集法 (Garbge Collection Algorithm)

堆積空間不足時的處理方式

- 記憶體聚集法 (Memory Compaction)：
  - 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)：
  - 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。
  - Java 的 JVM 與微軟的 .NET 平台 都使用垃圾蒐集法

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。
- 在管理記憶體的時候，通常需要硬體的記憶體管理單元 (MMU) 配合，才能有效管理記憶體，並防止異常的存取行為。

常見的 MMU 硬體

- 重定位暫存器
- 基底界限暫存器
- 分段表
- 分頁表

重定位暫存器

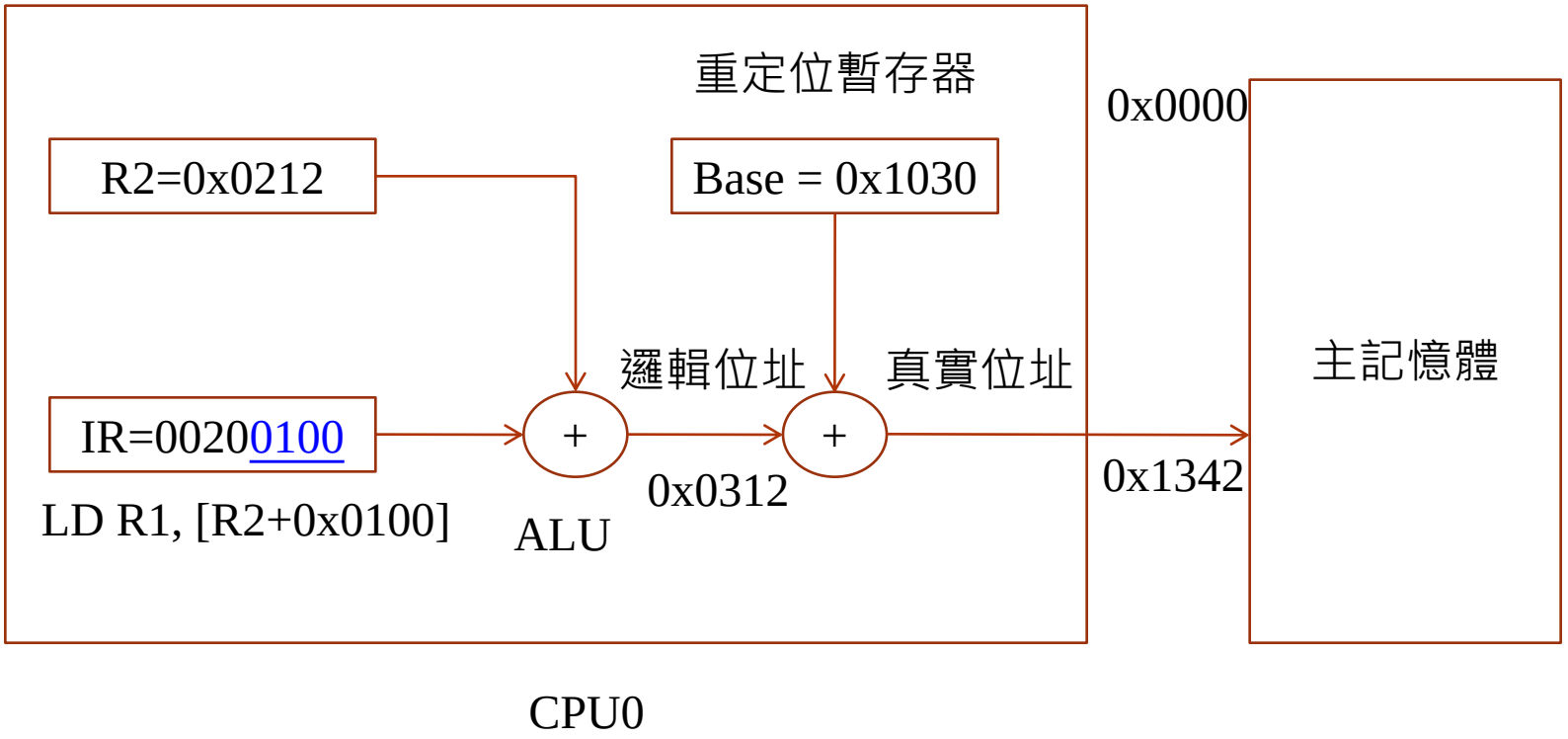


圖 10.6最簡單的MMU - 使用『重定位暫存器』

基底-界限暫存器

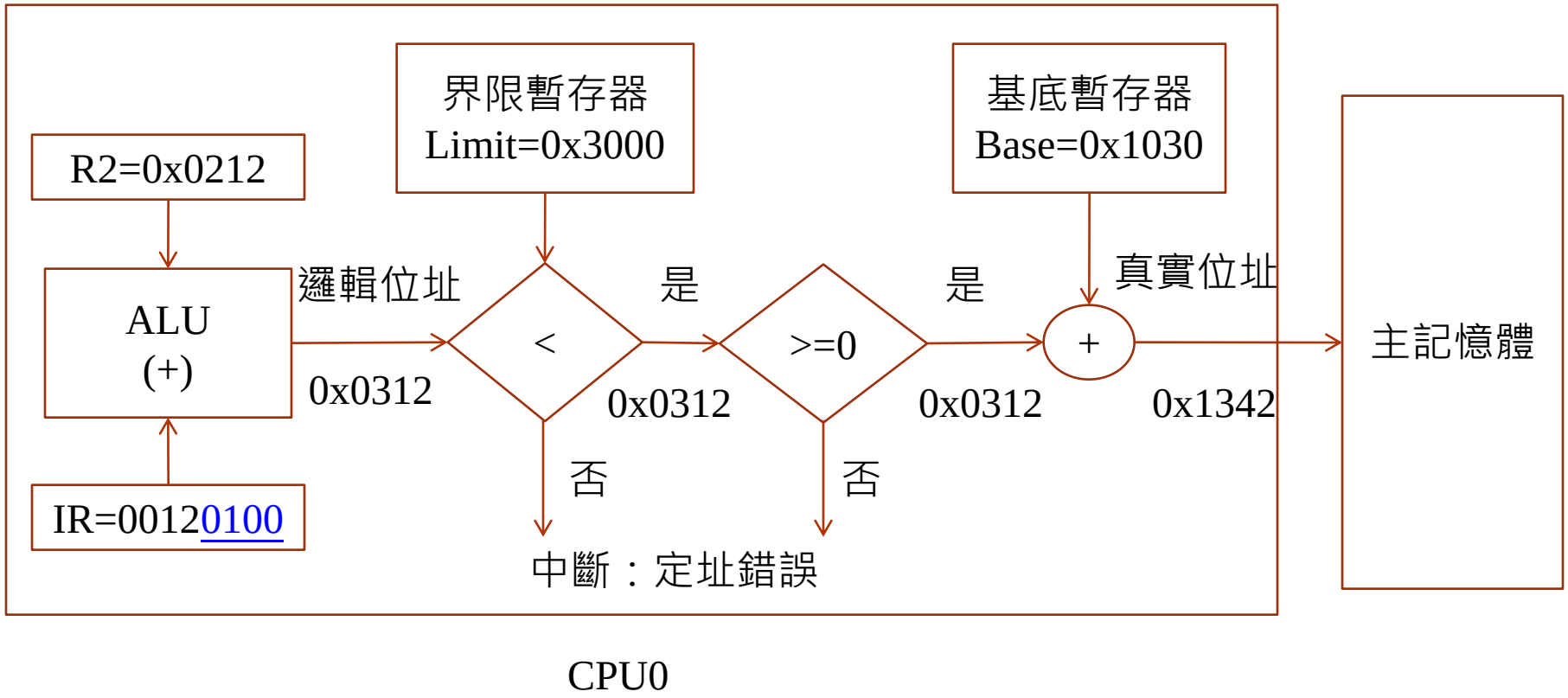


圖 10.7使用『基底-界限暫存器』提供硬體保護措施

分段機制

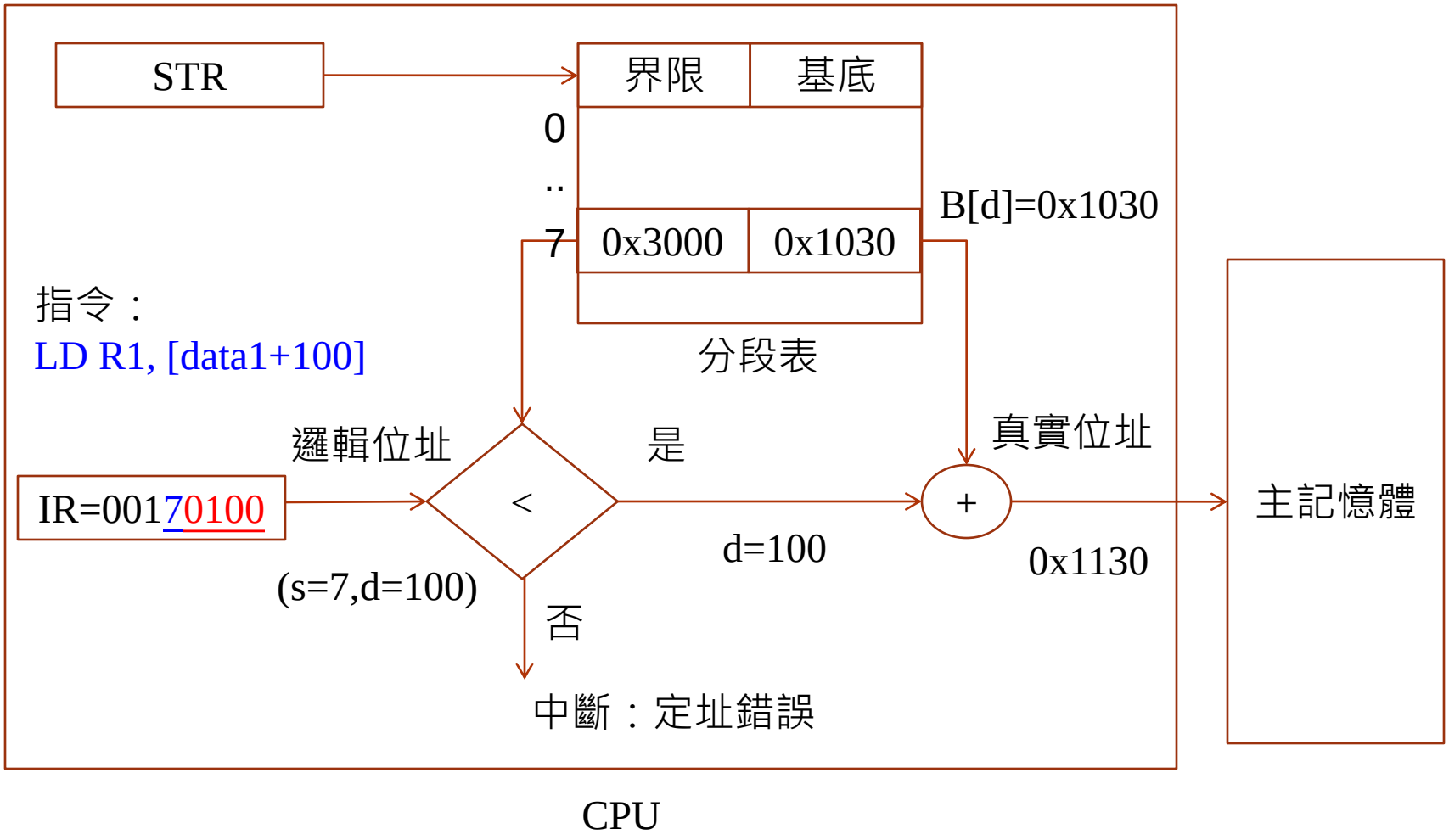


圖 10.8 使用分段表進行記憶體位址轉換的一個範例

分頁機制

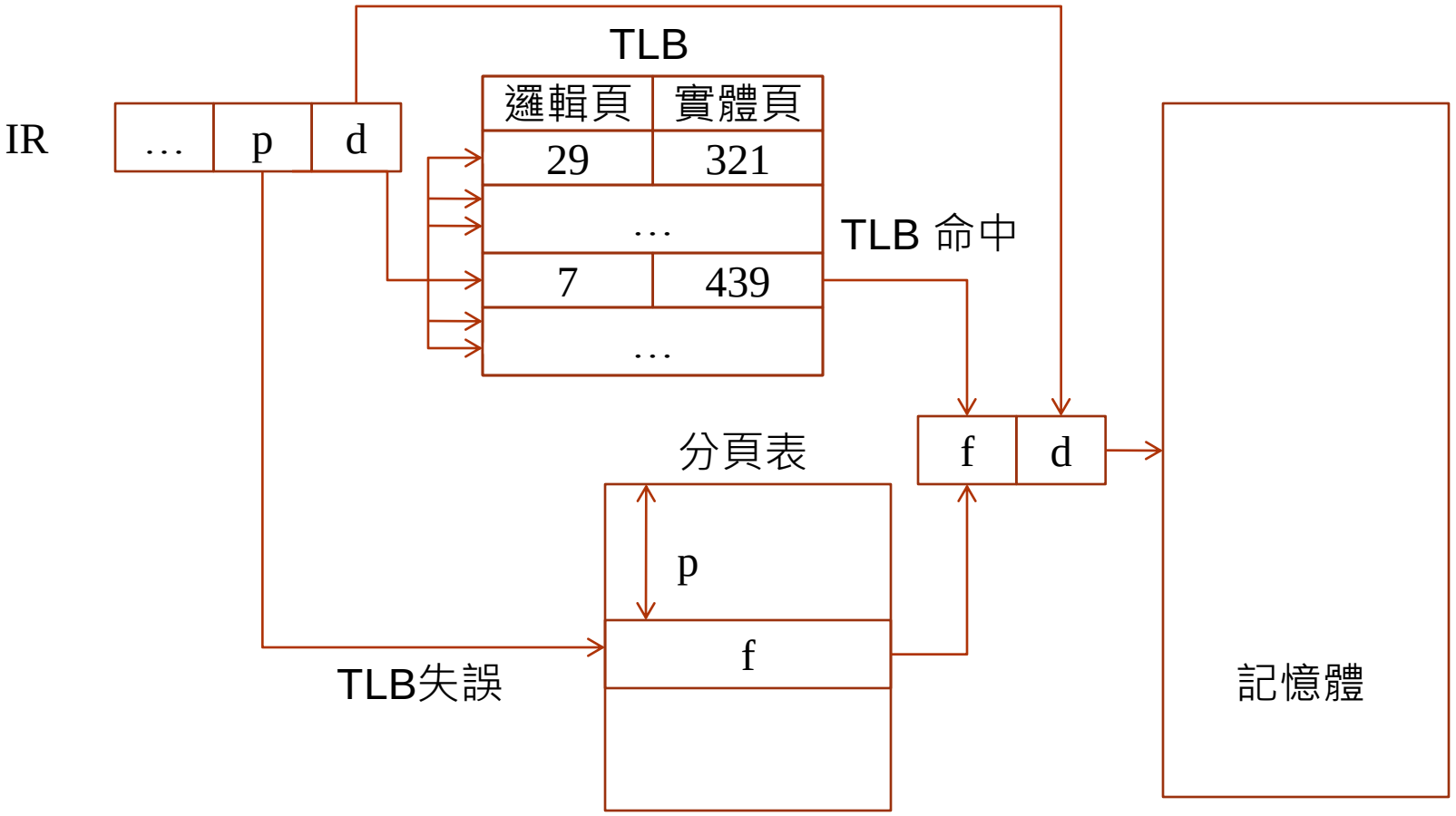
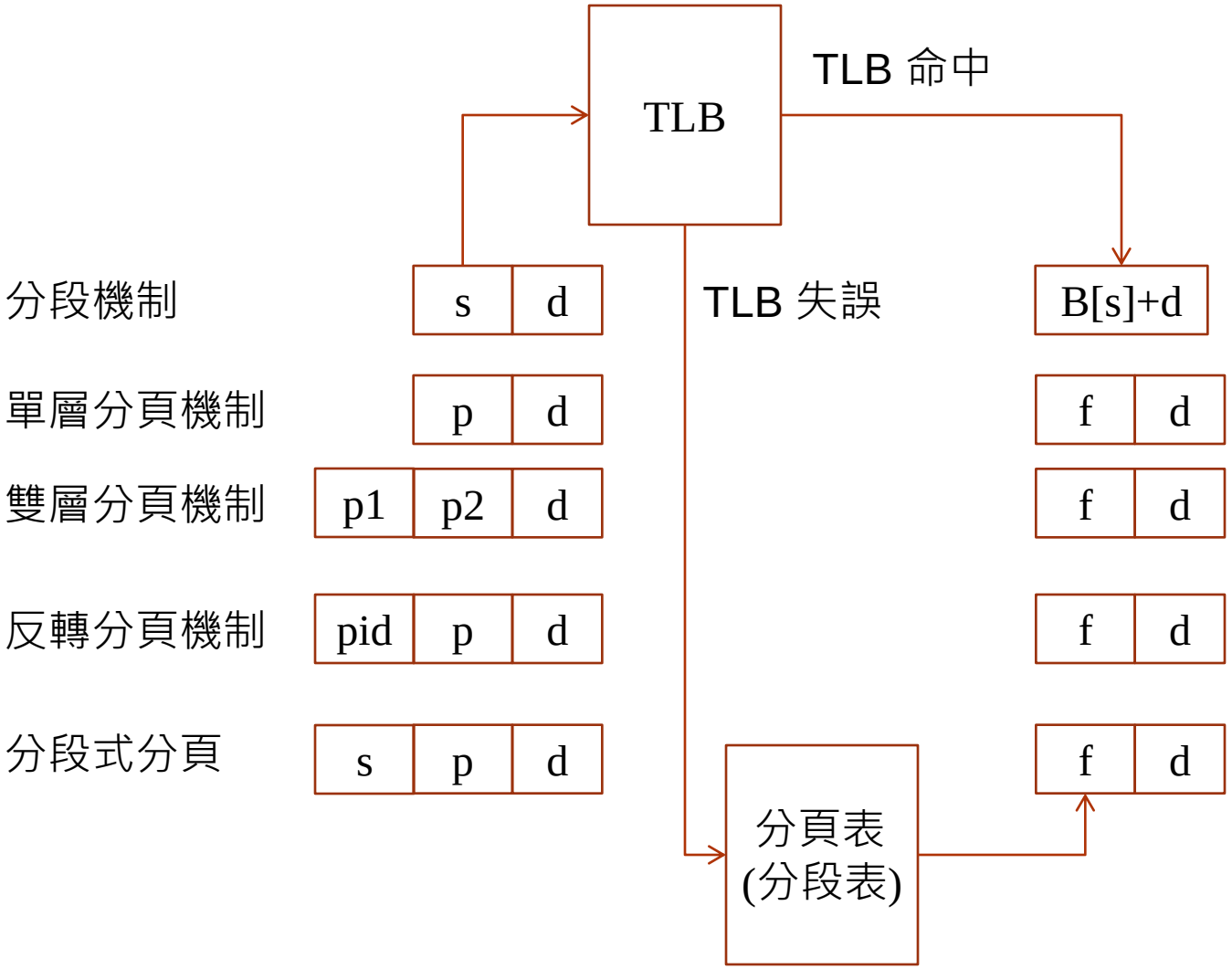


圖 10.9使用分頁表搭配TLB進行位址轉換的過程

圖 10.10各種分段與分頁機制的組合



10.4 檔案與輸出入

- 輸出入系統
  - 透過系統函式庫，規定輸出入介面，以便將驅動程式掛載到作業系統中
- 檔案系統
  - 將輸出入裝置封裝為「檔案」與「資料夾」的概念。

檔案系統

- 利用輸出入驅動程式建構出檔案系統
- 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。
- 檔案系統的主要核心概念為『目錄』與『檔案』

Linux 與 Windows 的檔案系統

- 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。

磁區結構

- 磁區的分配
  - 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。

- 磁區的組織方式
  - 要管理這些磁區, 必須使用某種資料結構, 以下是兩種常見的區塊管理結構。
  - 1. 鏈結串列法 (Linked List)
  - 2. 位元映射法 (Bit mapped)

磁區的組織方式 - 鏈結串列法

- 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。

磁區的組織方式 – 位元映射法

- 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。
- 範例
  - 如果磁碟的大小為 1G, 而每個磁區大小為 1K
  - 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

輸出入系統

- 輸出入系統的主要目的
  - 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。
  - 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。

- 假如沒有輸出入系統
  - 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，然後才能開始撰寫程式。
  - 很容易導致錯誤與 Bug 的發生。

驅動程式

- 何謂驅動程式？
  - 控制輸出入裝置，以供作業系統與程式設計師呼叫使用的程式。

- 作業系統呼叫驅動程式的方法
  - 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)。

10.5 實務案例：LINUX 系統

- Linux 的行程管理
- Linux 的記憶體管理
  - IA32 (x86) 的記憶體管理單元
- Linux 的檔案與輸出入

Linux 作業系統的歷史

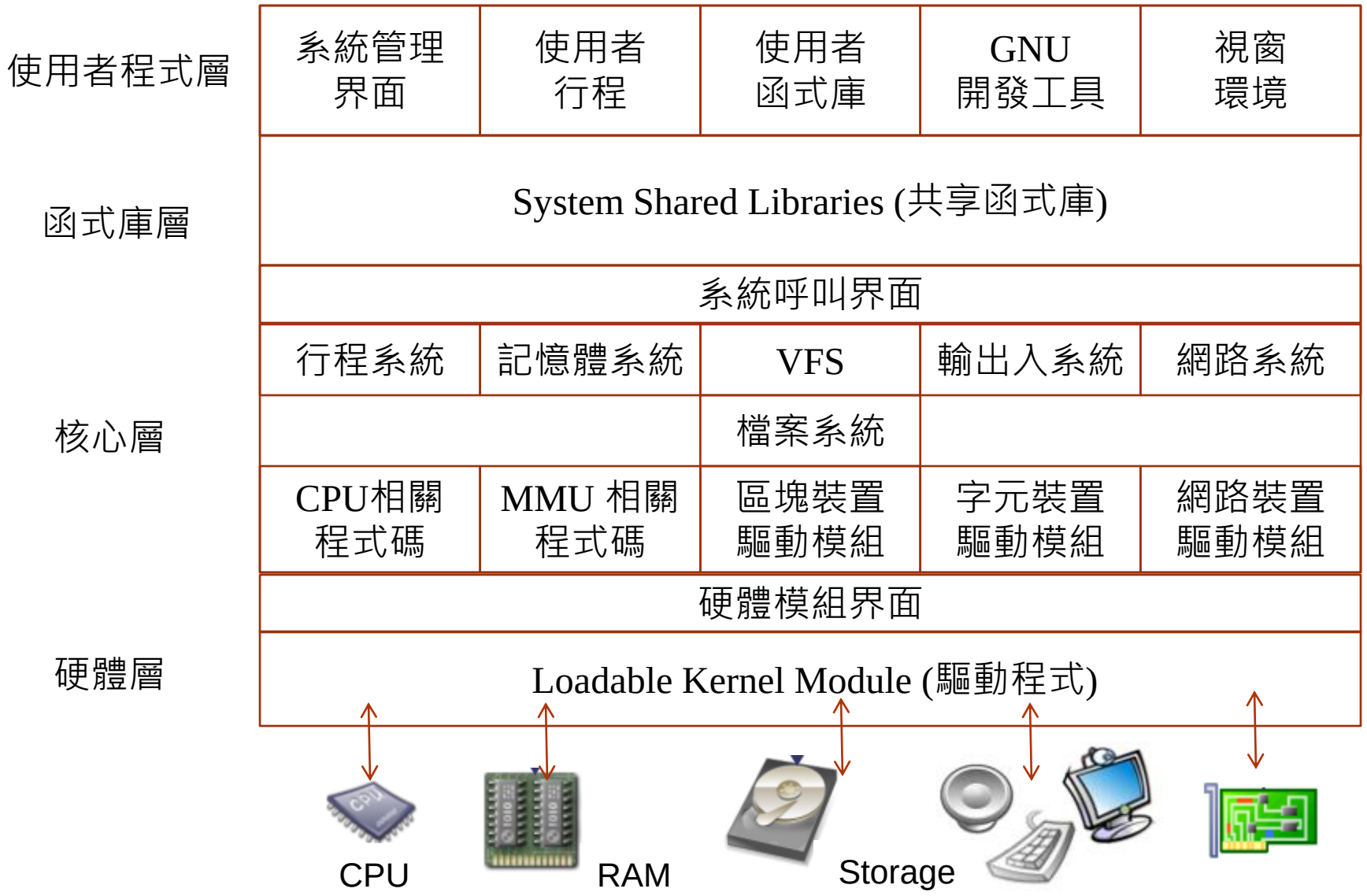
- 1990 年，Linus Torvalds 於芬蘭赫爾辛基大學當學生時希望在 IBM PC 個人電腦上實作出類似 UNIX 系統的一個專案。

- 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

- Linux 的系統架構
  - 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
- 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。
- 使用者程式層
  - 一般的應用程式。

圖 10.11 Linux 的基本架構



Linux 的行程管理

- 行程管理
  - 使用 fork() 分叉出新行程
  - 使用 execvp(prog, arg_list) 將新行程替換為另一個程式。
- 執行緒
  - 使用 pthread 函式庫
  - 使用 pthread_create() 建立新執行緒
  - 使用 sleep() 暫停一段時間

使用 fork 建立新行程 – spawn 函數

▶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()函數分支出子行程
	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

使用 fork 建立新行程 – 執行結果

```
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

▶範例 10.4 利用 pthread 函式庫建立執行緒的範例

檔案 ch10/thread.c

```
#include <pthread.h>
#include <stdio.h>

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;
}
```

說明

引用 pthread 函式庫

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

- 分段式分頁
  - Linux 原本是在 IA32 (x86) 處理器上設計的。
  - IA32 具有分段式分頁的 MMU 單元，包含 LDT 與 GDT 等兩個分段表。

- 分段表
  - LDT 分段表：(Local Descriptor Table)
    - 是給一般行程使用的
  - 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 (x86) 的記憶體管理單元

- 分段式分頁
  - IA32 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。

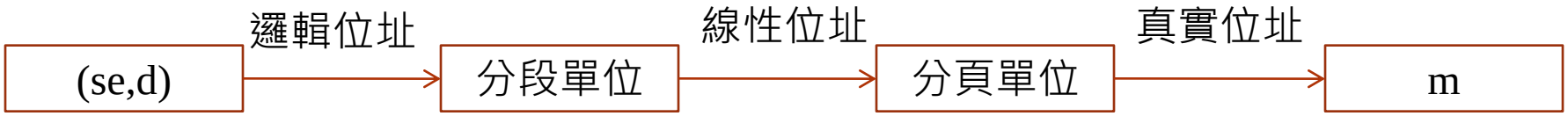


圖 10.12 IA32的兩階段位址轉換過程

圖 10.13 IA32分段模式

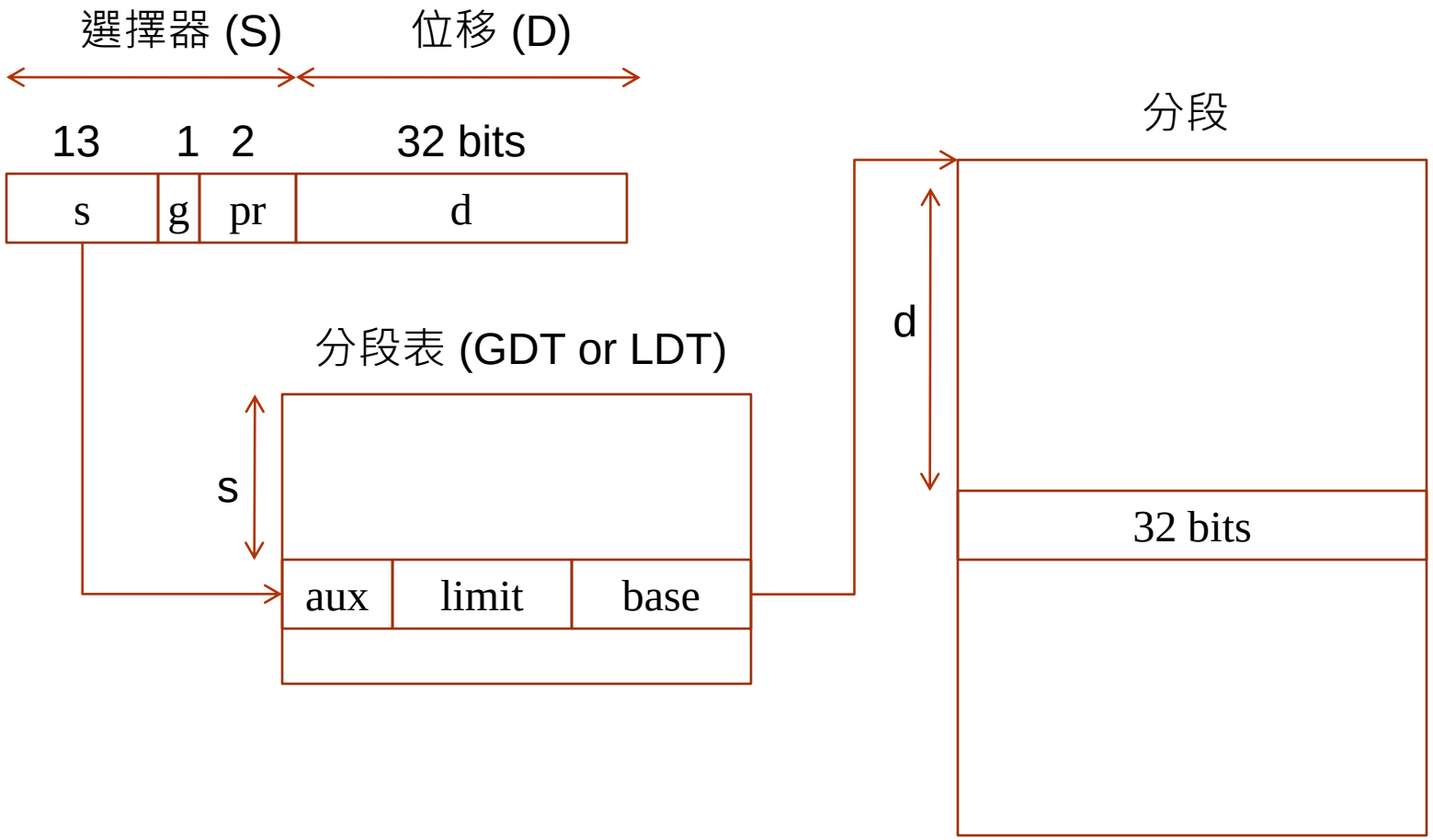


圖 10.14 IA32的分頁模式

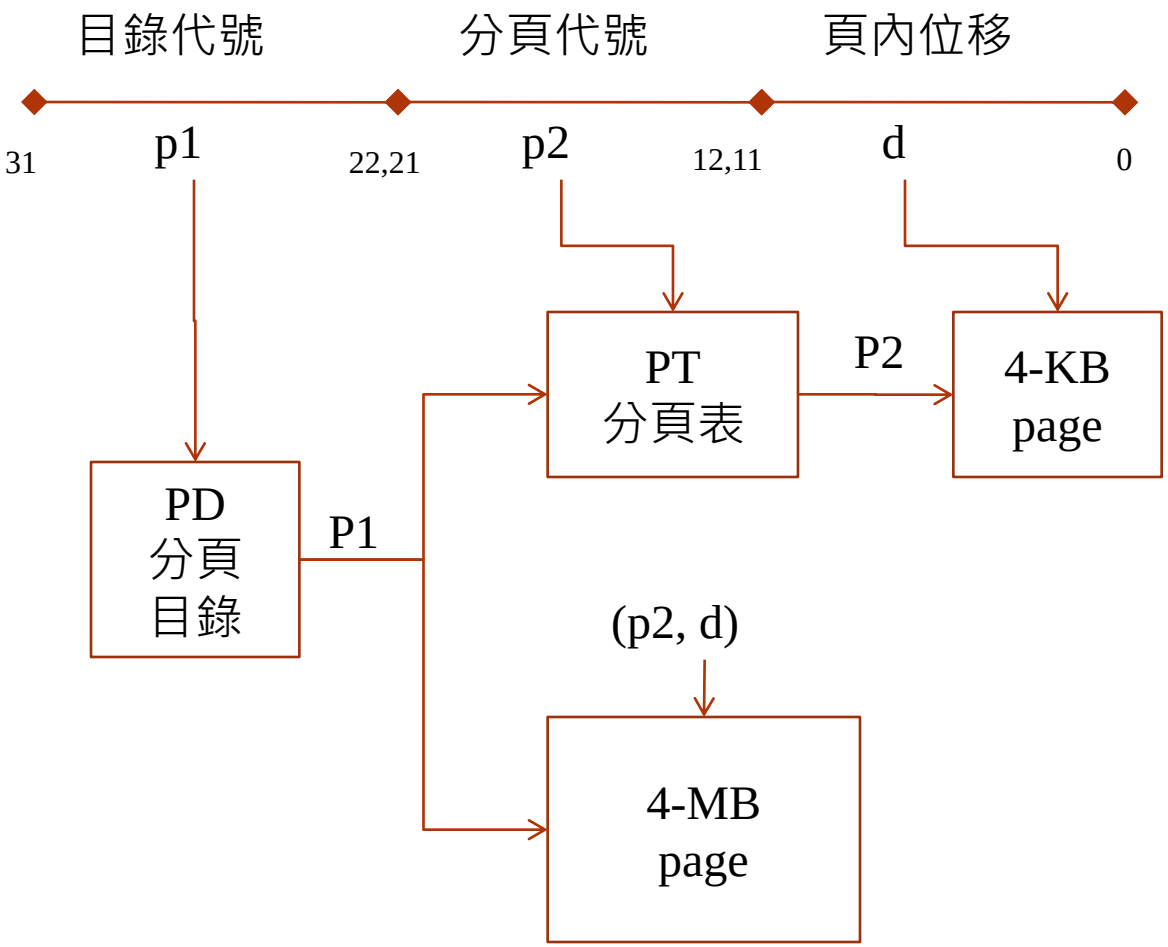
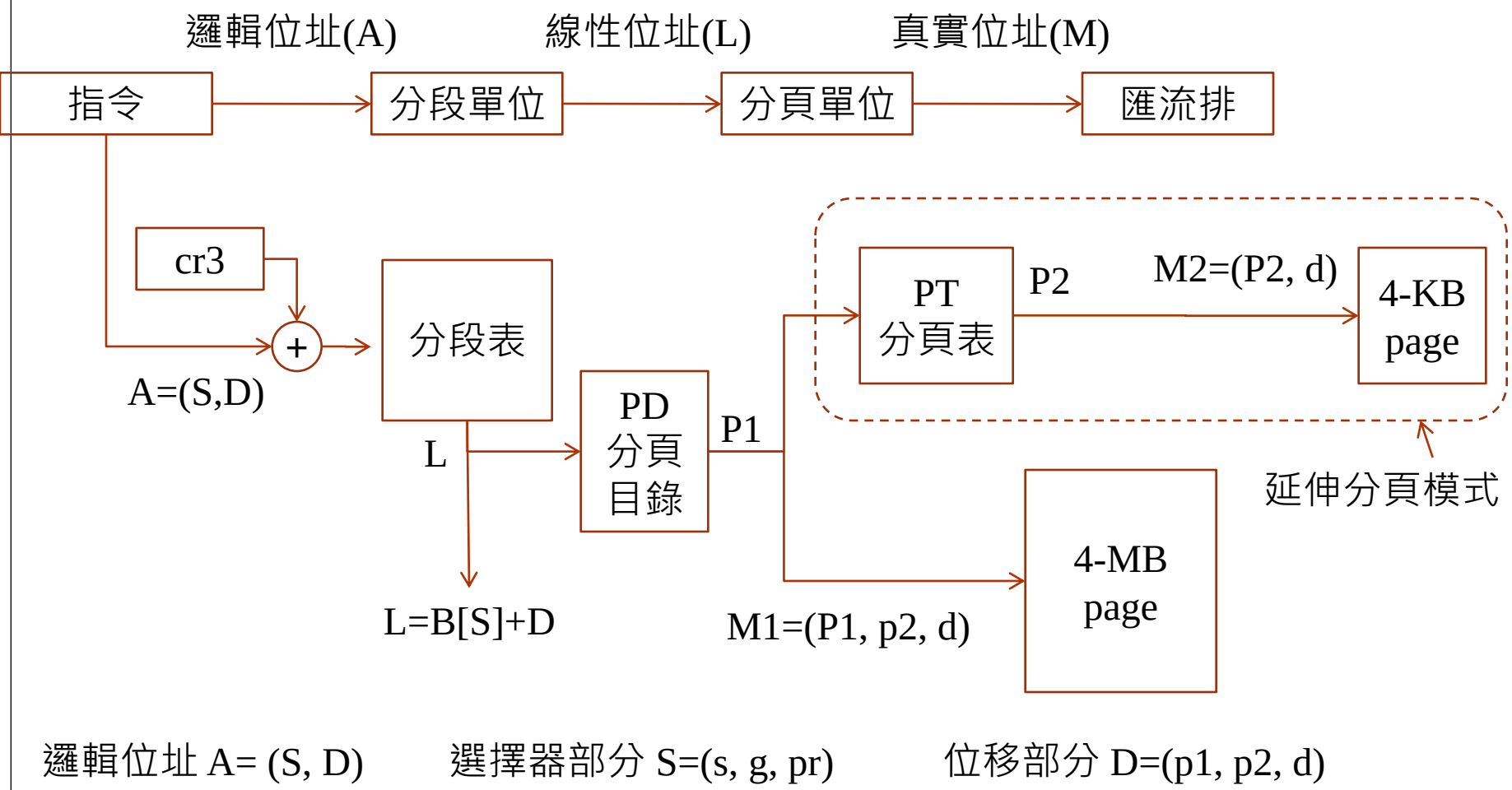


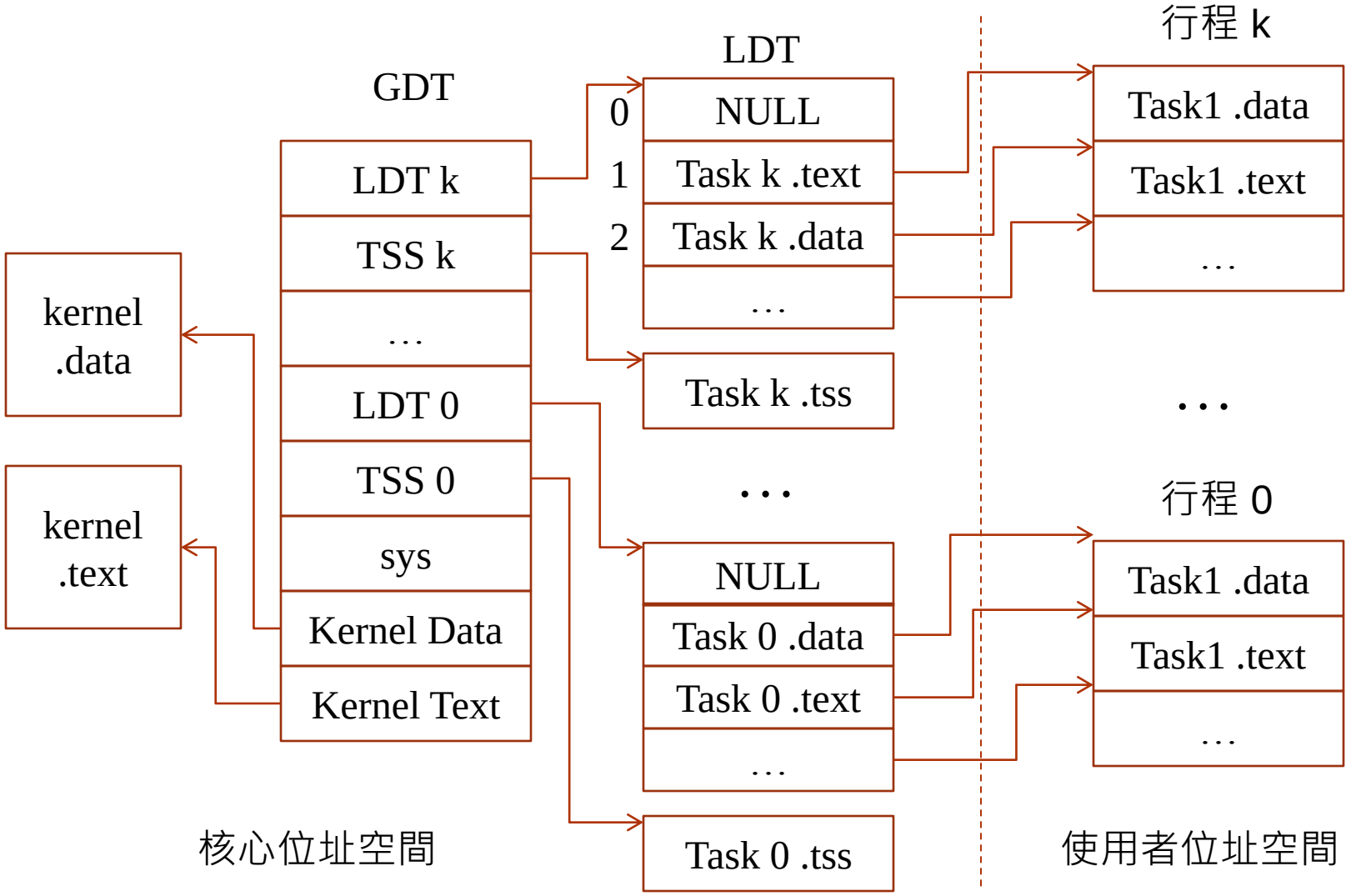
圖 10.15 IA32的分段分頁模式



Linux 的記憶體管理機制

- 分段式分頁
  - 在 IA32 上採用「分段+雙層分頁」的延伸記憶體管理模式, 每個頁框的大小為 4KB。
- LDT
  - 指向使用者行程的分頁
  - 記載的是各個行程的分段表, 以及行程的狀態段 (Task State Segment : TSS)。
- GDT
  - 指向核心的分頁
  - 記載 LDT 分段表的起始點, TSS 起始點, 以及核心各分段的起點。

圖 10.16 Linux 的記憶體管理機制



Buddy 頁框分配系統

- 使用時機
  - 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。

Buddy 系統的頁框大小

- 頁框大小
  - 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
  - 每個區塊的第一個頁框位置一定是區塊大小的倍數。

- 範例
  - 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 系統的分配方法

- 方法
  - 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
  - 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。

- 範例
  - 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。

Slab 記憶體配置器

- 使用時機
  - 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 Slab 配置器。
- 分配方法
  - 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
- 範例
  - 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, 然後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。

檔案系統中的基本邏輯概念

- 物件
  - 檔案、目錄、路徑、檔案屬性

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

表格 10.2 Linux 檔案系統的第一層目錄

目錄	全名	說明
/bin	Binary	存放 2 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)

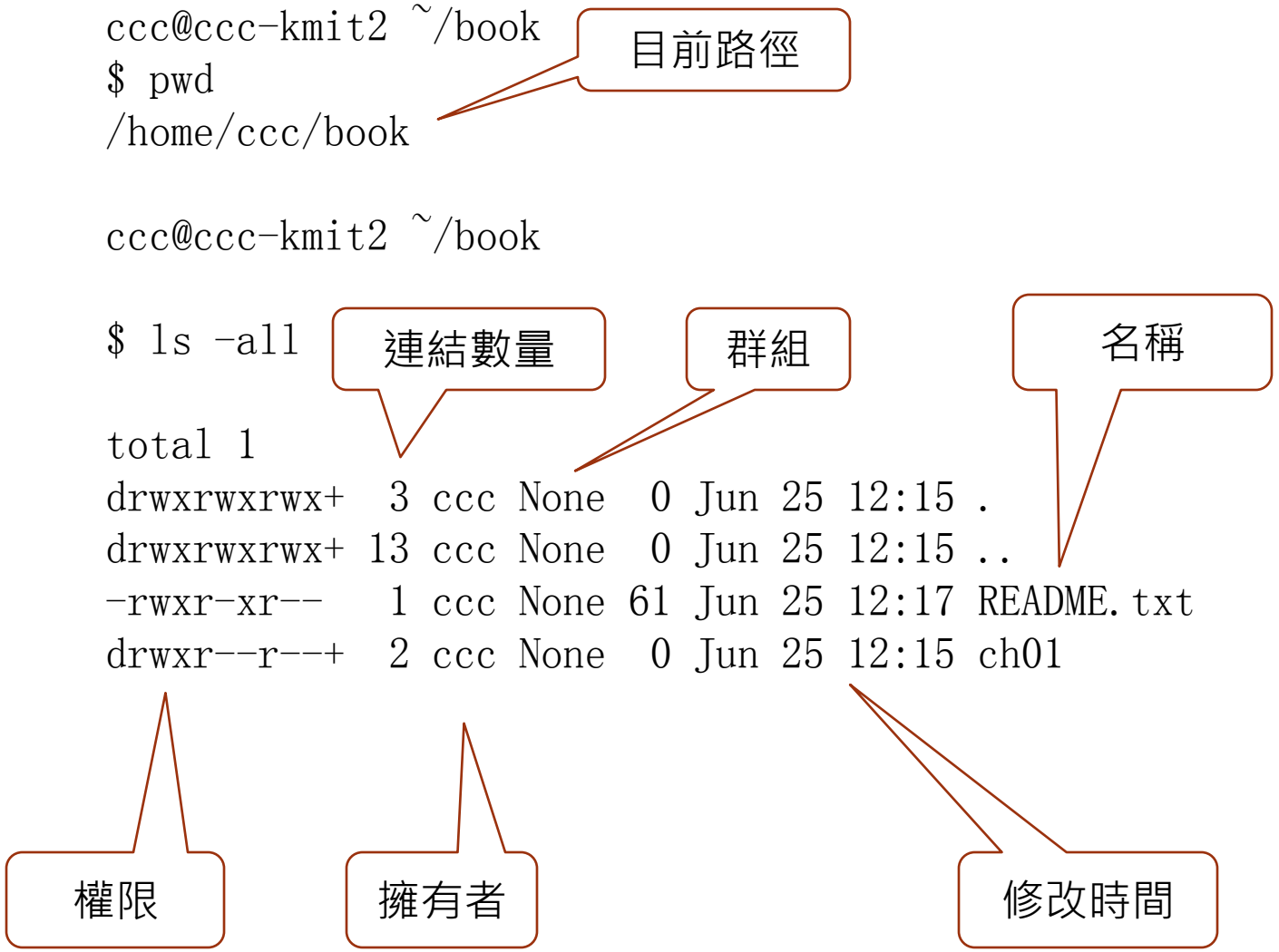
磁碟分割

- 何謂磁碟分割？
  - 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)。
- Windows 的磁碟分割
  - 槽：A: B: C: D:
- Linux 的磁碟分割
  - 方法：直接將裝置映射到 /dev 資料夾的某個檔案路徑中。
  - 第一顆硬碟：/dev/hda1, /dev/hda2, …。
  - 第二顆硬碟：/dev/hdb1, /dev/hdb2, …。
  - 軟碟： /dev/sda1, /dev/sda2, …./dev/sdb1, …. 。

掛載磁碟 (mount)

- Mount 的用途
  - 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。
- 範例
  - mount -t ext2 /dev/hda3 /mnt
    - 將 Ext2 格式的硬碟分割區 /dev/hda3 掛載到 /mnt 目錄下。
  - mount -t iso9600 -o ro /dev/cdrom /mnt/cdrom
    - 將 iso9600 格式的光碟 /dev/cdrom 以唯讀的方式掛載到 /mnt/cdrom 路徑當中
  - 可使用 unmount 將掛載的裝置移除

圖 10.17 UNIX/Linux 中的檔案與目錄概念



Linux 中檔案相關的系統呼叫

- open(), read(), write(), lseek(), stat(), opendir(), readdir() …

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

Linux 的輸出入系統

- 說明
  - 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
- 範例
  - 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。

Linux 的驅動程式

- 功能
  - 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。

- 範例
  - 字元類的裝置 (Character Device)
    - 又被稱為串流裝置 (Stream Device)
    - 像是鍵盤、滑鼠、印表機等
    - 必須支援基本的檔案操作, 像是 open(), read(), ioctl() 等。
    - 採用『註冊-反向呼叫』機制，掛載驅動程式。

結語

- 行程管理
  - 行程、執行緒 (thread)、切換、排程方法
  - 同步：競爭情況、鎖定、死結
  - 狀態：建立、執行、等待、就緒、死亡
- 記憶體管理
  - 分配策略：First Fit、Next-Fit、Best-Fit、Worst-Fit
  - MMU 硬體：重定位暫存器、基底界線暫存器、分段表、分頁表
- 檔案與輸出入
  - 檔案、目錄、磁區、驅動程式、註冊-反向呼叫

習題 (第1頁)

- 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？

習題 (第2頁)

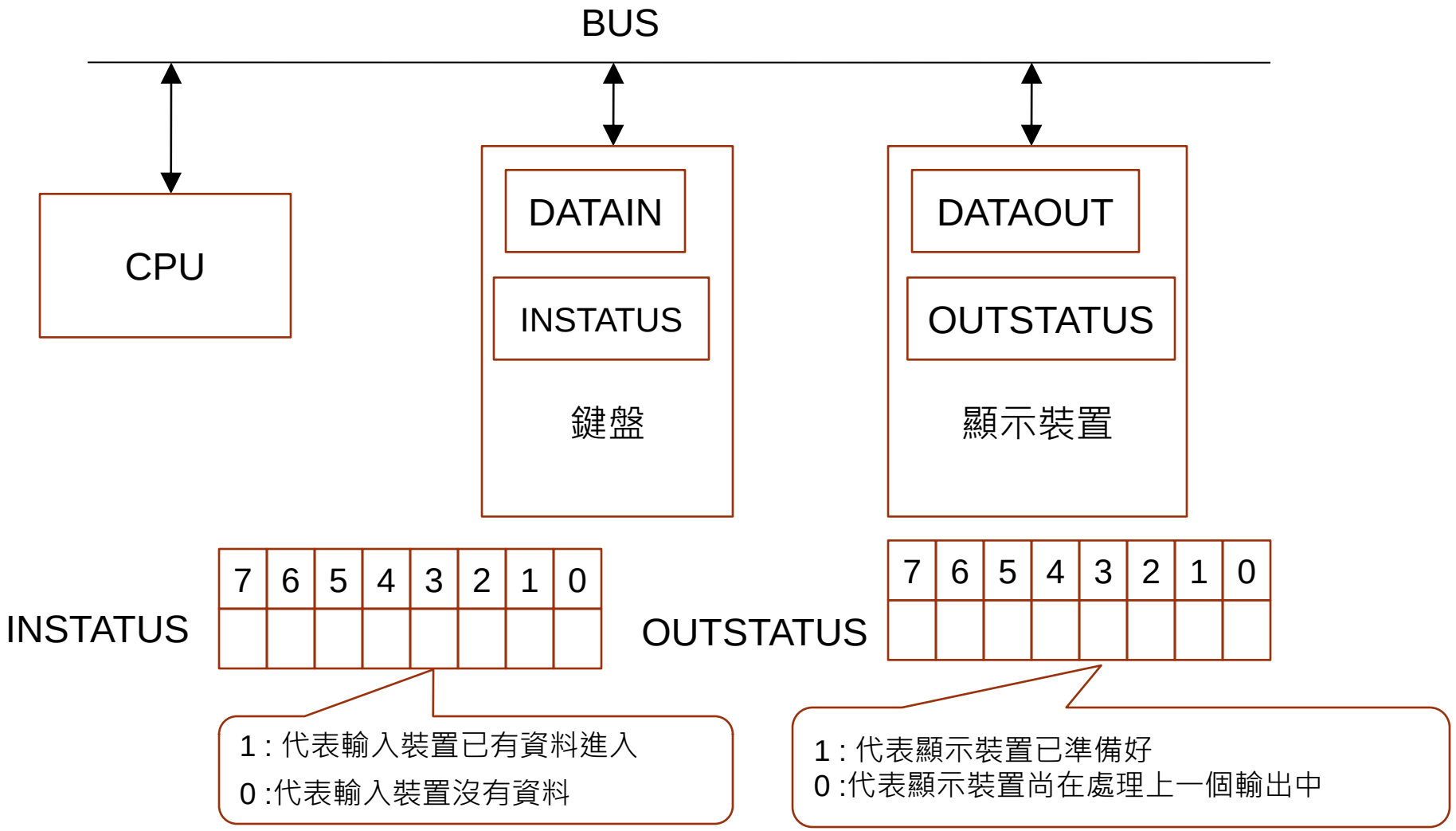
- 10.6　請說明 Linux 中的行程管理系統採用哪些策略？
- 10.7　請說明 Linux 中的記憶體管理系統採用哪些策略？
- 10.8　請說明 Linux 中的輸出入管理系統採用哪些策略？
- 10.9　請說明 Linux 中的檔案管理系統採用哪些策略？
- 10.10　請安裝 Ubuntu Linux 的最新的版本, 然後使用看看。

習題 (第3頁)

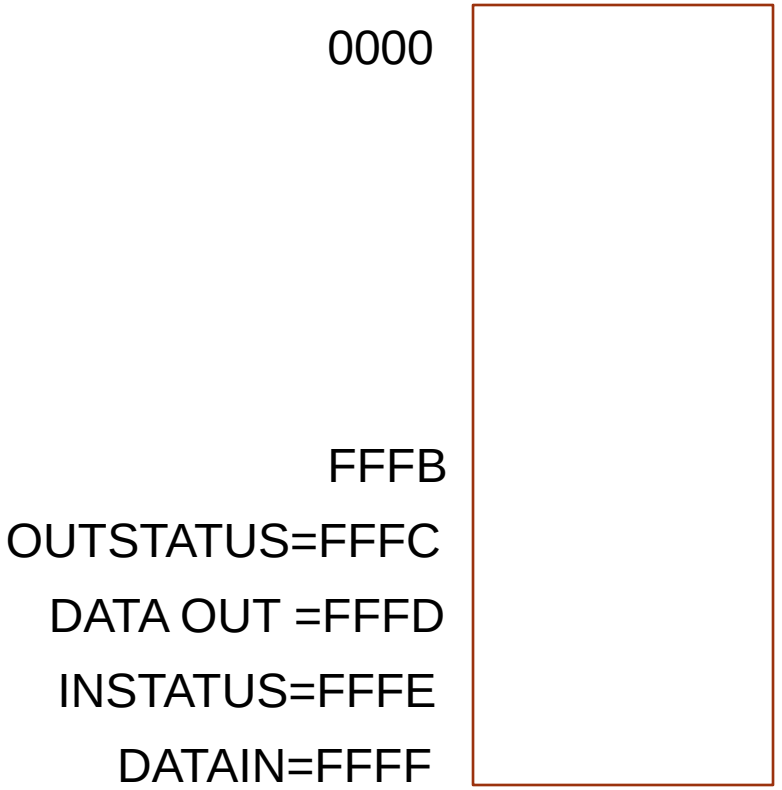
- 10.11 請於 Cygwin 環境下編譯 ch10/fork.c 檔案, 並執行看看。
- 10.12 請於 Cygwin 環境下編譯 ch10/thread.c 檔案, 並執行看看。
- 10.13 請於 http://www.kernel.org/ 網站中下載 Linux 核心原始碼的最新版本, 然後看看其中的 include/linux/sched.h、arch/x86/include/asm/ processor.h、arch/x86/include/asm/system.h 等原始碼, 試著理解其運作邏輯。

未納入書中

簡單的電腦輸出入模型


BUS
CPU
DATAIN
INSTATUS
鍵盤
DATAOUT
OUTSTATUS
顯示裝置
INSTATUS
7 6 5 4 3 2 1 0
1 : 代表輸入裝置已有資料進入
0 :代表輸入裝置沒有資料
OUTSTATUS
7 6 5 4 3 2 1 0
1 : 代表顯示裝置已準備好
0 :代表顯示裝置尚在處理上一個輸出中

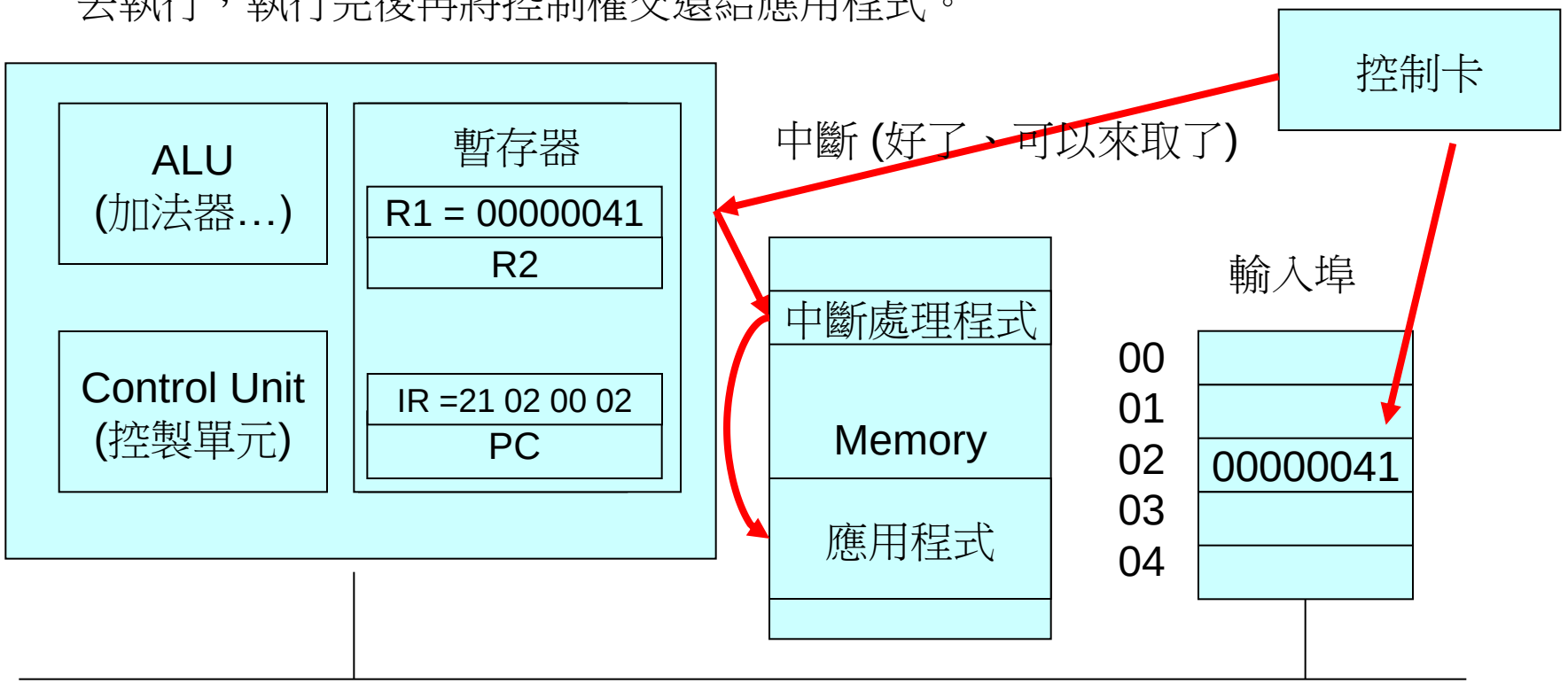
記憶體映射 I/O

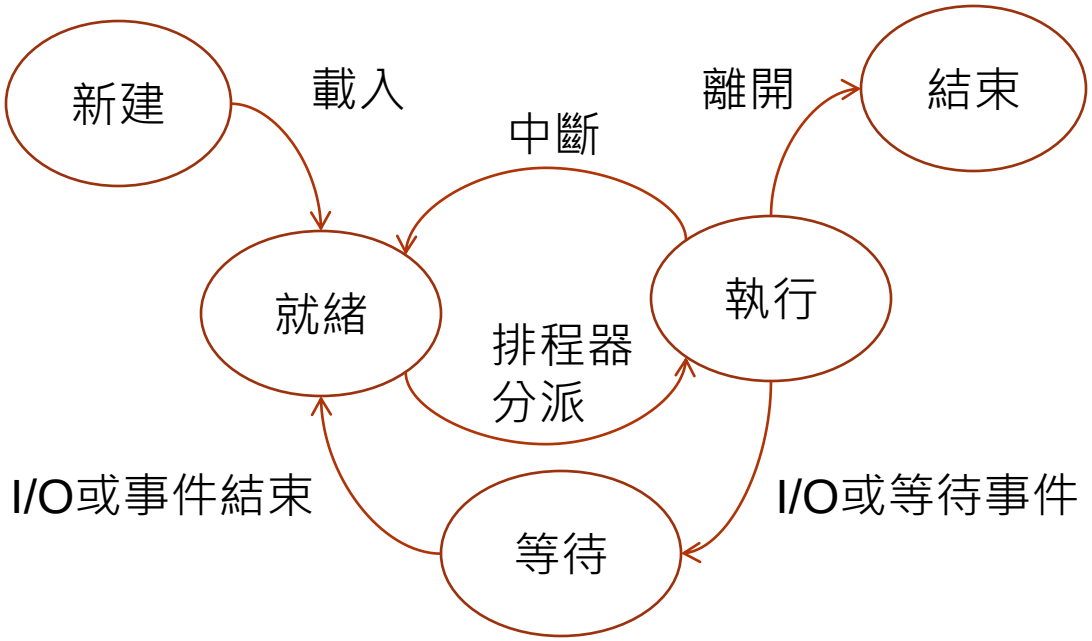

0000
FFFB
OUTSTATUS=FFFC
DATA OUT =FFFD
INSTATUS=FFFE
DATAIN=FFFF

中斷機制的原理

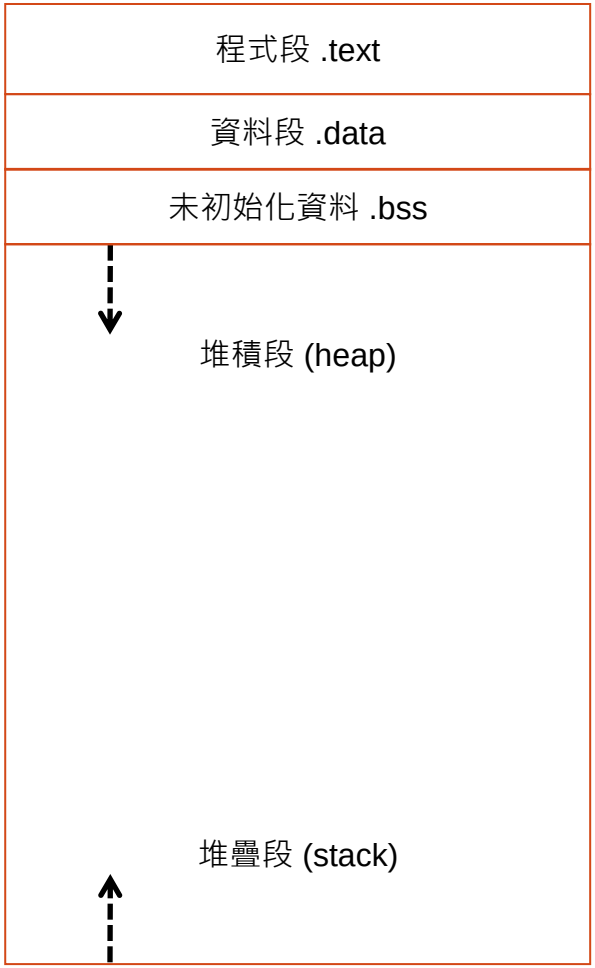
1. 必須靠硬體配合
2. 輸出入裝置取得資料後會產生中斷
3. 中斷造成 CPU 跳到特定的程式 (稱為中斷程式)
   去執行，執行完後再將控制權交還給應用程式。

Call READ INDEV

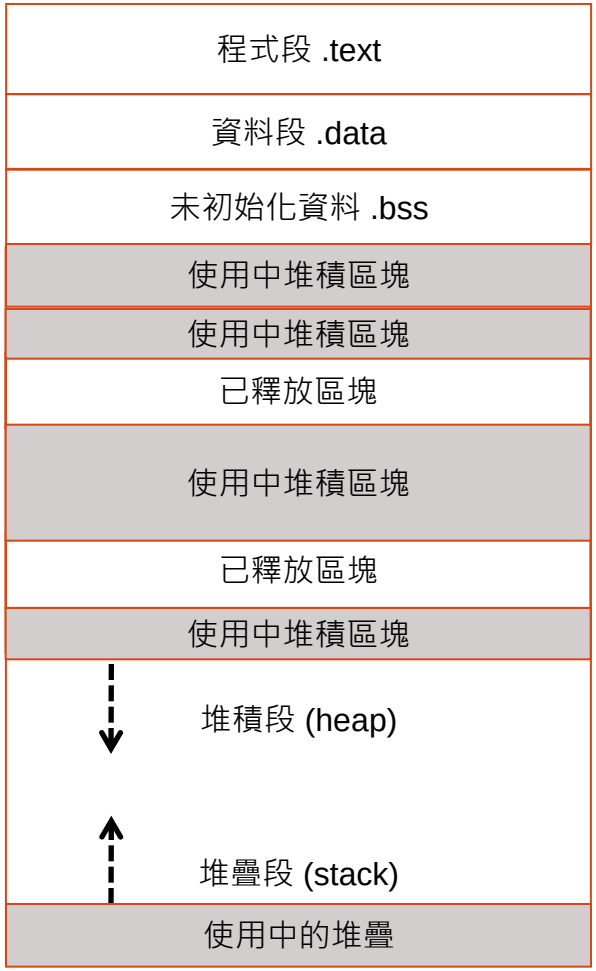



新建
載入
中斷
離開
結束
就緒
執行
排程器
分派
I/O或事件結束
I/O或等待事件
等待

行程的記憶體分配圖



(a) 程式開始時的記憶體分配情況



(b) 程式執行中的記憶體分配情況

程式的行為模式 – CPU 與 I/O 交替運行

```
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;
}
```

Linux 的行程切換程式碼 (IA32處理器)

Linux 2.6.29.4 檔案 arch/x86/include/asm/system.h

```
…
#define switch_to(prev, next, last) \
do {\
…
  unsigned long ebx, ecx, edx, esi, edi;
  asm volatile("pushfl\n\t"                    /* save flags */ \
    "pushl %%ebp\n\t"                          /* save EBP */ \
    "movl %%esp,%[prev_sp]\n\t"                /* save ESP */ \
    "movl %[next_sp],%%esp\n\t"                /* restore ESP */ \
    "movl $1f,%[prev_ip]\n\t"                  /* save EIP */ \
    "pushl %[next_ip]\n\t"                     /* restore EIP */ \
    "jmp __switch_to\n"                        /* regparm call */ \
    "1:\t" \
    "popl %%ebp\n\t"                           /* restore EBP */ \
    "popfl\n"                                  /* restore flags */ \
    \
….
while (0)
…
```

將新行程的程式計數器推入堆疊中

跳入 C語言的 switch_to() 函數中

由於 C語言函數在返回前會從堆疊中取出返回點，以返回上一層函數繼續執行。因此，switch 返回時會跳入新行程中。