Chapter 10

虛擬記憶體

章節目標

- 定義虛擬記憶體，並描述其好處
- 說明如何使用需求分頁將頁面載入記憶體中
- 應用先進先出、最佳和 LRU 頁面替換演算法
- 描述一行程的工作集，並說明其程式局部性的關係
- 描述 Linux、Windows 10 和 Solaris 如何管理虛擬記憶體
- 使用 C 程式語言設計虛擬記憶體管理模擬

10.1 背 景

- 整個程式的載入記憶體是不必要的。例如，考慮以下情形：
  - 程式中經常有程式碼去處理不尋常的錯誤狀況。針對錯誤管理而寫的程式碼幾乎也從未被執行
  - 陣列、串列和表格經常占有著比它們實際所需更多的配置
  - 程式的某些選項和特徵很少被使用到
- 雖然有時我們也需要用到整個程式，但畢竟不會同時需要整個程式的所有部份

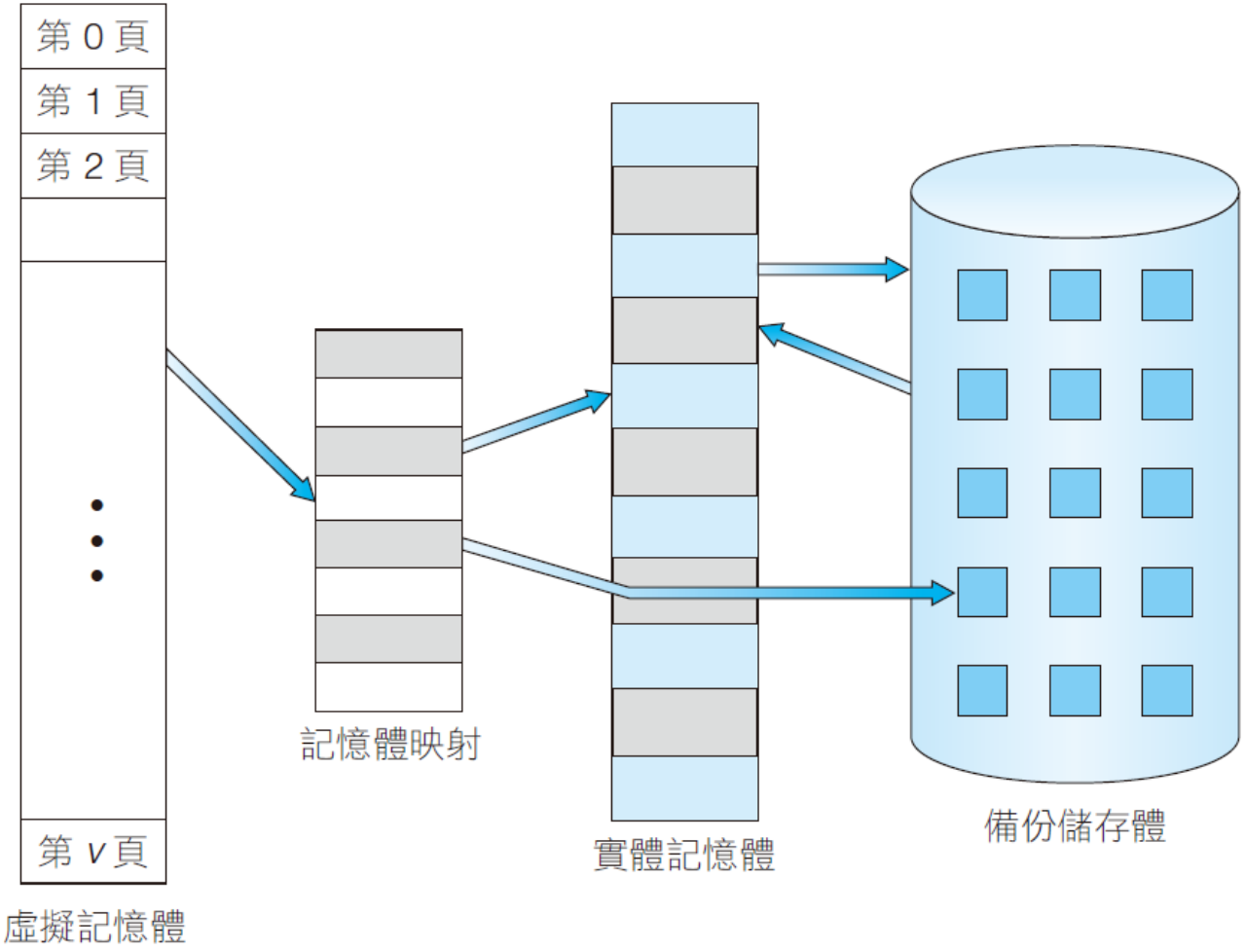
10.1 背 景

- 能執行的只有部份程式在主記憶體中的能力有著多項的優點：
  - 程式不再被可用實體記憶體的總量所限制
    - 使用者可以為一很大的虛擬 (virtual) 位址空間寫作程式，簡化程式寫作的工作
  - 因為各個使用者程式占有較少的實體記憶體，更多程式可同時一起執行
    - CPU 的利用率和輸出量也隨之增加，但是回應時間和回復時間則不因此而增長
  - 載入和置換各個使用者程式入記憶體的所需 I/O 會較少，所以使用者程式的執行將較快

10.1 背 景

- **虛擬記憶體** (virtual memory)
  - 邏輯記憶體與實體記憶體之間的分隔
    - 在一個比較小的實體記憶體提供大量的虛擬記憶體
    - 程式規劃工作變得十分容易
    - 程式設計師不需煩惱有多少實體記憶體空間可使用

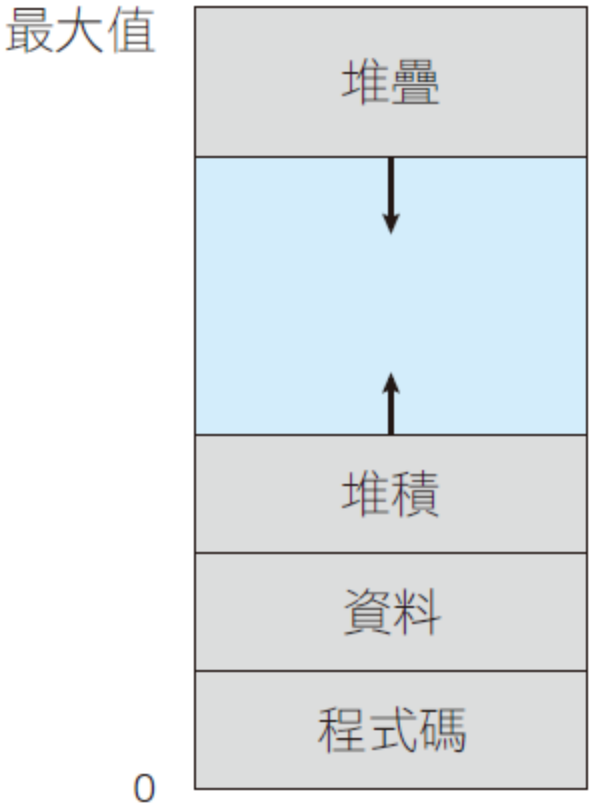
圖 10.1　本圖顯示大於實體記憶體的虛擬記憶體



10.1 背 景

- **虛擬位址空間** (virtual address space)
  - 涉及到行程如何儲存在記憶體的邏輯 (或者虛擬) 觀點
    - 通常是行程從一個特定的邏輯位址從位址 0 開始，而且存在於連續記憶體中
    - 實體記憶體可能以分頁欄來組織，而且這個實體分頁欄可能以下不連續的方式分配給一個行程
    - 把邏輯分頁映射到實際分頁欄是由記憶管理單位 (MMU) 來決定

圖 10.2　虛擬位址空間



10.1 背 景

- 讓堆積在記憶體向上成長
  - 堆疊透過連續的功能呼叫讓記憶體向下成長
  - 在堆積和堆疊之間大的空白空間是虛擬位址空間的一部份
    - 只有在堆積或堆疊成長時，這段虛擬位址空間才需要真實的實體分頁
  - 包含區間的虛擬位址空間即是**鬆散** (sparse) 的位址空間
    - 使用鬆散的位址空間是有益的
    - 因為堆疊或堆積的成長區段、或者在程式執行時
    - 如果我們想要動態鏈結到程式庫 (或其它共用物件)，皆能將區間填滿

10.1 背 景

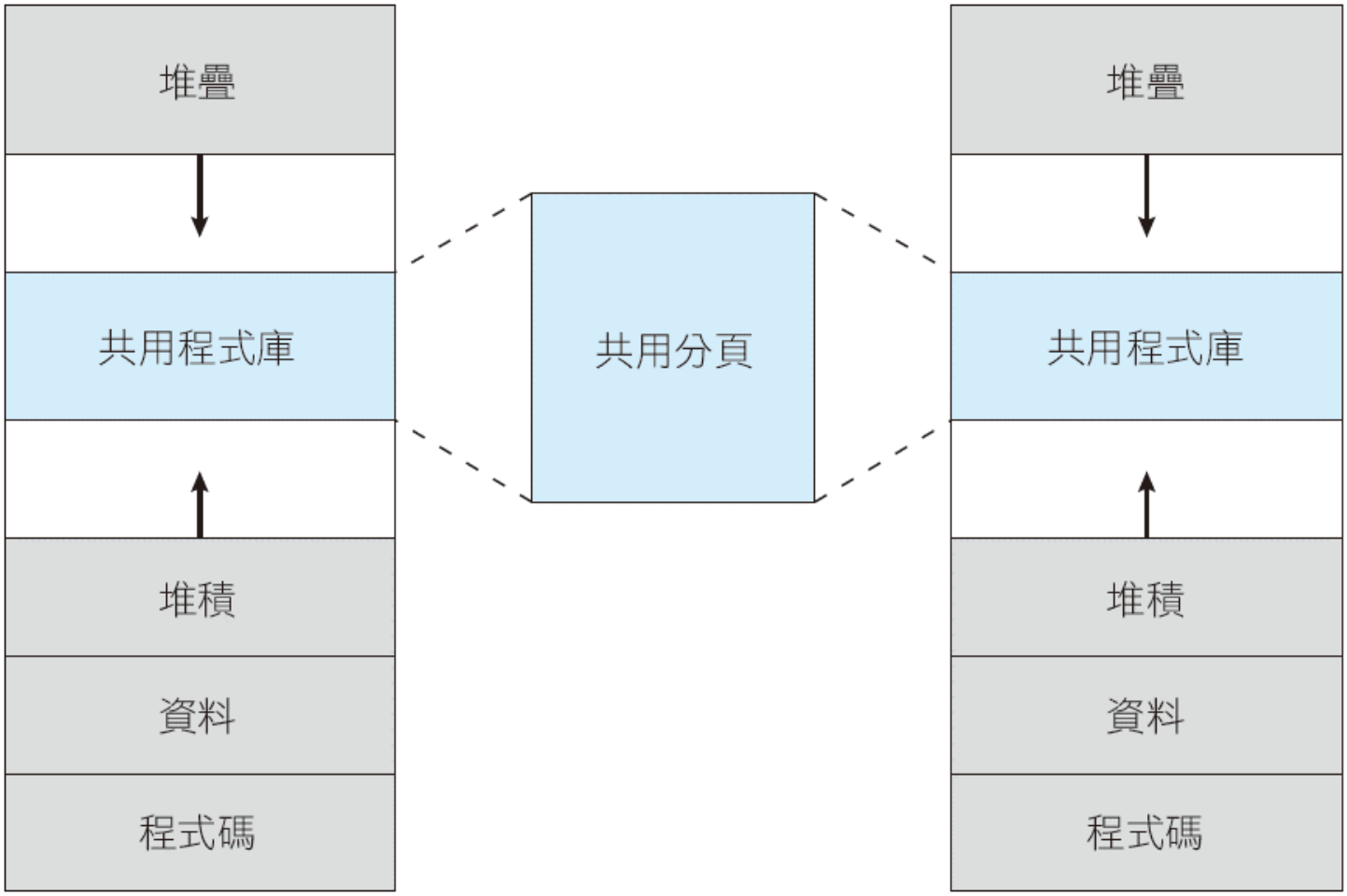
- 除了將實體記憶體和邏輯記憶體分隔之外，虛擬記憶體也允許檔案和記憶體經由分頁共用，讓 2 個或多個行程共用。這將產生下列優點：
  - 系統程式庫在經由共用物件的映射到虛擬位址空間之內，可以被幾個行程共用
    - 雖然每個行程認為被共用的程式庫是它的虛擬位址空間的一部份，存放在真實記憶體中的程式庫真實分頁被所有行程共用 (圖 10.3)
    - 通常程式庫會以唯讀方式映射到每一個行程的空間然後鏈結。
  - 虛擬記憶體使行程能夠共用記憶體，兩個或更多的行程能互相通信

10.1 背 景

- 虛擬記憶體讓行程產生與另一個行程共用的記憶體區域
- 分享這一個區域的行程，視它為自身虛擬位址空間的一部份，然而記憶體的真實實體分頁被分享

- 分頁可以在行程以 fork() 系統呼叫產生行程時被共用，因此加速行程產生

圖 10.3　使用虛擬記憶體的共用程式庫



10.2 需求分頁

- 考慮一個可執行的程式如何由磁碟載入到記憶體
  - 一個選擇是在程式執行時間內，在實體記憶體中載入整個程式
  - 這個方法的問題是，最初我們不需要整個程式在記憶體中。
  - 考慮從一列使用者可選擇的程式開始
    - 載入整個的程式進入記憶體，造成載入所有選擇的可執行程式碼，不管這個選擇是否最後被使用者選擇
    - 一個替代策略是，一開始只載入他們所需要的分頁，這個策略稱為**需求分頁** (demand paging)，並且普遍用於虛擬記憶體系統中

10.2 需求分頁

- 使用需求分頁的虛擬記憶體，分頁只在程式執行時有需要的時候才載入，因此從不被存取的分頁就從不載入到實體記憶體

- 需求分頁系統就像一種使用置換法的分頁系統
  - 行程存放在輔助記憶體
    - 通常是 HDD 或 NVM 裝置中
  - 當我們要執行某個行程的時候，就把那個行程置換至記憶體中
  - 需求分頁解釋了虛擬記憶體的主要好處之一
    - 即僅載入所需程式的一部分，而能可以更有效率地使用記憶體

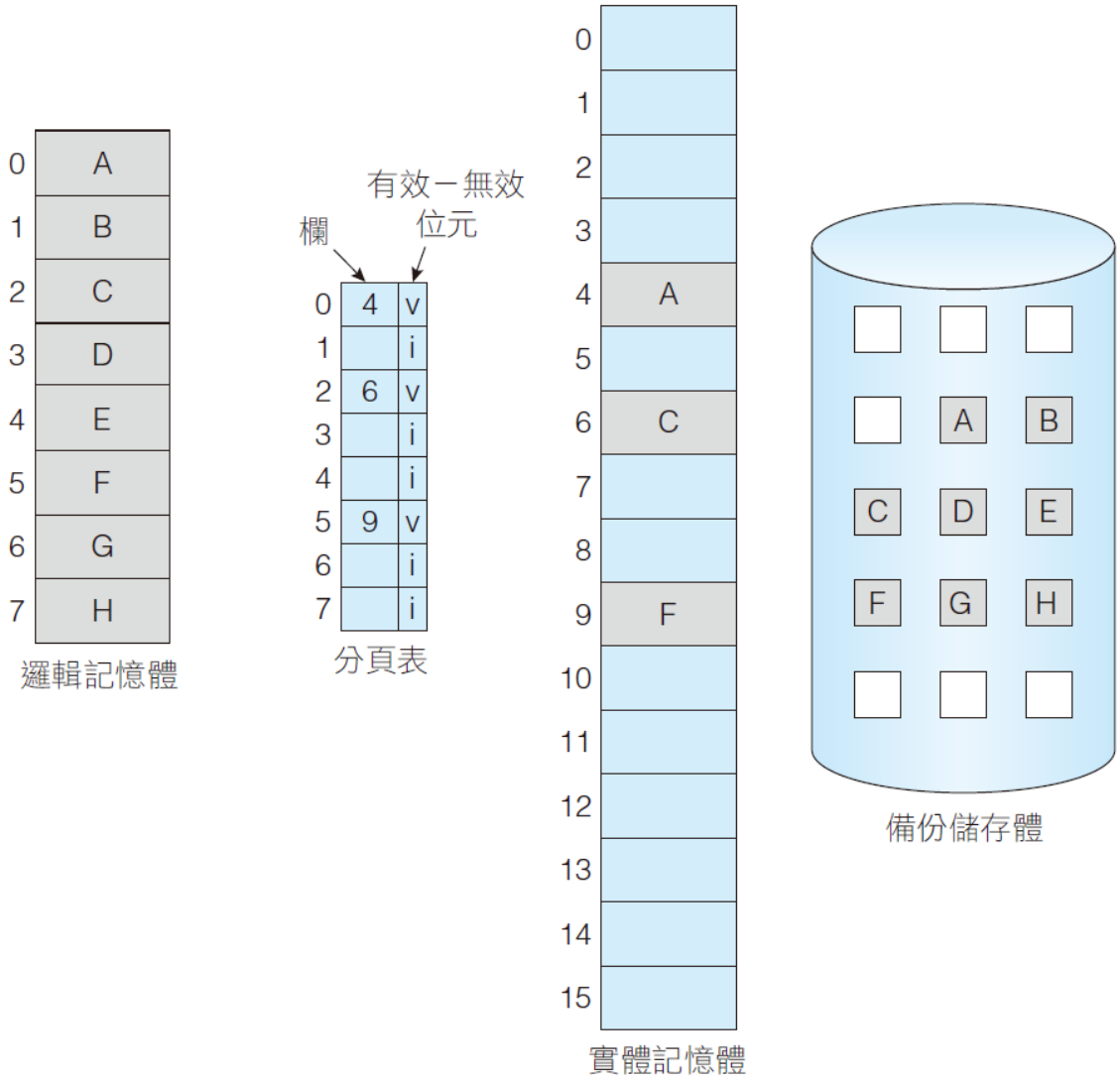
10.2.1 基本概念

- 需求分頁背後的一般概念是，僅在需求發生時才將頁面載入記憶體中
  - 結果當行程執行時，某些分頁將儲存在記憶體中，而某些分頁將儲存在輔助記憶體中
  - 需要某種形式的硬體來支援及區分這兩種狀況
    - 有效－無效位元方法可用於此種目的
      - 當該位元設置為“有效”時，關聯的分頁既合法又在記憶體中
      - 如果該位元設置為“無效”，則該分頁無效
        - » 即不在行程的邏輯位址空間中
      - 或有效但當前處於輔助記憶體

10.2.1 基本概念

- 照常設置進入記憶體的頁面的分頁表條目，但是當前不在記憶體中的分頁的分頁表條目僅標記為無效
  - 這種情況如圖 10.4 所示
    - » 如果進程從未嘗試存取該分頁，則將分頁標記為無效

圖 10.4　當某些頁面不在主記憶體中時的分頁表



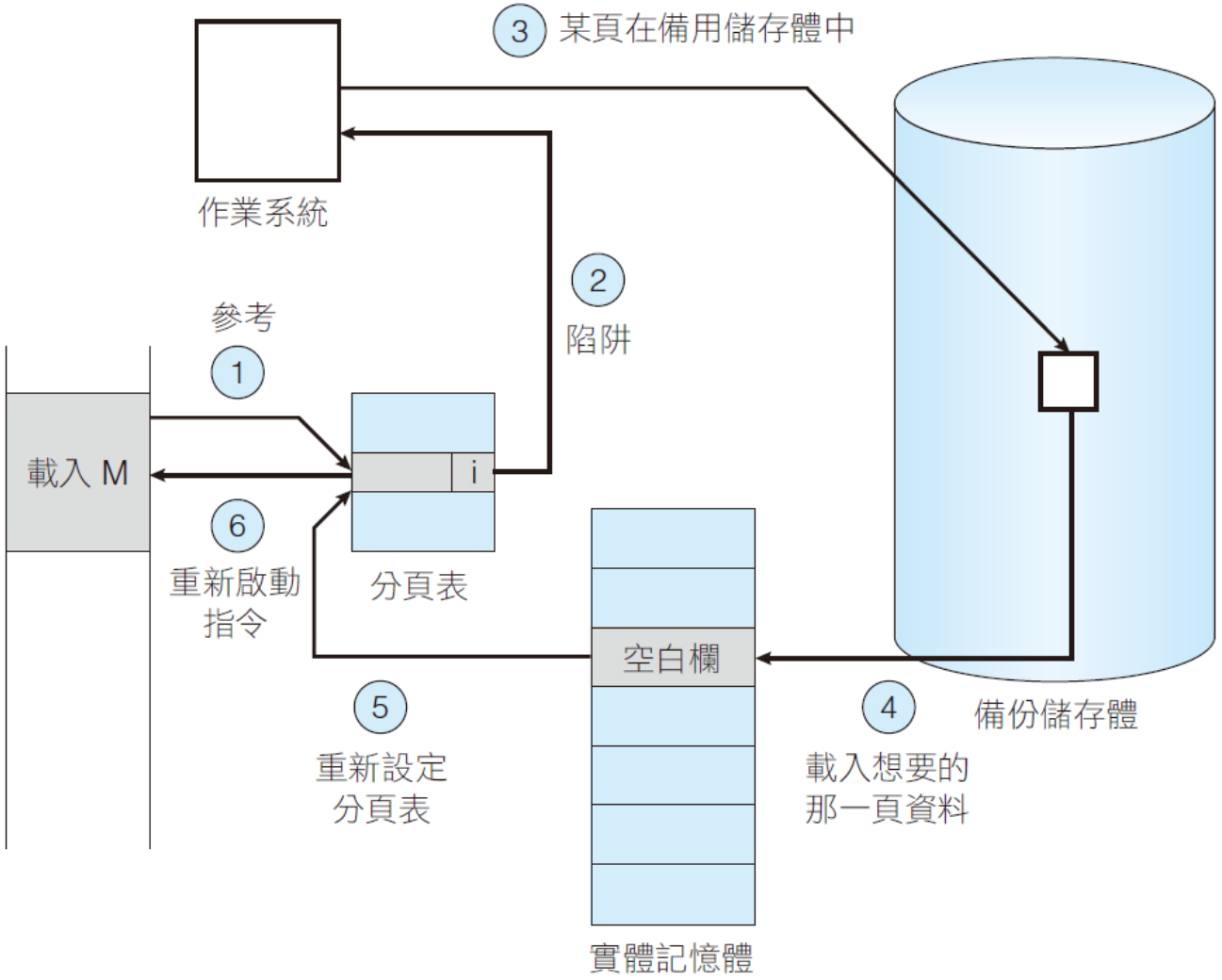
分頁錯誤

- 存取標示為無效的分頁，將產生**分頁錯誤** (page-fault)
  - 分頁用的硬體在經由分頁表轉換位址的時候，將會注意無效的位元已經被設定了
  - 這會引起因為無效位址的錯誤形式，而對作業系統發出陷阱
  - 這個陷阱是因為作業系統沒有把行程所需要的分頁載入記憶體
  - 處理分頁錯誤流程十分直接 (圖 10.5)：
    1. 首先我們檢查行程的內部表格 (通常保存在行程控制區段中)，來決定這項參考是屬於有效還是無效的記憶體存取

分頁錯誤

2. 如果是無效的記憶體參考，便終止該行程
   - 如果是有效的參考，但是我們尚未將該頁載入，現在就必須把該頁載入
3. 找一個空白欄
   - 例如，從空白欄列表中拿一個出來
4. 排定一項磁碟作業去將想要的那一頁讀入新找到的那個欄中
5. 當磁碟讀完了，修改與該行程有關的內部表格與分頁表，以反映該頁目前已在記憶體中了
6. 重新啟動因錯誤位址陷阱而被中斷的那個指令
   - 該行程現在就可存取該分頁，就如同該分頁一直都放在記憶體一樣

圖 10.5 處理頁面錯誤的步驟



純需求分頁

- 在極端的情況之下，可以開始執行一個在記憶體中沒有分頁的行程
  - 當作業系統設定指令指標到這個行程的第一個指令時，因為第一分頁不在主記憶體，所以馬上因該分頁而錯誤
  - 在這一頁載入記憶體之後，行程就可以繼續執行了，它在需要分頁時產生錯誤，直到它所需要的分頁都在記憶體內為止
  - 到那時候，我們就可以毫無錯誤地執行該程式了
  - 這就是**純需求分頁** (pure demand paging)：
    - 直到需要某分頁的時候才把它載入記憶體中

參考區域性

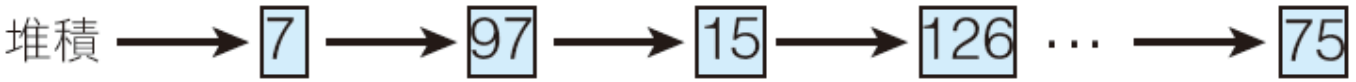
- 理論上，某些程式在執行每一個指令時都會存取一些新的分頁 (一個指令分頁，許多資料分頁)，每個指令都可能會造成許多分頁錯誤
  - 這種情況可能造成系統性能不能夠接受此結果
  - 執行行程的分析證明此行為是非常少見的
  - 程式通常具有**參考區域性** (locality of reference) ，而使需求分頁有不錯的能

10.2.1 基本概念

- 需求分頁法中所需要的硬體支援和分頁法及置換法中所需要的一樣：
  - 分頁表：
    - 此表經由有效－無效位元或保護位元的特殊數值可以標示某單元為無效
  - 輔助記憶體：
    - 此記憶體用來存放不在主記憶體中的那些頁
    - 輔助記憶體通常是一個高速磁碟或 NVM 裝置
    - 該磁碟稱為置換裝置，而為此目的而使用之磁碟區段稱為置換空間 (swap space)
- 需求分頁的重要要求是在分頁錯誤後必須能重新啟動任何指令

10.2.2 空白欄列表

- 解決分頁錯誤，大多數作業系統都維護一個**空白欄列表** (free-frame list)，這是一個滿足這些請求的空白欄
- 作業系統通常使用稱為“**零填充按需** (zero-fill-on-demand)”的技術配置空白欄
  - 零填充按需的技術在配置之前被“清空”，從而清除了先前的內容



10.2.3 需求分頁的性能

- 計算有效存取時間，必須知道處理分頁錯誤要花多久時間。一次分頁錯誤會引起下列的一連串事情發生：
  1. 對作業系統發出陷阱
  2. 保存使用者暫存器與行程的狀態
  3. 認定該中斷乃是一次分頁錯誤
  4. 核對其對某頁的參考是合規定的，並且決定該頁在輔助儲存器上的位置
  5. 產生一次由儲存體讀取可用空白欄的讀取動作
     a. 在讀出裝置的佇列中等待，直到讀出的要求被完成
     b. 等待該裝置的搜尋及/或潛伏時間
     c. 開始將該頁轉移至一個空白欄內

10.2.3 需求分頁的性能

6. 在等待的時候，可以把 CPU 分配給其它的使用者 (CPU 排班)
7. 接收從儲存 I/O 子系統來的中斷信號 (I/O 做完了)。
8. 將其它使用者的程式與暫存器的狀態保存起來 (如果步驟 6 已執行)
9. 確認中斷信號來自於輔助儲存裝置
10. 更正分頁表與其它表格，以表示所要的某頁已在記憶體中
11. 等待 CPU 再次分配給這個行程
12. 重新存入使用者暫存器、行程狀態和新的分頁表，然後恢復中斷指令

10.2.3 需求分頁的性能

- 令一次分頁錯誤出現的機率為 p $(0 \le p \le 1)$。我們當然希望 p 越接近 0 越好
  - 只有一點點分頁錯誤出現。於是有效存取時間 (effective access time) 就是

$$\text{有效存取時間} = (1 - p) \times ma + p \times \text{分頁錯誤的時間}$$

- 三個主要成分：
  1. 處理分頁錯誤中斷
  2. 讀取某頁
  3. 重新啟動該行程

10.2.3 需求分頁的性能

- 在平均分頁錯誤處理時間是 8 毫秒和記憶體存取時間 200 奈秒時，有效存取時間是：

$$\begin{aligned} \text{有效存取時間} &= (1-p) \times (200) + p\,(8\text{ 毫秒}) \\ &= (1-p) \times 200 + p \times 8{,}000{,}000 \\ &= 200 + 7{,}999{,}800 \times p \end{aligned}$$

  - 有效存取時間和**分頁錯誤比率** (page-fault rate) 成正比

- 如果一千次存取裡面有一次引起分頁錯誤，那麼有效存取時間就等於 8.2 微秒
  - 因為需求分頁而造成電腦速度減緩的因數是 40！如果希望有小於百分之十的減緩

10.2.3 需求分頁的性能

- 保持分頁錯誤比率在以下的層次：

$$220 > 200 + 7{,}999{,}800 \times p$$

$$20 > 7{,}999{,}800 \times p$$

$$p < 0.0000025$$

  - 也就是說，若要保持因分頁而造成的緩慢在一個合理的水準，就必須每 399,990 次存取中只准產生一次分頁錯誤

10.2.3 需求分頁的性能

- 以 I/O 處理置換空間，通常要較以檔案系統來處理為快
  - 因為置換空間通常以較大區塊分配，而不用牽涉檔案搜尋及間接配置
- 利用這種技巧，檔案系統本身有如備份儲存
  - 無論如何，置換空間依然必須使用在沒有結合檔案的分頁
  - 稱為**匿名記憶體** (anonymous memory)
- 行動作業系統通常不支援置換。取而代之的是，這
  - 系統從檔案系統做需求分頁，如果記憶體受限制時，從應用程式唯讀分頁 (例如程式碼)

10.3 寫入時複製

- 想想 fork() 系統呼叫是以複製父行程的方式產生子行程
  - fork() 藉由為子行程複製父行程位址空間的方式工作
  - 考慮許多子行程在產生之後立即呼叫 exec() 系統呼叫，父行程位址空間的複製可能是不需要的
- **寫入時複製** (copy-on-write)：是讓父行程和子行程在最初時共享相同的分頁
  - 這種共享的分頁被標為寫入時複製的分頁，這表示如果有行程寫入共享的分頁，共享分頁的複製就產生了

10.3 寫入時複製

- 提供系統呼叫 fork() 的變形
  - vfork() [用在 (virtual memory fork)]
  - vfork() 在寫入時複製的操作**虛擬記憶體的 fork** 與 fork() 不同
- 使用 vfork() 時，父行程被暫停，而子行程使用父行程的位址空間
  - 因為 vfork() 沒有使用寫入時複製，如果子行程改變任何父行程位址空間的分頁，被更改的分頁在父行程恢復執行時都看得見
  - vfork() 必須小心地使用，以確保子行程不會修改父行程的位址空間

10.3 寫入時複製

- vfork() 在子行程一產生之後就呼叫 exec() 時被使用到
- 因為不會發生分頁的複製，所以 vfork() 是一種非常有效率的行程產生方法

圖 10.7　在行程 1 修改分頁 C 之前

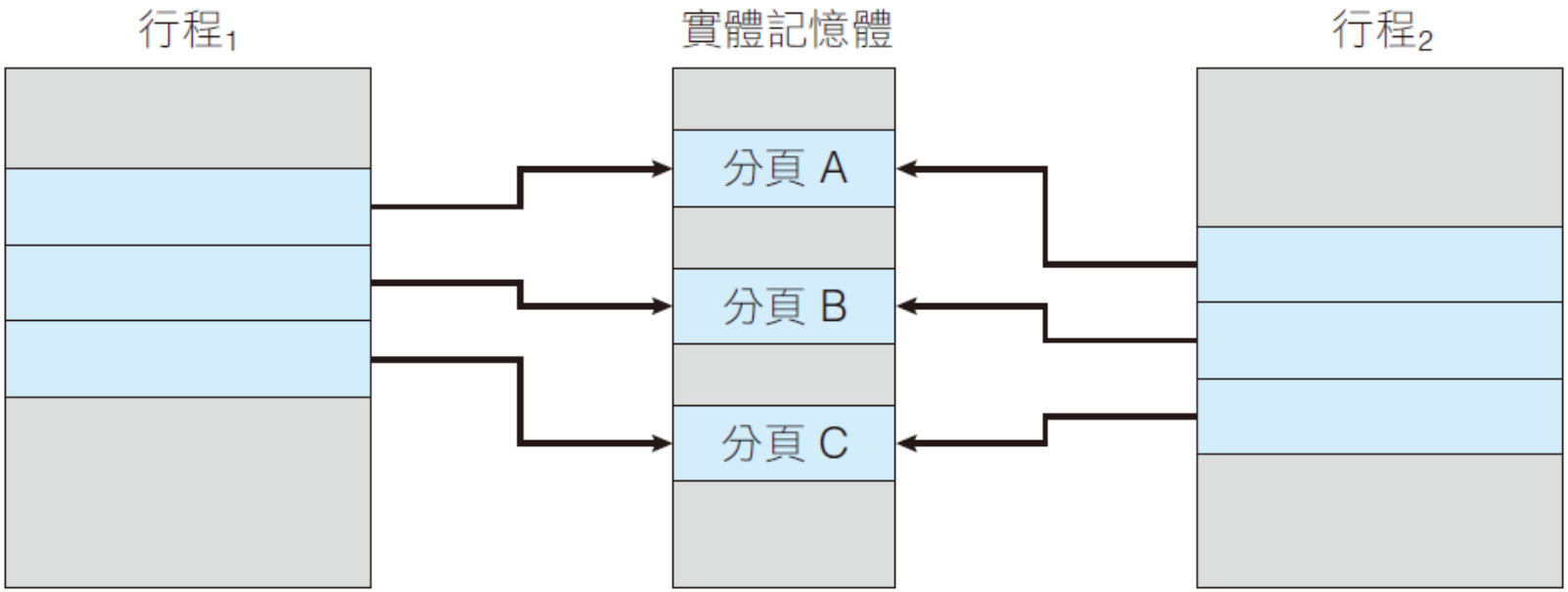
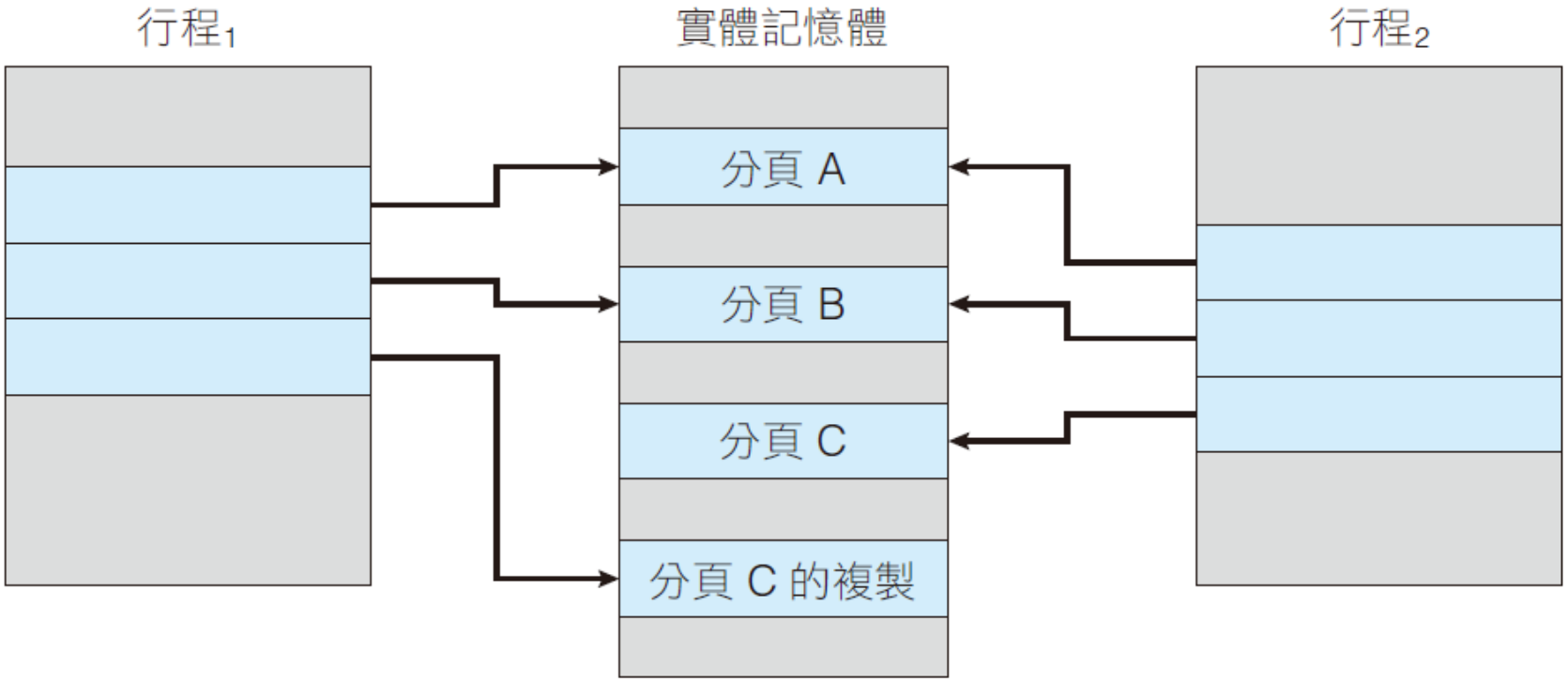


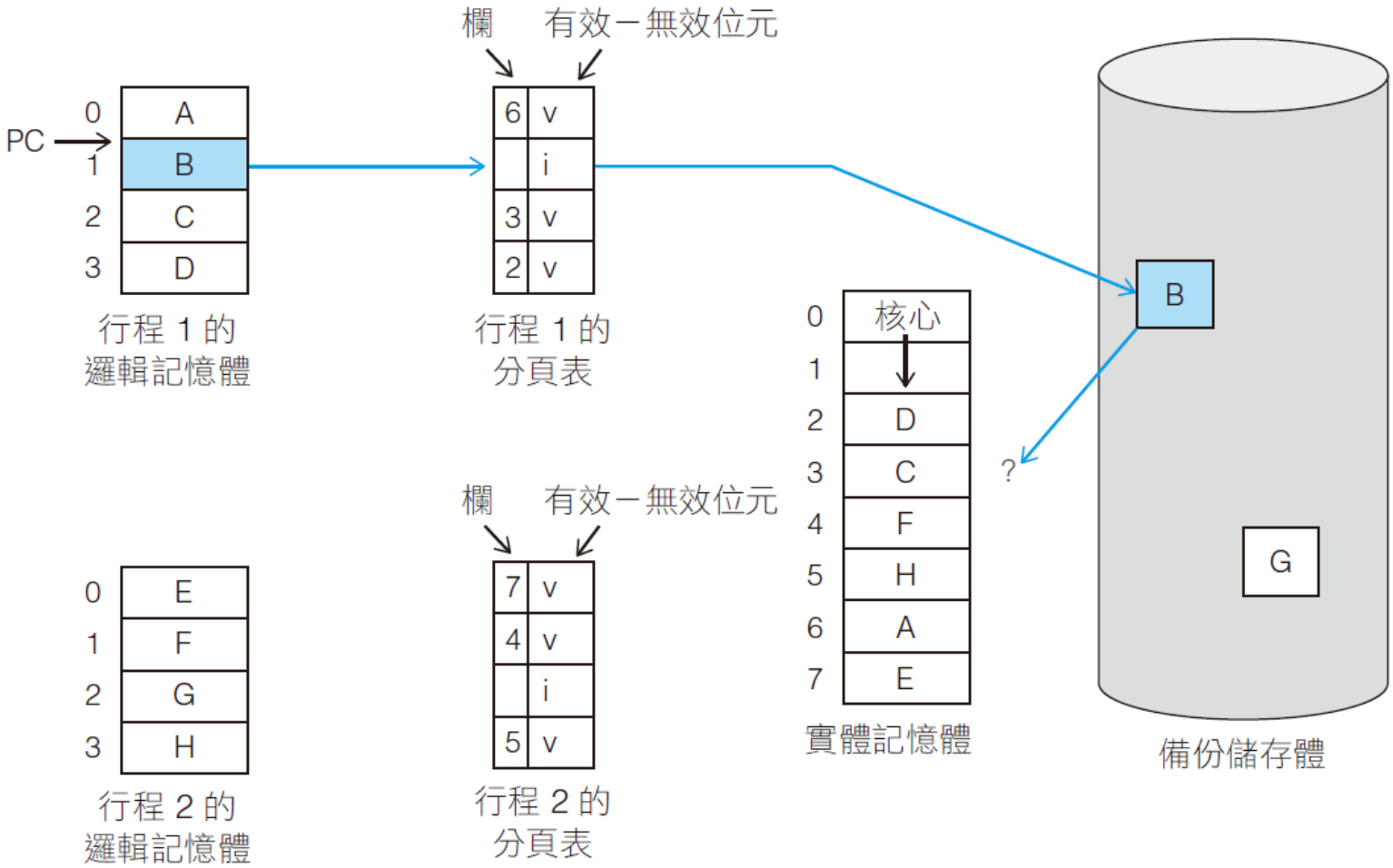
圖 10.8　行程 1 修改後的分頁 C



10.4 分頁替換

- 如果我們增加多元程式規劃的程度，我們就**過度配置** (over-allocating) 記憶體
- 過度配置將以下列方式展現出來
  - 當執行一個使用者行程的時候，出現一個分頁錯誤
  - 作業系統確定所要的那一頁存在輔助記憶器的什麼地方
  - 但是卻發現在可用空白欄的表中已無空白欄了
    - 所有記憶空間都已被使用了

圖 10.9　分頁替換的需要



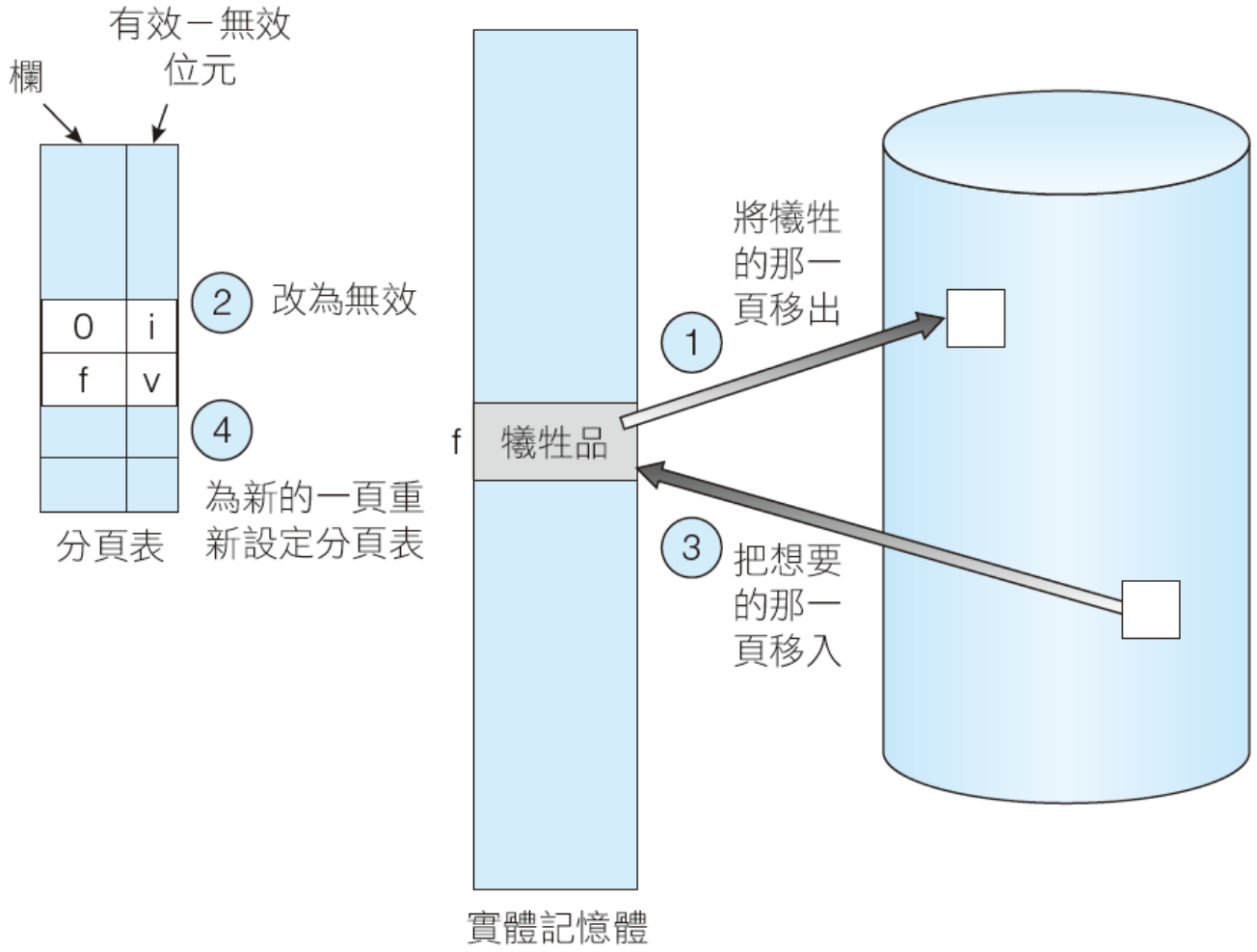
10.4.1 基本分頁替換

1. 找出想要的那一頁在輔助儲存器中的什麼地方。
2. 找出一個空白欄：
   a. 如果有空白欄就直接使用它
   b. 若沒有可用的欄，就使用分頁替換演算法去找一個當作**犧牲品的欄** (victim frame)
   c. 將犧牲品寫入輔助儲存器中，並且更改分頁表與分頁欄
3. 將想要的那一頁讀入空白 (新的) 欄中；更改分頁表與分頁欄
4. 從分頁錯誤的地方繼續執行使用者的行程

10.4.1 基本分頁替換

- 如果沒有空白欄可以使用，就需要轉移兩頁
  - 一頁進來和一頁出去
- 在這個情況之下，將會使得分頁錯誤處理時間增加一倍，同時也會增加有效存取時間。

圖 10.10　分頁替換


有效－無效
位元
欄
0 i
f v
2
改為無效
4
為新的一頁重
新設定分頁表
分頁表
f
犧牲品
1
將犧牲
的那一
頁移出
3
把想要
的那一
頁移入
實體記憶體

10.4.1 基本分頁替換

- 欄的配置演算法 (frameallocation algorithm) 與分頁替換演算法 (page-replacement algorithm)
  - 也就是說，如果我們有很多行程在記憶體中，我們必須決定每個行程要分給它多少欄使用
- 我們評估一個演算法是藉著執行一連串對記憶體的參考，並且計算分頁錯誤的次數來評估它
  - 這一連串的對記憶體參考稱為**參考字串** (reference string)

10.4.1 基本分頁替換

- 第一，對於某個大小的頁來說(頁的大小一般都是被硬體或系統所固定)，我們只要考慮頁數的問題，而不是整個位址
- 第二，如果我們對第 p 頁做參考，那麼任何立即跟著的對第 p 頁的參考都不會造成分頁錯誤
- 在第一次參考之後第 p 頁就在記憶體中了，所以接下來的參考當然不會產生分頁錯誤

10.4.1 基本分頁替換

- 為了要確定某參考字串和分頁替換演算法的分頁錯誤的次數，我們同時也需要知道有多少分頁欄可以使用
  - 當可用的欄數增加之後，分頁錯誤的次數將會減少
- 為了描述各種分頁替換演算法，我們將使用下列的參考字串

  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

圖 10.11　分頁錯誤與欄數的關係

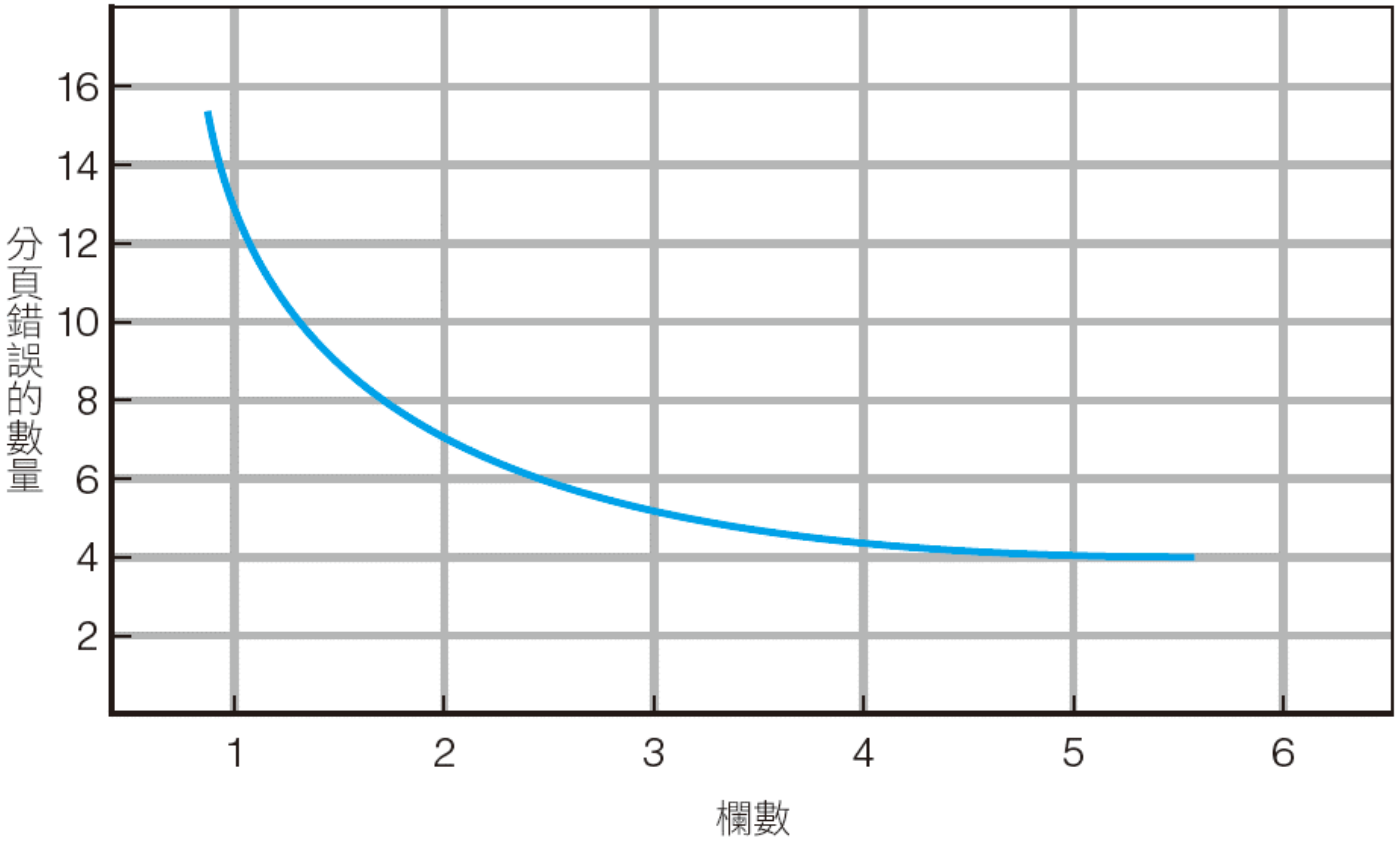
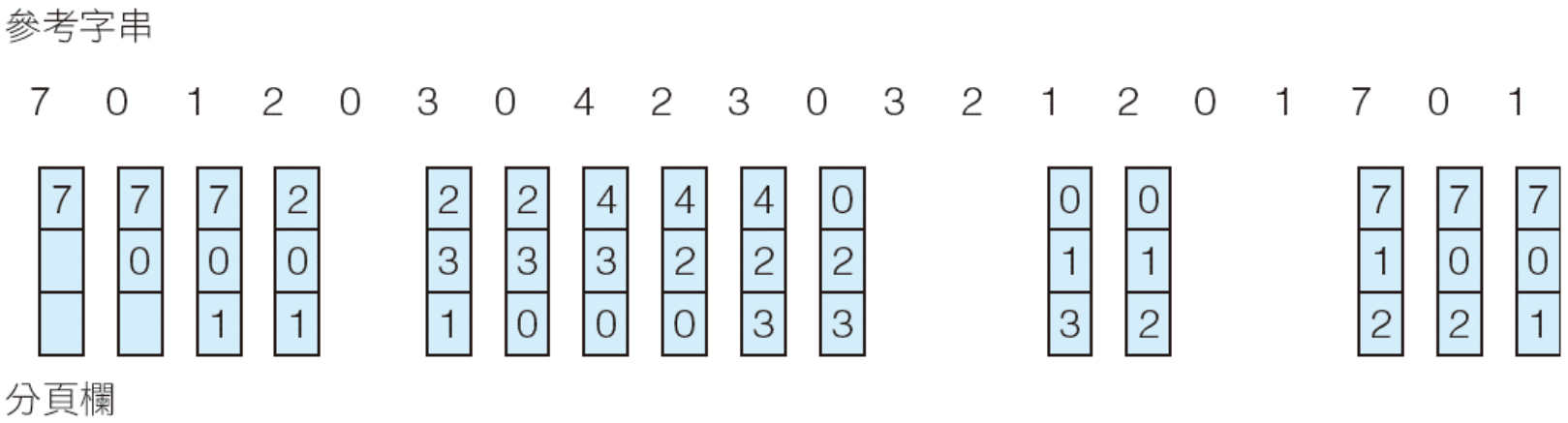


圖 10.12　FIFO 分頁替換演算法



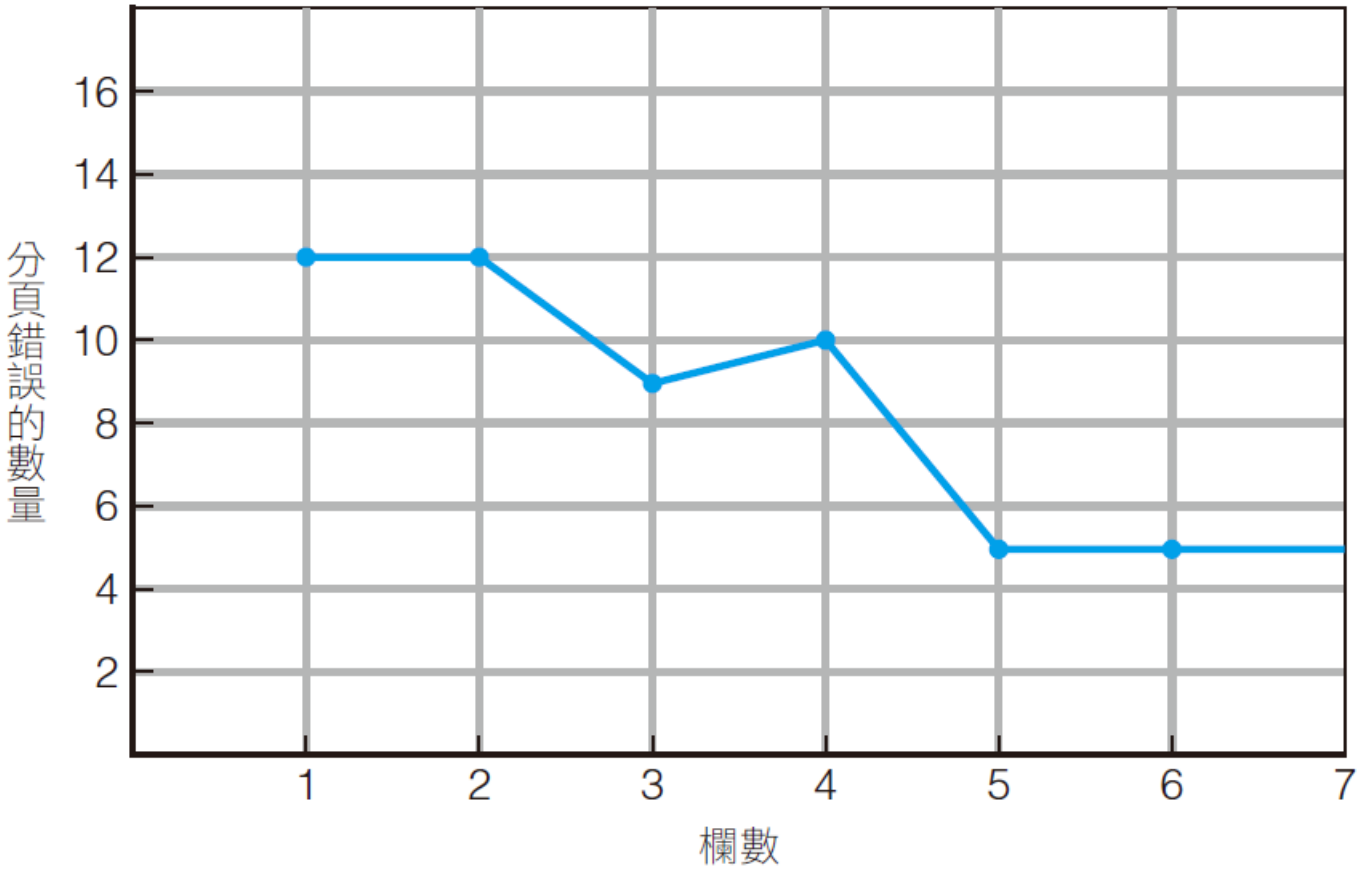
10.4.1 基本分頁替換

- 為了描述這個在 FIFO 分頁替換演算法可能遭遇的問題，試考慮下列的參考字串：

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

  - 圖 10.13 中就是分頁錯誤對可用欄數量所做的曲線
  - 我們注意到四個欄 (10) 的分頁錯誤比三個欄 (9) 的分頁錯誤大！
  - 這個結果是最無法預料的，也就是有名的**畢雷地異常** (Belady's anomaly) 現象

圖 10.13　在參考字串上替換 FIFO 的分頁錯誤曲線



10.4.3 最佳分頁替換

- 把未來最長時間之內不會被用到的那一頁替換掉
  - 以先前那個參考串為例子，最佳分頁替換演算法將產生 9 次分頁錯誤

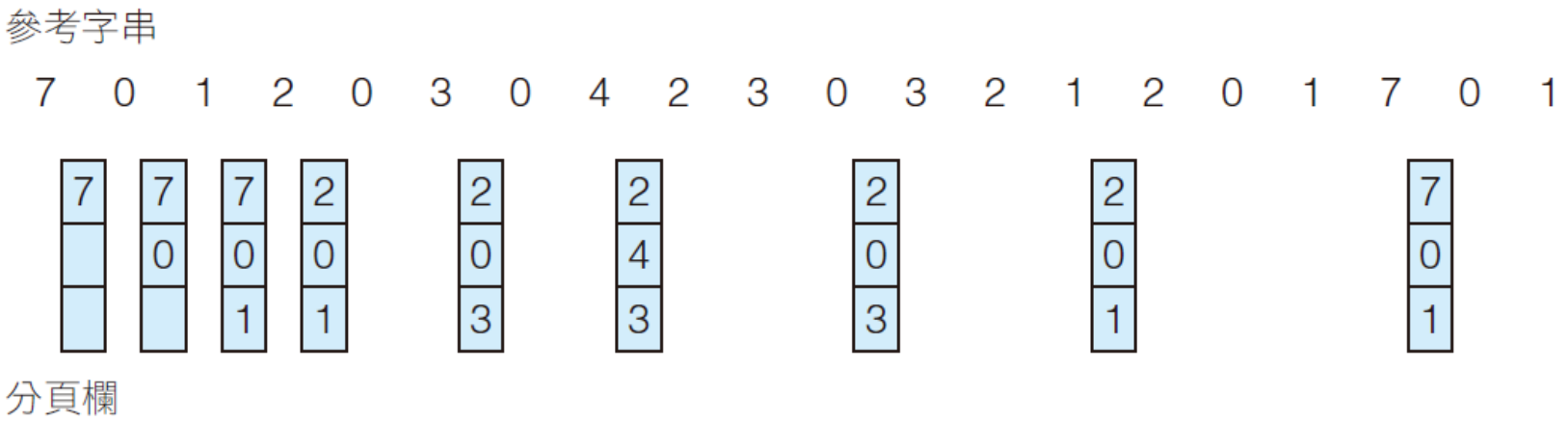
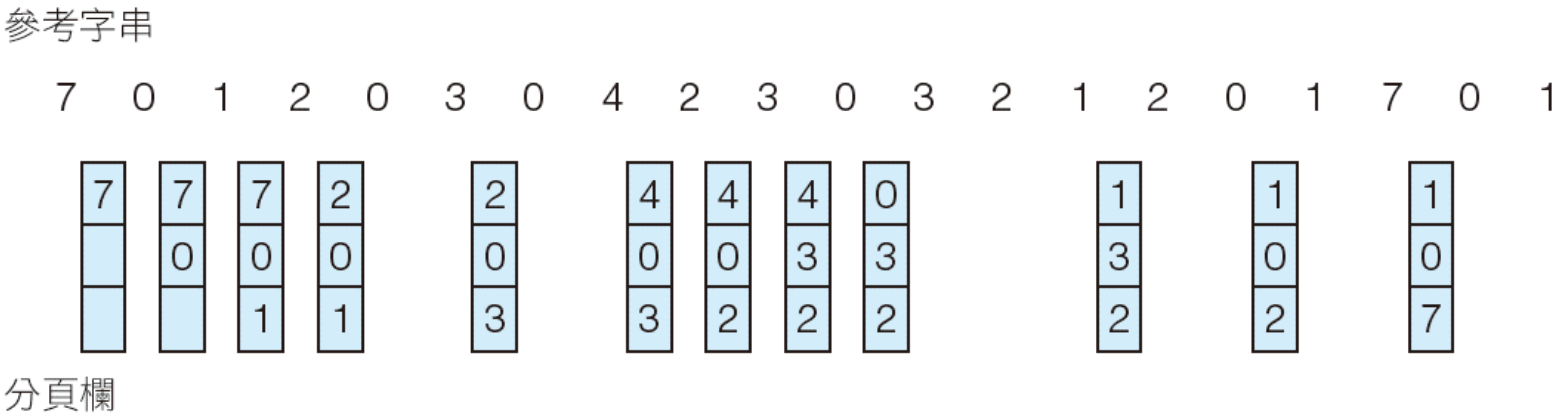


圖 10.14 最佳分頁替換演算法

10.4.4 LRU 分頁更換

- 將最近的過去當做最近的將來
- 替換的是最久未被使用的那一頁
- 與每一頁前一次被使用的時間極有關聯



- LRU 的十二次錯誤仍然勝過 FIFO 的十五次錯誤
- 如何去執行

10.4.4 LRU 分頁更換

- 計數器 (counters)
  - 每個單元都附著一個使用時間暫存器，並且在 CPU 上加入一個邏輯時鐘或計數器
  - 每經過一次記憶體參考之後時鐘就增加一次
    - 時鐘暫存器內的值就會為該頁複製至它的分頁表中的使用時間暫存器在這種方式下
    - 我們始終擁有每一分頁最後一次參考的“時間”我們替換掉最小時間數值的分頁
    - 出溢位 (overflow) 的問題必須要考慮

10.4.4 LRU 分頁更換

- 堆疊 (stack)
  - 每當某一頁被參考過後，它就被從堆疊中移出並放在堆疊的頂端
  - 在堆疊頂端的就是最近經常被使用的頁，而在堆疊底部的則是最近被使用的
  - 以一個雙向的鏈結串列來執行其工作
    - 其中有一個頭指標和一個尾指標
  - 要移動某一頁並將其放在堆疊的頂端時，在最壞的情形下需要更改六個指標
  - 每次更新都必須花費許多時間
  - 特別適合以軟體或微程式碼來執行 LRU 的替換

10.4.4 LRU 分頁更換

- LRU 替換和最佳替換一樣，都不會遭受到畢雷地異常現象
- 這兩種演算法都屬於一種稱為**堆疊演算法** (stack algorithms) 的分頁替換演算法類別

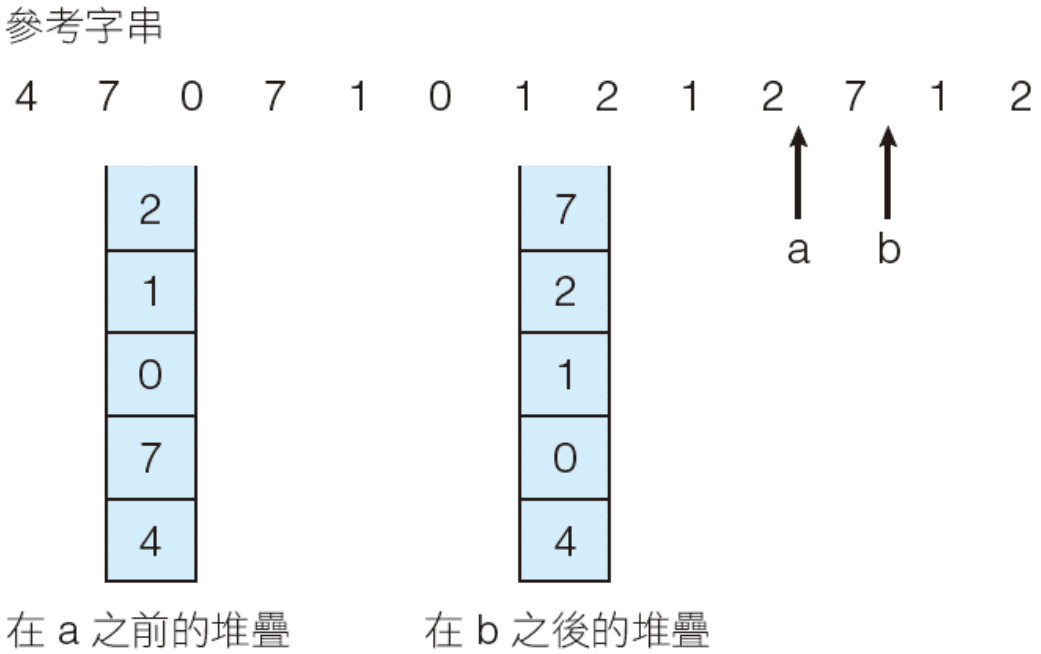
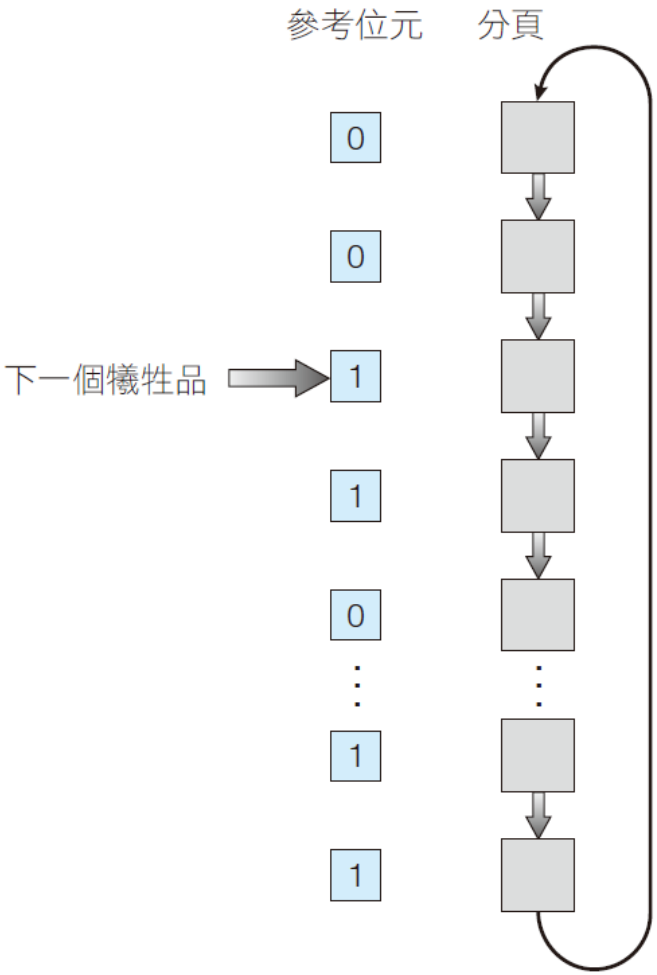


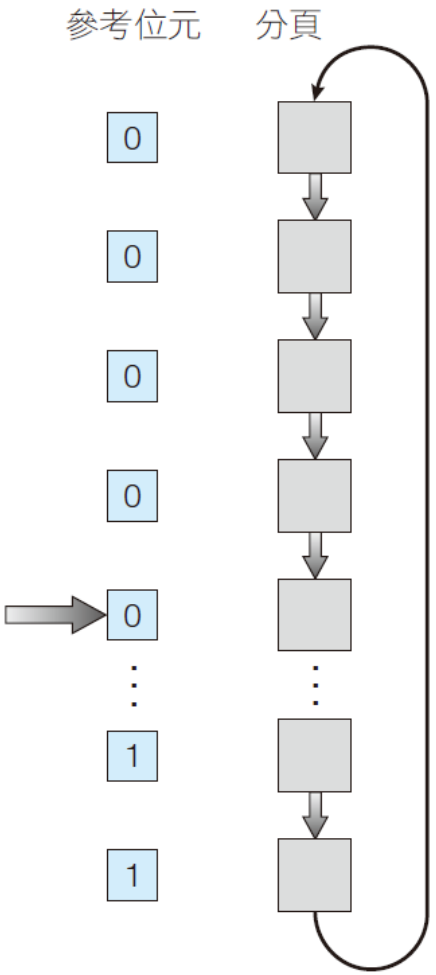
圖 10.16 使用堆疊來記錄最新的分頁參考

10.4.5 LRU 近似分頁替換

- 額外的參考位元演算法
- 第二次機會演算法
  - FIFO 替換演算法，當某頁被選出來之後，檢視它的參考位元
    - 參考位元為 0，進行替換這一頁的工作
    - 參考位元是 1，給那一頁第二次機會，並且繼續去選出下一個 FIFO 頁
      - 當某頁得到第二次機會之後，它的參考位元就被清除掉，並且將它的到達時間重置為目前的時間，因此被給予第二次機會的頁，直到其它所有的頁都被替換掉之後才會被替換掉
      - 如果某頁經常被使用，以至於參考位元一直都是 1，那它就永遠不會被替換掉

圖 10.17　第二次機會 (時鐘) 分頁替換演算法





加強第二次機會演算法

- 把參考位元和修改位元視為一組有序對來加強第二次機會演算法
  1. (0, 0) 表示未使用過也未被修改過——最佳替換分頁
  2. (0, 1) 表示近來未被使用過但曾被修改過——沒有那麼好，因為此頁需要被寫出，才可以被替換
  3. (1, 0) 表示近來被使用但未被修改過——可能很快會被再使用到
  4. (1, 1) 表示曾被使用過且被修改過——可能會被再使用到，再替換掉此頁之前，必須把它寫出輔助儲存體
- 當需要做分頁替換時，我們使用和時脈演算法相同的技巧

10.4.6 基於計數的分頁替換

- 還有許多其它演算法可用來處理分頁替換。例如，我們可以使用一個計數器來記錄每一頁被參考過的次數，並發展出以下兩種方法
  - **最不常用使用** (least frequently used, LFU) 分頁替換演算法讓次數最少的那一頁被替換掉
  - **最常使用** (most frequently used, MFU) 分頁替換演算法認為次數最少的頁可能才被載入不久，並且將要被使用

10.4.7 分頁緩衝演算法

- 這個觀念的一種延伸就是為被修改過的分頁保存一個表
- 每當分頁用的裝置閒置之後，就選出曾經被修改過的一頁並將它寫入輔助儲存器中
- 當出現分頁錯誤的時候，我們就先檢查看看需要的那一頁是否在空白欄庫存中
  - 如果不在的話，我們就必須選出一個空白欄，並且將資料讀入其中

10.4.8 應用程序和分頁替換

- 應用程式經由作業系統的虛擬記憶體存取資料
- 一個典型的例子，一個資料庫提供自己的記憶管理和 I/O 緩衝
- 作業系統正在緩衝 I/O，而且應用程式也正在這麼做，那麼對一組 I/O 使用了兩次記憶體
- 作業系統給予特別的程式有能力使用輔助儲存器分割作為邏輯區塊的較大循序陣列，而沒有任何的檔案系統資料結構
  - 這陣列有時被稱為**原始磁碟** (raw disk)

10.5 欄的配置

- 最少量的欄數
- 配置演算法
- 全域和區域的配置
- 不一致的記憶體存取

10.5.1 最少量的欄數

- 指令本身可能會跨越兩個欄
  - 如果它的兩個運算元中的每一個都可以是間接參考，則總共需要六個欄
- 最大數目由可用實體記憶體數量定義

10.5.2 配置演算法

- 如果有 93 個欄和 5 個行程，那麼每個行程可以分到 18 個欄
  - 剩下的 3 個欄可以當作一個空白欄緩衝區的庫存
  - 這就稱為**同等分配** (equal allocation)
- 比例分配 (proportional allocation)

$$S = \sum s_i$$

  - 可用欄的總數是 m，a_i 近似約為

$$a_i = s_i / S \times m$$

- 因而傷害到低優先權的行程
  - 以其優先權，或是在其大小與優先權的結合來決定

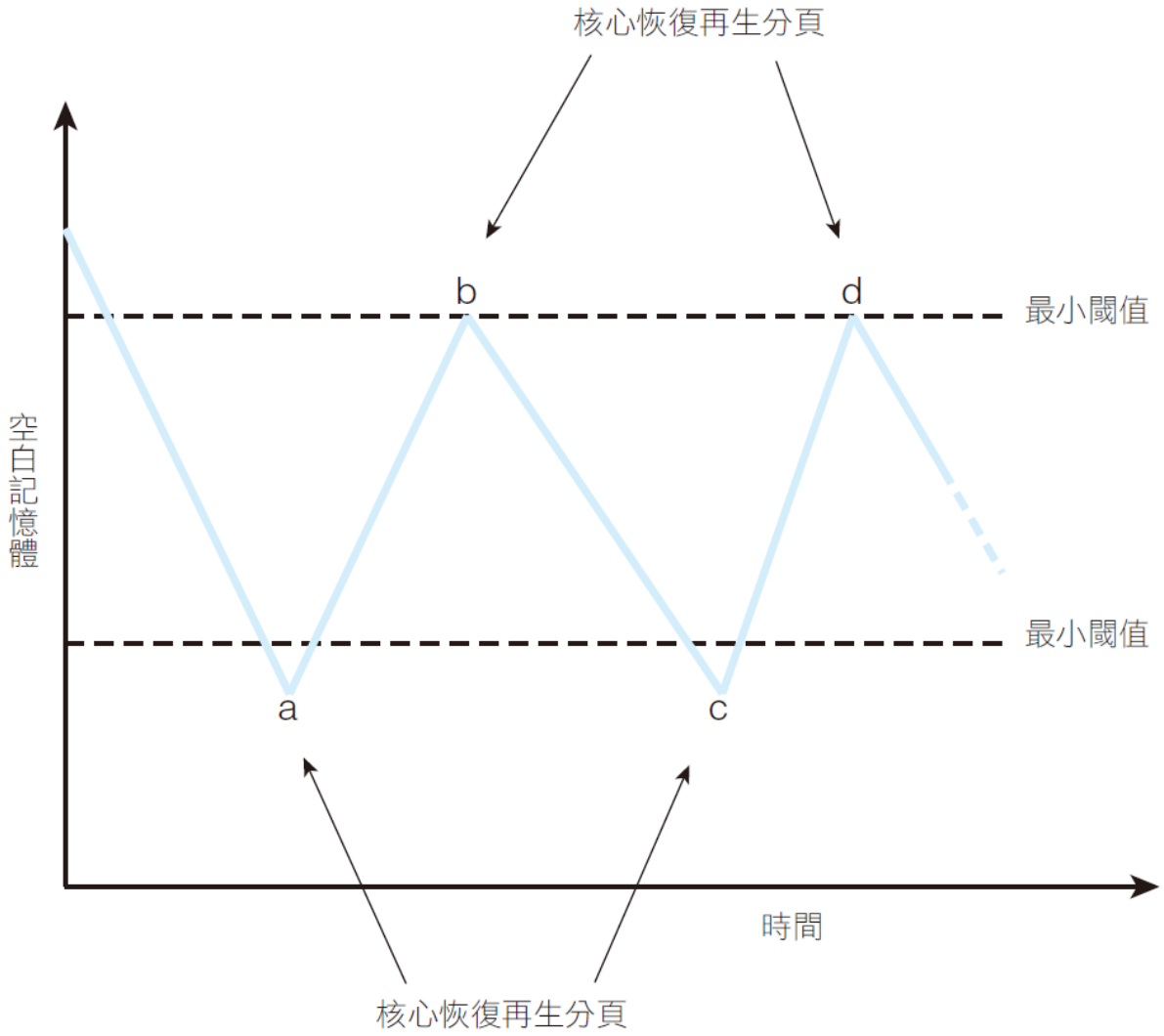
10.5.3 全域和區域的配置

- 全域替換法 (global replacement)
  - 允許一個行程從所有欄數中選出一個替換欄，即使目前該欄正分配給其它某個行程使用中
  - 一個行程可以從其它行程獲得一欄
- 區域替換法 (local replacement)
  - 要求每一行程只能從它自己選出欄
  - 有可能因為頁數不足而延遲行程的執行

再生分頁

- **收割者** (reaper)
  - 讓可使用記憶體數量都保持在最小閾值以上
    - 當它降至該閾值以下時，將觸發核心常式，該常式一開始從系統中的所有行程
- 當可用記憶體數量下降到極低水準時，稱為**記憶體不足殺手** [out-of-memory (OOM) killer] 的行程將選擇終止，進而釋放記憶體

圖 10.18　再生分頁



10.5.4 不一致的記憶體存取

- 目前為止，我們已經假設所有的主記憶體一致地被產生——或至少平等地被存取
  - 在不一致的記憶體存取 (NUMA) 系統通常在多 CPU 的系統中
  - 這些性能差異取決於系統中的 CPU 如何和記憶體互相連結。通常這類的系統是由數個系統板所組成
  - 用 CPU 是使用共享系統的互連結構的，如您預期，CPU 來存取本地記憶體比存取另一個 CPU 的本地記憶體更快

10.5.4 不一致的記憶體存取

- Solaris 藉由在核心中創造一個**地址群組** (lgroup) 入口來解決這個問題
  - 每個地址群組聚集接近的 CPU 和記憶體
  - 將 CPU 和記憶體聚集在一起，在該群組中的每個 CPU 都可以在定義的等待時間間隔內存取該群組中的任何記憶體

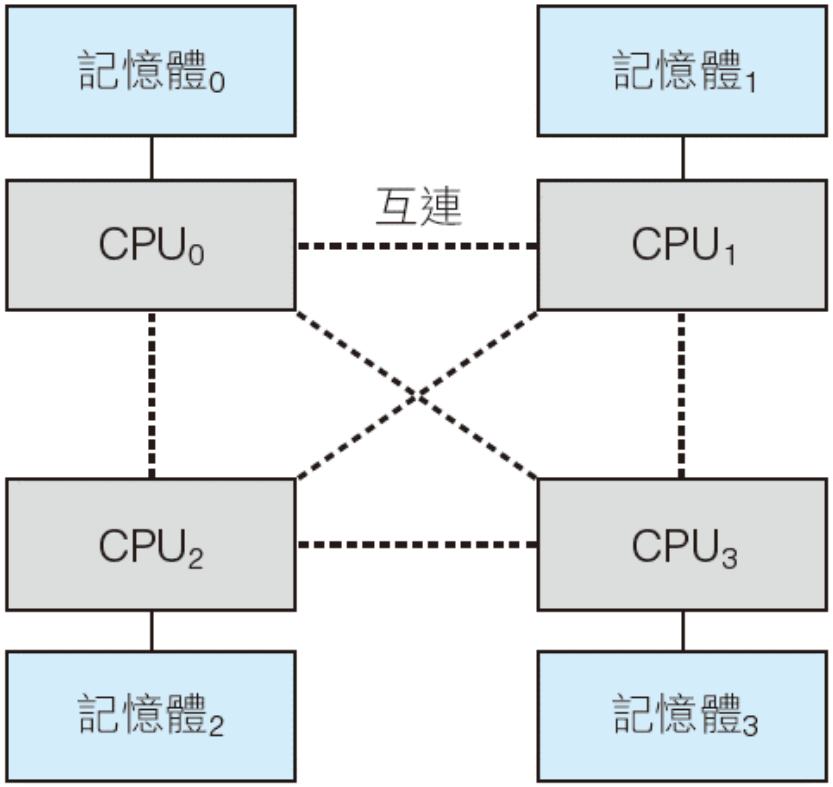


圖 10.19 非統一記憶體存取架構的多處理架構

10.6 輾轉現象

- 考慮如果行程沒有“足夠”的欄位時發生的情況，過程中將快速出現分頁錯誤
  - 即它沒有支援工作集中分頁所需的最小欄位數
- 替換某些分頁立即替換接下來的需要分頁
  - 但很快地一次又一次地出現錯誤，必須立即替換並帶回分頁

10.6.1 輾轉現象之原因

- CPU 的使用率太低了，我們就藉著引入一個新行程來增加多元程式規劃的程度
  - 這些產生錯誤的行程必須使用分頁裝置來將頁置換進來和置換出去

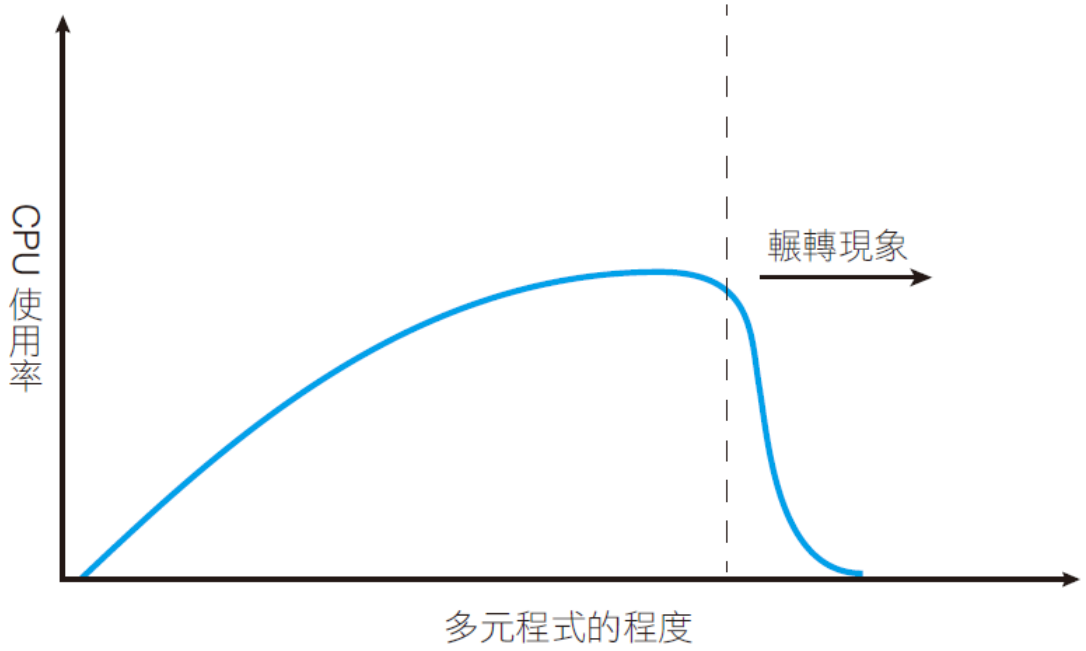
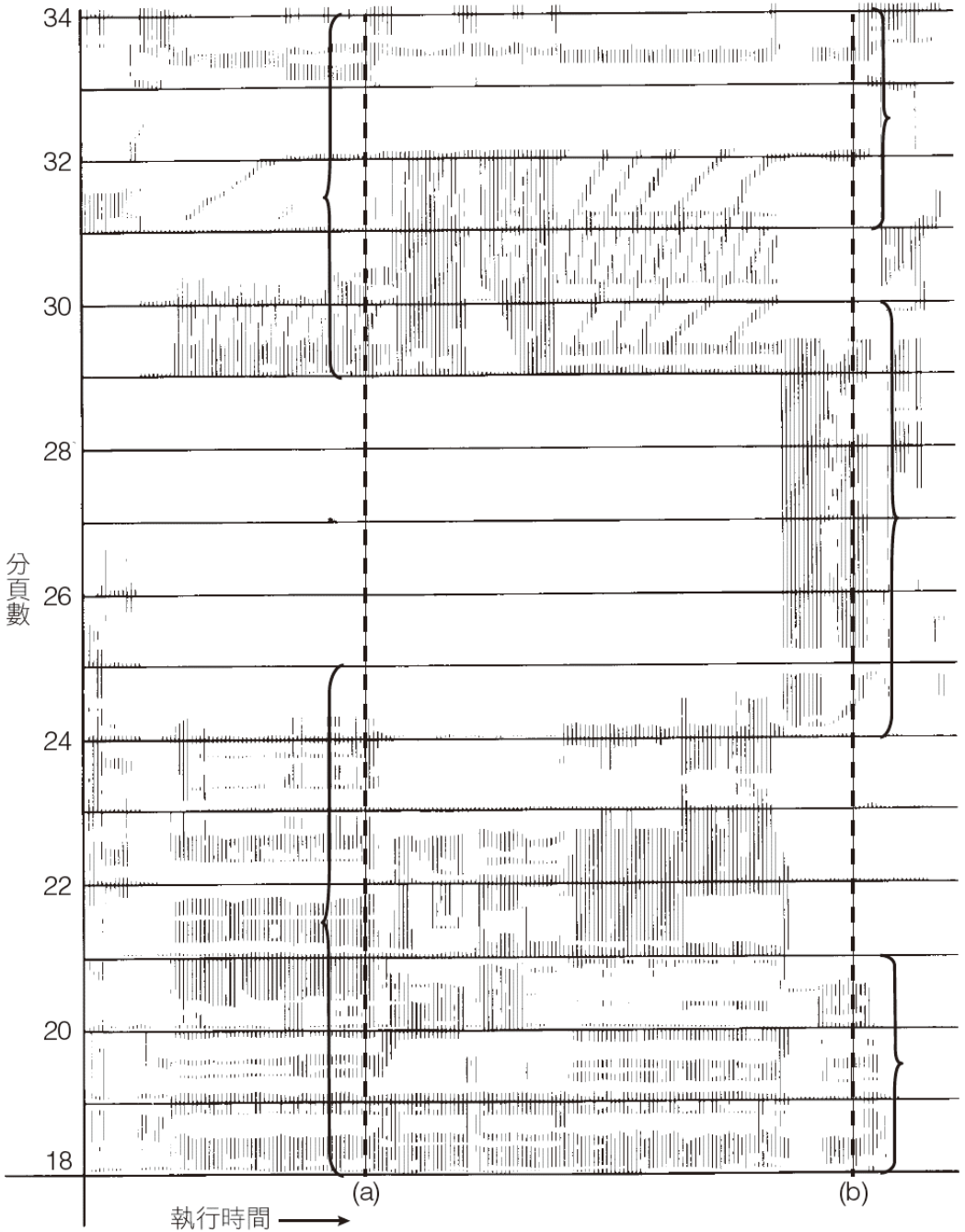


圖 10.20 輾轉現象

10.6.1 輾轉現象之原因

- 藉著使用**區域性替換演算法** (local replace algorithm)來限制輾轉現象所造成的影響
  - 或稱為**優先權替換演算法** (priority replacement algorithm)
- 為了防止輾轉現象發生，我們必須提供行程所需的一切的欄
  - 藉著觀察實際上一個行程正在使用那些頁而開始
  - 這個方法定義行程執行中的**局部區域模式** (locality model)

圖 10.21 記憶體引用模式中的局部性



10.6.2 工作集模型

- 這個模式中使用一個參數，Δ，來定義**工作集欄框** (working-set window)
  - 在最近 Δ 頁參考中的頁所組成的集合就是**工作集** (working set)
- 工作集的精確度乃是依 Δ 的選擇而定
  - 如果 Δ 太小，它將無法包含整個工作組
  - 如果 Δ 太大的話，它將會重疊許多區域
  - 在極端的情況下，如果 Δ 是無限大，那麼工作組就是整個程式了

10.6.2 工作集模型

- 工作集最重要的性質就是它的大小
  - 如果我們要為系統中的每個行程計算它們的工作集大小，WSS_i

$$D = \sum WSS_i$$

- $(D > m)$，輾轉現象將會發生

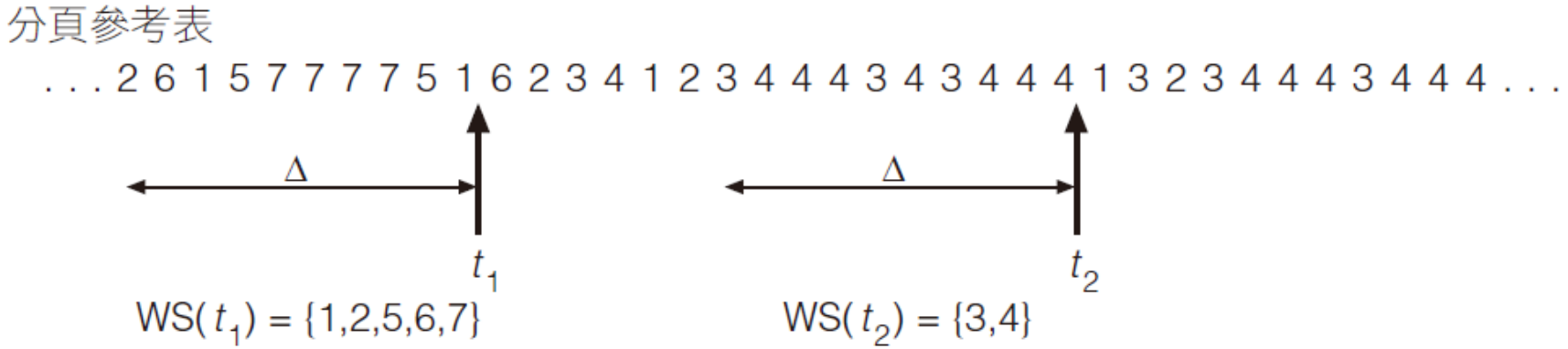


圖 10.22 工作集模型

10.6.2 工作集模型

- 一個固定的計時器中斷和一個參考位元來近似工作集的執行情形
  - 舉例來說，假設 Δ 是 10,000 次參考，並且我們可以每隔 5,000 次參考產生一次計時器中斷
- 如果它曾被使用過，那麼這些位元中至少有一個會是 1
- 如果沒有被使用過，所有的位元都將是 0
  - 那些至少有一個位元是 1 的頁將會被認為是在工作組中
- 10 個位元和中斷每 1,000 次參考

10.6.3 分頁錯誤頻率

- 當它太高的時候，我們就知道該行程需要更多的欄了
  - 同樣地，如果分頁錯誤比率太低，就表示該行程擁有太多欄了

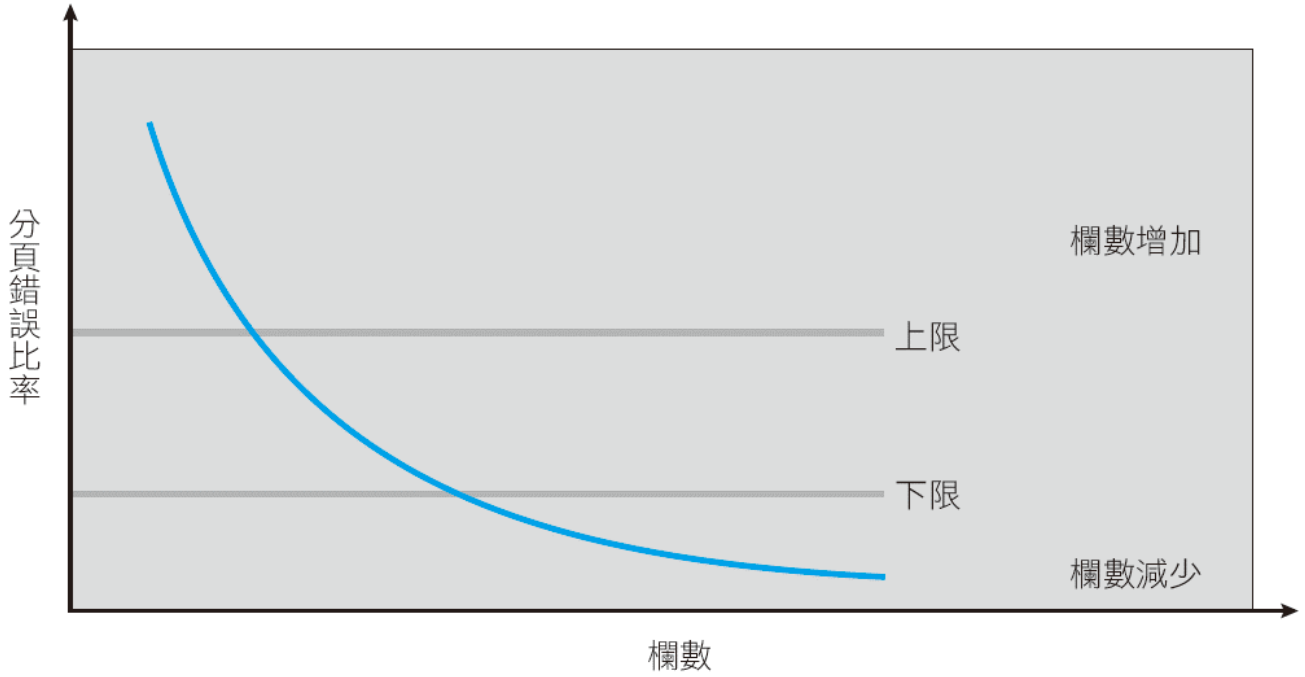


圖 10.23 分頁錯誤頻率

10.7 記憶體壓縮

- 分頁的替代方法是進行**記憶體壓縮** (memory compression)

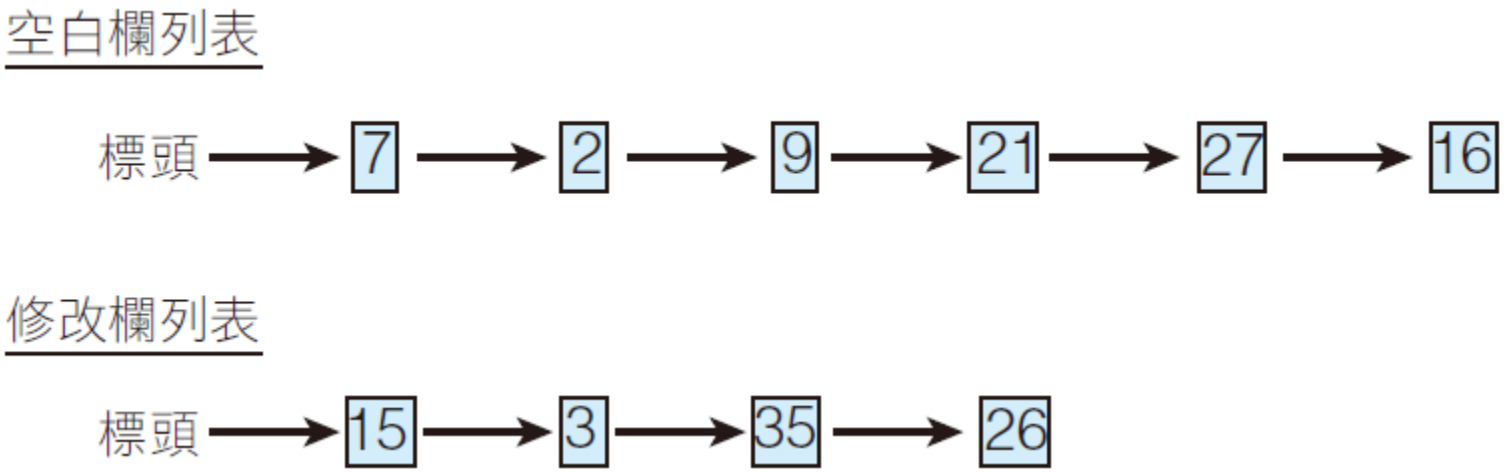


圖 10.24 壓縮前的空白欄列表

10.7 記憶體壓縮

- 對於 Windows 10，Microsoft 開發**通用型視窗平台** (Universal Windows Platform, UWP) 架構
  - 該架構為執行 Windows 10 的裝置 (包含行動裝置) 提供通用的 APP 平台
- 與任何形式的資料壓縮都一樣，在壓縮演算法的速度和可以實現的減少量 [稱為**壓縮率** (compression ratie)] 之間存在一些爭議

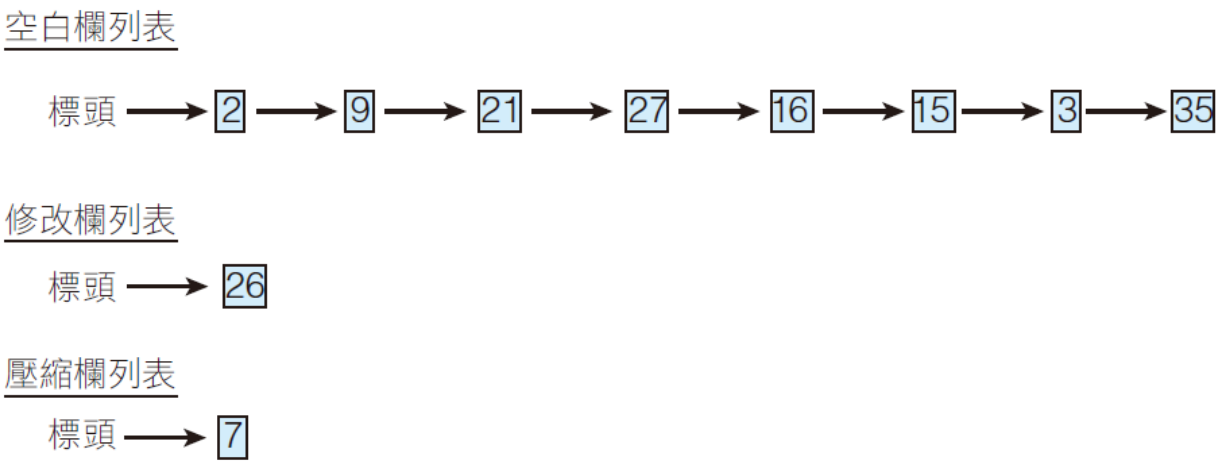


圖 10.25 壓縮後的空白欄列表

10.8 核心記憶體的配置

- 核心記憶體通常從可用記憶體池配置的，這與平常滿足使用者模式行程的串列是不同
  1. 核心為不同大小的資料結構需求記憶體
  2. 配置到使用者模式行程的分頁並不一定在連續實體記憶體中

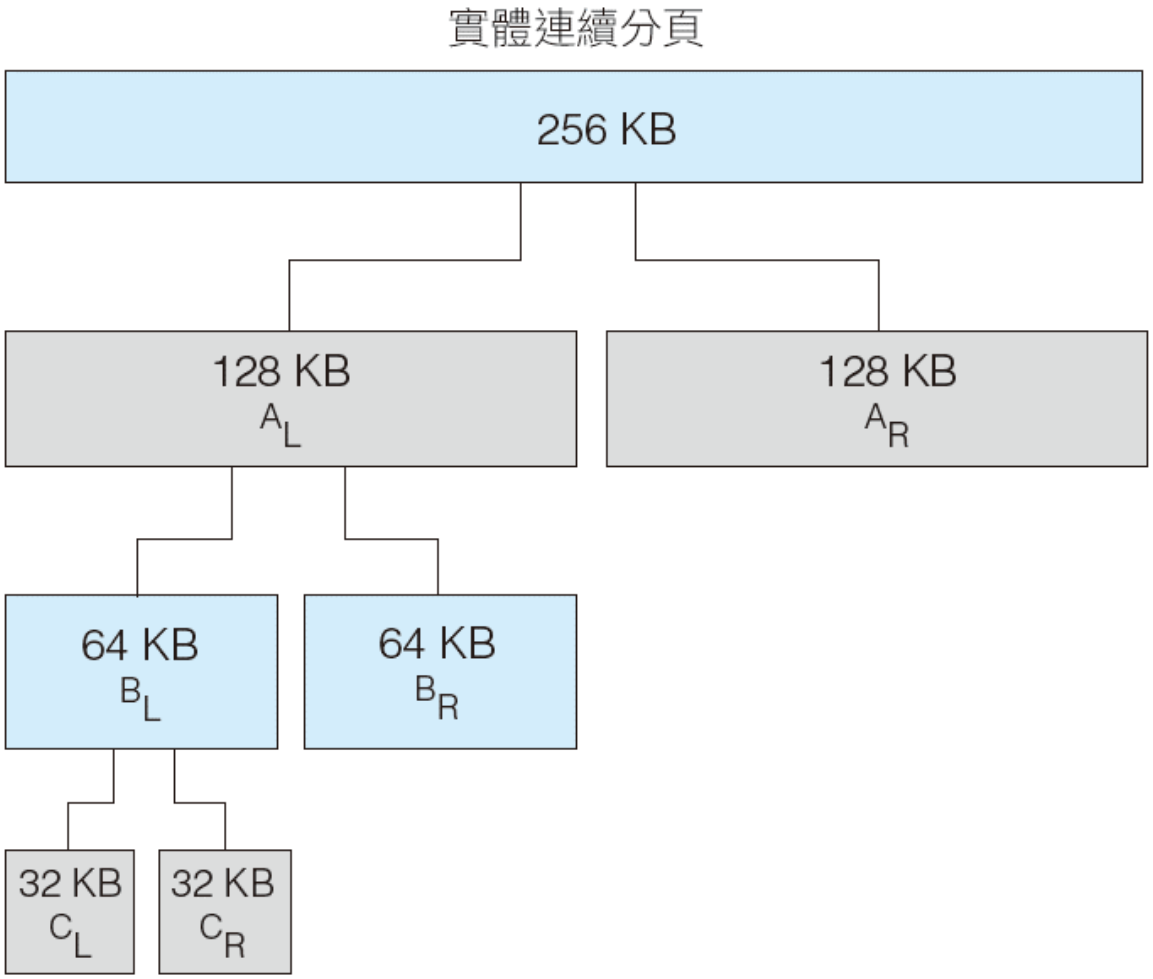
10.8.1 夥伴系統

- 夥伴系統以固定大小區段來配置記憶體
  - 記憶體使用 **2 的次方配置器** (power-of-2 allocator) 的區段來配置
  - 配置器滿足大小以 2 的次方為單位的需求 (4 KB、8 KB、16 KB 等等)
  - 如果需求的大小不適合單位，則以 2 為底，四捨五入到下一個更高的次方

10.8.1 夥伴系統

- 假定記憶體區段的大小最初是 256 KB 以及核心需求 21 KB 的記憶體
  - A_L 和 A_R —— 每個大小128 KB
  - 64 KB 的夥伴 —— B_L 和 B_R
  - 32 KB 的夥伴，C_L 和 C_R
- 優點是利用一種稱為**合併** (coalescing) 的技巧，可以將毗連的夥伴聯合形成更大的區段

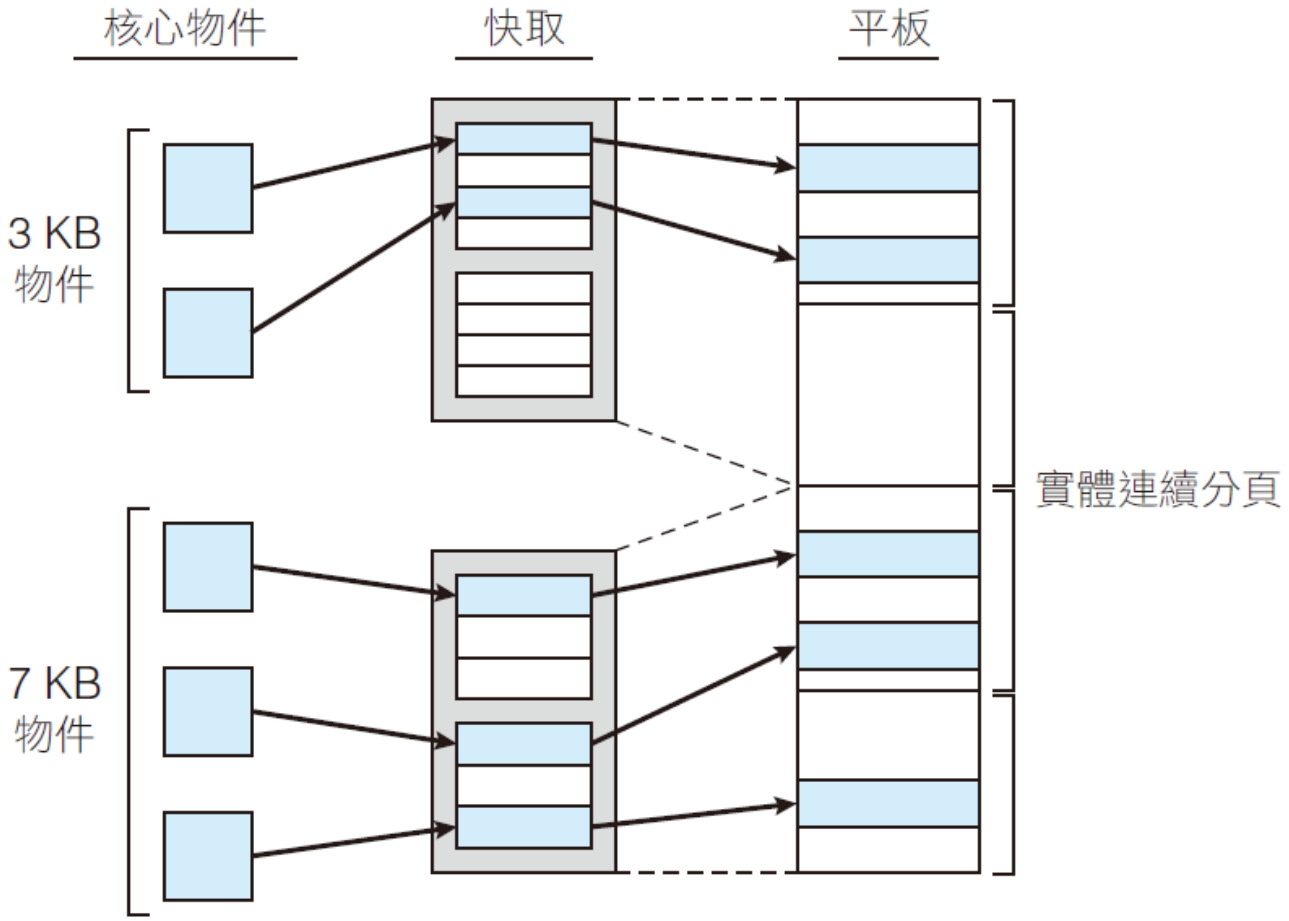
圖 10.26　夥伴系統配置



10.8.2 平板配置

- 配置核心記憶體的第二個策略稱為**平板配置** (slab allocation)
  - **平板**由一個或更多個實體連續分頁組成
  - **快取** (cache) 由一個或更多個平板所組成
- 每個快取與**物件** (object) 一起存在，此物件是快取代表的核心資料結構的實例化

圖 10.27　平板配置



10.8.2 平板配置

- Linux 系統中，一個行程描述符類型 struct task_struct，這大約需要 1.7 KB 的記憶體
  - 新的任務需要為來自快取的 struct task_struct 物件配置所需的必要記憶體
  - struct task_struct 物件已配置在平板中且被標示為可用時，快取將使用 struct task_struct 物件滿足需求
    1. 填滿。平板中所有的物件被標示成已使用
    2. 空的。平板中所有的物件被標示成閒置
    3. 部份。平板包括兩者已使用和閒置這兩種物件
  - 如果可用的物件不存在，則從空的平板配置一個可用的物件
  - 如果沒有空的平板可用，從連續實體分頁配置一個新的平板並分配到一個快取

10.8.2 平板配置

- 平板配置器首先在 Solaris 2.4 核心中出現
  - Linux 2.2 版開始
  - Linux 核心採用平板配置器
- Linux 的最近版本包含兩種其它的核心記憶體配置
  - SLOB
  - SLUB 配置器 (Linux 稱它的平板製作為 SLAB)

10.8.2 平板配置

- SLOB (它表示 Simpe List of Blocks) 藉由維護三個串列物件來工作：
  - small (少於 256 個位元組的物件)
  - medium (少於 1,024 個位元組的物件)
  - large (多於 1,024 個位元組的物件)
- SLUB 移除 SLAB 配置器為維護每一快取之物件所做的每一個 CPU 佇列

10.9 其他考慮的因素

- 預先分頁
- 分頁的大小
- TLB 範圍
- 反轉分頁表
- 程式結構
- I/O 交互上鎖和分頁上鎖

10.9.1 預先分頁

- **預先分頁** (prepaging) 就是想要防止這種高度的一開始需求分頁
- 當這個行程將要恢復執行的時候，我們就自動把它的整個工作集載回記憶體中
- 最可能出現的情形就是許多被預先分頁載入記憶體的頁根本就未被使用到
- 假設有 s 頁被預先分頁了，而其中只有 α 部份被使用到 $(0 \leq \alpha \leq 1)$
  - $s*\alpha$ 儲存分頁錯誤的代價是大於或小於處理 $s*(1-\alpha)$ 非必要頁的代價呢？
  - α 很接近0，預先分頁就輸了

10.9.2 分頁的大小

- 設計師很少有對分頁大小做選擇的機會
  - 分頁的大小
  - I/O 時間
  - 解析度 (resolution)
- 分頁的大小絕對是 2 的次方，一般範圍是在 4,096 (2^{12}) 至 4,194,304 (2^{22}) 位元組之間
- 還有許多因素必須考慮
  - 內部斷裂
  - 局部區域
  - 表格大小
  - I/O 時間

10.9.3 TLB 範圍

- TLB 的命中率 (hit ratio)
  - 回想 TLB 的命中率和使用 TLB 解決虛擬位址轉譯的百分率有關
- TLB 範圍 (TLB reach)
  - 指 TLB 可以存取的記憶體數量
  - TLB 的項數乘上分頁的大小
- 增加頁面大小或提供多個頁面大小
  - 對於某些不需要這麼大分頁大小的應用程式可能導致碎片增加
  - 或者大多數結構都提供對一個以上分頁大小的支援，並且可以讓作業系統支援這樣的配置

10.9.5 程式結構

- 128 乘以 128 的陣列

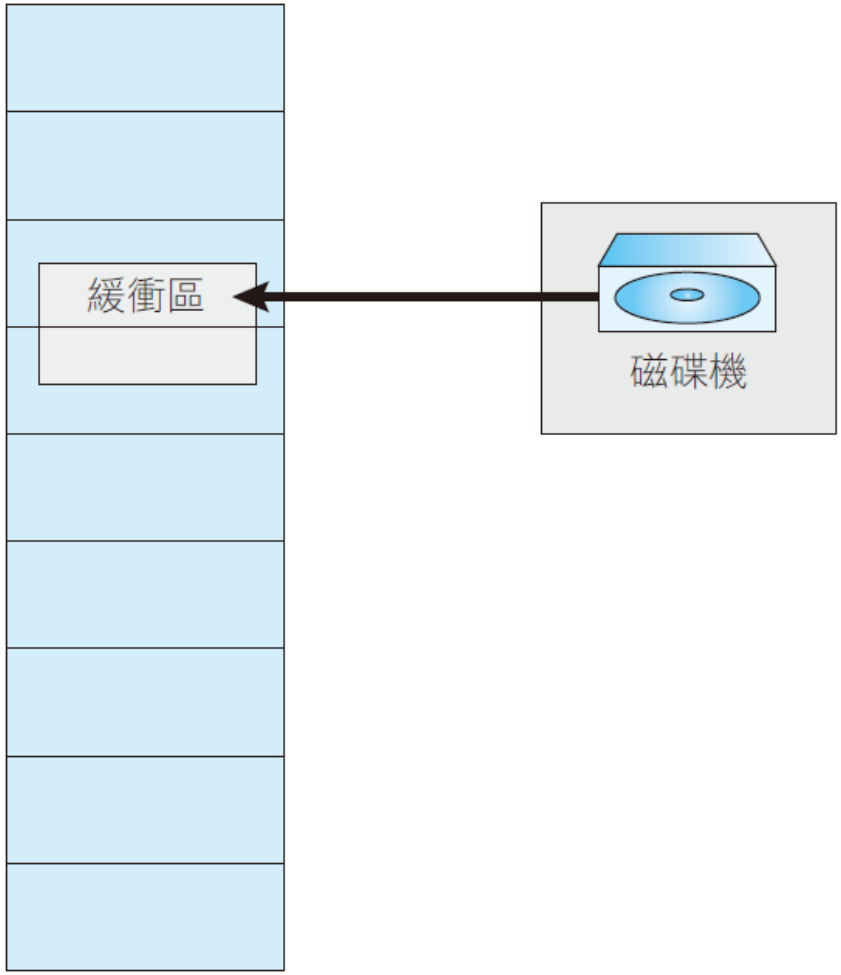
```
int I, j;
int[128][128] data;

for (j = 0; j < 128; j++)
   for (i = 0; i< 128; i++)
data[i][j] = 0;

int I,j;
int[128][128] data;

for (i = 0; i < 128; i++)
   for (j = 0; j < 128; j++)
      data[i][j] = 0;
```

- 分頁錯誤減少為128 次

10.9.6 I/O 交互上鎖和分頁上鎖

- 用需求分頁法的時候，有時候需要使它的某些分頁被**鎖** (locked) 在記憶體中
- 這些行程會造成分頁錯誤，並且使用一個全體替換演算法，它們其中之一會替換掉存放著正在等待的行程所要的記憶體緩衝區的那一頁
- **鎖定** (pinning) 記憶體分頁很常見

圖 10.28　用於 I/O 的欄必須在記憶體中的原因



10.10 作業系統的例子

- Windows
  - Windows 10 使用**叢集** (clustering) 方式的需求分頁來實作虛擬記憶體
  - **工作集最小值** (working-set minimum) 是保證行程在記憶體中具有的最小分頁數
    - 如果有足夠的記憶體可用，則可以為一個行程分配與其**工作集最大值** (working-set maximum) 一樣多的分頁
  - 與該列表相關聯的是一個閾值，該閾值可顯示是否有足夠的可用記憶體

10.10 作業系統的例子

- 如果某個行程的分頁錯誤低於其工作集最大值，則虛擬記憶體管理器會從可用的空白分頁列表中分配一個分頁

- 當空白記憶體量降至閾值以下時，虛擬記憶體管理器將使用稱為**自動工作集修整** (automatic working-set trimming) 的全域替換策略將值恢復到閾值以上的水準
  - 如果分配給行程的分頁超過其工作集最小值，則虛擬記憶體管理器將從工作集中刪除分頁

10.10 作業系統的例子

- Solaris
  - 在 Solaris，當執行緒產生分頁錯誤時，核心從它所維護的空白頁串列指定一個分頁給產生錯誤的執行緒
  - lotsfree —— 它表示開始分頁的臨界值
  - **分頁置出** (pageout)
  - 分頁置出演算法利用幾個參數來控制分頁掃描的速率 (稱為 scanrate)
    - 掃描速率用每秒的頁數來表示，介於慢掃描 (slowscan) 到快掃描 (fastscan) 的範圍

10.10 作業系統的例子

- 如果系統不能維持可用記憶體的數量在 minfree，則分頁置出行程在每一次要求新頁時就被呼叫
- 另一種加強是區分出分配給行程的分頁，和分配給一般檔案的分頁
  - 這就是所謂的**優先權分頁** (priority paging)

圖 10.30 Solaris 的分頁掃瞄器