Chapter 9

主記憶體

章節目標

- 探討邏輯位址和實體位址的差異，以及記憶體管理單元 (MMU) 在轉換位址中的角色
- 探討最先 (first) 配合、最佳 (best) 配合、最差 (worst) 配合策略來連續配置記憶體
- 說明內部斷裂與外部斷裂的差異
- 透過分頁系統包括轉譯旁觀緩衝區 (TLB) 將邏輯位址轉換到實體位址
- 描述階層式分頁、雜湊分頁和反轉分頁表
- 描述 IA-32、X86-64 和 ARMv8 架構的位址轉換

9.1 背景說明

- 主記憶體和建立在處理器內的暫存器，是 CPU 唯一可以直接存取的儲存體
  - 機器指令使用記憶體位址作為參數，但沒有使用磁碟位址作為參數
  - 任何執行的指令和任何被這些指令使用的資料必須放在這些直接存取儲存裝置內
- 大部份的 CPU 可以在每個時脈片段內，用一個或多個的操作速率對一個指令解碼，並且在暫存器內容上完成一些簡單的操作

9.1 背景說明

- 在此情形下，處理器通常需要**停頓** (stall)，因為它無法取得完成這個正在執行指令所需的資料
- 用來配合速度差異的記憶體緩衝器稱為**快取記憶體** (cache)
- 關心存取實體記憶體的相對速率，也必須確定正確操作

9.1.1 基本硬體

- **基底暫存器** (base register) 是用來存放最小的合法實體記憶體位址
  - **界限暫存器** (limit register) 則含有範圍的大小
- 記憶體空間的保護可以藉由 CPU 硬體比較每次使用者模式產生的位址與暫存器來完成

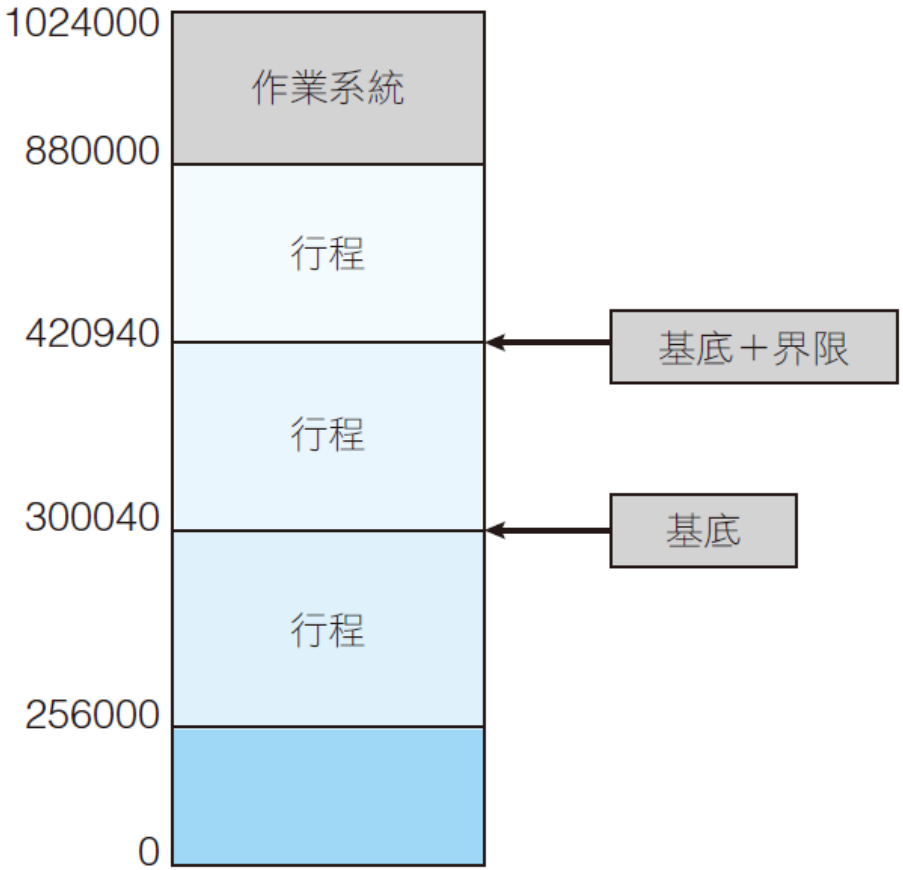
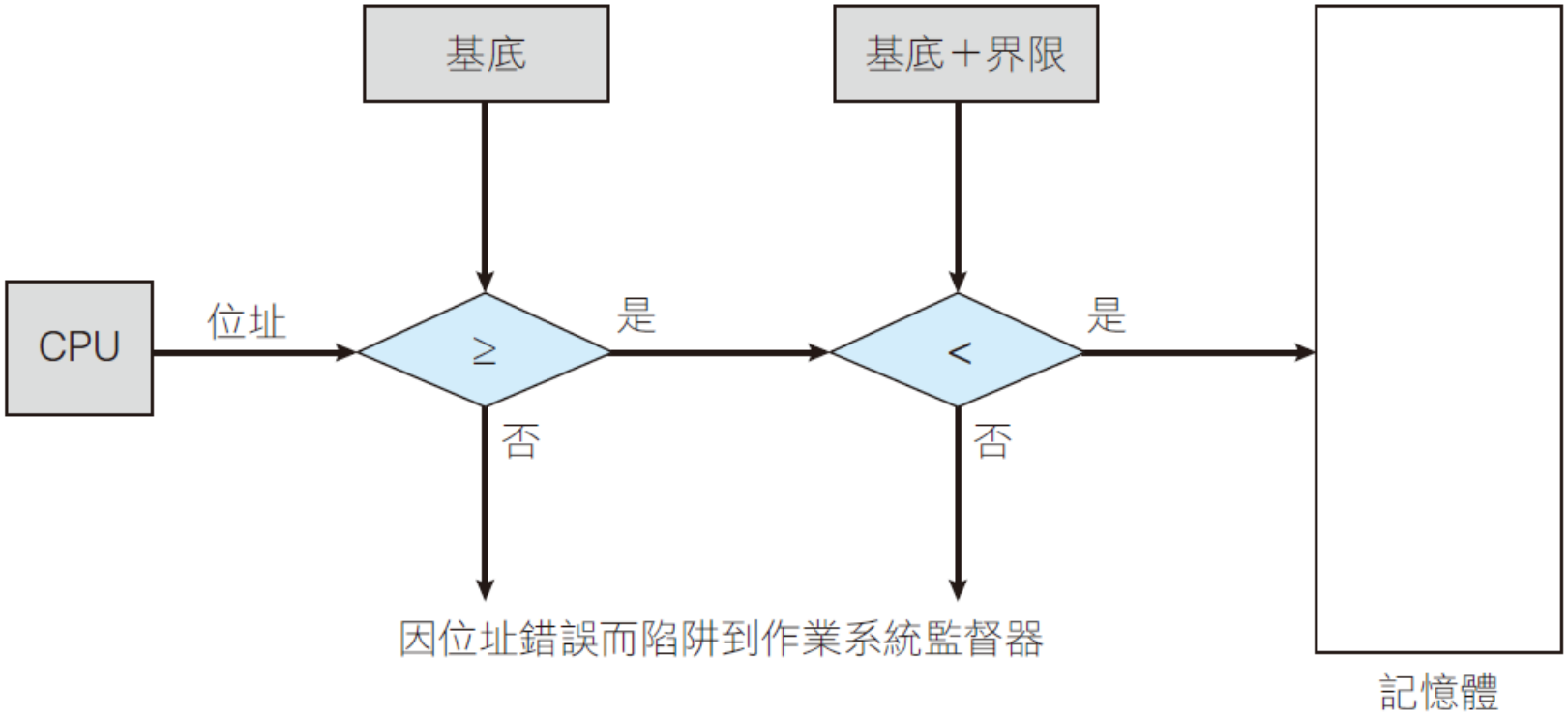


圖 9.1 一個基底及一個界限暫存器定義一個邏輯位址空間

圖 9.2　使用基底及界限暫存器的硬體位址保護



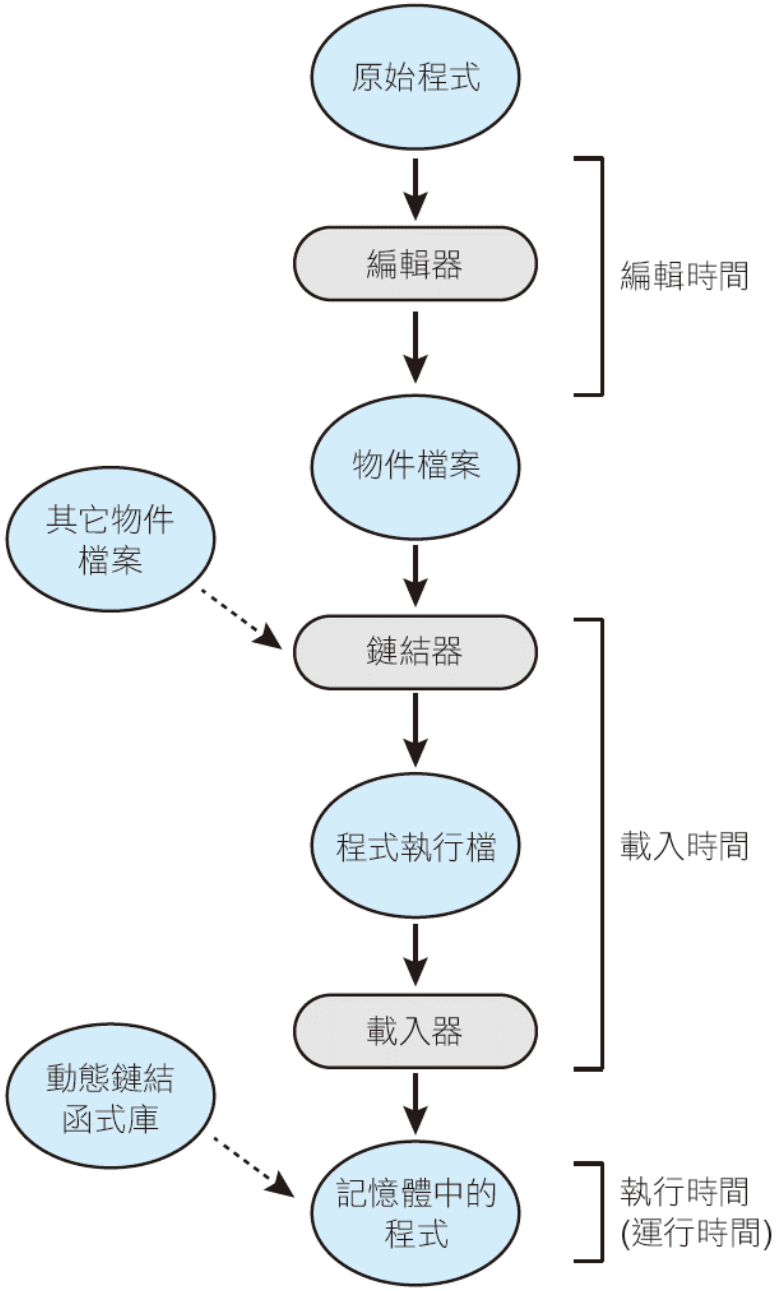
9.1.2 位址連結

- 隨著行程執行從記憶體存取指令和資料，最後該行程終止同時回收記憶體來給其它行程使用
- 大多數系統允許使用者行程存在實體記憶體中，雖然電腦的位址空間可以從 00000 開始，但使用者行程的第一個位址不一定是 00000
- 編譯器通常將這些符號化的位址**連結** (bind) 可重新定位的位址 (例如“離這個模組的開始處 14 個位元組”)
  - 鏈結器 (linker) 或載入器 (loader) 再依序將這些可重新定位的位址連結到絕對位址 (例如 74014)
  - 每一次定位的工作都是從一個位址空間映射到另一個位址空間

9.1.2 位址連結

- 傳統上，指令和資料至記憶體位址的連結可在下述方式的任何步驟中完成：
  - 在編譯時間時，若已確知程式在記憶體中的位置，絕對碼 (absolute code) 即可產生
  - 程式在編譯時間若不能確定在記憶體中的位置，則編譯器就必須產生可重定位程式碼 (relocatable code)
  - 執行時間 (execution time)

圖 9.3 一個使用者程式的多重步驟處理過程



9.1.3 邏輯位址空間和實體位址空間

- CPU 產生的位址通常稱為**邏輯位址** (logical address)，而記憶體單元看到的位址
  - 也就是載入到記憶體的**記憶體位址暫存器** (memory-address register) 之數值
  - 通常叫作**實體位址** (physical address)

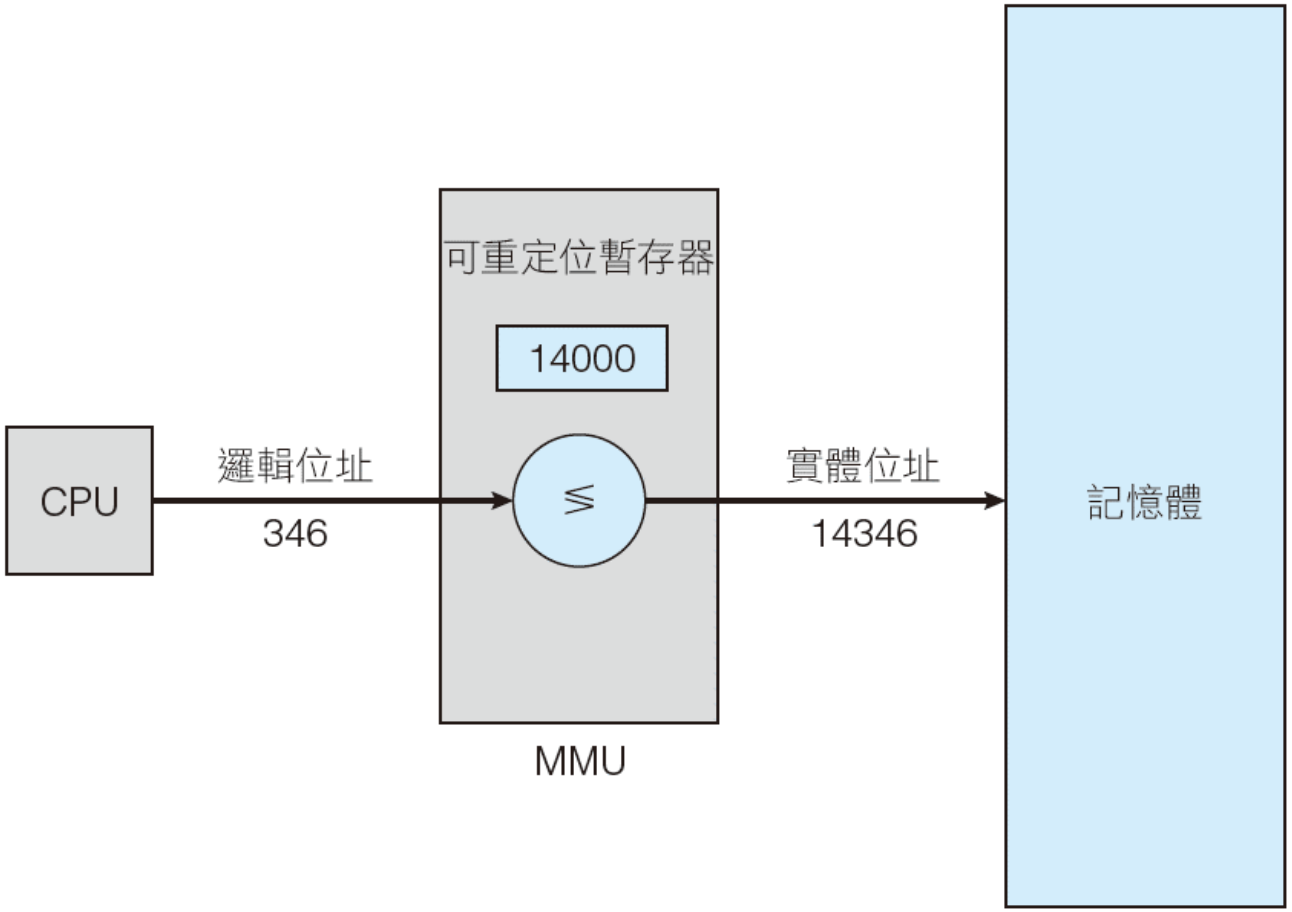
9.1.3 邏輯位址空間和實體位址空間

- 在編譯或載入時間的連結位址，一般具有相同的邏輯和實體位址
  - 執行時位址連結會導致邏輯和實體位址不同
  - 叫作**虛擬位址** (virtual address)
- 把邏輯位址和虛擬位址交替地使用
  - 一個程式產生的所有邏輯位址形成的集合叫作**邏輯位址空間** (logical address space)
  - 這些邏輯位址對應的實體位址之集合就稱為**實體位址空間** (physical address space)

9.1.3 邏輯位址空間和實體位址空間

- 在程式執行時，由虛擬位址映射到實體位址的工作是由一種稱為**記憶管理單元** (memory-management unit, MMU) 的硬體裝置來執行
- 目前我們將使用一個簡單的 MMU 技巧來說明 MMU
- 現在基底暫存器被叫作**可重定位暫存器** (relocation register)

圖 9.5　使用可重定位暫存器來動態地重新定位



9.1.4 動態載入

- 行程大小受限於實體記憶體大小。要得到較佳的記憶體空間使用率，可採**行動態載入** (dynamic loading)
- 動態載入的方式是將所有程式以可重定位載入的格式儲存在磁碟內
  - 一個常式只有在需要時會被載入，這種方法尤其是對於處理不常發生情況的大量程式碼特別有效

9.1.5 動態鏈結和共用程式庫

- 有些作業系統僅支援**靜態鏈結** (static linking)，其中系統程式庫就像其它目的模組一樣皆可由載入程式將之併入二進制的程式內
- DLL 也稱為**共享程式庫** (shared libraries)，並且在 Windows 和 Linux 系統中廣泛使用
- 這樣程式就能執行新的程式碼，如果程式庫的版本不相容，則程式和程式庫中都包含版本訊息

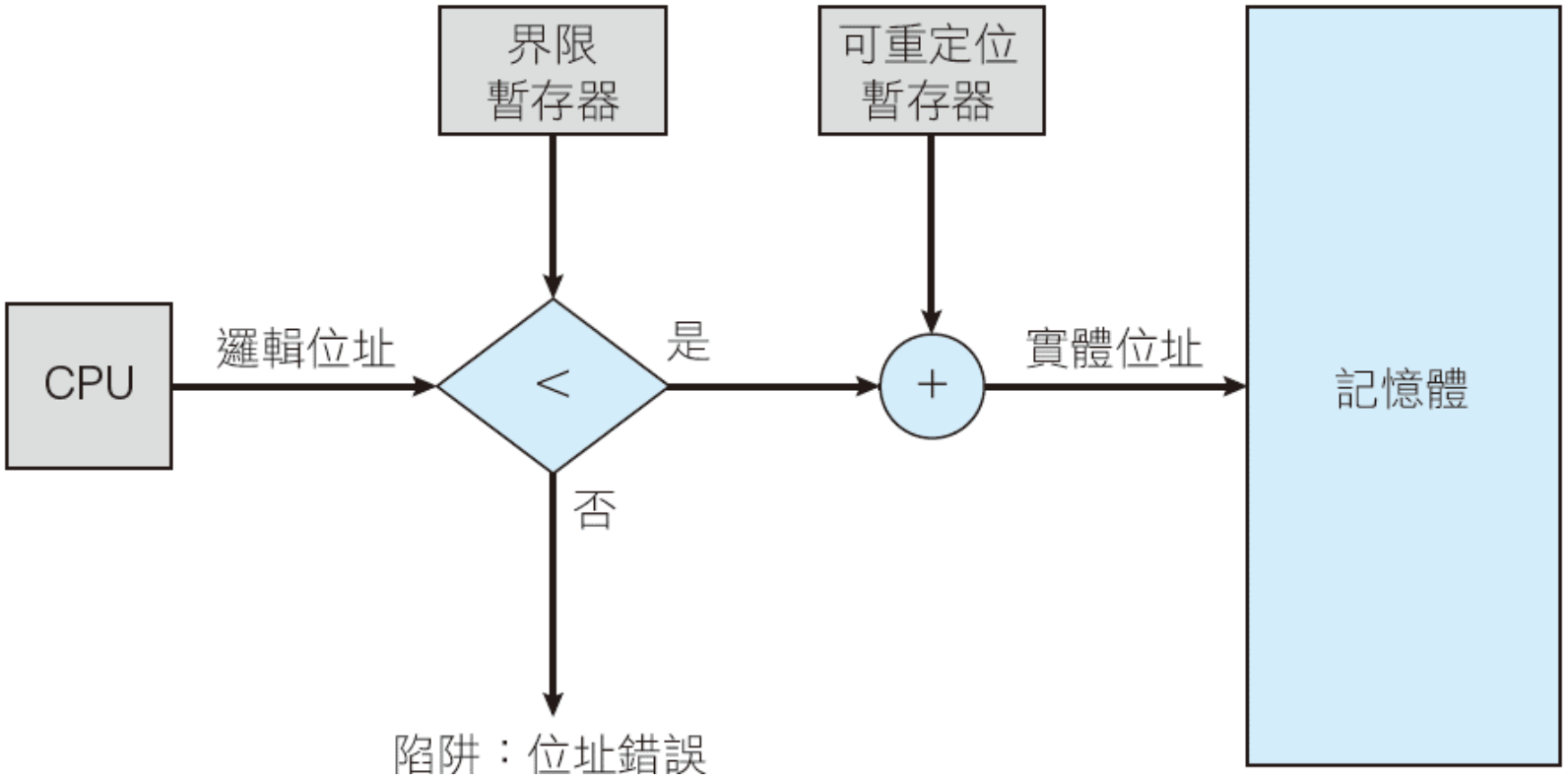
9.2 連續記憶體配置

- **連續記憶體配置** (contiguous memory allocation) 中，每個行程包含在記憶體中一個單獨連續區間
- 記憶體通常被分成兩部份：
  - 一份給作業系統
  - 一份給使用者行程

9.2.1 記憶體保護

- 一個系統，有可重定位暫存器 和界限暫存器
- 可重定位暫存器存放著最小的記憶體實體位址，而界限暫存器則存放著邏輯位址的範圍
- 每個邏輯位址都必須小於界限暫存器設定的範圍
  - MMU 藉著加上可重定位暫存器中的值來動態地重新定位，這個映射位址於是被傳送到記憶體

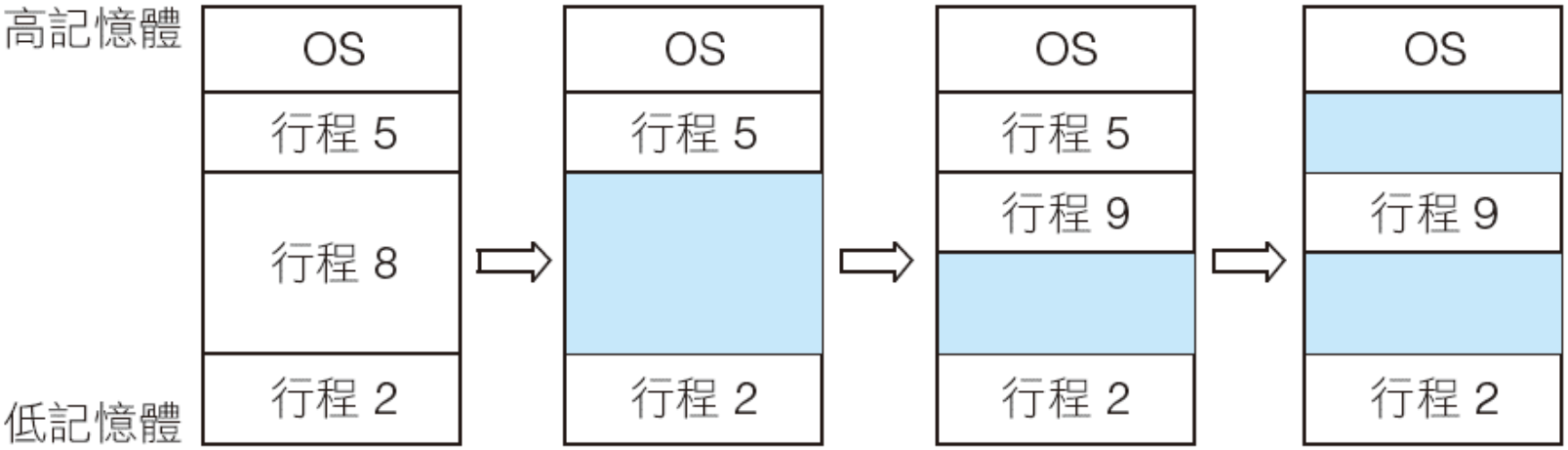
圖 9.6　可重定位暫存器和界限暫存器的硬體支援



9.2.2 記憶體配置

- 記憶體配置中最簡單的配置方法就是把主記憶體分成一些固定大小的分割 (partition)
  - 每個分割可能有一個行程要執行在**可變分割** (variable-partition) 方法中
- 整個記憶體就像是一大塊未劃分的區間，稱為**洞** (hole)
- 動態儲存體配置問題是指如何從可用區間去滿足 n 個大小。此問題有許多解決方法
  - 最先配合 (first-fit)
  - 最佳配合 (best-fit)
  - 最差配合 (worst-fit)

圖 9.7　可變分割



9.2.3 斷　裂

- 記憶體配置的最先配合和最佳配合這兩種策略，都會遭遇到外部斷裂 (external fragmentation) 的問題
  - 斷裂現象也將遺失其它的 0.5 N，則三分之一的記憶體可能沒有使用到，這就是著名的**百分之五十規則** (50-percent rule)
- 內部斷裂 (internal fragmentation)——小部份內部的記憶體沒有被使用

9.2.3 斷　裂

- 解決外部斷裂的問題，最常用的一種方法就是**聚集**(compaction)
  - 聚集的目的在於收集記憶體內零零散散的可用空間，使成一大區間
  - 聚集的動作並不是隨時可以進行的
  - 如果可重定位的動作是組譯或載入時完成，亦即可重定位方式採用的是靜態，則聚集的動作便不可行

9.3 分 頁

- **分頁** (paging) 是一種記憶體管理方法，允許行程的實體位址空間是不連續的
  - 分頁避免外部斷裂和相關的壓縮需求，這兩個問題困擾著連續記憶體配置
- 製作分頁的基本方法牽涉到實體記憶體被打散為許多稱為**欄** (frame) 的固定大小區塊
  - 和邏輯上的記憶體也被打散為大小相同稱為**頁** (page) 的區塊
- 頁數被當作指向**分頁表** (page table) 的索引

9.3 分 頁

- 任何由 CPU 產生的位址都分為兩個部份：
  - 一個**頁數** (page number, **p**)
  - 一個**頁偏移量**(page offset, **d**)
- 記憶體上的基底位址，這個基底位址再和頁偏移量結合在一起，而定義出送往記憶單元中的實體記憶體位址
- 如果邏輯位址空間的大小是2^m，而頁的大小是 2^n 位元組

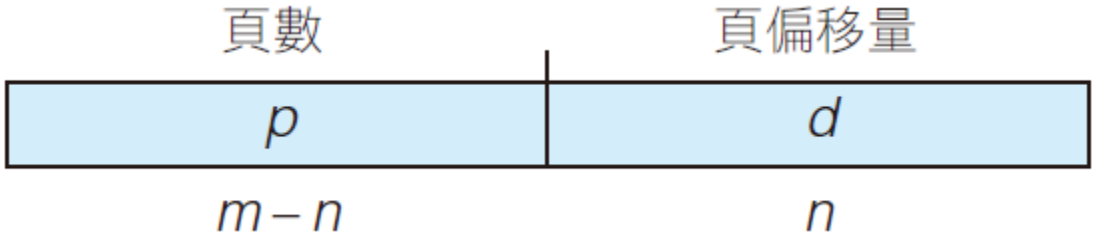


圖 9.8　分頁硬體

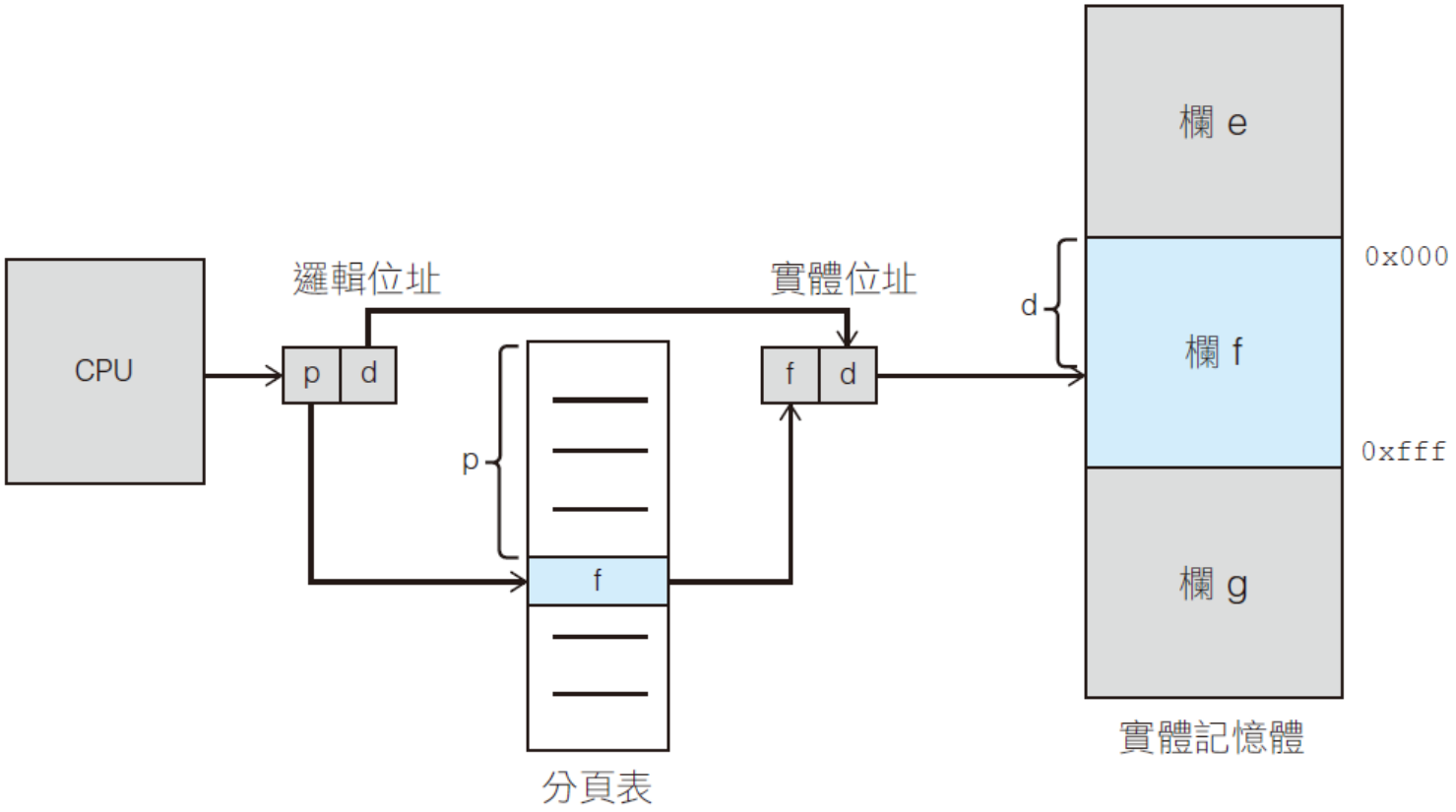


圖 9.9　邏輯與實體記憶體中的分頁模式

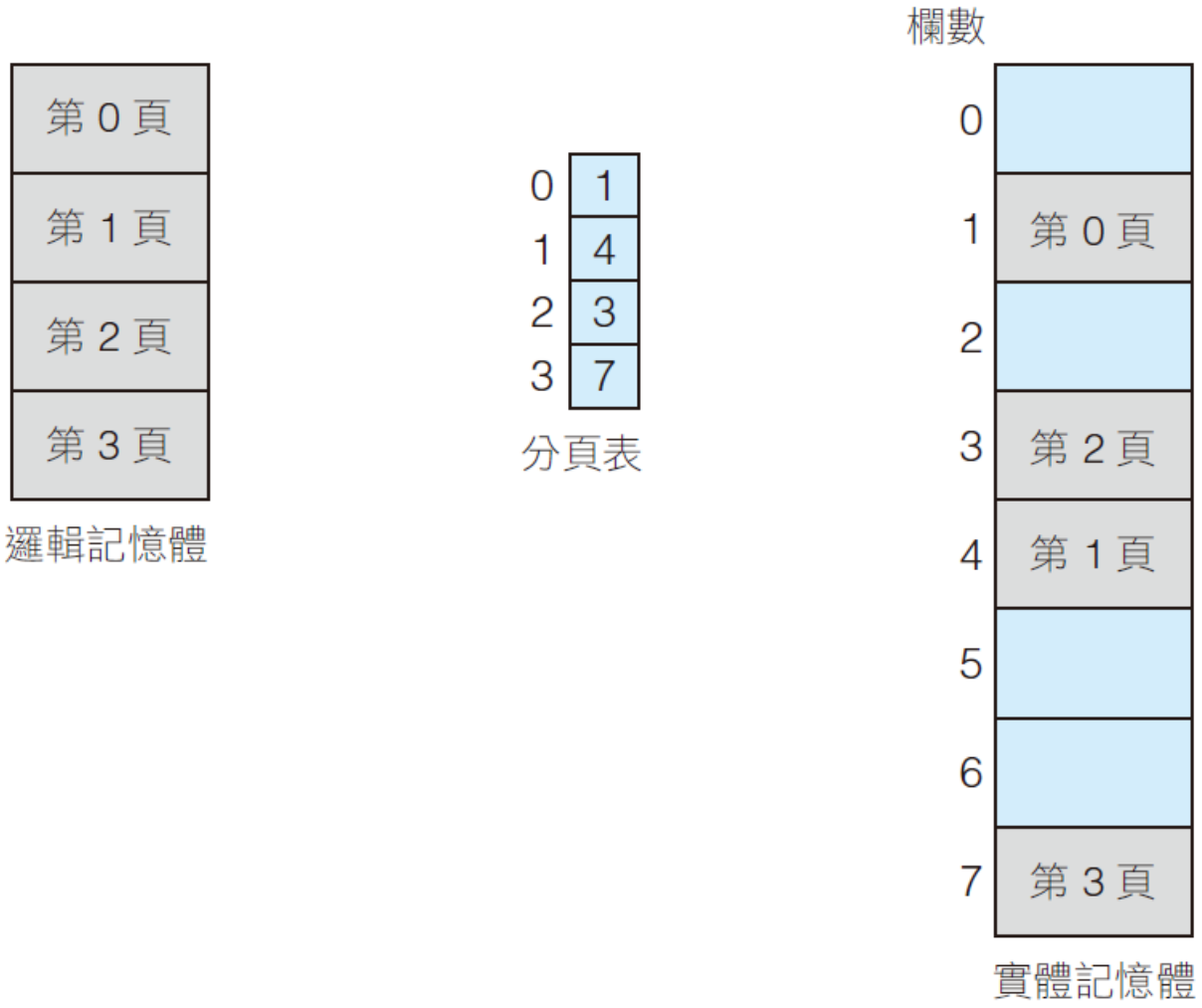
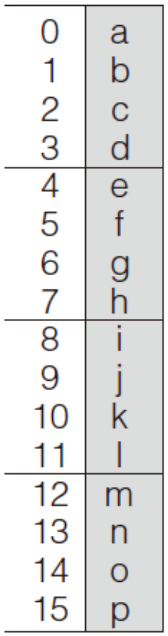
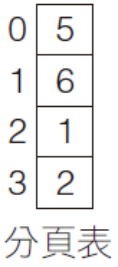
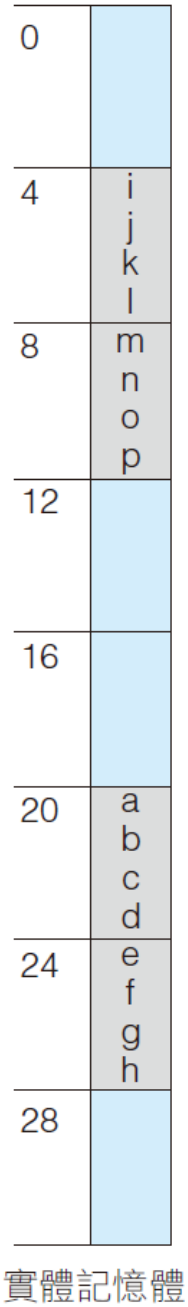


圖 9.10　以每頁 4 位元組分頁之 32 位元組記憶體的例子



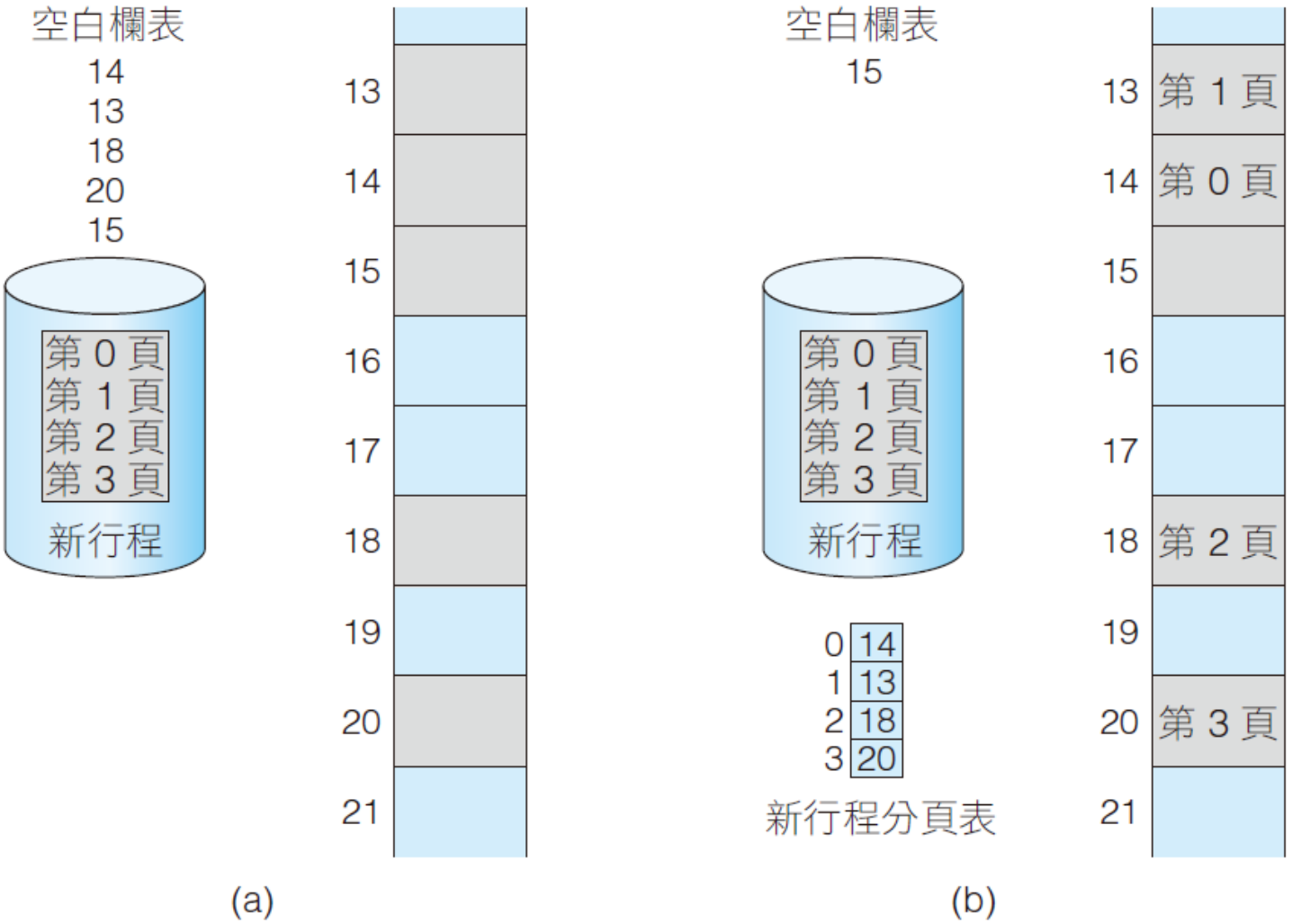




9.3 分 頁

- 舉例來說，如果一頁有 2,048 位元組，72,766 位元組的行程就需要 35 頁外加 1,086 位元組
- 當然要分配給它 36 個欄才可以，因而造成 2,048－1,086 = 962 位元組的內部斷裂
- 最差的情況就是，一個行程需要 n 頁外加 1 位元，就得分配 $n + 1$ 個欄，造成幾乎等於一整個欄的內部斷裂

圖 9.11　空白欄 (a) 之前的配置和 (b) 之後的配置



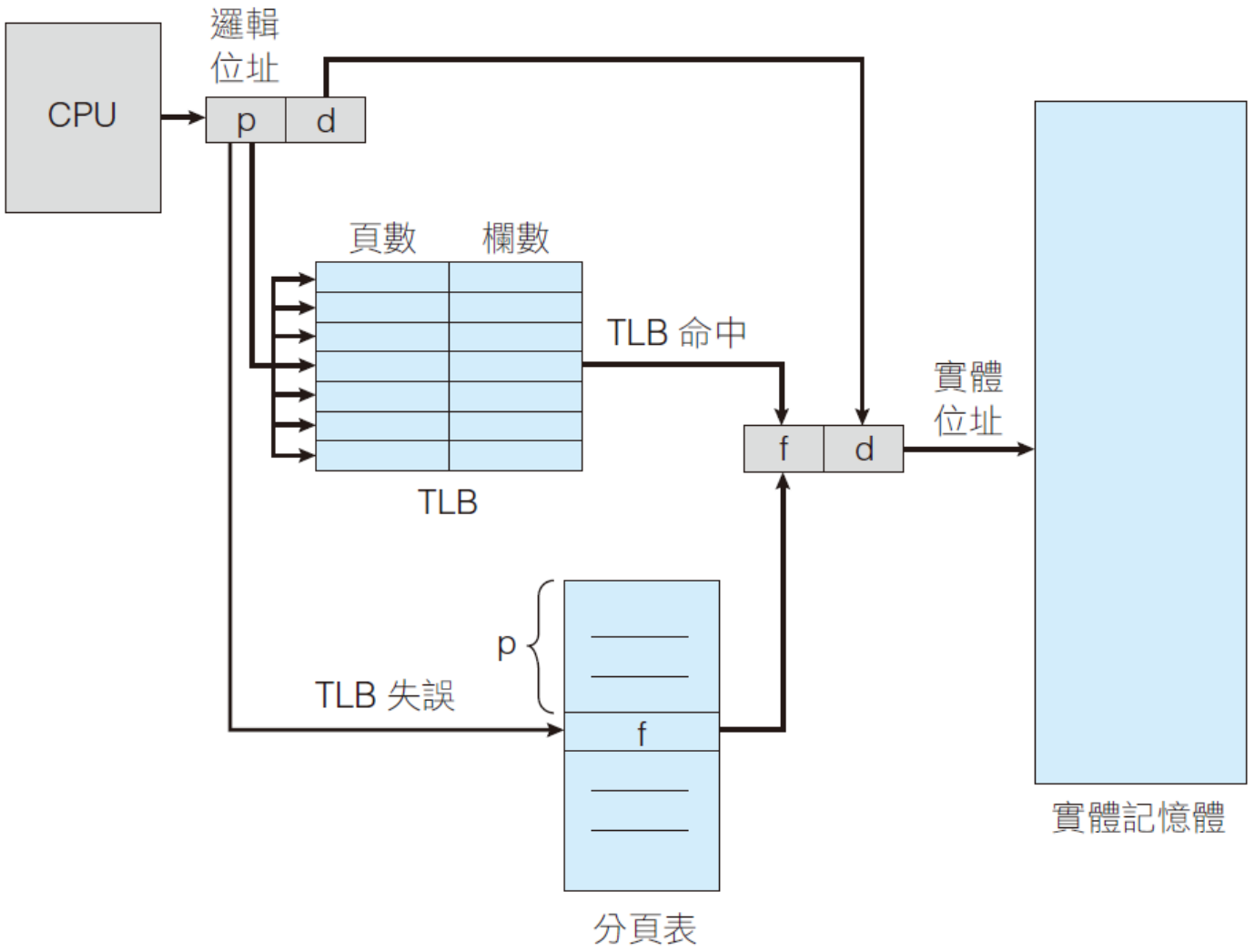
9.3 分 頁

- 若是將分頁表保存在主記憶體中**分頁表基底暫存器** (page-table base register, PTBR) 來指向分頁表
- 使用這個方法時，需要兩次記憶體存取來存取資料 (一次是為了分頁表，一次用於實際資料)
  - 因此速度上就慢了一倍，這種延遲在大多數情況下都是無法容忍的
- 使用一個特殊的小型硬體快取記憶體，稱為**轉譯旁觀緩衝區** (translation look-aside buffer, TLB)

9.3 分 頁

- 有些 TLB 允許其內容以**硬體繞線固定** (wired down)，這表示它們不能從 TLB 移除
- 有些 TLB 在每個 TLB 項目中儲存**位址空間識別碼** (address-space identifier, ASID)
  - 一個 ASID 可以唯一地識別該行程，並且用來提供該行程的位址保護
- 如果 TLB 沒有支援個別的 ASID 時，每次選中新的分頁表時，TLB 就必須**全部清除** (flushed)，以確保下一個執行的行程不會使用錯誤的轉換資訊

圖 9.12　使用 TLB 的分頁硬體



9.3 分 頁

- 相關暫存器中找出一個頁數所花時間的百分比稱為**命中率** (hit ratio)
  - 所謂 80% 的命中率，就是 80% 的時間中，我們都可以在相關暫存器裡找到想要的頁數
- 如果存取記憶體要花 10 奈秒 (ns)，則當有頁數在相關暫存器時做的映射式記憶體存取，一共要花 10奈秒

9.3 分 頁

- **有效記憶體存取時間** (effective memory-access time)

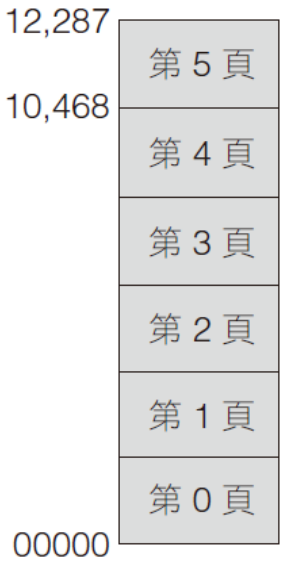
$$\text{有效存取時間} = 0.80 \times 10 + 0.20 \times 20$$
$$= 12 \text{ ns}$$

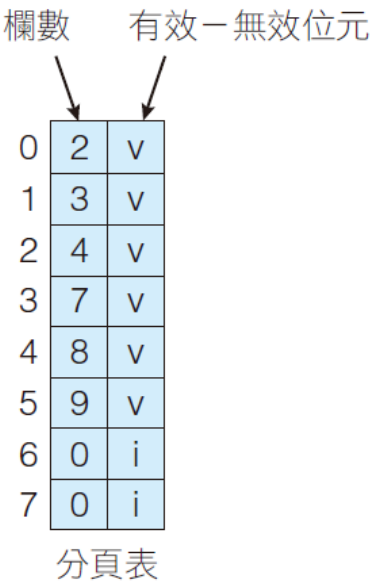
$$\text{有效存取時間} = 0.99 \times 10 + 0.02 \times 20$$
$$= 10.1 \text{ ns}$$

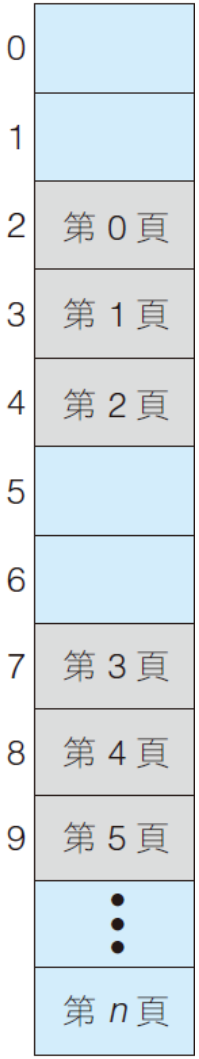
保　護

- 在分頁的環境中，記憶體的保護是靠每欄上的保護位元來完成的
- 分頁表中的每一項都會加上另一個額外的位元：**有效—無效** (valid-invalid)位元
  - 有效時，則表示這一頁是在此行程的邏輯位址空間內，因此是一個合法 (或有效) 的頁
  - 如果這個位元被設定成無效，則表示這一頁不在此行程的邏輯位址空間內
- 有些系統以**分頁表長度暫存器** (page-table length register, PTLR) 的方式提供硬體，來表示分頁表的大小

圖 9.13　分頁表中的有效 (v) 或無效 (i) 位元



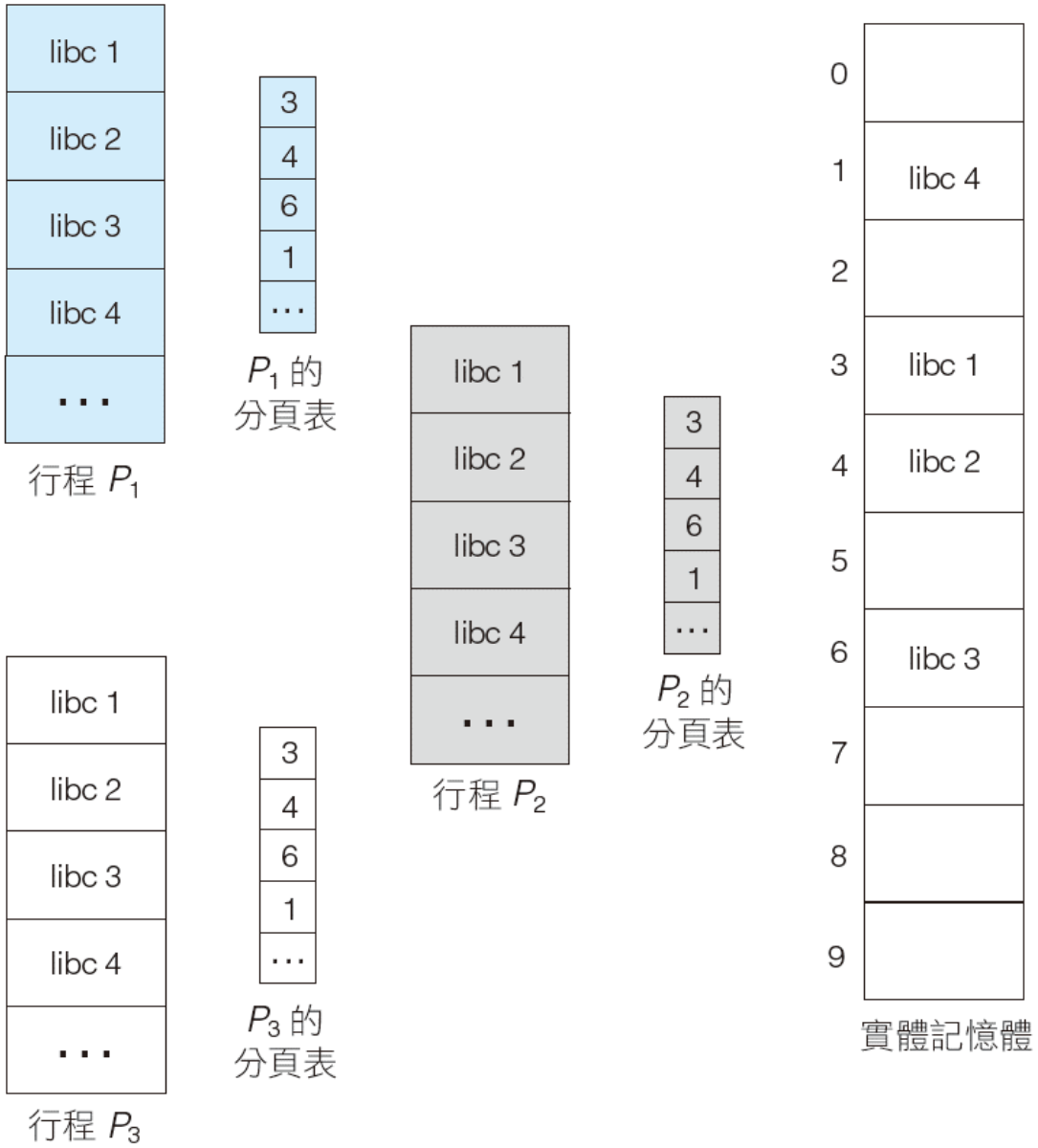




9.3.4 共用分頁

- 分頁的優點是可以共享通用程式碼，這在具有多個行程的環境中尤為重要
- 程式碼都是**可重進入的程式碼** (reentrant code)
  - 則程式碼可以被共用

圖 9.14 在分頁環境中共用標準 C 程式庫



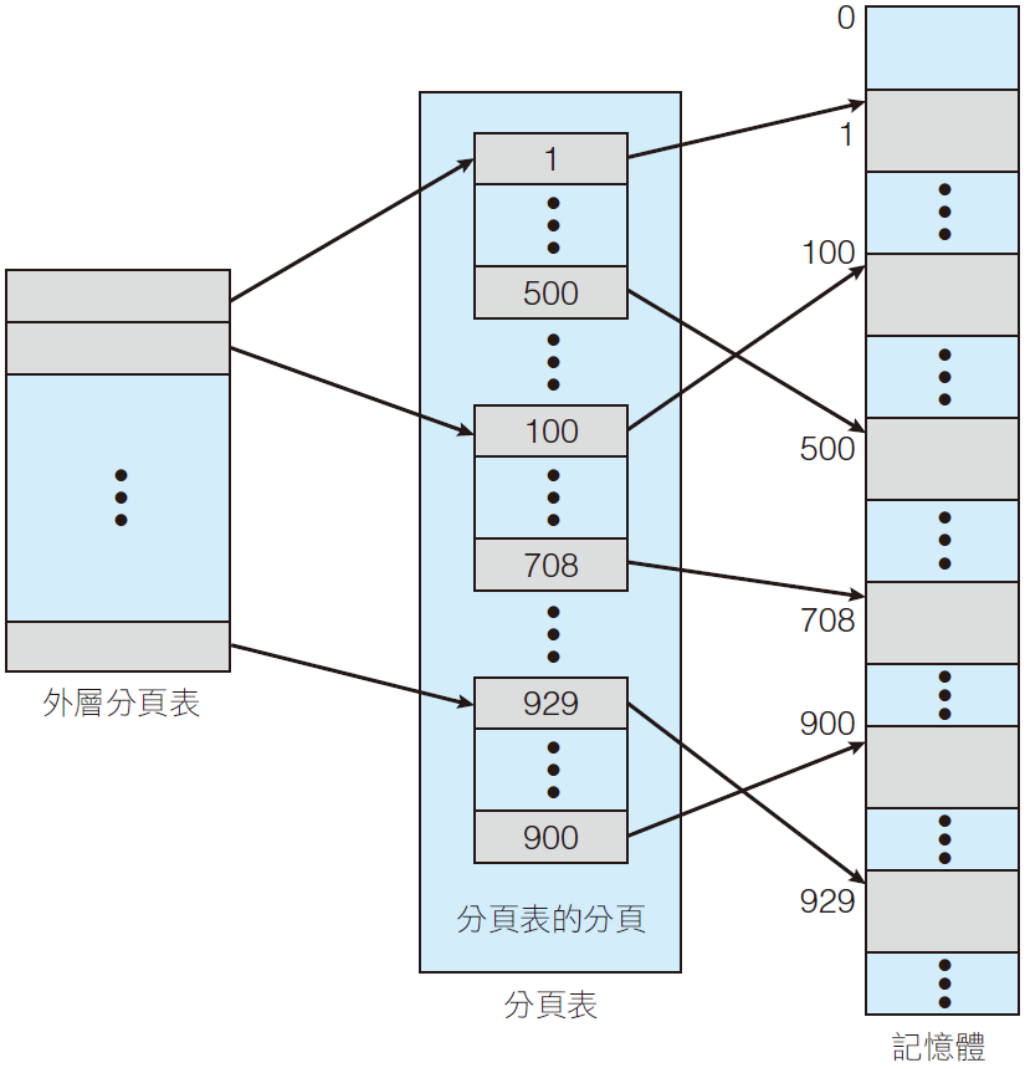
9.4 分頁表的結構

- 分頁表結構化的常見技巧，包括
  - 階層式分頁
  - 雜湊分頁表
  - 反轉分頁表

9.4.1 階層式分頁

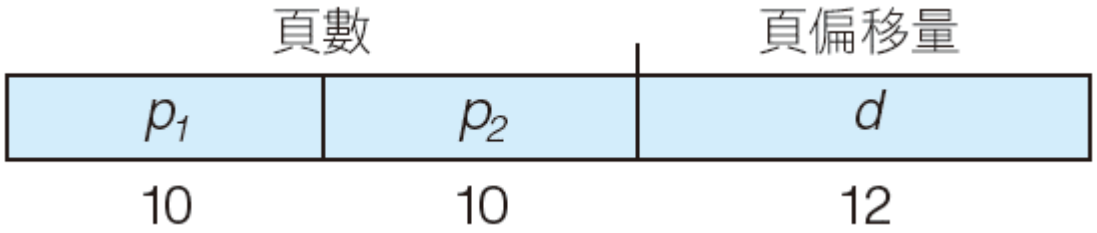
- 考慮一個有 32 位元邏輯位址空間的系統，如果在這個系統中每頁的大小是 4 KB (2^{12})，則分頁表必須有 100 萬項 ($20^{20} = 2^{32}/2^{12}$)
  - 因為每一項有 4 位元組，所以每個行程僅分頁表就需要 4 MB 的實體位址空間
- 問題有一種簡單的解決方法，就是把分頁表分成較小的片段。有幾個方法可達成以上的要求
- 一個方法就是使用兩層分頁演算法，也就是分頁表本身也由分頁的方法得到

圖 9.15　兩層分頁表的技巧



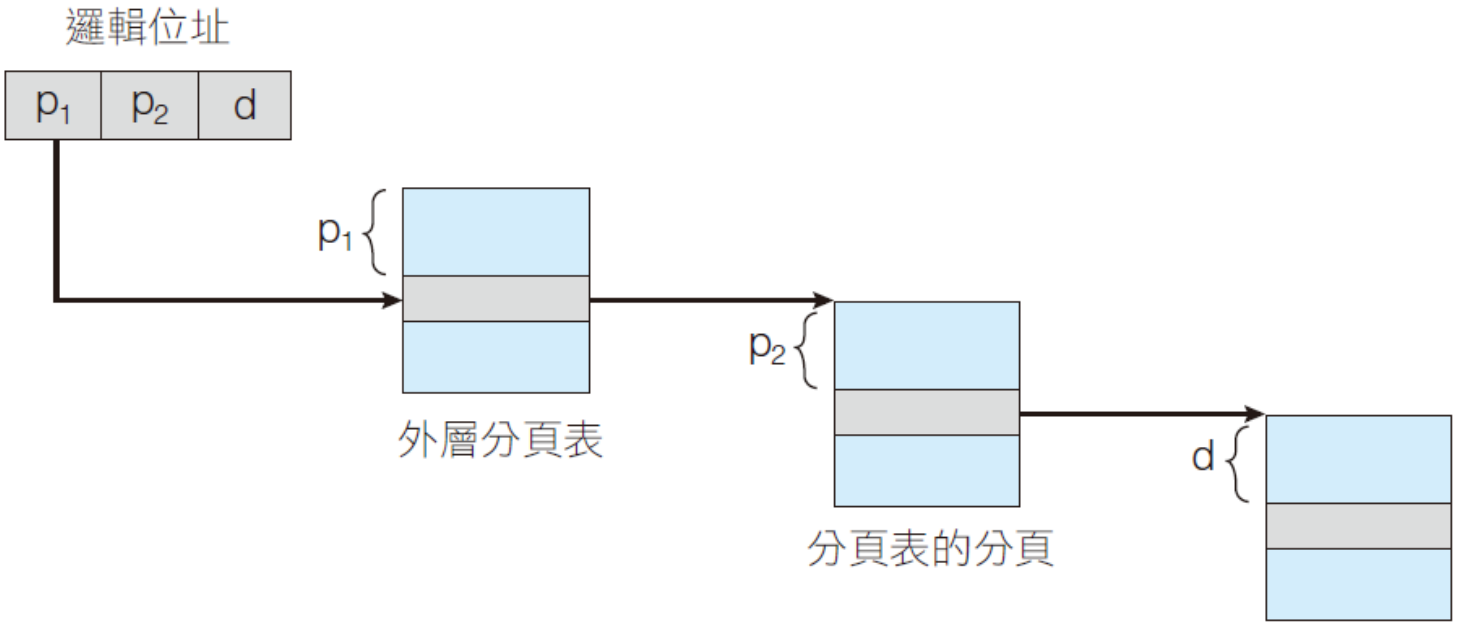
9.4.1 階層式分頁

- 前面 32 位元邏輯位址空間和每一頁的大小是 4 KB 的例子
- 邏輯位址被分成 20 位元的頁數和 12 位元的頁偏移量
- 因為我們將分頁表分頁，頁數進一步地被分成 10 位元的分頁數和 10 位元的頁偏移量



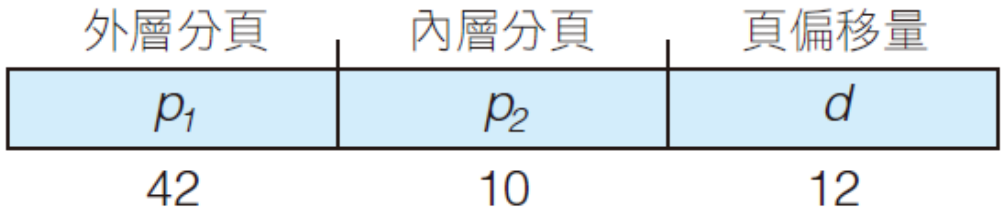
- 因為位址轉換由外層分頁表向內做起，這種方法也稱為**向前映射** (forward-mapped) 分頁表

圖 9.16　32 位元兩層分頁架構的位址轉換



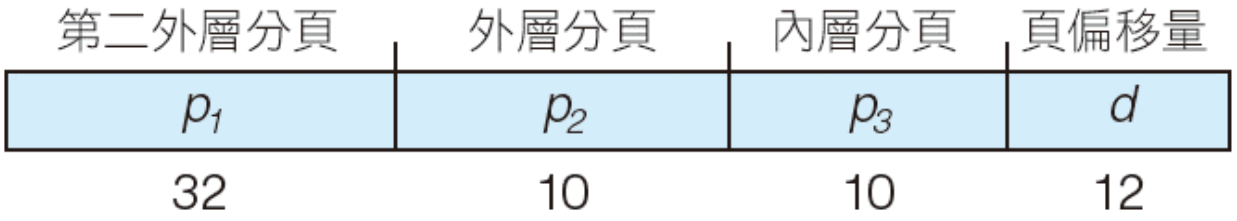
64 位元邏輯位址空間

- 兩層分頁的方法就不再適用
  - 4 KB (2^{12})
  - 2^{52}
  - 2^{10} 個 4 位元組



- 外層分頁表由 2^{42} 項 (或 2^{44} 位元組) 所構成。避免一個如此巨大表格的明顯方法是，將外層分頁表分為兩個較小的片段
  - 這種作法也用在一些 32 位元的處理器，以增加彈性和效率

64 位元邏輯位址空間



- 外層分頁表的大小仍是 2^{34} 位元組 (16 GB)
- 第二外層分頁表本身要再分頁

9.4.2 雜湊分頁表

- 處理位址空間大於 32 位元的一種常見方法是，使用**雜湊分頁表** (hashed page table)，其中雜湊值即是虛擬分頁值
- 雜湊表中每一項包括雜湊到相同位置 (以處理碰撞) 之單元的鏈結串列
- 每個單元由三個欄位組成：

  (1) 虛擬分頁的數值

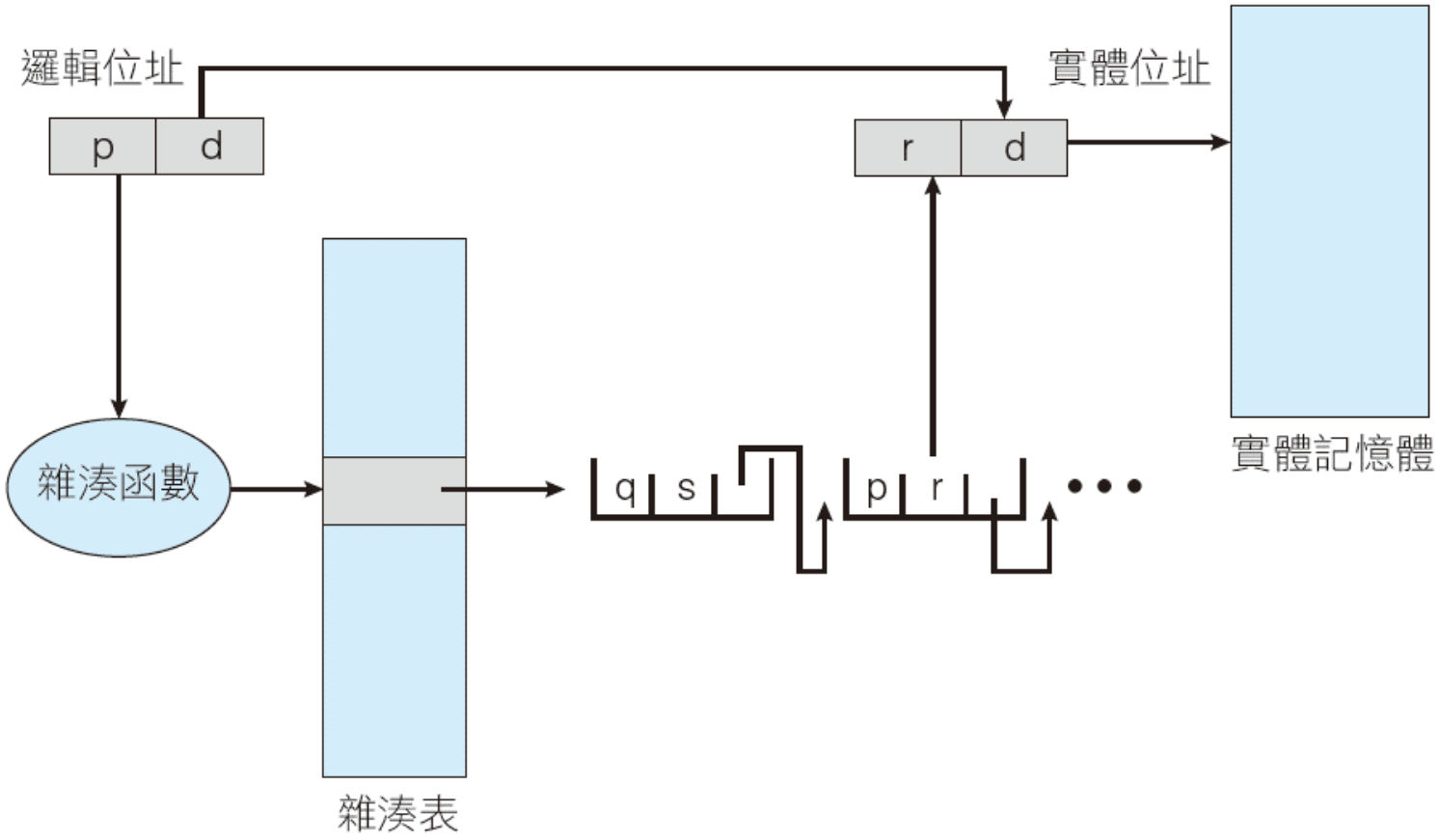
  (2) 映射分頁欄的數值

  (3) 指向鏈結串列下一單元的指標

9.4.2 雜湊分頁表

- 下列的方式進行：
  - 虛擬位址的虛擬分頁數值被雜湊 (hashed) 到雜湊表中
  - 虛擬分頁數值和鏈結串列中第一個單元的欄 1 做比較
  - 如果兩者相同，相關分頁欄位 (欄2) 就被用來形成所需要的實體位址
  - 如果不吻合，鏈結串列中接下來的單元會被搜尋，以找出符合的虛擬分頁數值

圖 9.17　雜湊分頁表



9.4.2 雜湊分頁表

- 一種適合 64 位元位址空間的此種技巧之變形已被提出，這種變型使用**叢集分頁表** (clustered page table)，與雜湊分頁表相似，但是雜湊分頁表中的每一項是參考到幾個分頁，而非單一頁
  - 一個單一分頁表的紀錄可以儲存許多實體分頁欄的映射
  - 叢集分頁表特別適合**鬆散** (sparse) 的位址空間，其中參考記憶體是不連續且散佈在整個位址空間中

9.4.3 反轉分頁表

- 每個行程都有自己的一份分頁表，這份分頁表對於此行程正在使用的每一頁都有一項進入點
  - 或者說是每個虛擬位址，無論是否有效都有一個對映位址
- 雖然這種方法降低儲存每一份分頁表需要的記憶體量，但卻增加參考到一頁時，搜尋分頁表需要的時間
  - 為了增進效能，我們可以在參考雜湊表之前先搜尋TLB

9.4.3 反轉分頁表

- 該分頁表允許將多個虛擬位址映射到同一實體位址
- 此方法不能用於反轉分頁表
  - 因為每個實體分頁只有一個虛擬分頁項目，所以一個實體分頁不能具有兩個 (或多個) 共享虛擬位址
  - 因此對於反轉分頁表而言，僅一個虛擬位址映射共用實體位址可能在任何時間發生

9.4.4 Oracle SPARC Solaris

- 一個近代 64 位元 CPU 和作業系統緊密結合，以提供低負擔的虛擬記憶體
  - 在 **SPARC** CPU 上執行的 **Solaris** 是一個完整的 64 位元作業系統
  - 因此必須在不會用完所有的實體記憶體以保存多層次分頁表的前提下，解決虛擬記憶體的問題
- 雜湊分頁表有效地解決問題。有兩個雜湊表
  - 一個給核心使用
  - 一個給所有的使用者行程

9.4.4 Oracle SPARC Solaris

- 一個從虛擬記憶體映射記憶體位址到實體記憶體
- 每一個雜湊表的項次代表一個對映到虛擬記憶體的連續區域
  - 這比每一分頁有一個獨立的雜湊表項次有效率
- 每個項次有一個基底位址和一個跨度指示此項次表示的分頁數目

9.4.4 Oracle SPARC Solaris

- 如果每一位址都要求搜遍雜湊表，則虛擬到實體轉換將花費很長時間
  - 所以 CPU 製作一個存放轉換表格項次 (translation table enty, TTE) 以做快速硬體查詢的TLB
- 這些 TTE 存放在轉換儲存緩衝區 (translation storage buffer, TSB) 做快取，其中包含每個最近存取分頁的項次
- 當虛擬位址的參考發生時，硬體搜尋 TLB 以做轉換
  - 如果找不到時，硬體會走遍記憶體中的 TSB，搜尋映射到造成此搜尋之虛擬位址的 TTE，這種 **TLB 走遍** (TLB walk) 功能在許多近代 CPU 都可以找到

9.4.4 Oracle SPARC Solaris

- 如果在 TSB 找到吻合的，CPU 會複製 TSB 項次到 TLB，然後就完成記憶體轉換
- 如果在 TSB 沒有找到吻合的，核心被中斷以搜尋雜湊表
- 核心從適當的雜湊表產生 TTE，儲存此 TTE 到 TSB，以便被 CPU 的記憶體管理單元自動載入 TSB
- 中斷處理器交還控制權給 MMU，MMU 完成位址轉換，並從主記憶體取出要求的位元組和字元組

9.5 置 換

- 行程指令和資料必須在記憶體被執行
  - 一個行程或是一部份的行程可能會暫時被置換 (swapped) 出記憶體到**備份儲存體** (backing store)，
  - 再回到記憶體被繼續執行置換讓所有行程的實體記憶體可以超出系統的真實實體記憶體
  - 因此增加一個系統多元程式的程度

9.5.2 分頁置換

- 大多數系統，包括 Linux 和 Windows 都使用置換的變形，即可以在一個分頁中而不是整個流程分頁中進行置換
  - 置換一般是指標準置換
  - 分頁是指與分頁的置換
  - **移出** (page out) 作業方式是將分頁從記憶體移動到備份儲存體
  - 反向過程稱為**移入** (page in)

圖 9.19　利用磁碟當備份儲存體來置換兩個行程

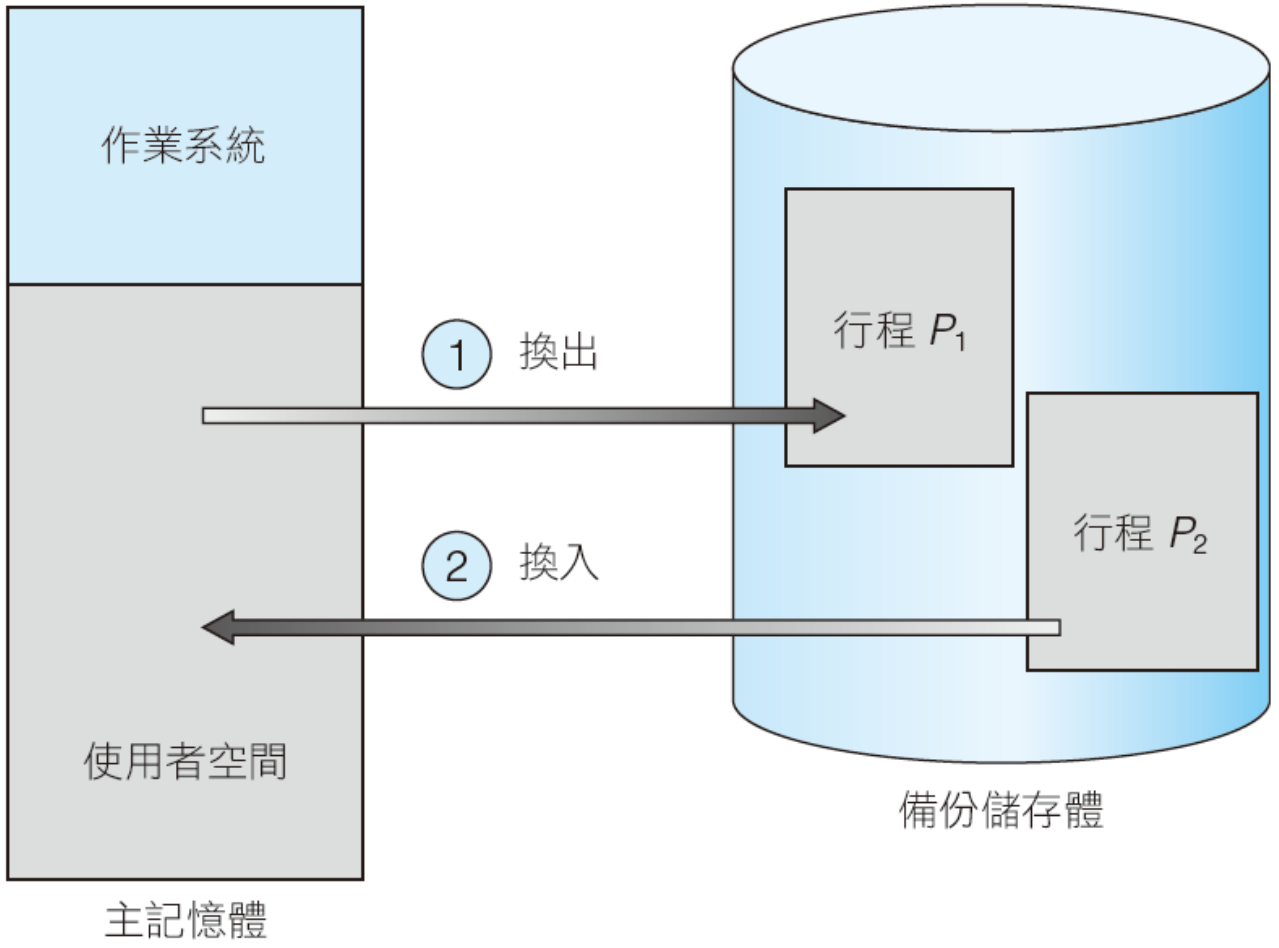
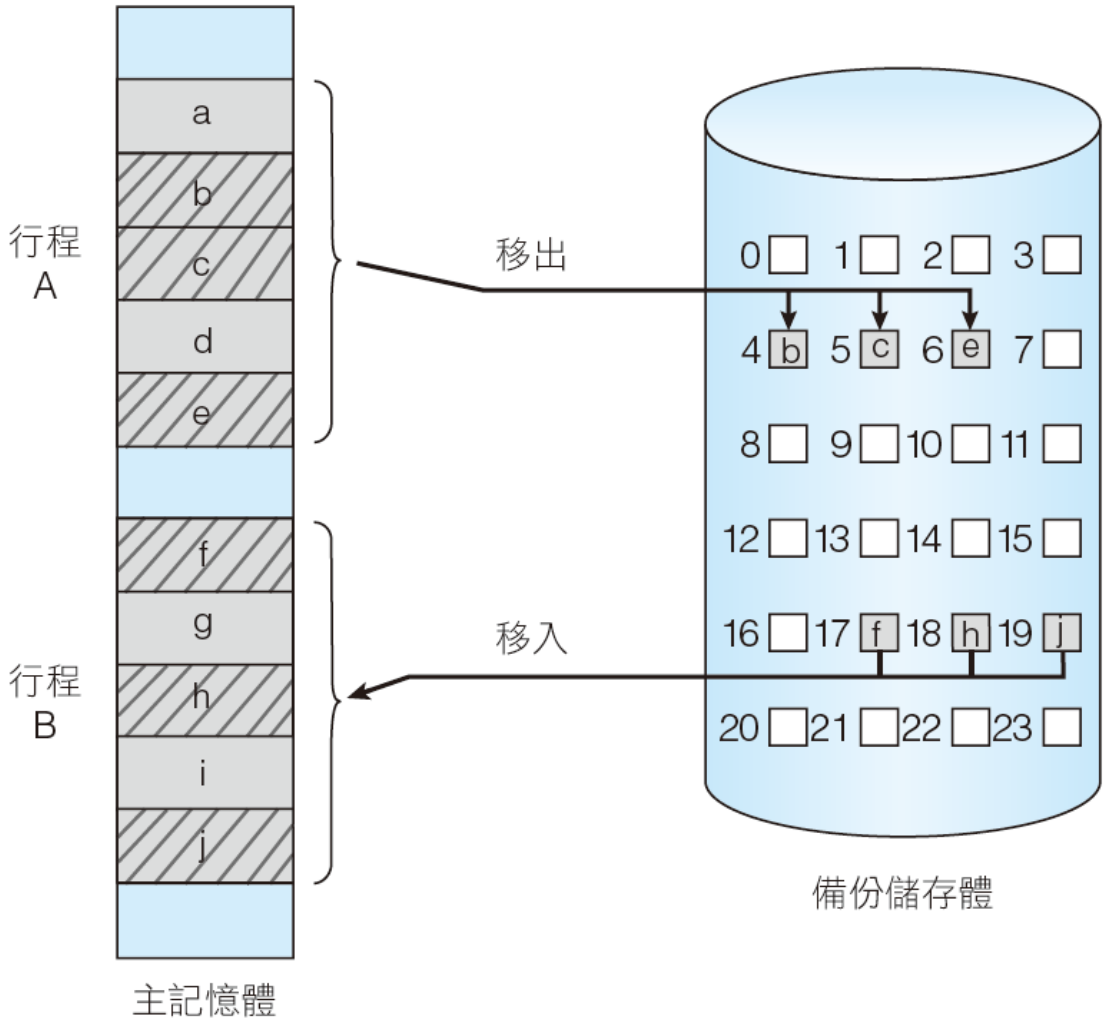


圖 9.20　置換分頁


行程
A
a
b
c
d
e
行程
B
f
g
h
i
j
主記憶體
移出
移入
0 1 2 3
4 b 5 c 6 e 7
8 9 10 11
12 13 14 15
16 17 f 18 h 19 j
20 21 22 23
備份儲存體

置換下的系統效能

- 雖然分頁置換比置換整個行程更有效，當系統經歷任何形式的置換時，通常活動行程會比可用實體記憶體更多
- 一般要處理這種情況有兩種方法：

  (1) 終止某些行程

  (2) 獲得更多的實體記憶體

9.5.3 行動系統的置換

- Apple 的 iOS 在可用記憶體降低到一定門檻時，要求應用程式自願放棄所配置的記憶體來取代置換
- 唯讀資料 (例如程式碼) 從系統被移除，稍後需要時再重新從快取記憶體載入
- 已經被修改的資料 (例如堆疊) 絕不會被移除。然而，任何不能釋放足夠記憶體的應用程式可能會被作業系統終止

9.5.3 行動系統的置換

- Android 不支援置換，並採取類似 iOS 使用的策略
- 如果沒有足夠的可用記憶體時，它可能終止一個行程
- 在終止一個行程前，Android 會寫入它的**應用程式狀態** (application state) 到快取記憶體，因此應用程式可以快速地重新啟動

9.6 範例：Intel 32 和 64 位元架構

- Intel 晶片的架構已經主宰個人電腦領域好幾年了
  - Intel 稍後生產一系列 32 位元的晶片
    - IA-32
      - 其中包括 32 位元 Pentium 處理器
  - Intel 已經生產一系列基於 x86-64 架構的 64 位元晶片

9.6 範例：Intel 32 和 64 位元架構

- 須注意因為 Intel 在這些年已發佈這些架構的許多版本和變形
  - 我們不能提供所有晶片的記憶體管理架構之詳細描述
  - 也沒辦法提供所有 CPU 的細節，因為這種資訊最好留待計算機架構的書籍
- 反之，我們提供這些 Intel CPU 的主要記憶體管理觀念

9.6.1 IA-32 架構

- IA-32 系統的記憶體管理被分成兩個元件
  - 分段
  - 分頁

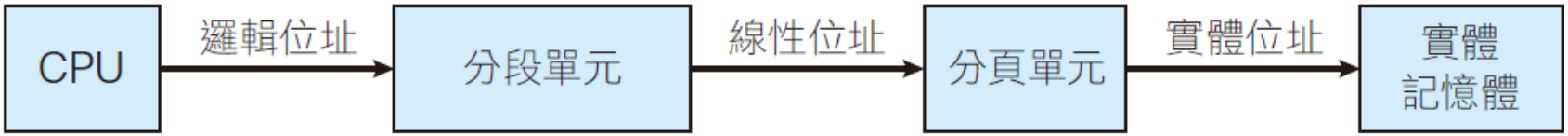


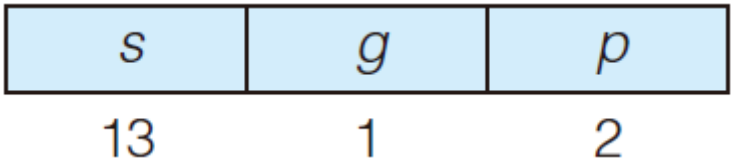
圖 9.21 在 IA-32 中邏輯對實體位址轉譯

9.6.1.1 IA-32 分段法

- IA-32 架構允許區段可以大到 4 GB 位元組，每個行程的分段數目最多是 16 K
  - 一個行程的邏輯位址被分成兩部份：
    - 第一部份由至多 8 K 所組成，是該行程的私有分段
      - 第一部份的資訊存放在區域描述表
    - 第二部份由最多 8 K 所組成，是所有行程共同使用的區段
      - 第二部份的資訊存放在全域描述表
  - LDT 和 GDT 中的每一項皆占用 8 個位元組，其中包含關於某一特殊區段的詳細資訊，包括
    - 基底位址
    - 該區段的限制

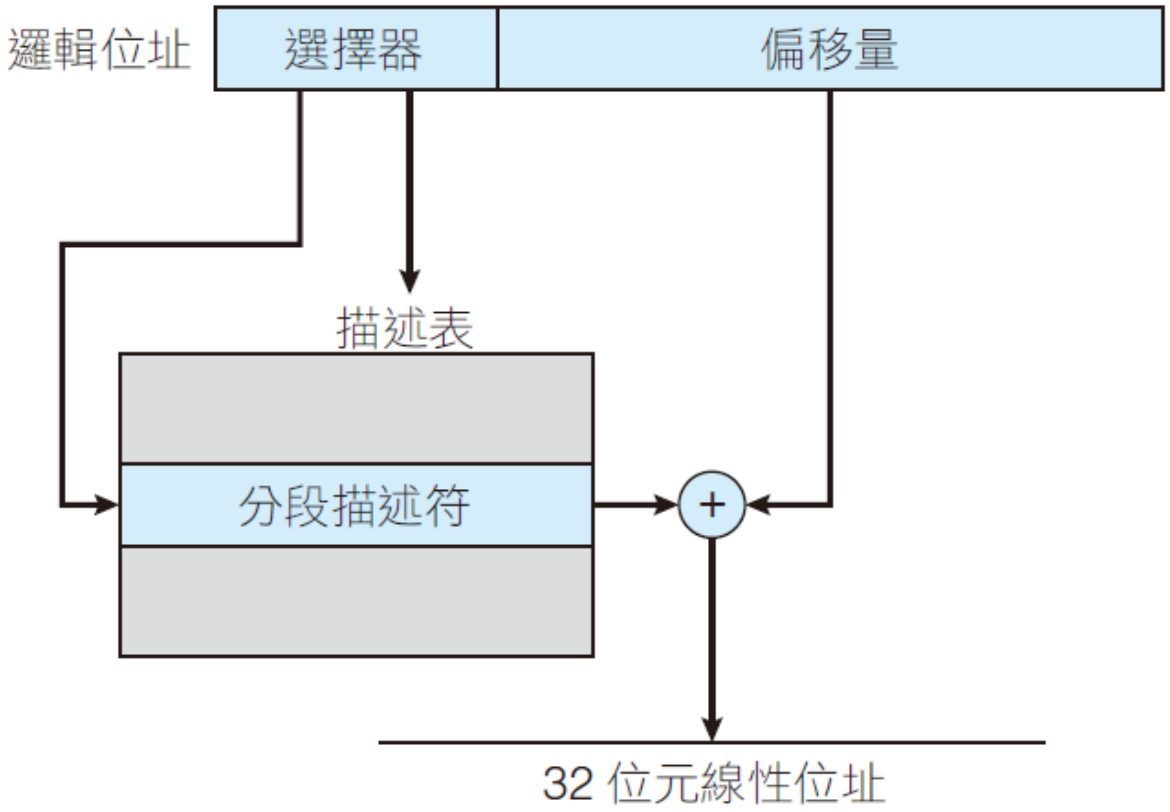
9.6.1.1 IA-32 分段法

- 邏輯位址是一組 (選擇器、偏移量)，其中選擇器 (selector) 是一個 16 位元數字：



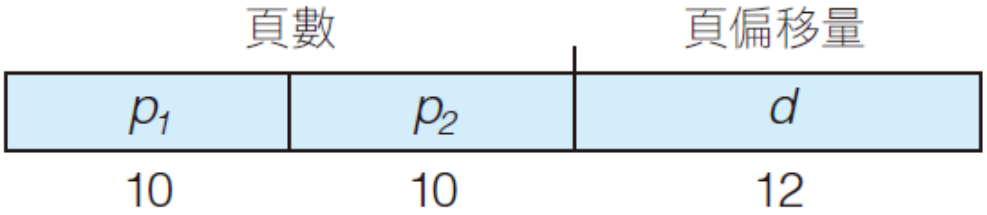
  - 此處 s 表示區段號碼，g 用來指示此區段是在 GDT 或 LDT 中，而 p 是處理保護
  - 偏移量為 32 位元，是用來設定所要的位元組在區段中的哪一個位址
- 該區段的基底和限制資訊被用來產生**線性位址** (linear address)。首先，限制資訊被用來檢查位址是否有效

圖 9.22 IA-32 分段



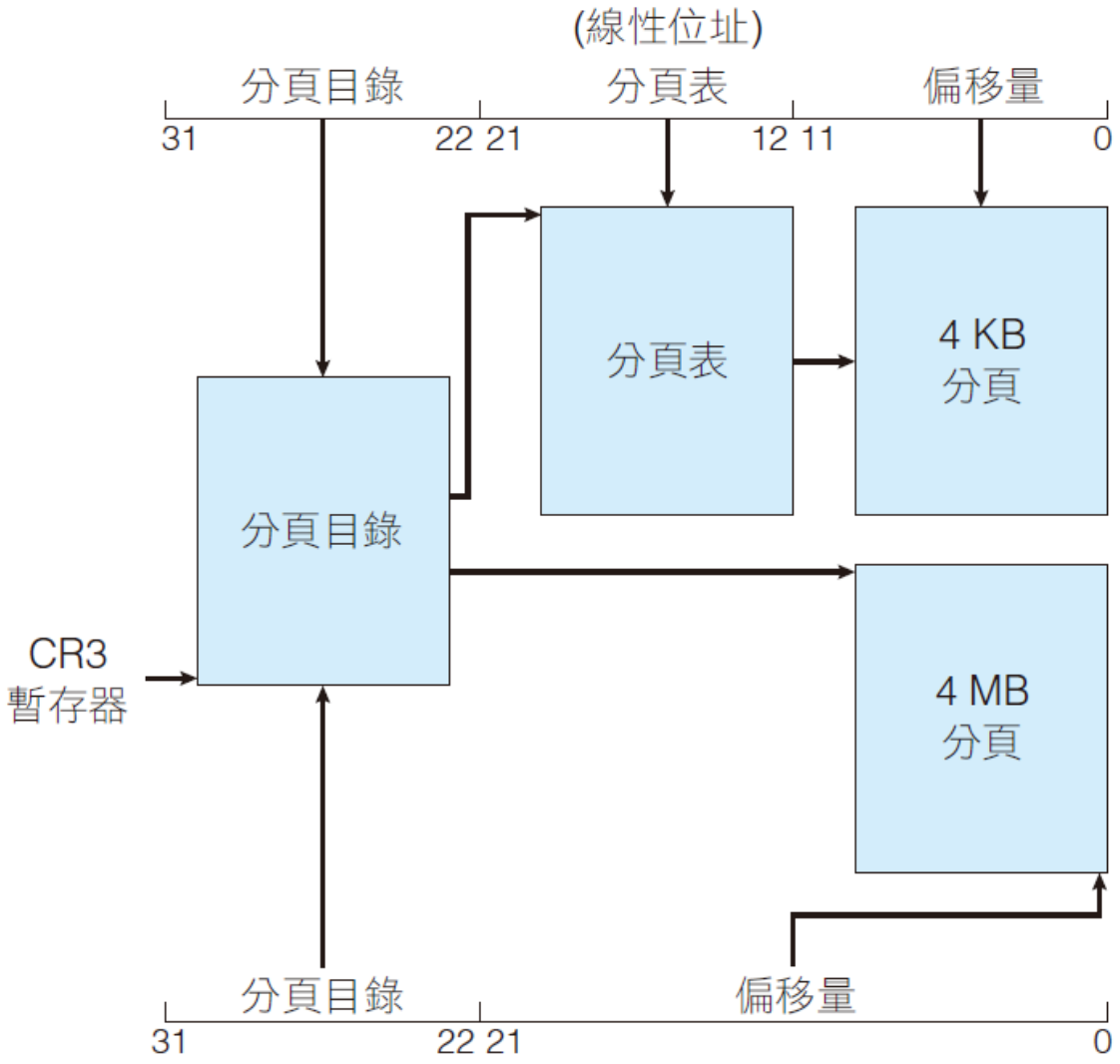
9.6.1.2 IA-32 的分頁

- IA-32 使用兩層分頁法，32 位元的線性位址分割如下列：



  - 10 個高階位元參考最外層分頁表的進入點，IA-32 稱為**分頁目錄** (page directory)
- 標準的 4 KB。如果設定旗標，分頁目錄直接指向 4 MB 分頁欄

圖 9.23　A-32 架構中的分頁



Intel 採取分頁位址擴展

- 讓 32 位元處理器可以存取大於 4 GB 的實體位址空間
- 引入 PAE 支援的基本差別是分頁由兩層技術 (如圖 9.23 所示) 變成三層技術
  - 其中最高的兩位元參考到**分頁目錄指標表格** (page directory pointer table)
  - 圖 9.24 描述有 4 KB 的 PAE 系統 (PAE 也支援 2 MB 分頁)
  - 結合 12 位元的偏移量，加上 PAE 支援 IA-32 增加位址空間到 36 位元，這可以支援到 64 GB 的實體記憶體

圖 9.24　分頁位址擴展

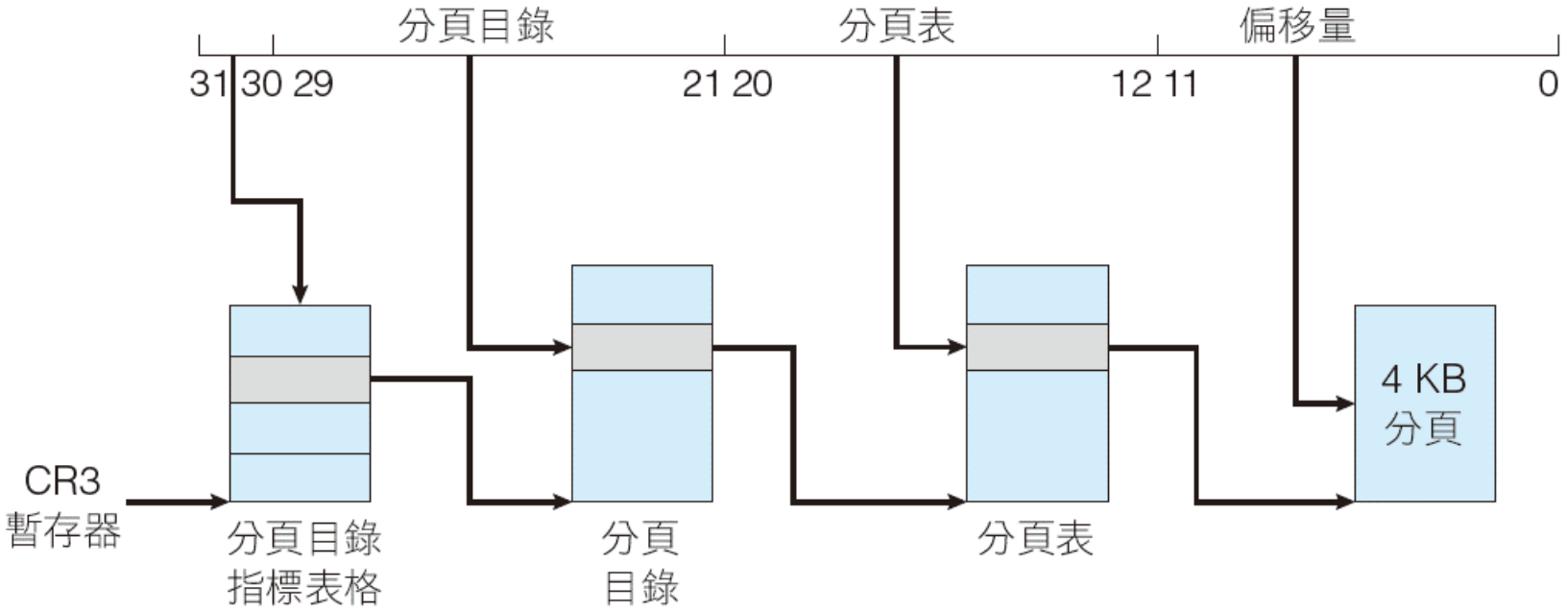
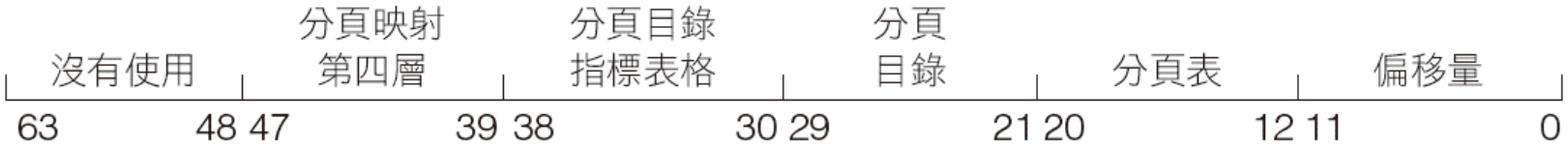

分頁目錄
分頁表
偏移量
31
30 29
21 20
12 11
0
CR3
暫存器
分頁目錄
指標表格
分頁
目錄
分頁表
4 KB
分頁

圖 9.25　x86-64 線性位址



9.6.2 x86-64

- 開始發展 64 位元架構，稱為 x86-64，它是以擴展現有 IA-32 指令集為基礎
- x86-64 架構目前提供 48 位元的虛擬位址，使用四層分頁結構支緩 4 KB、2 MB 或 1 GB 的分頁
- 線性位址的表示顯示在圖 9.25
- 因為這個定址技術可以使用 PAE，虛擬位址的大小是 48 位元，但支援 52 位元的實體位址 (4096 兆位元組)

9.7 範例：ARM 架構

- 通常是 32 位元的 ARM 處理器
- ARMv8 具有三種不同的**轉換顆粒** (translation granules)：
  - 4 KB
  - 16 KB
  - 64 KB
- 每個轉換顆粒提供不同的分頁大小，以及較大的連續記憶體部分
  - 稱為**區域** (regions)
- 但請留意層級 1 和層級 2 的表可能使用另一個表或 1 GB 區域 (層級 1) 或 2 MB 區域 (層級 2)

9.7 範例：ARM 架構

- 位址轉換在微 TLB 開始
  - 在失誤發生時，主 TLB 就被檢查。
  - 果兩個 TLB 都產生失誤，分頁表走訪必須由硬體執行

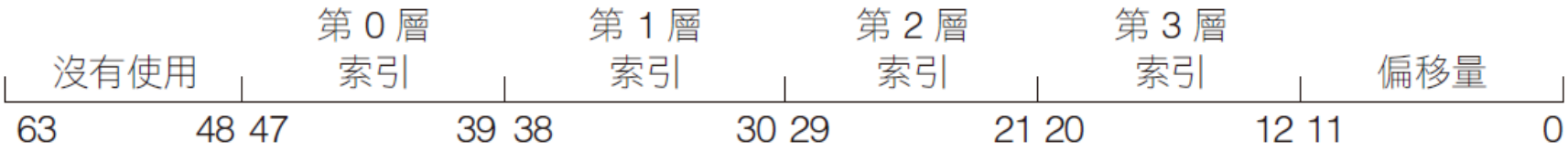


圖 9.26 ARM 4 KB 轉換顆粒

圖 9.27 ARM 四階分層分頁