第5章 虛擬記憶體

5-1 基本概念

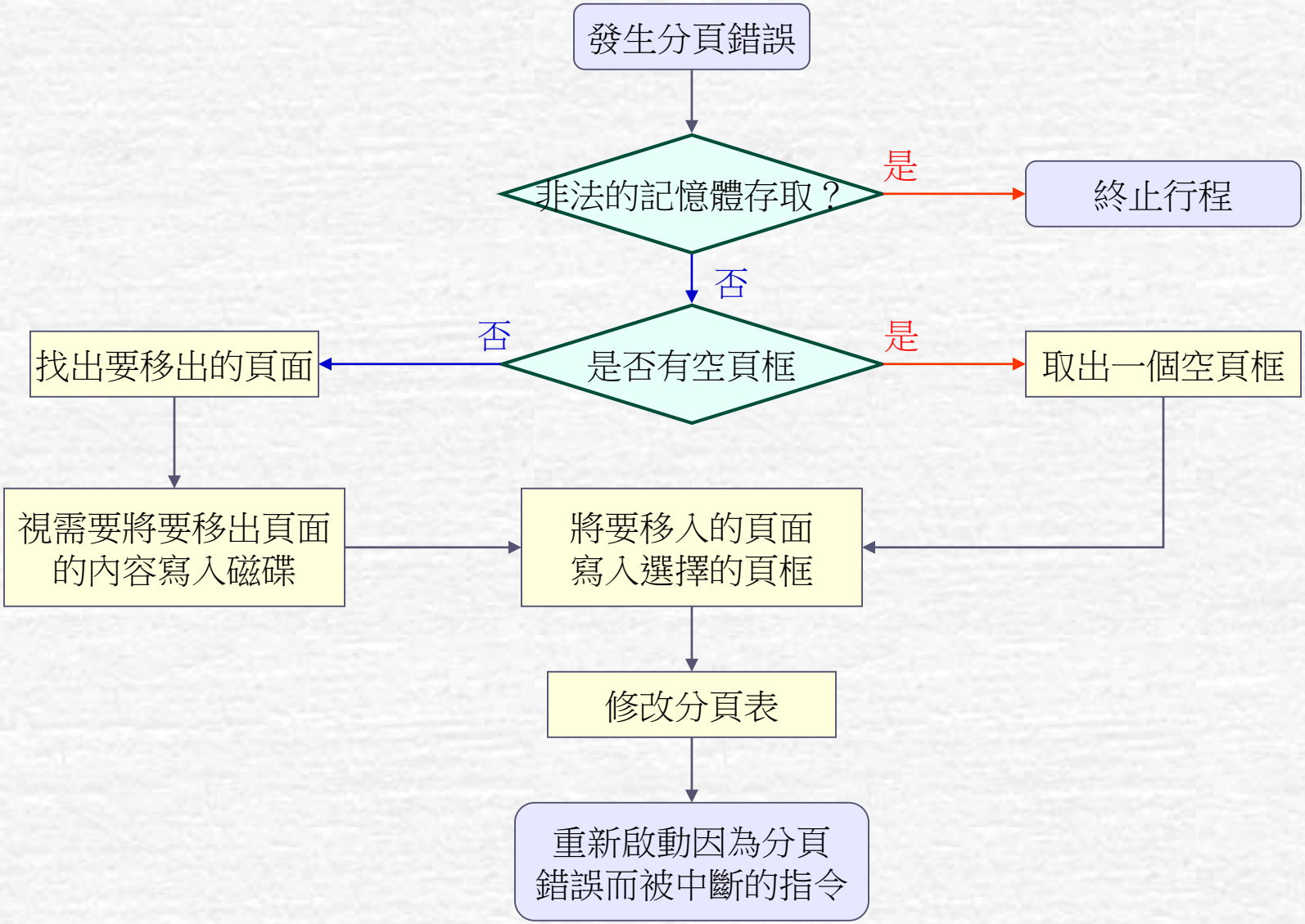
- 把整個程式完整放入實體記憶體的缺點：
  - 在行程的執行過程中，未必會用到程式的所有部份
  - 程式設計師必須小心斟酌程式的長度，以免超過實體記憶體所能容納的範圍

基本概念

需求分頁(Demand Paging)
純粹需求分頁(Pure Demand Paging)

- 虛擬記憶體：一塊連續、而且巨大的邏輯位址空間，程式設計師可以撰寫程式而不必顧慮到程式長度限制等問題
  - 利用需求分頁，將行程暫時沒有用到的頁面「打包」放在輔助記憶體中
- 分頁程式：負責將頁面移入/移出記憶體的程式
  - 在分頁表的每個項目中增加一個有效位元，由記憶體管理硬體負責檢查
- 純粹需求分頁：一開始執行程式的時候，完全不將任何頁面放入記憶體中

圖5-1 分頁錯誤的處理流程



分頁程式與置換程式

- 置換程式：負責整個行程在主記憶體與輔助記憶體間的置換，被換出的行程會進入置換狀態，並且不會被納入短期排程的考量
- 分頁程式：在需求分頁機制下，負責行程頁面在主記憶體與輔助記憶體間的搬移

虛擬記憶體的運作考量

- 希望能夠讓較多行程能夠進入記憶體執行
  - 要配置給每個行程的頁框數量？
  - 要配置固定數目或變動數目的頁框？
- 希望將每個行程常用或最近還會用到的頁面留在主記憶體中，以改進虛擬記憶體的存取效率
  - 頁面替換的運作設計

分頁錯誤的處理時間

- 記憶體存取時間：介於10ns到200ns之間
- 假設記憶體的存取時間為m，當沒有發生分頁錯誤時，從記憶體中讀取所需字組的時間就是m
- 如果發生分頁錯誤時，行程必須先等待作業系統將所需的頁面載入
- 假設處理分頁錯誤的時間為t，則該字組的存取時間就變為t+m
- 影響t的因素：處理因為中斷所造成的控制權移轉，搜尋頁框，進行磁碟讀取，以及重新開始執行等 ，大約需要25ms
- 假設m=25ns，則主記憶體存取與輔助記憶體存取的時間比大約是1:1000000

  如果在1000次記憶體存取中發生了1次分頁錯誤：t+m=25000000+25

  則平均的存取時間就變成大約25微秒：(25000000+25+25*999)/1000

  也就是單純記憶體存取的1000倍

5-2頁面替換

- 局部性原則
- 頁面替換演算法
- 頁面大小的選擇

局部性原則(Locality of Reference)

- 局部性原則：行程在某一時刻所進行的記憶體存取，多半會聚集在幾個主要的位址附近，也稱為參照的局部性
- 形成局部性原則的原因：
  - 程式在執行的時候，除非是遇到分支或函式呼叫，不然都是循序執行。因此，多半是存取連續的指令位址
  - 如果程式執行到迴圈等指令時，會不斷在一小段程式碼間反覆
  - 程式中所使用到的大型資料結構，通常也都是使用連續的記憶體位址

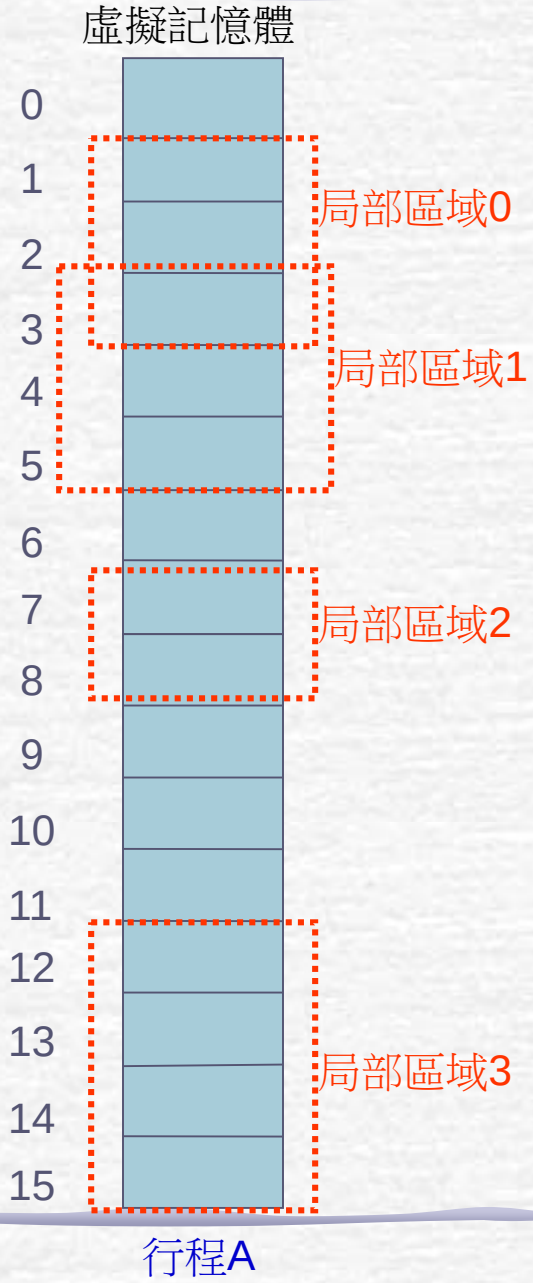
局部性的類型

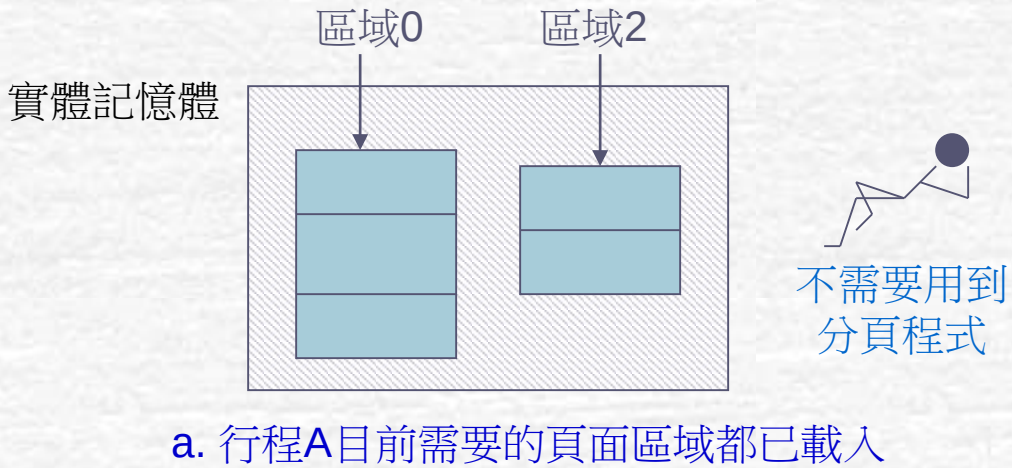
- **時間的局部性**：如果記憶體的某個位置剛被存取過，則不久之後可能還會再被存取，例如存放程式中主要變數的記憶體位置很可能會經常被存取
- **空間的局部性**：當某個記憶體位置被存取過之後，它附近的位置可能很快就會被存取，例如循序執行的指令或是陣列資料

以局部區域的角度看行程的執行

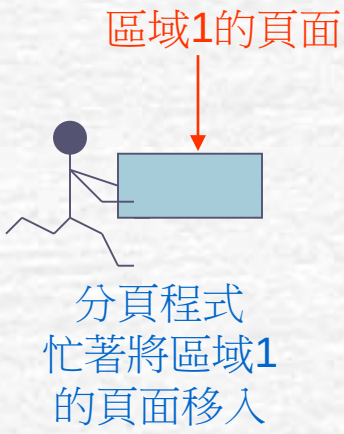
- 局部區域：在某段時間內會同時使用到的一組頁面
- 一個程式可能包含許多不同的局部區域，且這些局部區域彼此之間又可能會相互重疊
- 行程實際上的執行順序，就好像是在不同的局部區域間移動

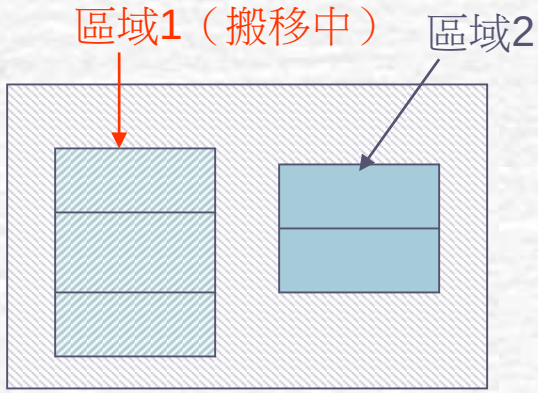
圖5-2 以局部區域的角度看行程的執行





a. 行程A目前需要的頁面區域都已載入





b. 行程A的指令區域由區域0移到區域1

頁面替換演算法

- 最佳替換演算法：理想上最好是能將以後永遠不會用到的頁面移出，退而求其次則是將最久以後才會用到的頁面移出
  - 只能用來作為比較的基準，無法實作
  - 範例：

參考的頁面編號	1	2	1	3	5	1	0	5	2	1	5	1
頁框	1	1	1	1	1	1	0	0	0	1	1	1
		2	2	2	2	2	2	2	2	2	2	2
				3	5	5	5	5	5	5	5	5

一共發生6次的分頁錯誤

FIFO演算法(First–in First-out)

- 最直覺的方式，也就是將待在記憶體中最久的頁面置換出去，以新的頁面內容取代
- 分頁表的項目中必須加入載入時間的欄位
- 並沒有考慮到頁面的使用情況，所以可能造成常用的頁面不斷被移出/移入，使得整體的效能變差

參考的頁面編號	1	2	1	3	5	1	0	5	2	1	5	1
頁框	1	1	1	1	5	5	5	5	2	2	2	2
		2	2	2	2	1	1	1	1	1	5	5
				3	3	3	0	0	0	0	0	1

一共發生9次的分頁錯誤

LRU演算法(Least Recently Used)

- 將最久沒有用到的頁面移出
- 根據局部性原則，如果某個頁面最近剛被使用過，則它可能在不久之後還會被使用
- 是表現相當良好的頁面替換演算法，但是因為要追蹤最後的存取時間，所以在實作上比較複雜
  - 最簡單的做法是在記憶體管理硬體中加入一個計數器，只要有發生記憶體存取，它就會自動加1
  - 在分頁表中的每個項目則加入1個欄位儲存計數器的值
  - 當存取某個頁面的時候，就將當時的計數器值存入該頁面對應的分頁表項目中
  - 需要替換頁面時，LRU演算法會找出欄位值最低的頁面進行取代

圖5-5 LRU演算法範例

一共發生7次的分頁錯誤

LRU近似法 - 老化演算法

- 老化演算法是為記憶體中的每個頁框加入一個參考位元組
- 當頁面被存取時，硬體會將位元組中的最重要位元設為1，並且利用計時器中斷，定期將整個參考位元組向右平移1個位元
- 當需要進行頁面替換時，參考位元組中2進位值最小的頁面就會被取代

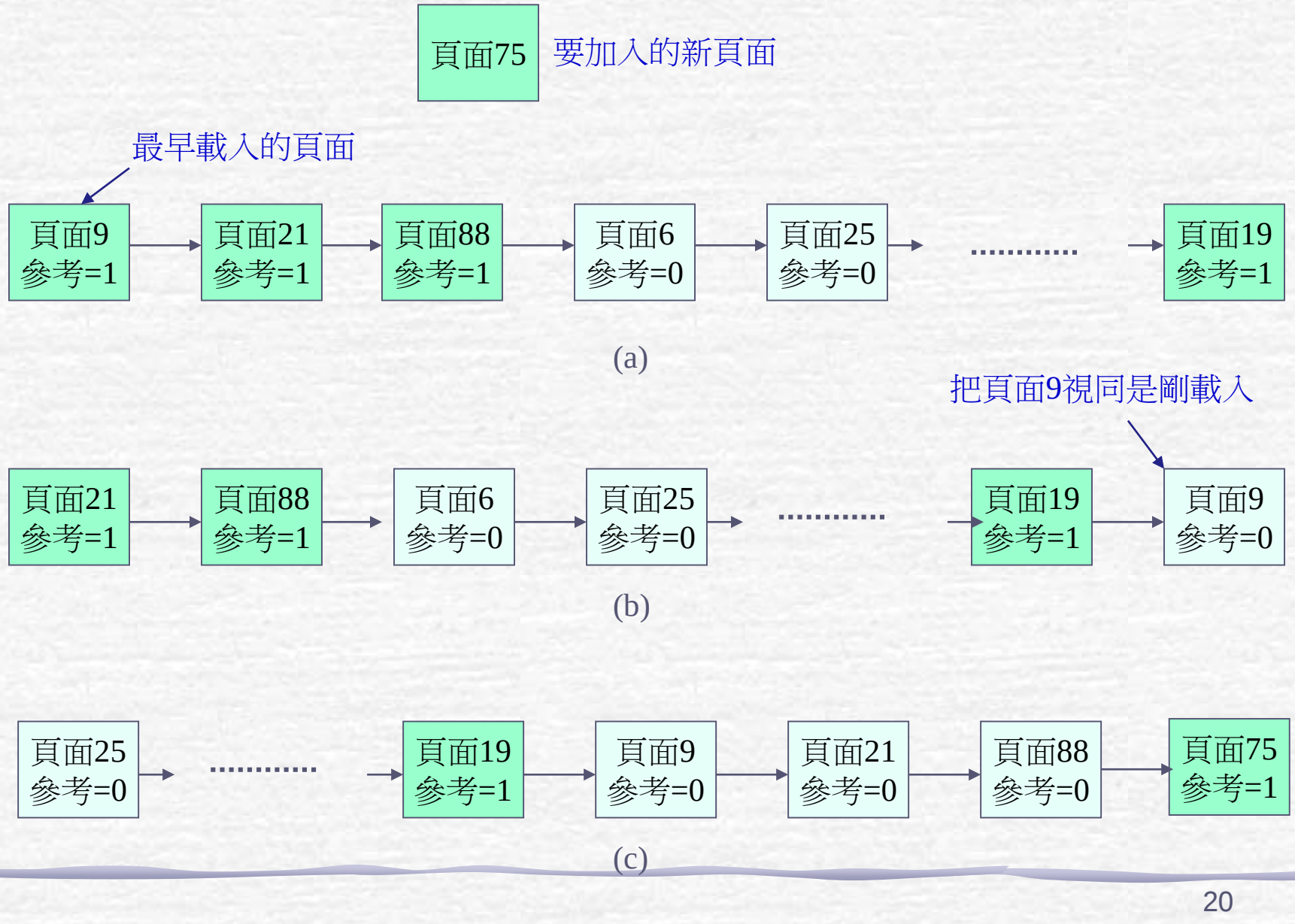
圖5-6 老化演算法範例


參考位元組
主記憶體
00000000
頁面18
00000000
頁面9
00000000
頁面29
00000000
頁面3
(a) 啟始狀況
參考位元組
主記憶體
10000000
頁面18
00000000
頁面9
10000000
頁面29
00000000
頁面3
(b) 頁面18與29被存取
01000000
頁面18
00000000
頁面9
01000000
頁面29
00000000
頁面3
(c) 計時器中斷，位元平移
11000000
頁面18
10000000
頁面9
01000000
頁面29
10000000
頁面3
(d)頁面3、9與18 被存取

LRU近似法 - 二次機會演算法

- 記憶體的每個頁框只對應到1個參考位元，且其初始值為0
- 當頁面被存取的時候，它所對應的參考位元會被設為1
- 當系統需要進行頁面替換時，作業系統會先依照FIFO的順序，尋找在記憶體中待得最久的頁面
  - 如果該頁面的參考位元為0，就會被新頁面取代
  - 如果它的參考位元是1，則會給它第二次機會：將它的參考位元設為0，其載入時間改為目前的時間
  - 再繼續搜尋下一個可能的候選頁面

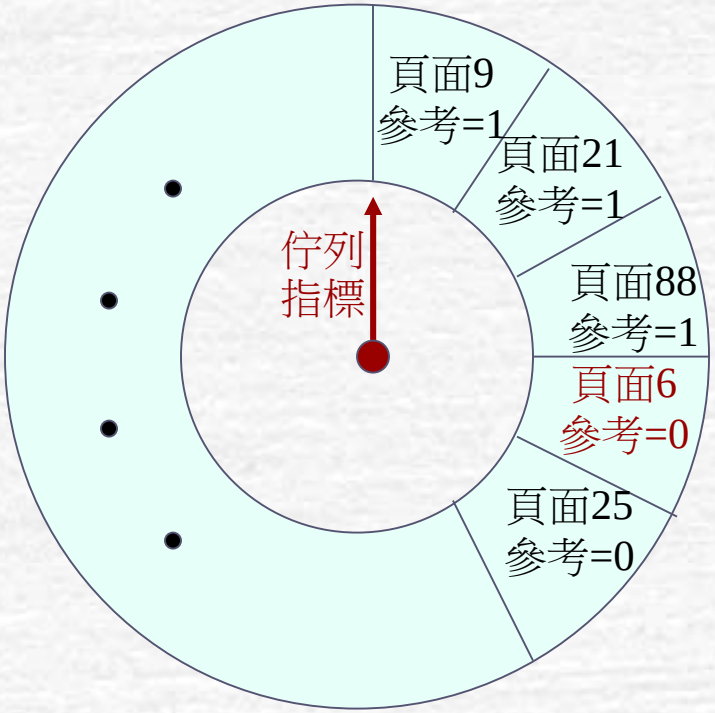
圖5-7 二次機會演算法—鏈結串列



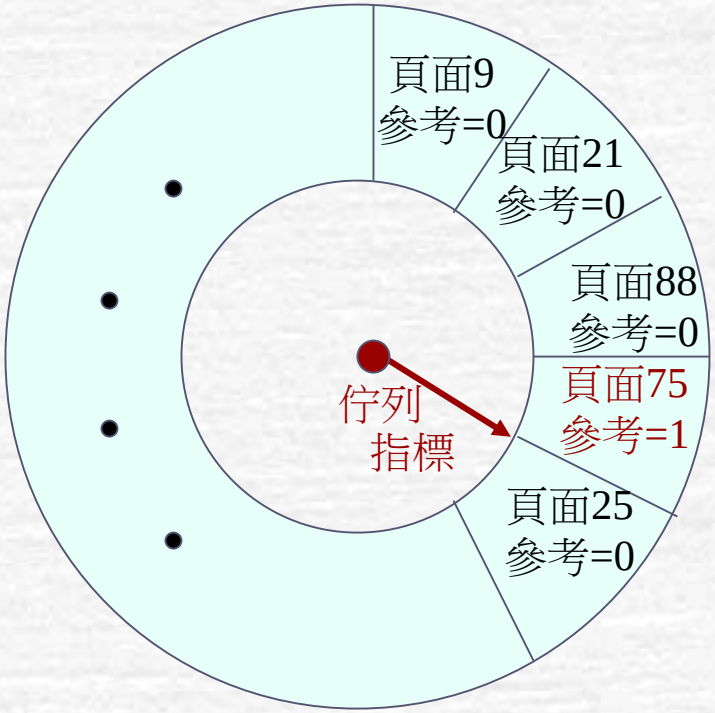
二次機會演算法的實作

- 用鏈結串列來實作二次機會演算法比較沒有效率
- 另一種可行的方式是採用環狀佇列來實作--也稱為clock演算法
  - 在每一次的頁面替換之後，將佇列指標前移到下一個頁框，以節省搜尋的時間

圖5-8 二次機會演算法—環狀佇列



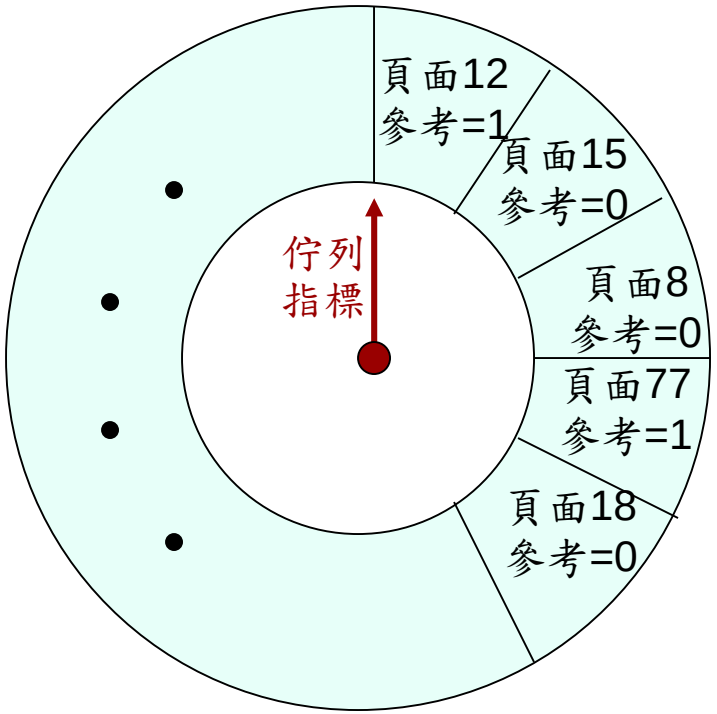
(a) 相當於圖5-7.a



(b) 相當於圖5-7.c

課堂練習

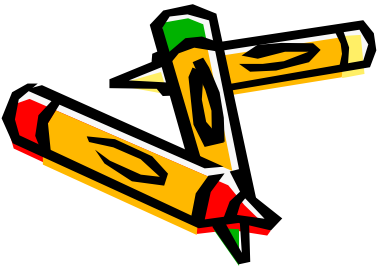
- 假設在使用二次機會演算法進行頁面替換時，目前環狀佇列的狀態如下。當新頁面22、33依序要進入記憶體時：
  - 哪兩個頁面會被替換掉？
  - 替換完成之後，各頁面的參考位元值各為多少？
  - 佇列指標會指向哪個頁面？



練習解答

- 頁面15、8會分別被22、33替換掉
- 佇列指標會指向頁面77
- 新的參考位元值為：

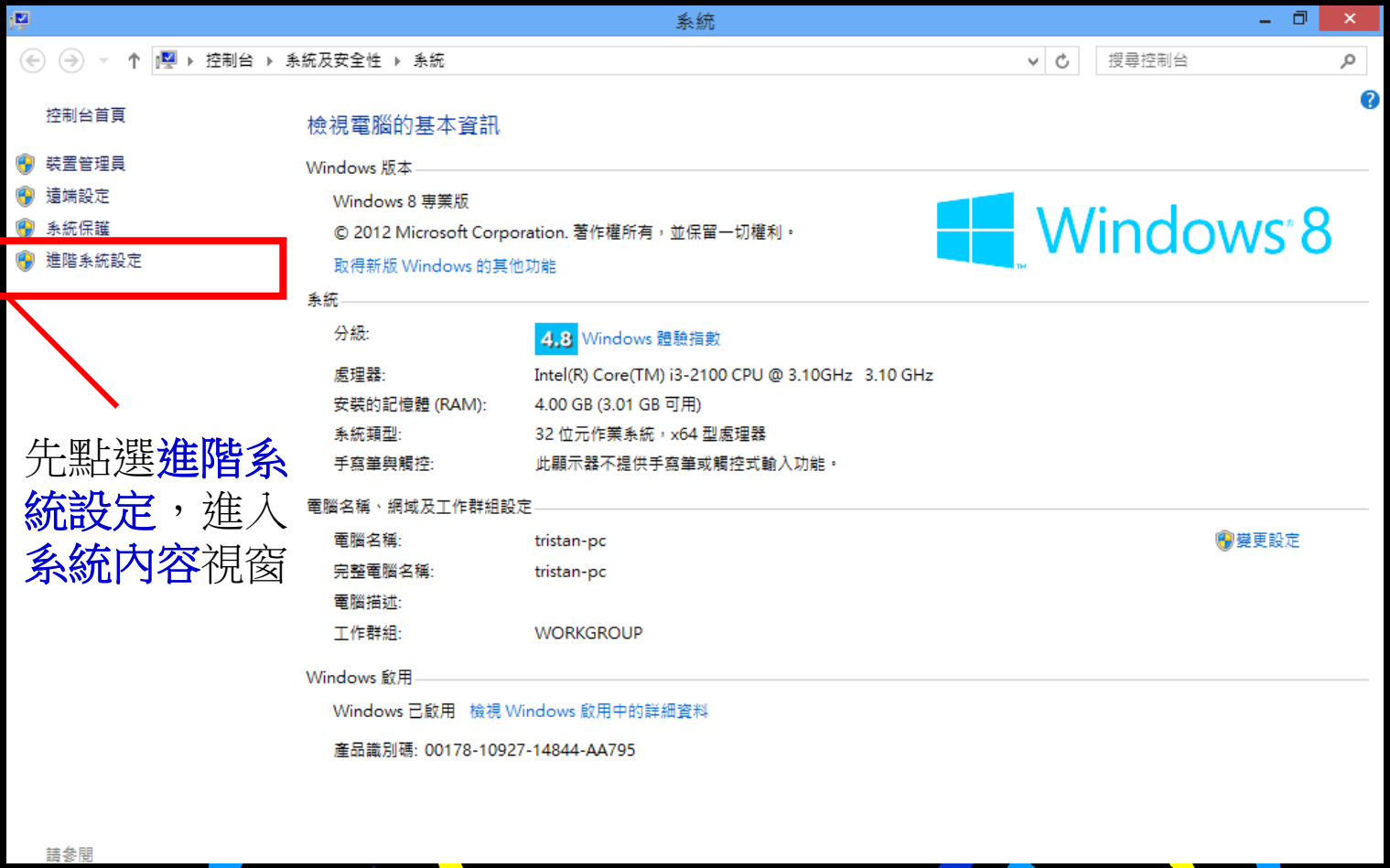
頁面	12	22	33	77	18
參考位元	0	1	1	1	0



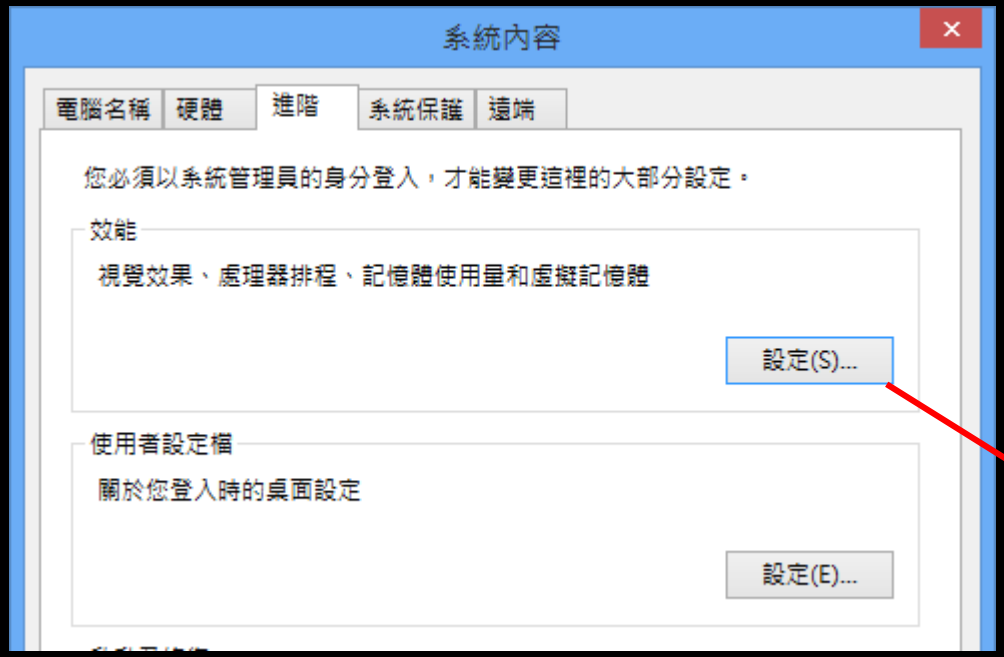
動手做做看
在Windows中調整虛擬記憶體的大小

- 從開始畫面中點選桌面動態磚
- 打開檔案總管
- 在電腦項目上按滑鼠右鍵，並點選內容

動手做做看
在Windows中調整虛擬記憶體的大小

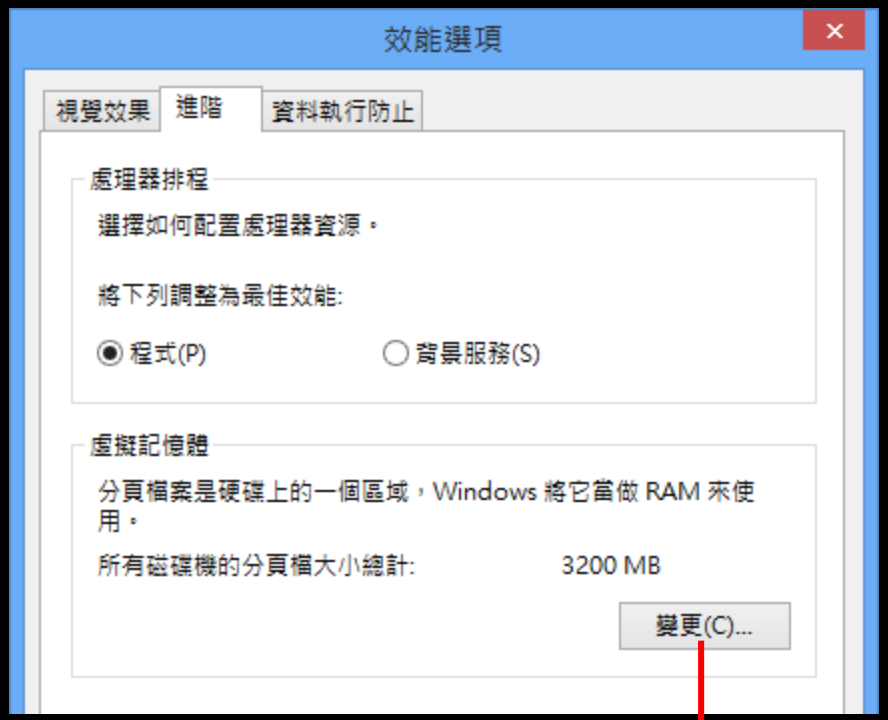


先點選**進階系統設定**，進入**系統內容**視窗



切換到**進階**頁籤，
按下**效能**欄位中的
設定鈕

圖5-22



開啟**效能選項**視窗後，在**進階**頁籤下選擇**變更**鈕

圖◆23

圖5-25 Windows的虛擬記憶體設定畫面