記憶體管理

大綱

- 記憶體基本概述
- 記憶體的運作模式
- 重疊與置換
- 斷裂的問題
- 記憶體分頁
- 記憶體的分段
- 分頁與分段的整合使用
- 記憶體管理單元範例
- 本章重點回顧

前言-為何需要記憶體管理

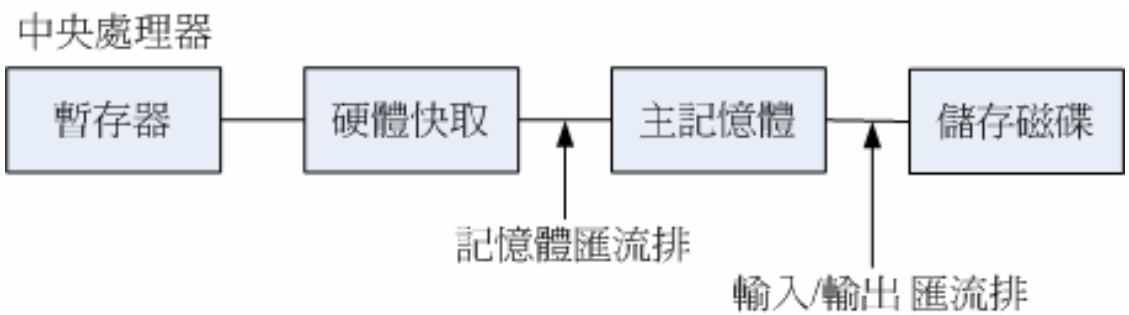
- 作業系統的功能之一如下：
  - 提供程序在執行時所需要的資源分配
  - 第三章在講排程與程序在取得CPU使用權的關係
- 因為CPU讀取的程序在記憶體中，所以：
  - 眾多的程序在記憶體中如何建立、如何退出
  - 需一個管理的機制來掌握實體記憶體的運用資訊

記憶體基本概述

- 電腦系統中記憶體的基本形式：
  - 主記憶體 (Main Memory)：
    - 例如動態隨機處理記憶體(Dynamic Random Access Memory, DRAM)
    - 容量比較大，速度比較慢，現在電腦都有1GB以上的容量
    - 用來儲存由磁碟系統所讀取的資料或CPU等待處理以及處理完畢的資料
  - 快取記憶體 (Cache Memory)
    - 例如靜態隨機處理記憶體(Static Random Access Memory, SRAM)
    - 容量比較小，單價比較高，可能整合到CPU內部了(第二層快取)
    - 儲存最近使用過的記憶體位置的位址與內容

CPU與記憶體之間的關係

- CPU如何存取記憶體內的資料(程式)
  - CPU內有暫存器，速度最快，能夠記錄程式碼與資料來進行處理。
  - 暫存器的資料由快取當中取得，因為快取常整合到CPU內部，因此暫存器與快取的速度非常快
  - 主記憶體透過匯流排(通常在北橋)將資料傳遞給快取，快取如果效能夠好，就能夠保存常用的資料，以使系統效能提升
  - 主記憶體的資料由硬碟來的(通常由南橋)，速度較慢。

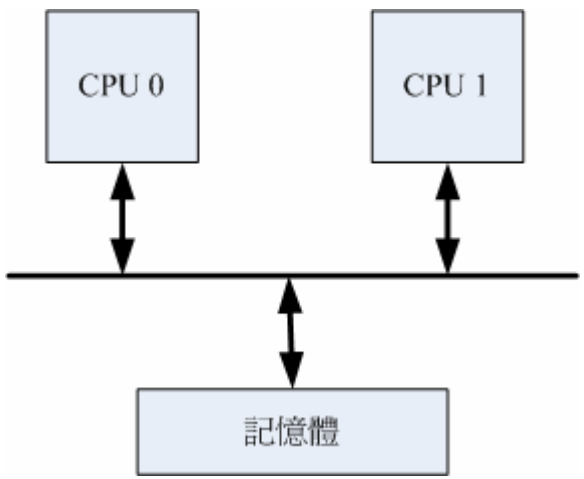


快取記憶體與主記憶體的回寫

- CPU處理完畢的資料如何寫回主記憶體？
  - 寫透式(write-through)
    - 直接將快取記憶體內的運算結果寫回主記憶體中
    - 下次要再使用此資料，則需要到主記憶體中查詢
  - 寫回式(write-back)
    - 將資料暫時保持在快取記憶體中，而不回寫到主記憶體
    - 避免資料不同步的機制：
      - clean data：快取/主記憶體中的資料一致
      - dirty data：快取/主記憶體中的資料不一致
      - 系統會定期將dirty data寫回到主記憶體中，讓資料一致！

多處理器分享記憶體架構

- 非統一記憶體處理技術(Non-Uniform Memory Access, NUMA)
  - 多工處理的計算機架構
  - CPU可快速的存取同一個區域記憶體內的資料
  - 有優良的擴充性
  - 可支援到256顆處理器



記憶體運作模式--記憶體的定址範圍

- 一部主機支援多大的記憶體？
  - 與CPU的定址匯流排寬度有關(？位元電腦)
    - 80286：CPU為24位元，定址空間2^{24}=16MB
    - 80386：CPU為32位元，定址空間2^{32}=4GB
    - 定址空間：
      - 記憶體就像蜂窩一樣是很多資料格點所組成的，每個資料格點可以放置程式碼與資料，每個資料格點也有位址，作業系統能夠記錄的最大記憶體位址，就是定址空間了。
    - 問：為何32位元的作業系統無法支援4GB以上的記憶體空間？

程序-記憶體-CPU的關係

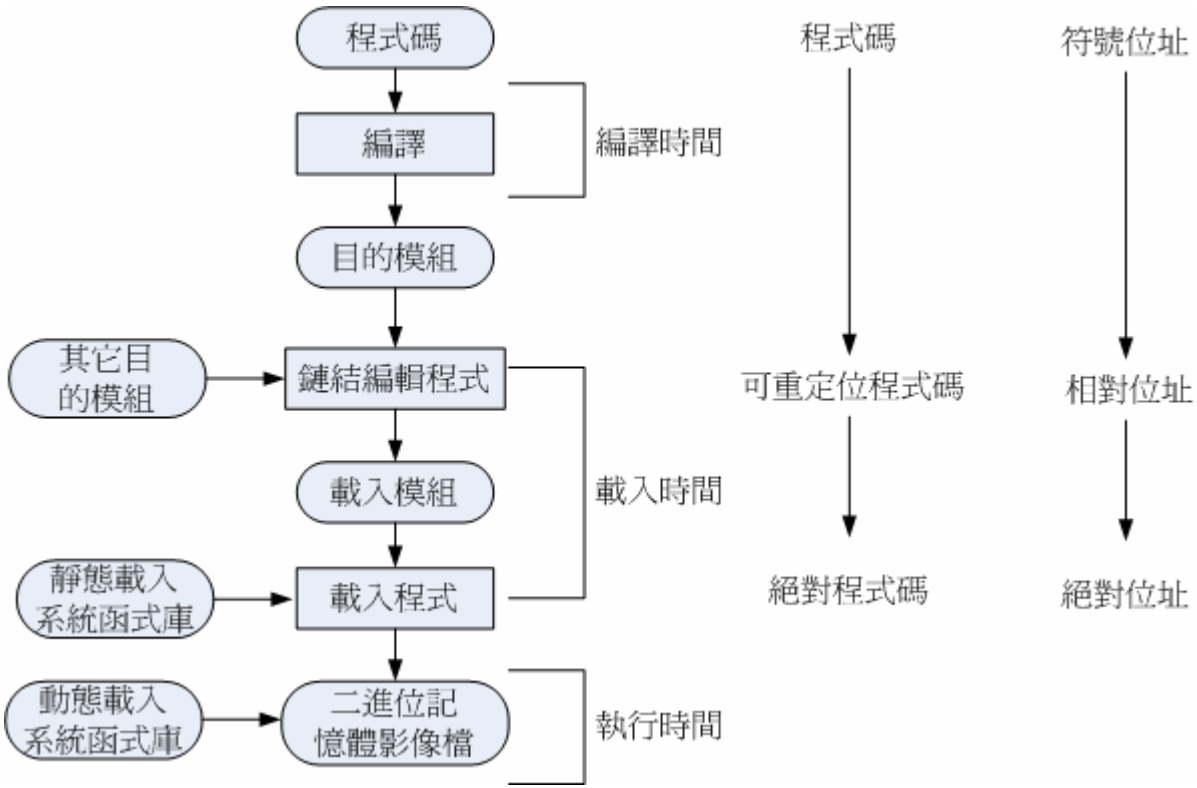
- 程序被觸發執行後，載入到記憶體中
  - CPU要運作這個程序：
    - 必須由主記憶體中找到該程序所在的記憶體位置，
    - 然後將裡面的資料讀出
  - CPU運作完畢後，也需要將結果寫回該程序所能使用的某個記憶體位置(位址)
  - 可能還需要將記憶體內的資料寫回某些儲存器，如硬碟、軟碟等。

程式如何載入記憶體成為程序-1

- 程式由編譯到執行的三階段：
  - 編譯時間(compile)：
    - 透過編譯器將程式碼編譯成為目的模組(Object module)
  - 載入時間(load)：
    - 與其他模組鏈結成為可執行的程式，並載入其他**靜態**的系統函式庫
  - 執行時間(execute)：
    - 整個資料已經被載入到記憶體中，同時載入其他的**動態**函式庫
- 重點是：**如何將程式的資料載入到記憶體中？**

程式如何載入記憶體成為程序-2

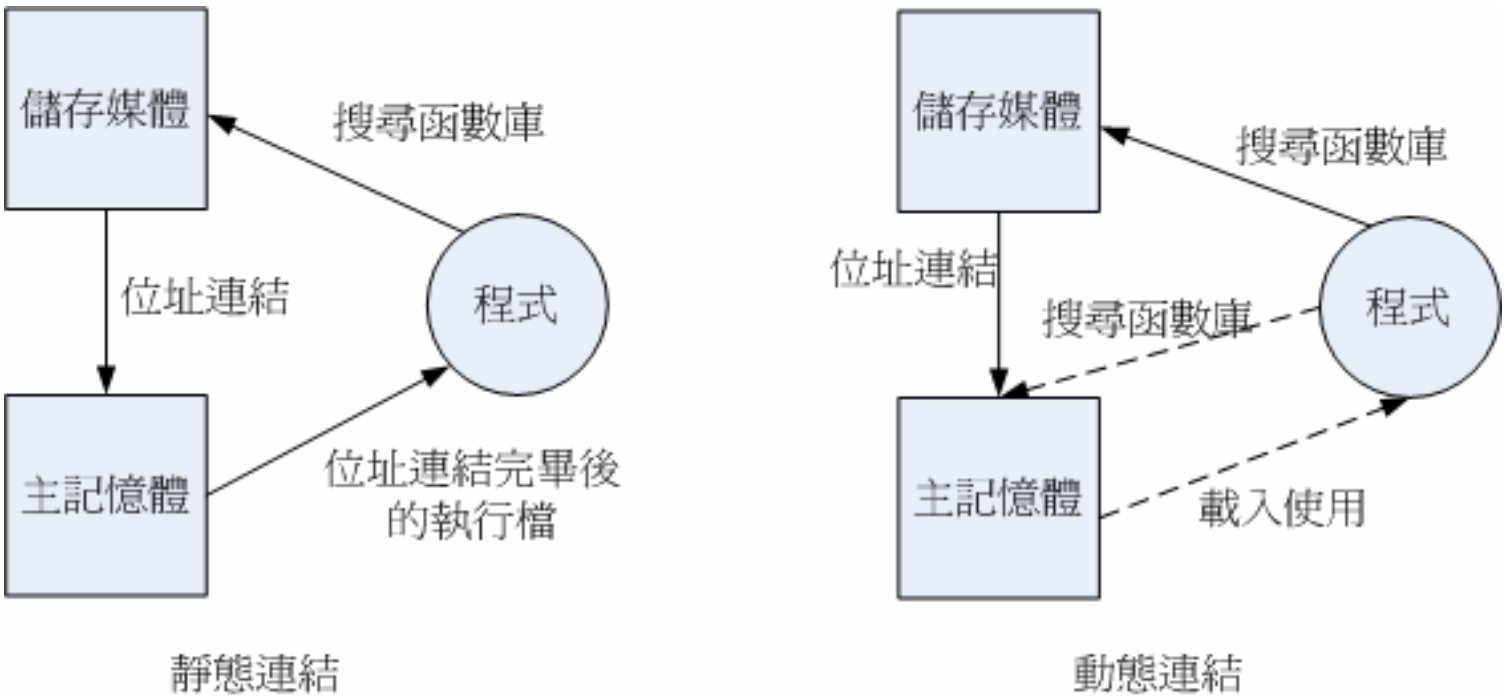
- 位址連結
  - 將程式碼編譯成為可以被載入到主記憶體當中的『可重定位程式碼』
  - 將該程式碼實際載入主記憶體當中的『絕對位址』後，就能運作了。



函式庫的分別

- 函式庫：→由其他軟體提供的程式段功能
  - 靜態函式庫：
    - 使用者程式在編譯時直接將此函式庫整合到程式中
    - 如果函式庫升級，需要重新編譯使用者程式
  - 動態函式庫：
    - 使用者程式在編譯時，只是將函式庫所在的位置編譯到程式碼中
    - 當程式被執行時，才去搜尋這個動態函式庫檔案，並且載入到主記憶體中
    - 動態函式庫被載入主記憶體後，可以被重複利用而不需重新載入
    - 函式庫升級時不需重新編譯

函式庫的運作模式圖示


儲存媒體
搜尋函數庫
位址連結
程式
主記憶體
位址連結完畢後
的執行檔
靜態連結
儲存媒體
搜尋函數庫
位址連結
搜尋函數庫
程式
主記憶體
載入使用
動態連結

預防記憶體不足的處理模式

- 情境：
  - 如果一個程式所需要的記憶體資源大於系統剩餘的記憶體容量，則此程式所觸發成為程序可能就會被放入等待佇列中
- 預防的方法：
  - 重疊：
    - 分次載入這個程式，而不是一次載入全部的程式碼與資料
  - 置換：
    - 一次載入程式全部的資料，但是比較不重要的資料先被移出主記憶體，以空出更多可用記憶體給系統使用。

重疊—舊的方法

- 透過兩次編譯的機制，將程式碼分成：
  - 第一次程式碼：第一階段會用到的
  - 第二次程式碼：第二階段會用到的
  - 符號表：各次程式碼之間的參考資訊
  - 其他：可能是共用資料等

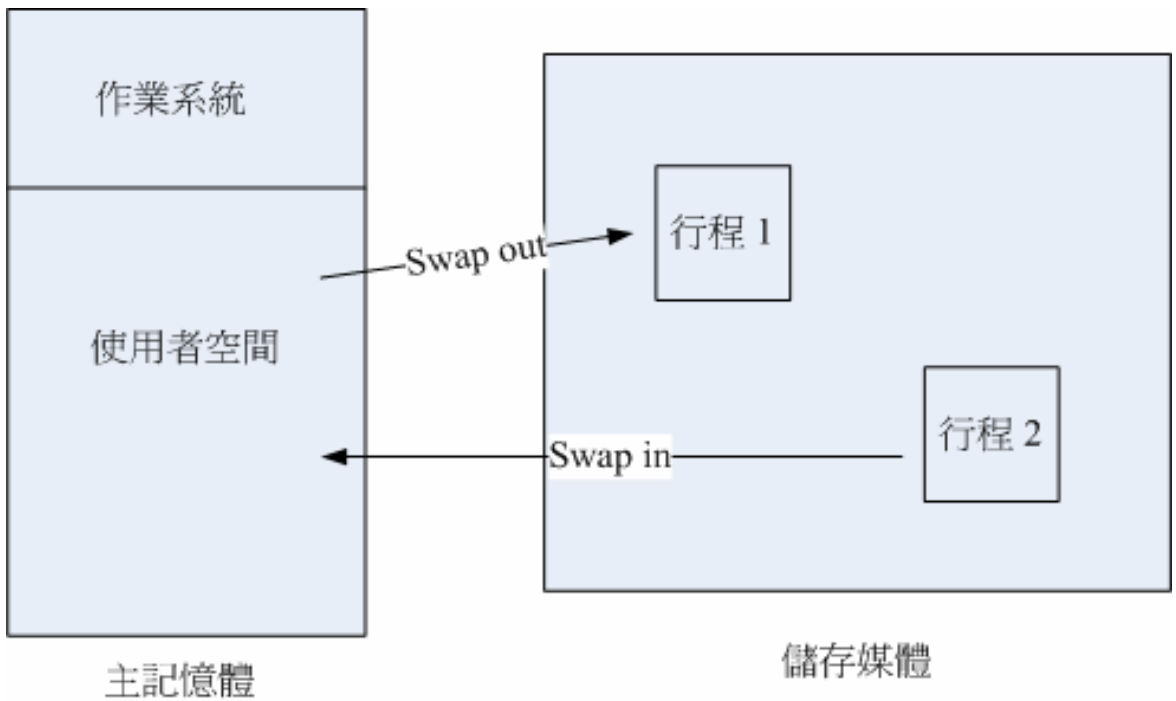
重疊—產生的記憶體處理情況

- 案例探討
  - 全部載入需花費：
    - 170Kbyes記體體
  - 兩次載入，每次的記憶體
    - 第一：127Kbyes
    - 第二：108Kbyes
  - 總結：
    - 記憶體使用量降低
    - 但由於多了很多參考鏈結、符號處理等，所以效能可能較差
    - 新的作業系統已經很少使用這個方法了。

項目	記憶體大小 (Byte)
第一次處理的程式碼	62K
第二次處理的程式碼	43K
符號表	10K
程式本身和其他	55K

置換(swap)—目前常用的方法

- 將較低優先權的程序移到外部儲存媒體，讓主記憶體給高優先權的程序使用；等高優先權程序運行完畢，再將低優先權的程序移回主記憶體中運作。



swap的效能-時間成本的考量

- 置出(swap out)
  - **RAM→儲存媒體**
  - 儲存媒體的資料搜尋
  - 平均潛伏時間
  - 寫入儲存媒體的時間
  - 釋放記憶體的時間
- 置入(swap in)
  - 儲存媒體尋找資料time
  - 平均潛伏時間
  - **儲存媒體→RAM**
  - 寫入記憶體的時間
  - 消除儲存媒體所佔空間的時間

問：為何記憶體不足時，或者同時開啟很多很大的程式時，你的系統中，硬碟為何跑不停？速度感覺很慢的原因？

什麼程序可以被swap？

- 不可被swap的程序
  - 當此程序正在進行I/O時：
    - 如果程序正在進行I/O，再將這個程序寫入swap，可能會導致原來的I/O產生問題，導致程序執行上的錯誤。
- 作業系統的處理方法：
  - 不允許置換正在I/O中的程序
  - 只有進入緩衝區的程序才能被置換

Linux的swap

- Linux的swap觀察
  - free
  - top
- swap所使用的裝置/檔案
  - swapon –s
  - cat /proc/swaps
- swap啟動的時機：
  - 當系統的記憶體不足，會將不活動的程序移動到swap

因為swap的速度比記憶體實在慢太多了，因此一般時候系統並不會用到swap。但如果你的swap被用到10%以上時，表示你的實體記憶體太小了！需要加裝更多的RAM了！

程序在記憶體中的存在方式

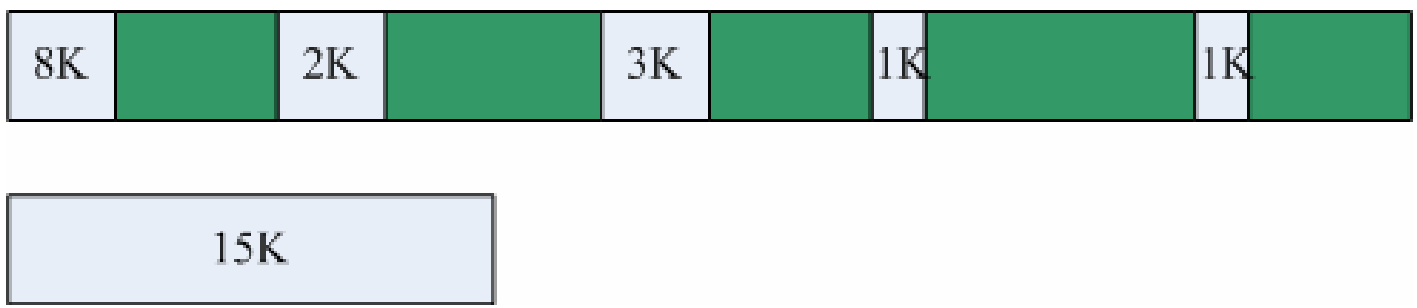
- 程式可以被觸發成為程序被執行的條件：
  - 記憶體資源必須足夠：
    - 程序載入時，作業系統需先配置一個可用的記憶體空間給該程序使用
    - 若記憶體不足，該程序會被丟入佇列中等待
  - 該記憶區段必須為連續：
    - 該程序必須連續的存在於記憶體區塊中。

斷裂—記憶體不連續的困境

- 情境：
  - 程序所要求的記憶體配置是系統還能夠支援的記憶體空間，但是這些空間是分散的
  - 這將導致程序無法被載入而被置入佇列中。
  - 這個記憶體空間分散的現象稱為：『斷裂』
  - 當發生斷裂現象時，會造成系統資源的浪費。

斷裂—記憶體不連續的困境

- 斷裂的案例：
  - 新的程序要求配置 15K 的記憶體資源
  - 系統目前有的資源『8+2+3+1+1=15K』
  - 因為不連續，所以該程序被丟入佇列中等待了



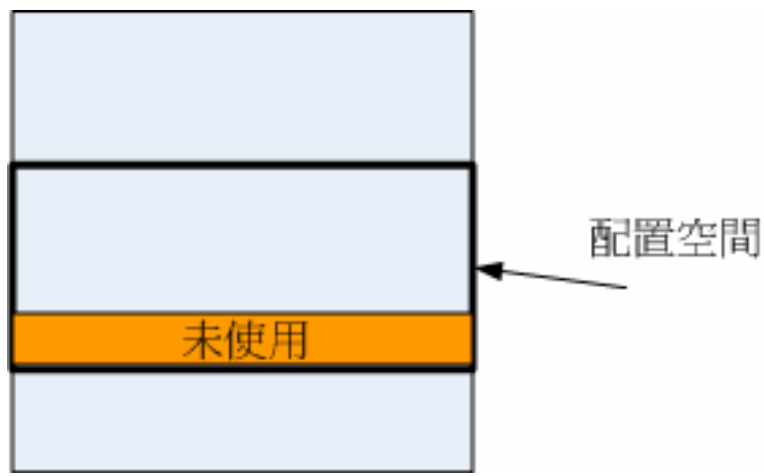
斷裂—50%法則

- 50%法則 (50-Percent Rule)：
  - 記憶體配置的演算法與配置的載入方式都會影響到斷裂空間的不同
  - 在最佳的配置狀態下，配置 n 單位的記憶體空間，會因為外部斷裂導致 n/2 單位空間的浪費
  - 浪費的空間約為1/3 (0.5n浪費/1.5n全部=1/3)
  - 此一現象稱為『50%法則』
  - 解決方法：『內部斷裂』或『可重定位分割法』

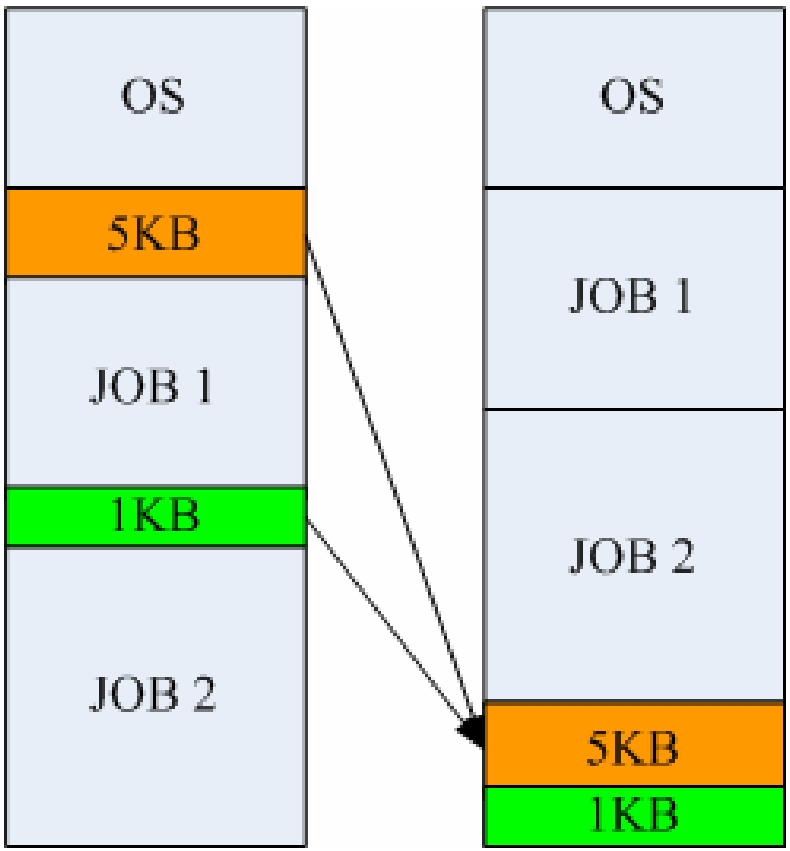
內部斷裂(Internal Fragmentation)

- 記憶體處理方式：
  - 使用一個較大的空間配置給單一程序來使用
  - 目的：
    - 將記憶體空間連續在一起
    - 程序釋放後的整塊空間較大

因有沒用到的空間，故稱為內部斷裂



可重定位分割法 (Relocatable Partition)



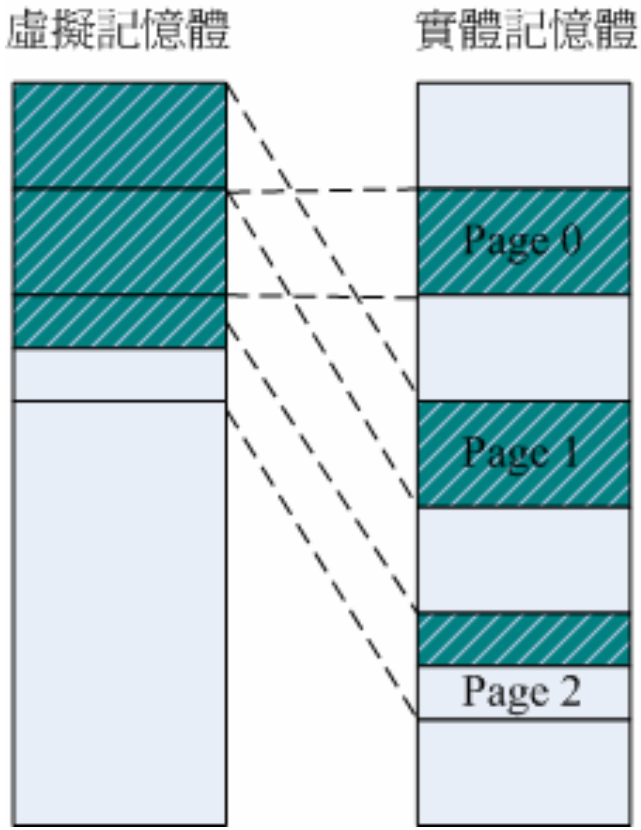
- 記憶體處理方式：
  - 隨時分析系統，將執行中的程序重新定位，以排列在一起
  - 此時原本斷裂的空間就會集中在一起了。
  - 缺點是：系統常要在記憶體中挪動程序，可能會影響到效能

斷裂的困境與解決之道

- 困境：
  - 外部斷裂的問題雖可使用內部斷裂與重新定位分割法來處理，但是總是會**影響到效能**。
- 解決方案：
  - 透過分頁技術來處理！

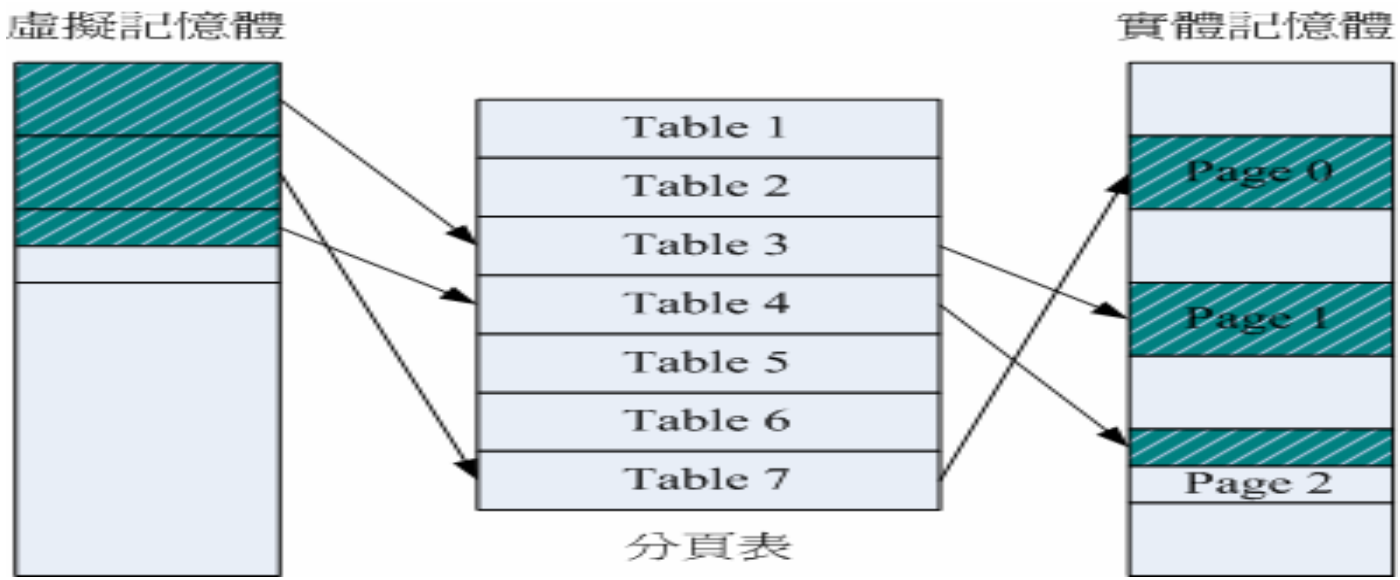
記憶體的分頁技術

- 分頁的作法：
  - 製作出虛擬記憶體，且虛擬記憶體與實體記憶體區塊大小相同
  - 實體記憶體區塊：欄位(Frame)
  - 虛擬記憶體區塊：分頁(Page)
  - Linux的分頁區塊大小通常為
    - 4KBytes
- 因為虛擬記憶體的配置是連續的，只是指向到不同的實體記憶體。因此可以使用不連續的實體記憶體區塊！



記憶體的分頁技術—分頁表

- 直接分頁的問題：
  - 因為記憶體載入為動態配置，透過分頁技巧直接指定固定(4K)記憶體位址可能會出錯；
  - 解決方案：透過中間的一個對照表來管理。



CPU對程序的定址

- 問題：
  - CPU怎麼知道程序所在的記憶體空間？
  - 前提：程序所在的虛擬記憶體空間是連續的
    - 只要知道『{開始的位址}+{連續空間的大小}』即可
- CPU處理方式：
  - 分頁頁數(Index)：相當於程序開始的位址
  - 分頁偏移量(Offset)：相當於連續空間的大小
  - 一支程序的讀取只要知道上述兩個值即可讀取
  - 透過『基底暫存器(Base Register)』記錄此兩數值

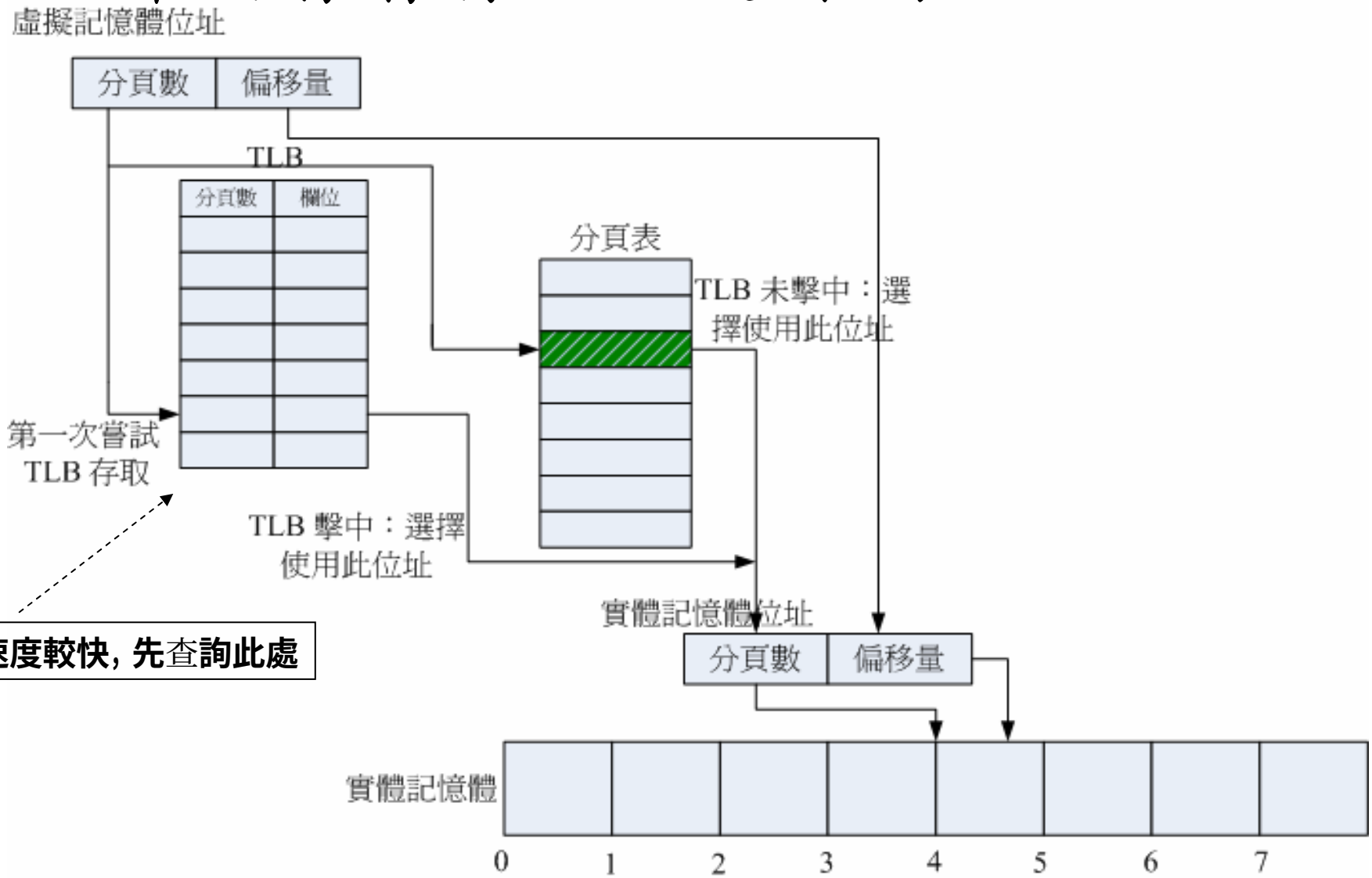
暫存器的處理方式

- 問題：
  - 暫存器的數目是固定的，所以無法容納系統這麼多的程序位址定位。
  - 解決方案：
    - 將分頁表放置到主記憶體中，讓系統管理
    - 只分配一個暫存器來管理上述的分頁表。
    - 此暫存器稱為：『分頁表基底暫存器(Page Table Base Register, PTBR)』

翻譯側看暫存區 (Translation Look-aside Buffer, TLB)

- 功能：
  - 因為分頁表基底暫存器處理單一用戶的程序效率不佳
  - 解決方案：
    - 透過硬體記憶體(TLB)來記錄單一用戶的程序定址
    - TLB的速度比主記憶體要快的多
- TLB的運作過程
  - CPU依舊取得分頁數與偏移量
    - 先到TLB查閱分頁數與欄位，若查到(擊中)及直接存取實體RAM
    - 若TLB沒有查到(未擊中)則直接到分頁表中查詢。

翻譯側看暫存區—運作圖示



速度較快，先查詢此處

暫存器效能優劣的評估

- TLB擊中率高，代表效能好
  - 常用的分頁資訊會被載入到TLB中
  - 擊中率越高，花費的時間越少，效能越佳！
  - 評估方式：
    - 有效記憶體存取時間（Effective Memory Access Time）

[有效記憶體存取時間]=[擊中率]×[於TLB中找到分頁的記憶體存取總時間]+(1−[擊中率])×[在TLB中找不到分頁的記憶體存取總時間]

Linux的記憶體管理單元

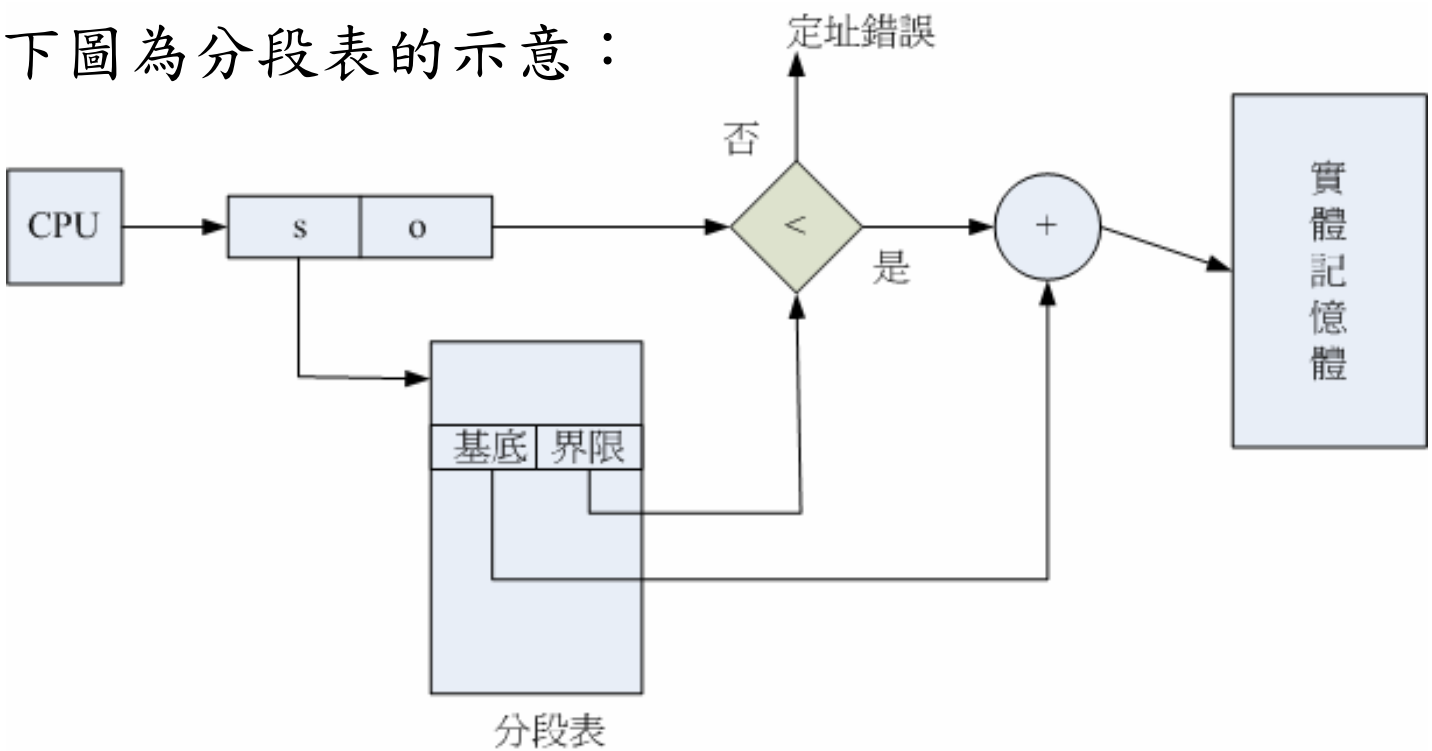
- Linux Memory Management Unit, MMU
  - 以分頁作為管理的單位
  - 32位元系統分頁區大小： 4KBytes
  - 64位元系統分頁區大小： 8KBytes
  - 範例：32位元系統的 1GB 實體記憶體
    - 共有 1x1024x1024(K)/4(K)=206144個分頁區塊

使用者程式的角度看記憶體

- 分頁技術：
  - 由『程序』連續的角度來看記憶體
- 分段技術 (Segmentation)：
  - 由『使用者程式』的角度來看記憶體
    - 一個軟體是由多個程式所構成的
    - 這些程式原先都放在儲存媒體當中
    - 當需要執行某程式時，該程式才會被載入到RAM中
    - 所以使用者看起來，RAM就是一段一段的。

記憶體的分段

- 分段法 (Segmentation)
  - 與分頁類似，透過分段表來記錄每一段程式所在的虛擬位址放置的實體位址對照！
  - 下圖為分段表的示意：



分段法的優點

- 可以對資料/程式碼進行保護
  - 資料的查詢透過分段表
  - 分段表中有個保護位元，依據保護位元的值，可覺得該資料/程式碼的保護狀態！ex>唯讀
- 可以對資料/程式碼進行共用
  - 因為是以程式段落來進行記憶，所以程式區段是可以共用的！
  - 分頁法無法達到共用與保護！

分頁/分段的共用原因

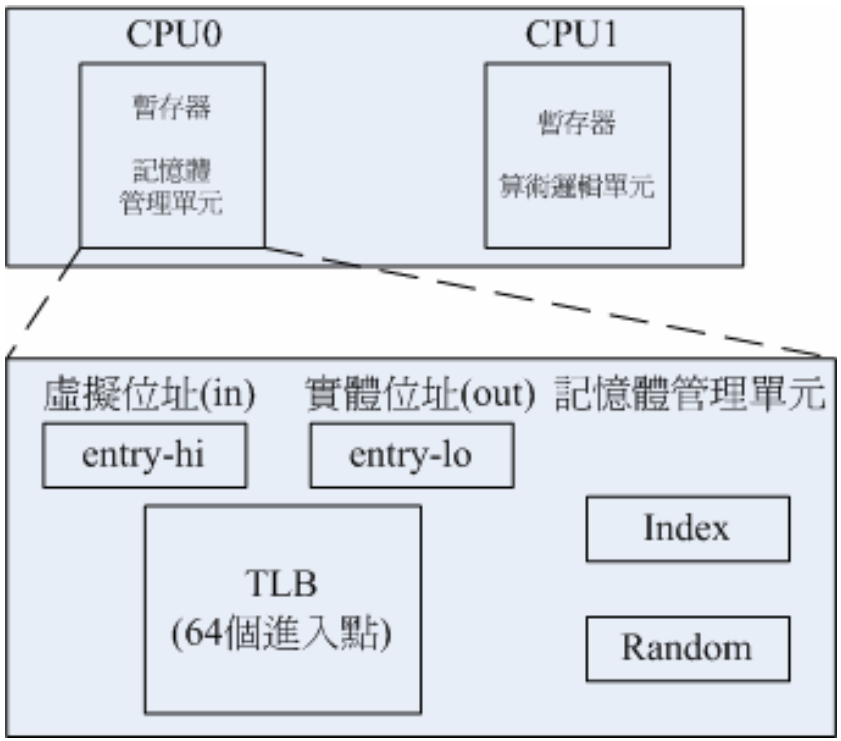
- 分頁法：
  - 可以避免記憶體斷裂的問題
  - 但是程式/資料無法保護與共用
- 分段法：
  - 程式/資料可以保護與共用
  - 記憶體斷裂的情況會非常嚴重。

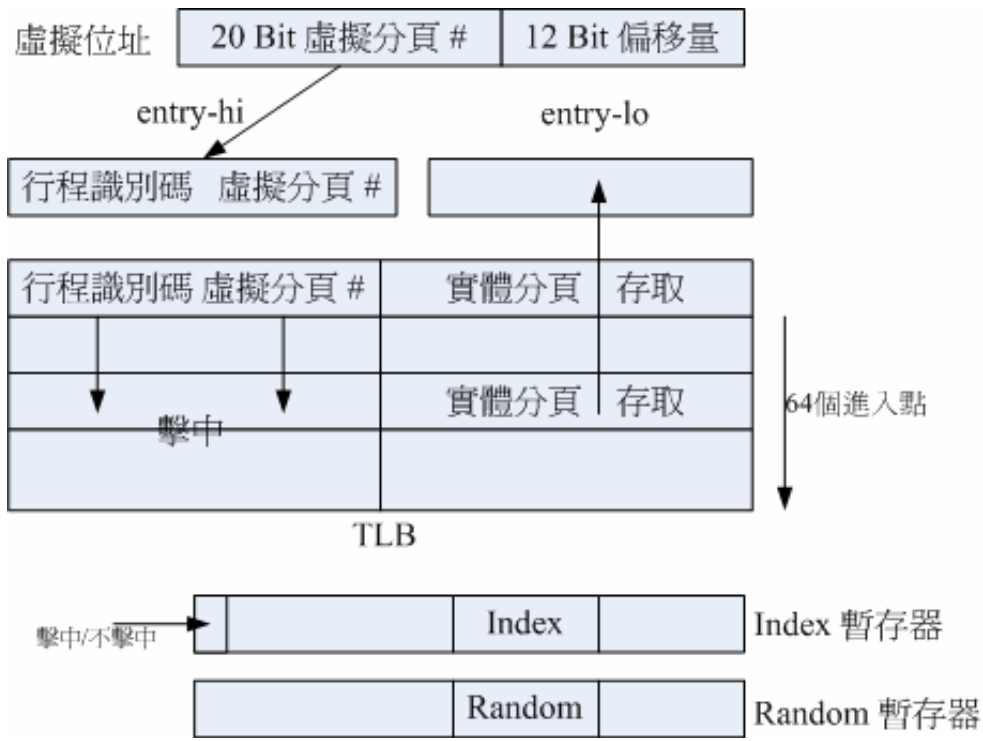
分頁與分段的整合使用

- 分頁／分段法的使用形態：
  - 硬體支援分段法／硬體支援分頁法
  - 軟體支援分段法／硬體支援分頁法

記憶體管理單元範例

- MIPS R2000/R3000 MMU





本章重點回顧

- 了解記憶體的分類與特性。
- 了解行程運行時於記憶體內的運作方式。
- 了解重疊與置換的技術。
- 了解何謂分頁與分段法，並學習其運作方式。
- 將透過記憶體管理單元的實例來了解記憶體管理的方式。