Operating Systems
作業系統

MEMORY MANAGEMENT
記憶體管理

記憶體管理

- 背景介紹
  - 位址空間
  - 位址連結
  - 重疊
  - 置換
- 連續配置
- 分頁
- 分段
- 摘要

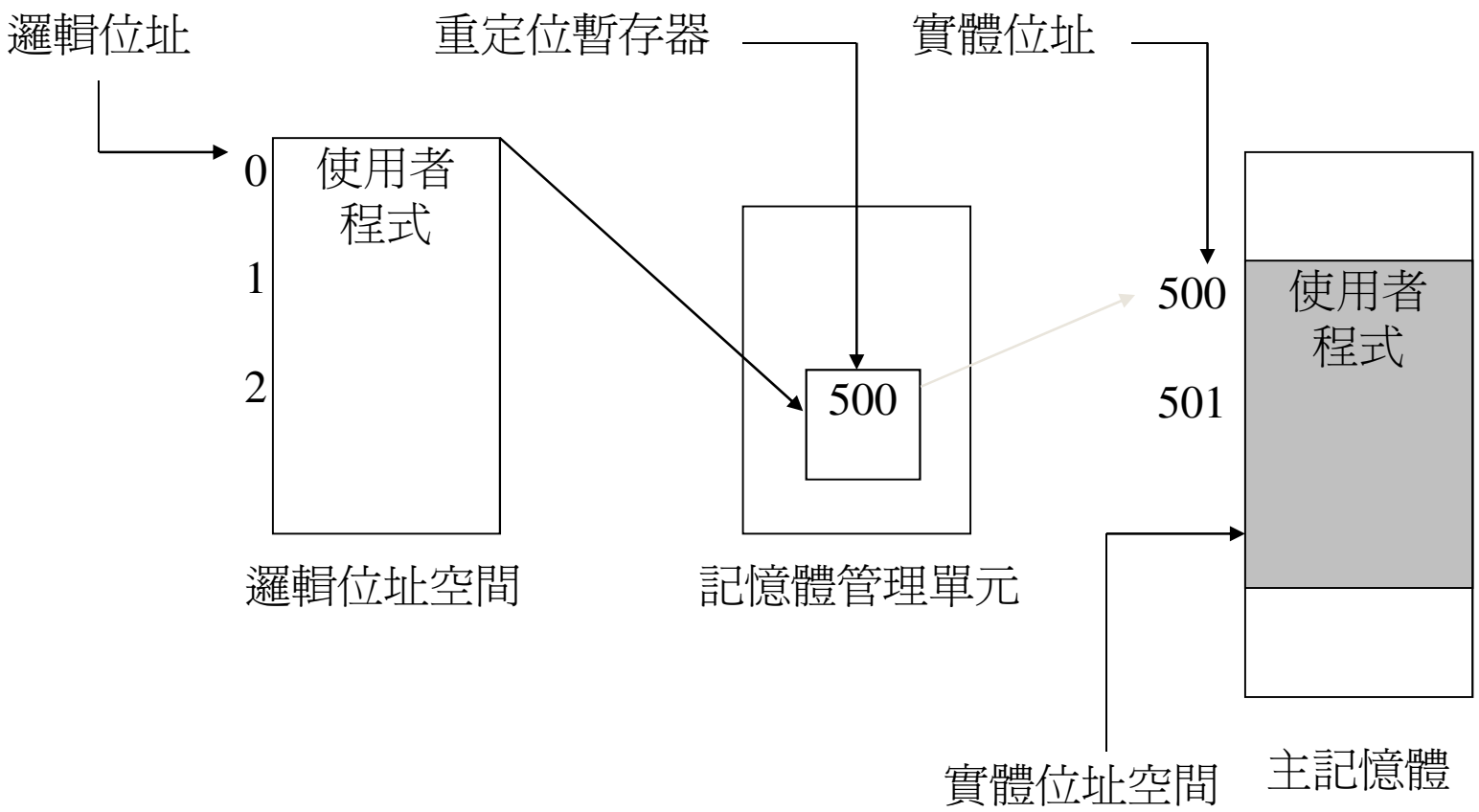
目標與趨勢

- 目標
  - 追蹤記憶體空間使用與否
  - 配置記憶體給需要的行程
  - 回收行程釋放出的記憶體
  - 有效率的置換（swapping）方法
- 趨勢
  - 程式成長的速度快於記憶體成長的速度
  - 多媒體應用環境，使用更多的記憶體

位址空間

- 記憶體位址
  - 邏輯位址，邏輯位址空間
  - 實體位址，實體位址空間
- 執行程式時，**邏輯**與**實體空間**的位址轉換
  - **載入器**（loader）：在主記憶體中尋找一塊可供使用的記憶體空間來載入程式
  - **基底暫存器**（base register）：又名重定址暫存器，存放邏輯位址轉換成實體位址的基底值
  - **記憶體管理單元**（memory management unit, MMU）：負責將邏輯位址加上基底值，以轉換成實體位址

邏輯位址空間到實體位址空間的轉換



位址連結（１）

- 當許多行程都要求將程式載入記憶體時：
  - 行程均進入**輸入佇列**
  - 依據排程器的排程結果選擇一個行程載入
  - 行程執行時，從**記憶體**取得**指令**與**資料**；執行結束後，會**釋放**所佔有的記憶體空間
- 位址轉換的步驟：
  - 原始程式中的位址：以**符號**表示
  - **編譯器**或**組譯器**：將符號所指之位址連結（binding）到一可重新定址的**相對位址**
  - 鏈結編譯器或載入器：將可重新定址的位址連結到記憶體中的**絕對實體位址**

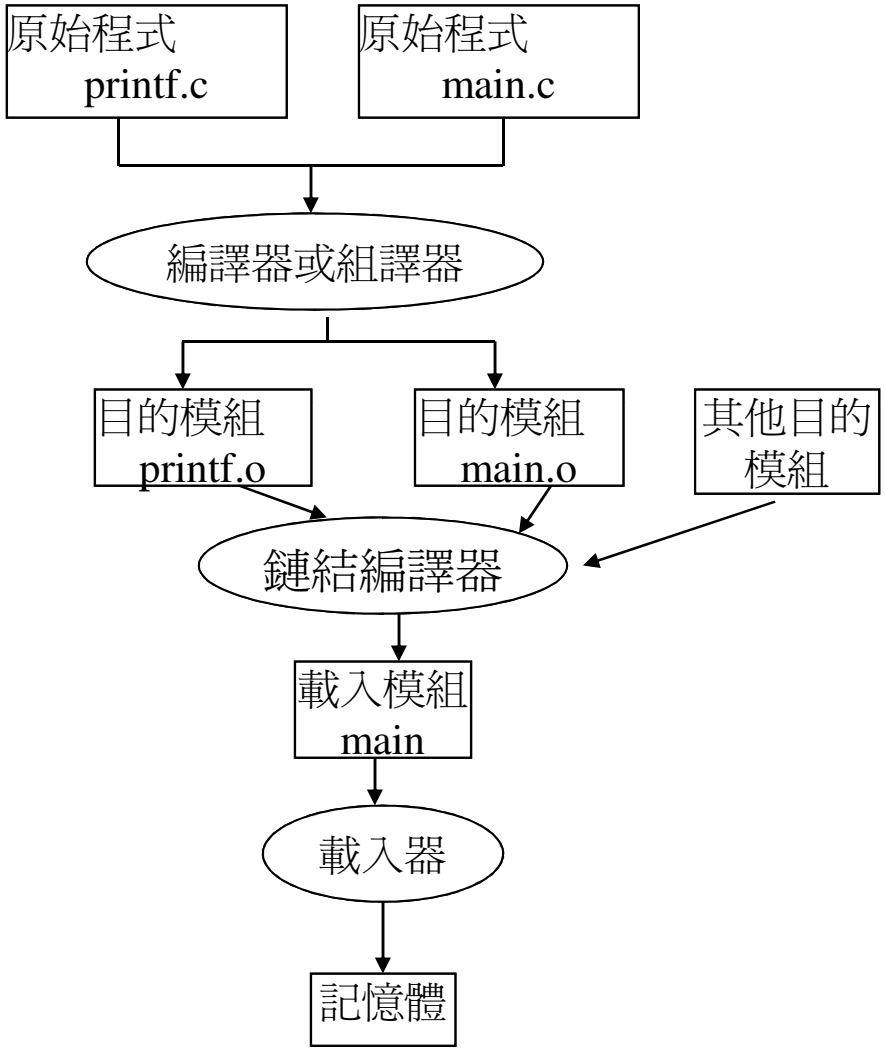
位址連結（２）

- 對於同一份資料或指令而言，所謂「位址」是隨時間而變的
- **資料**或**指令**連結到實體位址的動作可在下列任一階段完成
  - 編譯階段 (Compile time)
    - 已確定程式要在記憶體的某個位址執行
    - 當起始位址改變，程式必須重新編譯，以產生新的絕對位址的程式碼

位址連結（３）

- 載入階段 (Load time)
  - 不知道程式將在記憶體何處執行
  - 程式需編譯成可重新定址的程式碼，當起始位址改變，程式碼只需重新載入
  - 例：動態鏈結程式庫
- 執行階段 (Execution time)
  - 若在執行時，行程會從一記憶體區塊移動到另一區塊；或是內含執行時才能確定的資料型態
  - 例：大部分現代的作業系統中，動態產生行程或執行緒

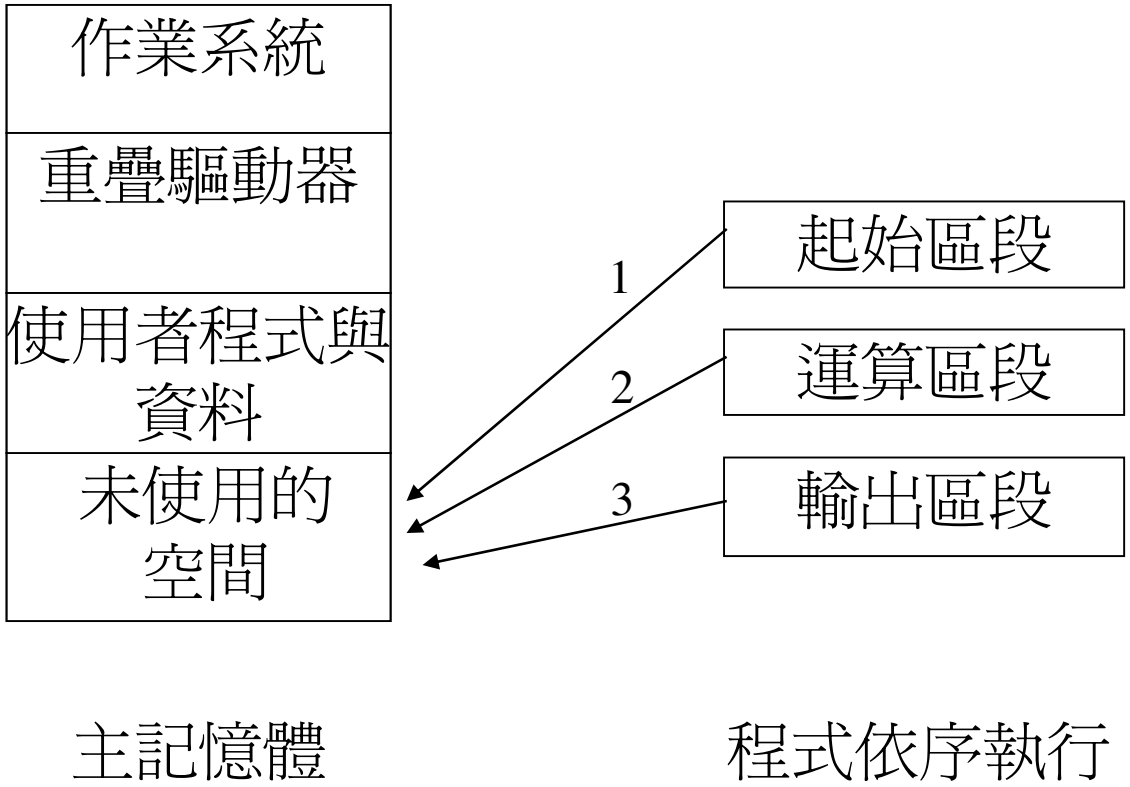
程式執行前的處理過程


原始程式
printf.c
原始程式
main.c
編譯器或組譯器
目的模組
printf.o
目的模組
main.o
其他目的
模組
鏈結編譯器
載入模組
main
載入器
記憶體

重疊 (Overlays)

- 目的：解決記憶體容量的限制
- 做法：在編譯時，將程式與資料分割成多個獨立區域；在執行時，記憶體中只保留有需要的區段
- 重疊驅動器：**載入目前要用的區段**
- 若有區段可共用記憶體，新載入的區段會覆蓋舊區段
- 多重重疊：造成程式設計師的負擔，一般會避免使用，因爲：
  - 區段分割太多→置換次數過多→降低程式執行效能
  - 區段分割太少→可重疊的程式部分過少→記憶體可能不夠，系統效能降低
- 除外：嵌入式系統（記憶體有限，沒有虛擬記憶體）

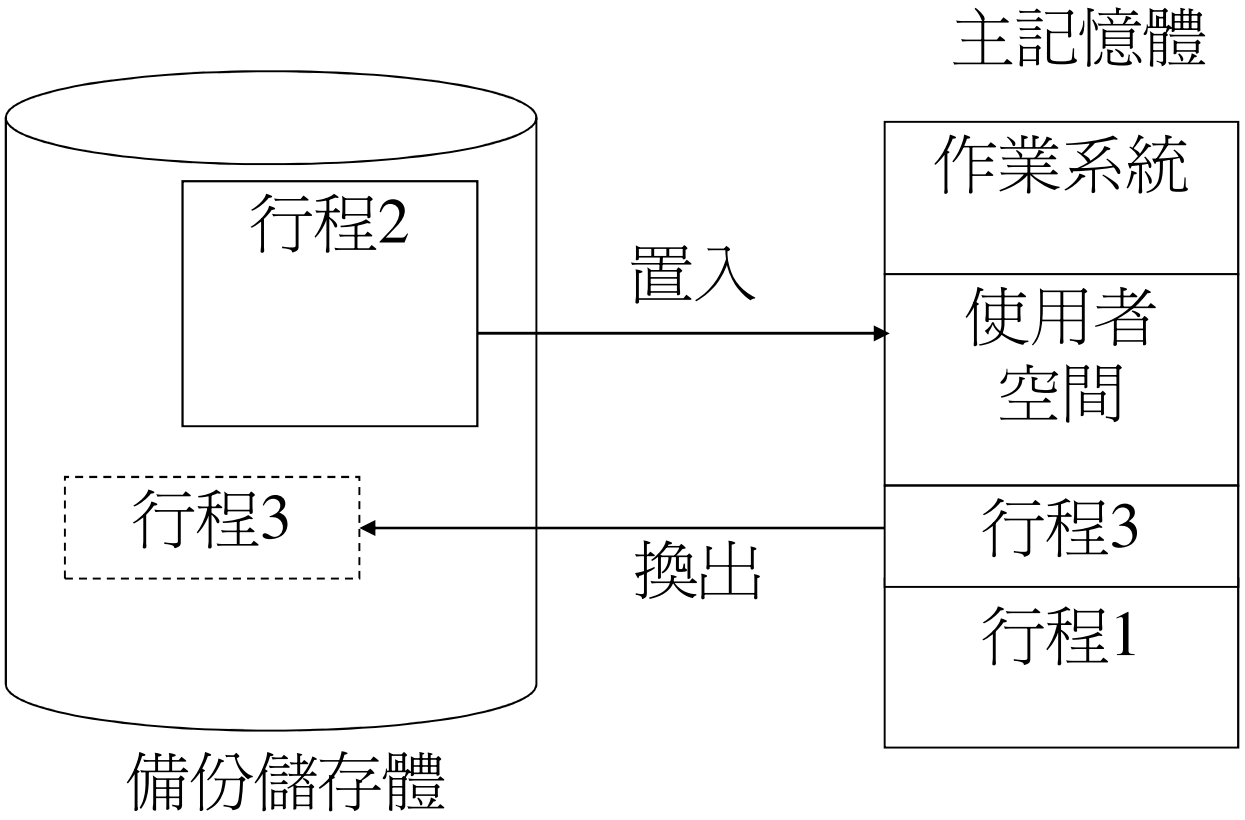
重疊 (Overlays) (Cont.)



置換（1）

- 時機：系統無足夠空間容納所有行程
  - 非執行中的行程暫時移到備份儲存體，要執行時再搬回記憶體中
  - 備份儲存體：一般而言指**磁碟**
- 換出、置入過程
  - CPU排程器決定下一個執行的行程
  - 分派程式到記憶體中尋找該行程
    - 若不存在且無足夠記憶體空間→先換出某些行程
    - 在置入該行程時，需重新載入暫存器內容，將控制權交給該行程

置換兩個行程



置換（２）

- 產生內文切換的額外負擔
  - 時間浪費在資料傳遞上
- 爲了提高效率
  - 行程必須隨時告知作業系統行程對記憶體需求的變化
  - 以便作業系統只置換實際所需的記憶體空間，節省置換所消耗的時間

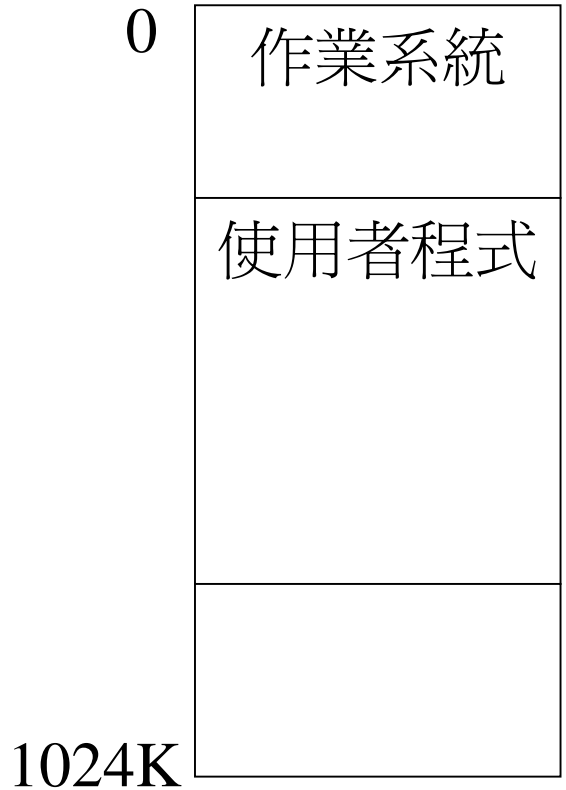
置換（３）

- 發生記憶體存取的錯誤
  - 原因：置換出不處於閒置狀態的行程（例：置換出的行程正在等待非同步的 I/O 操作）
- 解決方式：
  1. 任何企圖作I/O操作的行程不會被置換
  2. 只有進入作業系統緩衝區的行程才可作I/O操作，而作業系統與行程記憶體間的資料傳遞，只有在行程被置入時才可以進行

第八章 記憶體管理

- 背景介紹
- 連續配置
  - 單一分割配置
  - 多重分割配置
  - 斷裂
- 分頁
- 分段
- 摘要

記憶體分割



- 一部分供作業系統常駐使用
  - 放置位址常考量中斷向量的位址，因此作業系統常位於低位址
- 另一部分供使用者行程使用
- 配置方式：
  - 單一分割配置（單一使用者）
  - 多重分割配置（多元程式概念）

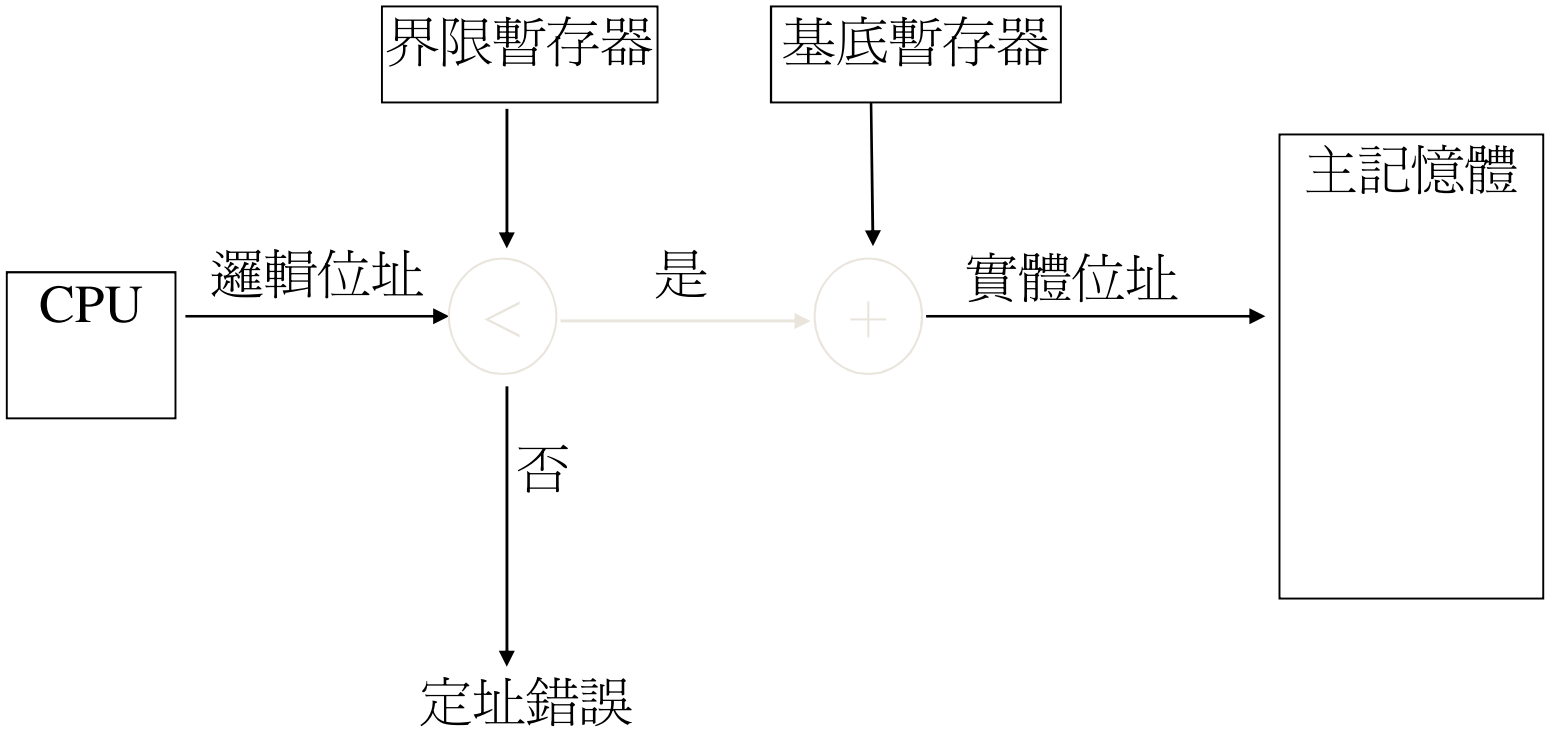
單一分割配置

- 單一使用者
- 記憶體分隔成兩部分，一用來常駐作業系統，剩餘僅供一個使用者行程執行
- 缺點：
  - 僅讓一個行程執行，造成記憶體空間的浪費
  - 若行程執行 I/O 操作， CPU 閒置，使整體系統效能降低
  - 若行程大小超過可用的記憶體空間，將導致程式無法執行（可能解決方法：重疊）

單一分割配置（２）

- 如何保護作業系統與使用者程式不會遭到對方不當修改？
  - 利用基底暫存器與界線暫存器的輔助
    - 基底暫存器：存放最小的記憶體實體位址
    - 界線暫存器：存放邏輯位址範圍
    - 每一個邏輯位址都須小於界線暫存器的邏輯位址範圍
    - 邏輯位址＋基底暫存器內的數值→記憶體的實體位址
  - CPU排程器選定一行程→分派程式將正確的值載入到基底和界線兩暫存器中→CPU每存取一次邏輯位址都經由兩暫存器的核對轉換，確保作業系統與使用者程式不會互相影響

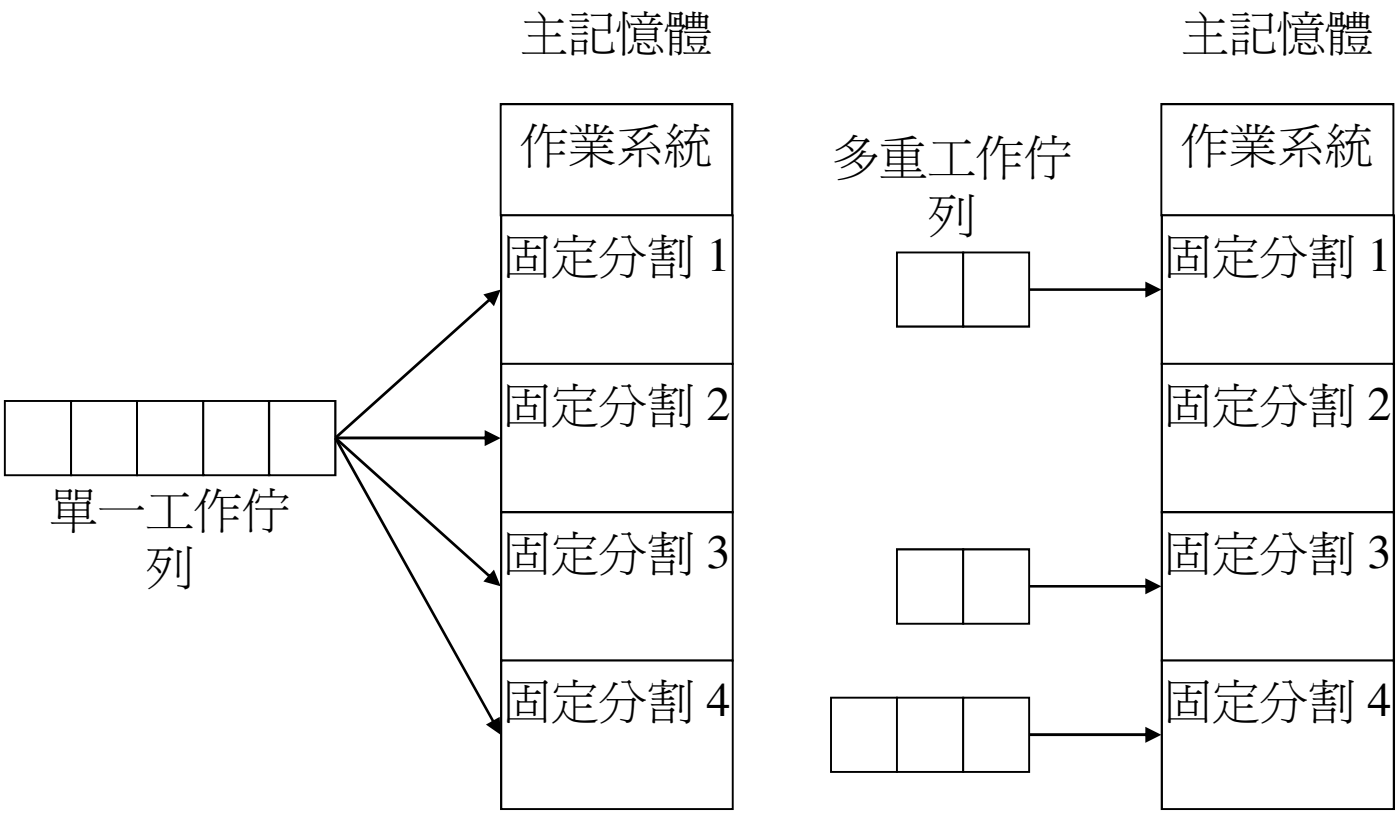
位址保護機制



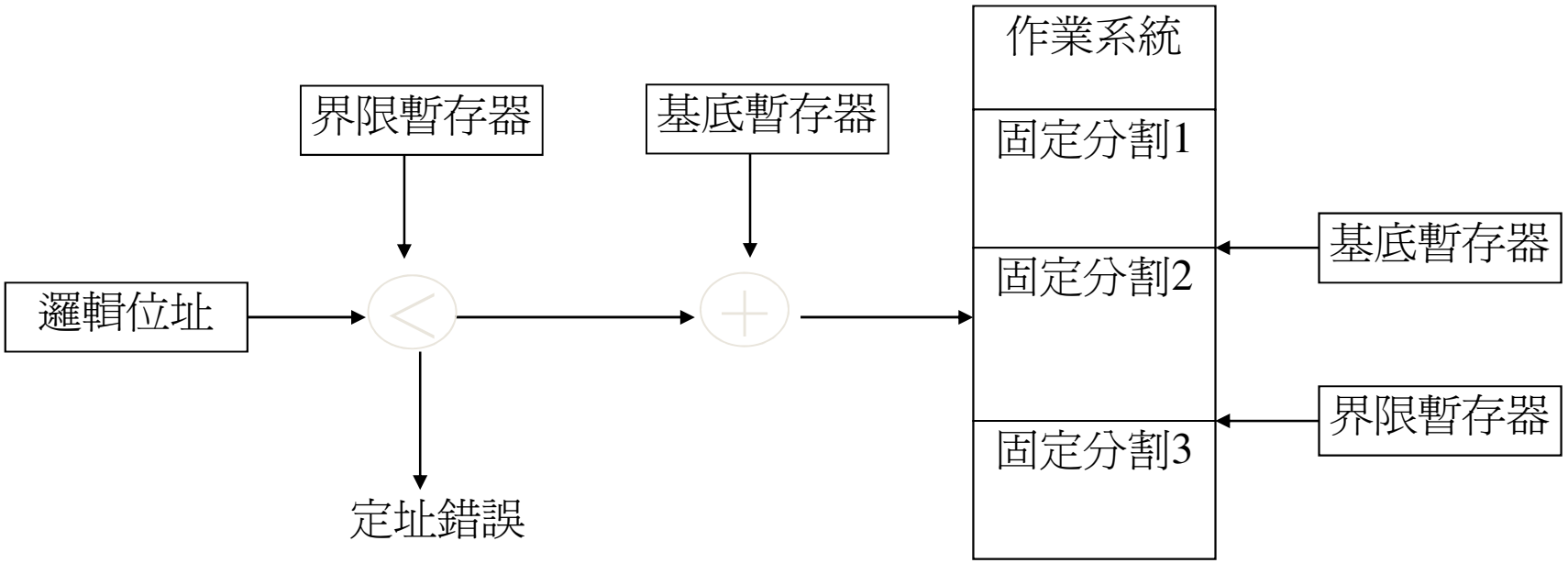
多重分割配置 (1)

- 多元程式概念
- 作業系統如何將可用的記憶體空間配置給正在輸入佇列中等待的多個使用者程式？
  - 將記憶體劃分成許多固定大小的區域或分割
  - 每個分割只能容納一個行程
  - 分割數目的多寡會影響到同時放置的使用者行程數量
- 設計輸入佇列兩方法：
  - 單一工作佇列：所有的分割都對應到同一個輸入佇列
    - 記憶體使用率較高；程式的等待時間較一致
  - 多重工作佇列：每個分割均有一個對應的輸入佇列
    - 缺點：作業系統須針對每個程式找到適合的輸入佇列，會造成記憶體中雖有可用空間卻無法使用的情況

單一與多重佇列


主記憶體
作業系統
固定分割 1
固定分割 2
固定分割 3
固定分割 4
單一工作佇列
主記憶體
多重工作佇列
作業系統
固定分割 1
固定分割 2
固定分割 3
固定分割 4

多重分割下的系統保護



多重分割配置（２）

- 記憶體劃分成固定大小的分割之問題：
  - 若程式小於分割大小的行程，會造成記憶體空間的浪費
  - 而程式大於分割大小的行程，則因爲需要跨越分割，增加系統額外的負擔
- 解決方法：動態分割法
  - 依照程式執行時的大小，在記憶體中找到夠大的可用區塊給此行程使用
  - 需在作業系統維護一個表格，隨時記錄記憶體中哪些區塊使用中、哪些區塊空閒

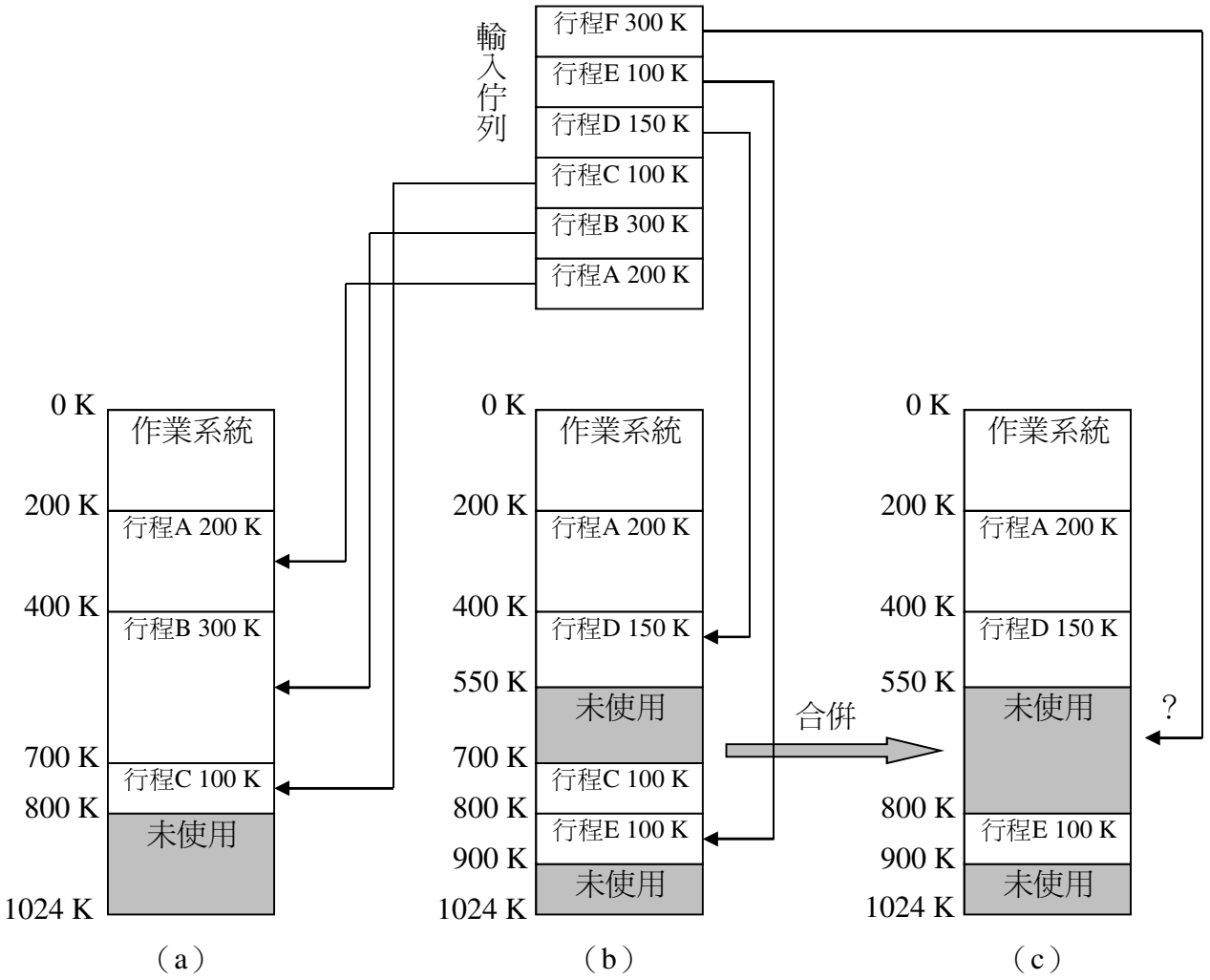
多重分割配置（3）

- 作業系統以動態分割法載入程式時，依據 3 種策略：
  - 最先符合法（First Fit）：從第一個可用的區塊開始循序找起，只要找到夠大的空間，就把程式載入。
  - 最佳符合法（Best Fit）：找到一個與載入程式大小最爲接近的區塊，再把程式載入。
  - 最差符合法（Worst Fit）：找到最大的區塊，再把程式載入。

多重分割配置（4）

- 根據電腦模擬分析
  - 最先符合法與最佳符合法在搜尋時間和記憶體空間的使用率都優於最差符合法
  - 最先符合法的執行速度通常比最佳符合法與最差符合法要快
  - 若不考慮搜尋時間，行程大小變化較大的適合最佳符合法；反之，則適合最差符合法。
- 行程執行結束後，作業系統將會
  - 釋放此行程所佔用的記憶體區塊
  - 檢查此被釋放區塊是否可與可用的相鄰區塊合併
  - 同時作業系統檢查輸入佇列中是否有程式正等待配置記憶體→如果有，則檢查此新合併的區塊大小是否夠該行程所用。

斷裂


輸入佇列
行程F 300 K
行程E 100 K
行程D 150 K
行程C 100 K
行程B 300 K
行程A 200 K
0 K
200 K
400 K
700 K
800 K
1024 K
作業系統
行程A 200 K
行程B 300 K
行程C 100 K
未使用
（a）
0 K
200 K
400 K
550 K
700 K
800 K
900 K
1024 K
作業系統
行程A 200 K
行程D 150 K
未使用
行程C 100 K
行程E 100 K
未使用
（b）
合併
0 K
200 K
400 K
550 K
800 K
900 K
1024 K
作業系統
行程A 200 K
行程D 150 K
未使用
?
行程E 100 K
未使用
（c）

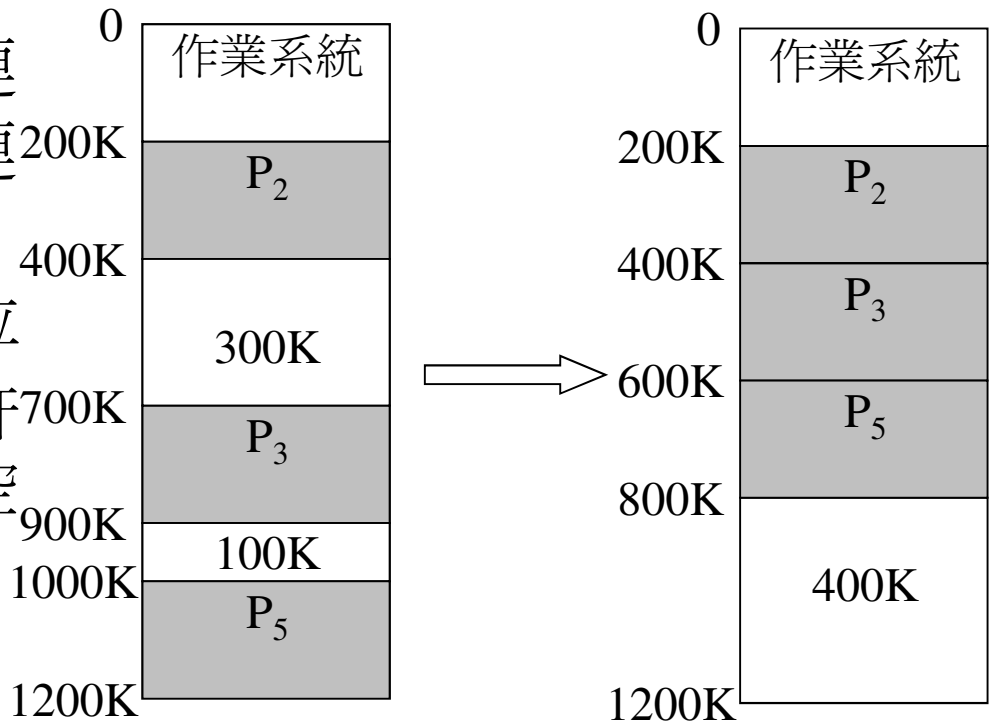
斷裂

- 有記憶體空間卻無法用（記憶體區塊太小）
- 外部斷裂
  - 因爲行程持續地被載入與置換，使得可用的記憶體空間被分割成許多不連續的區塊
  - 雖然記憶體所剩空間總和足夠讓此行程執行，卻因爲空間不連續，導致程式無法載入執行
- 內部斷裂
  - 發生在以固定分割方式配置的記憶體
  - 當一個程式載入到固定大小的分割，假如程式小於此分割，則此分割剩餘空間無法被使用
  - 不能使用的空間散佈在各個分割內，造成輸入佇列中的程式無法順利載入執行，造成浪費

內部斷裂

聚集

- 解決外部斷裂的問題
- 將記憶體中可用的不連續空間聚集成一大塊連續空間
  - 程式必須都可重新定位
  - 代價高，因爲要搬動許多行程的實體記憶體空間



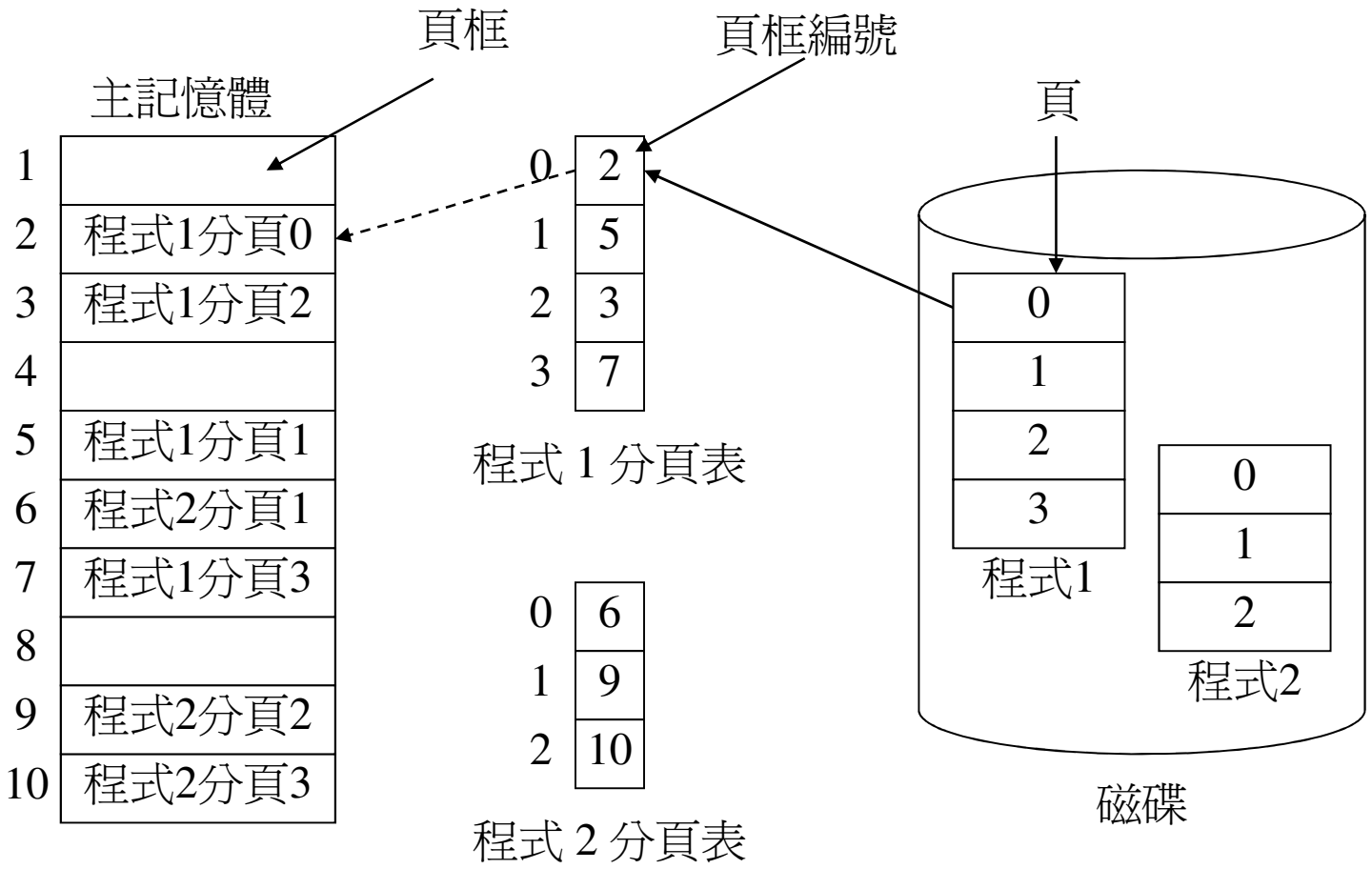
第八章 記憶體管理

- 背景介紹
- 連續配置
- 分頁
  - 基本方法
  - 分頁表的結構
  - 多層分頁法
  - 反轉分頁表
- 分段
- 摘要

基本方法（1）

- 程式可被不連續放置，沒有外部斷裂的問題
  - 將載入的程式分割成固定大小的分頁
  - 主記憶體也分割成固定大小的頁框，大小與分頁相同
  - 執行程式時，把程式所有的分頁載入記憶體任何可用的頁框中
- 每個程式有一個分頁表，存有每分頁在記憶體中的起始位址。當程式的分頁被載入到主記憶體時，程式分頁表中記錄該分頁被載入至主記憶體的哪一個頁框中
- 頁框表：作業系統須知道主記憶體中頁框使用與否、系統中總頁框數目有多少等資訊

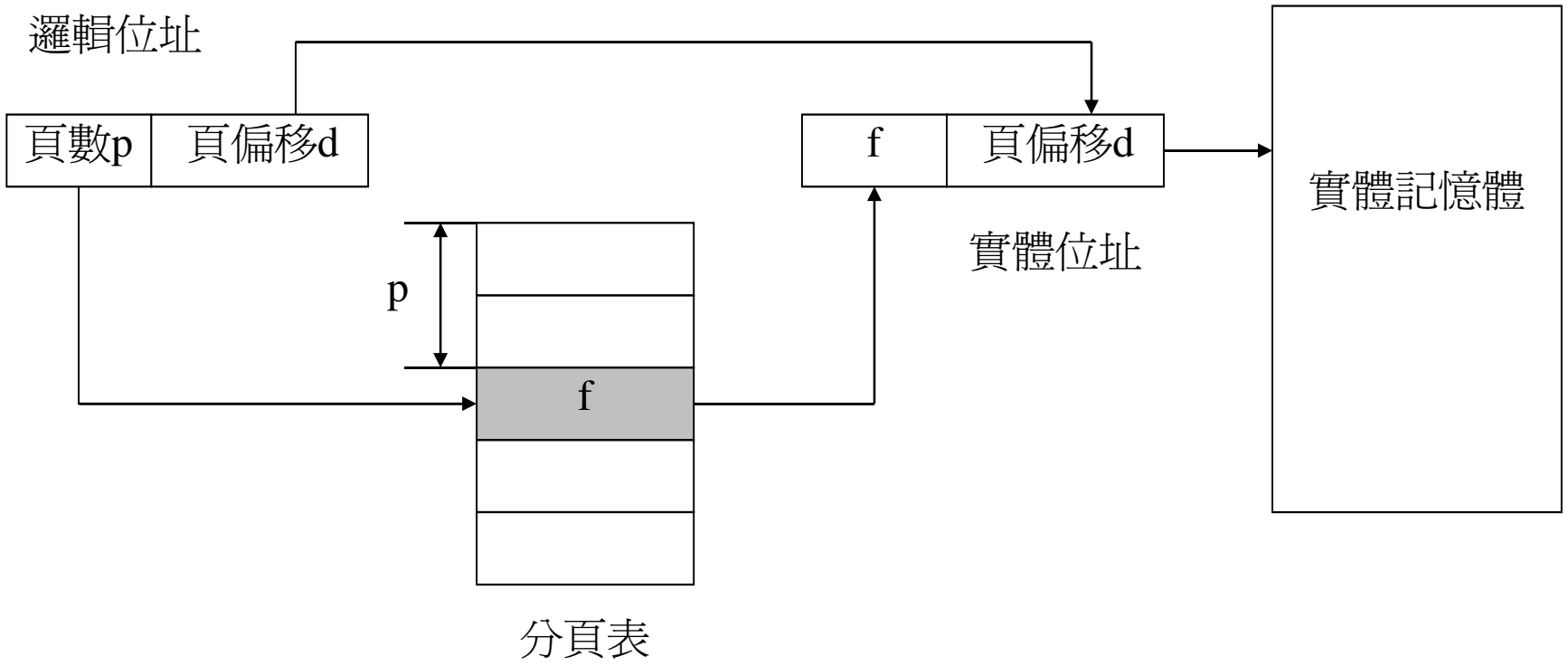
分頁法


頁框
頁框編號
主記憶體
頁
1
2 程式1分頁0
3 程式1分頁2
4
5 程式1分頁1
6 程式2分頁1
7 程式1分頁3
8
9 程式2分頁2
10 程式2分頁3
0 2
1 5
2 3
3 7
程式 1 分頁表
0 6
1 9
2 10
程式 2 分頁表
0
1
2
3
程式1
0
1
2
程式2
磁碟

基本方法（２）

- CPU 產生的邏輯位址分成兩部分：
  - 分頁碼爲指向分頁表的索引
  - 頁偏移表示與該分頁起始位址的距離
- 在分頁表中找到此分頁在實體記憶體中對應的基底位址後，再和頁偏移組合定義出實體記憶體位址
- 分頁法仍有內部斷裂的問題
  - 如果一個程式所需要的記憶體大小不能被頁框大小整除，最後一個頁框就不會全部被使用而形成內部斷裂
  - 若使分頁大小縮小，就可節省因內部斷裂而產生記憶體空間浪費；但須更大空間儲存分頁表（分頁的數量增加）
  - 每個程式平均會有 1/2 分頁的內部斷裂

分頁邏輯位址空間與實體位址空間的轉換



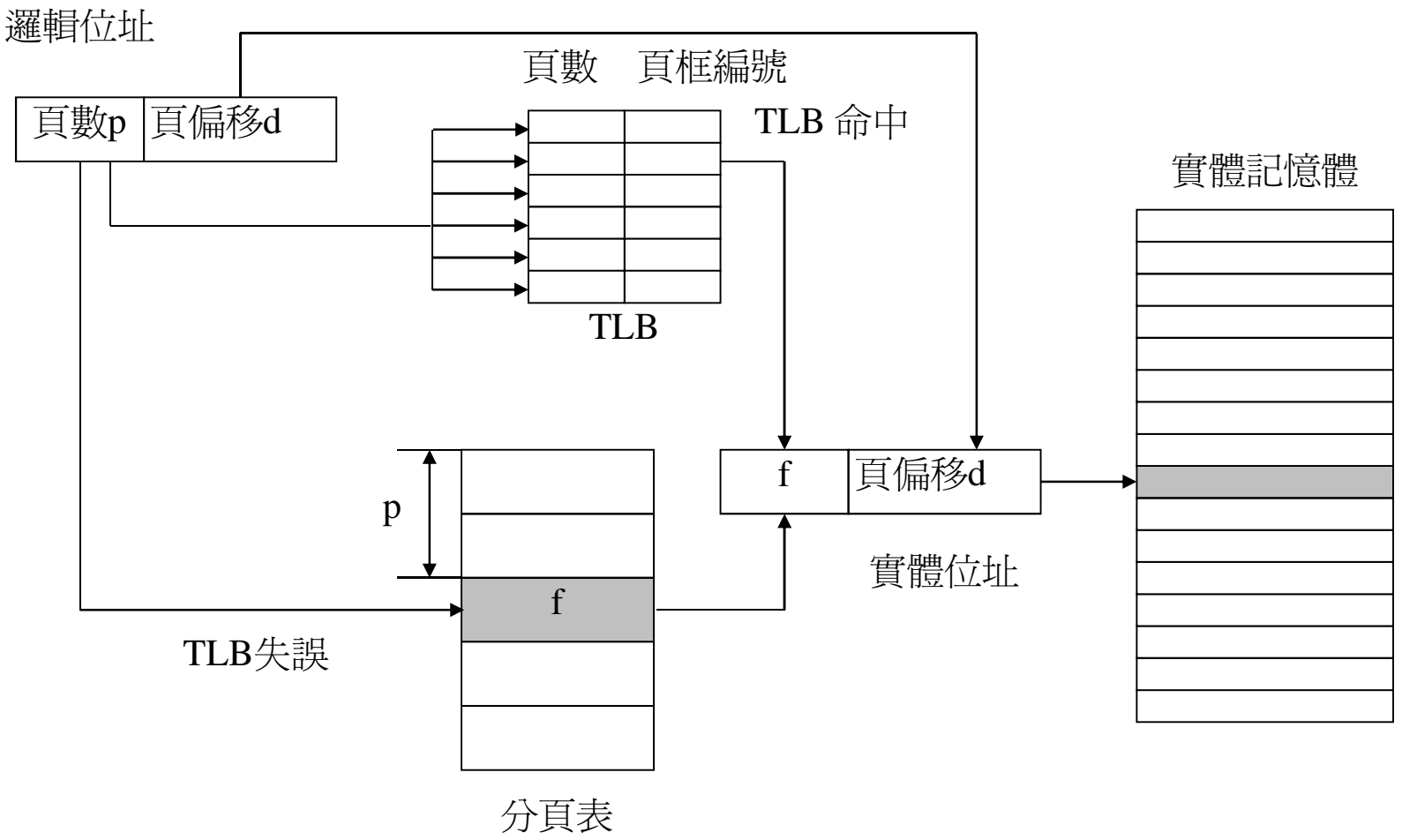
分頁表的結構（１）

- 儲存分頁表：若儲存在主記憶體中，系統需作兩次記憶體存取，效率低
- 解決方法：將分頁表放在關聯式記憶體（也稱位址查閱緩衝）中
  - 每筆資料有分頁碼，頁框編號兩欄位。關聯式記憶體以分頁碼爲索引，平行地在相對的頁框中找尋
  - 很短的時間就可找尋到，時間複雜度爲 O(1)

分頁表的結構（2）

- 實際上的做法（因關聯式記憶體昂貴）：使用主記憶體建立分頁表，將關聯式記憶體當成快取記憶體，只保存分頁表的部分內容
- 若頁框編號在關聯式記憶體中找得到，就直接與頁偏移相加得到實體記憶體位址，否則再到分頁表中找尋
- 在更新關聯式記憶體時，如果關聯式記憶體已經存滿，則系統必須置換掉其中一筆記錄（參考第 9 章的分頁置換規則）

使用 TLB 硬體支援的分頁法



分頁表的結構（３）

- 使用關聯式記憶體來儲存分頁表的效率：
  - 如果要尋找的分頁碼已經在關聯式記憶體中，則稱爲命中，否則稱爲失誤
  - 命中率的定義爲〔命中次數 /（ 命中次數 + 失誤次數）〕× 100%
  - 例：假設存取關聯式記憶體的時間爲 20 奈秒，直接存取主記憶體的時間爲 100 奈秒。假設在關聯式記憶體內命中率爲 95%，則所需的時間爲：

分頁表的結構（4）

- 0.95 × 120 ns + 0.05 × 220 ns = 125 ns
- 其中 120 奈秒的 20 奈秒花費在關聯式記憶體中找尋（命中），100 奈秒花費在存取主記憶體的資料；而 220 奈秒中的 20 奈秒花費在關聯式記憶體中找尋（失誤），100 奈秒花費在主記憶體中找尋，另外 100 奈秒花費在存取主記憶體的資料

- 適當地使用關聯式記憶體，可以降低主記憶體的存取時間，也可以節省成本

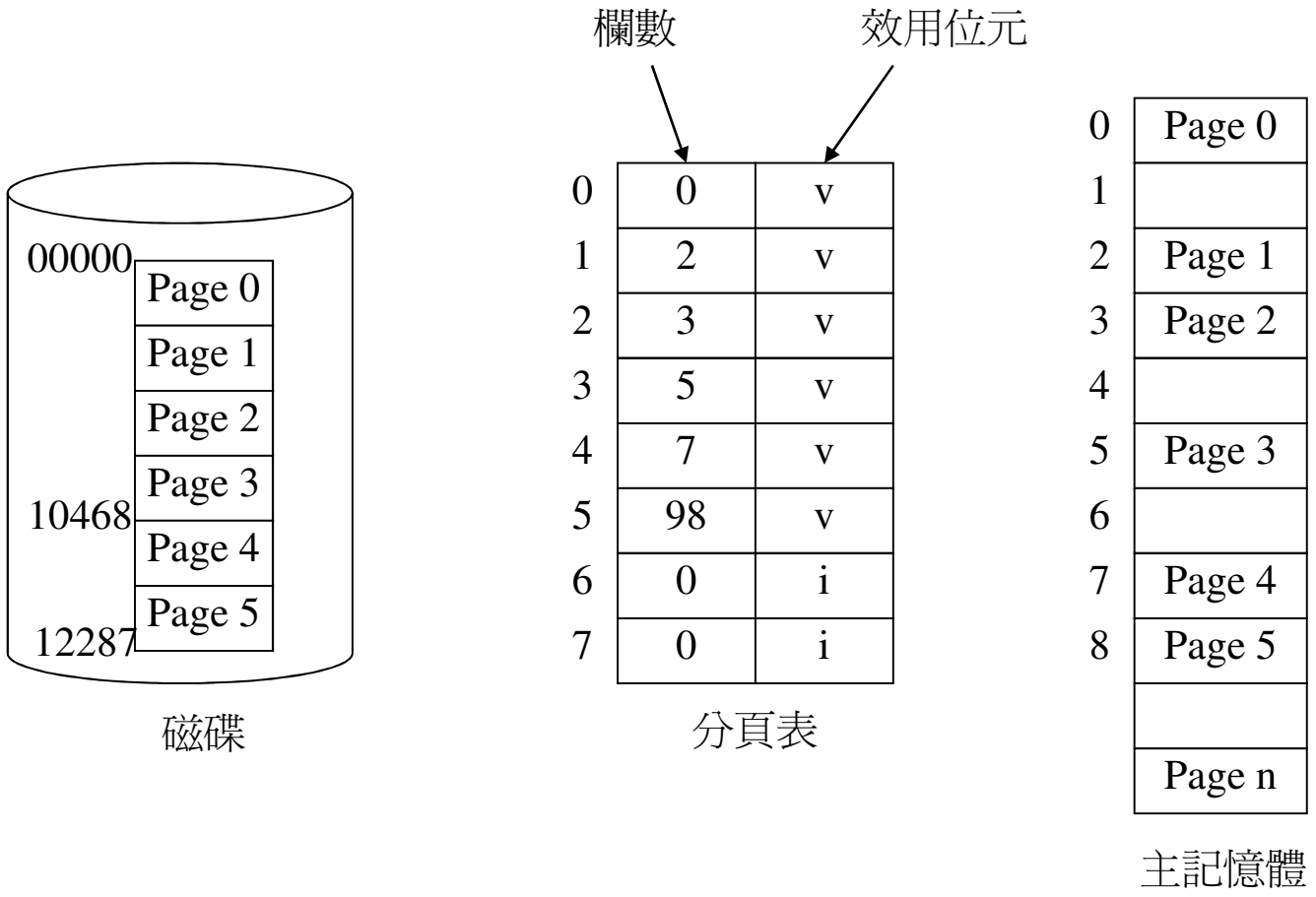
分頁表的結構（５）

- 分頁的環境中，記憶體的保護可以使用分頁上面的保護位元來完成（通常保存在分頁表）
- 保護位元可以定義某一分頁是可讀、可寫或者兩者皆可
- 每次的位址轉換均會去參考相對應的保護位元；企圖以非保護位元所提供的動作來存取此分頁，將會引發例外中斷

分頁表的結構（6）

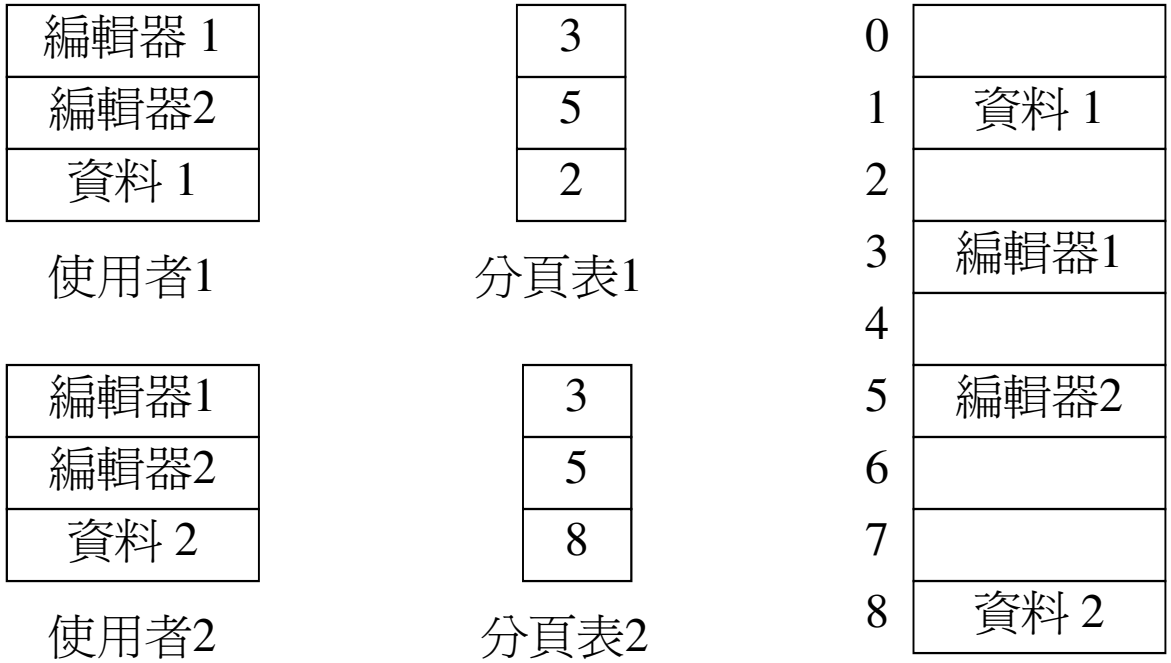
- 效用位元：
  - 當此位元被設定爲有效（v）時，表示所對應的分頁目前在主記憶體中；如果此位元被設定成無效（i），則表示所對應的分頁在輔助記憶體中
  - 作業系統會設定每分頁的有效位元，以核對該分頁的存取動作；當系統進行位址轉換時，也會去檢查此位元，若企圖存取無效的分頁也會引發例外中斷

分頁表中的效用位元


欄數
效用位元
00000
Page 0
Page 1
Page 2
Page 3
10468
Page 4
Page 5
12287
磁碟
0 0 v
1 2 v
2 3 v
3 5 v
4 7 v
5 98 v
6 0 i
7 0 i
分頁表
0 Page 0
1
2 Page 1
3 Page 2
4
5 Page 3
6
7 Page 4
8 Page 5
Page n
主記憶體

兩個使用者共用頁框

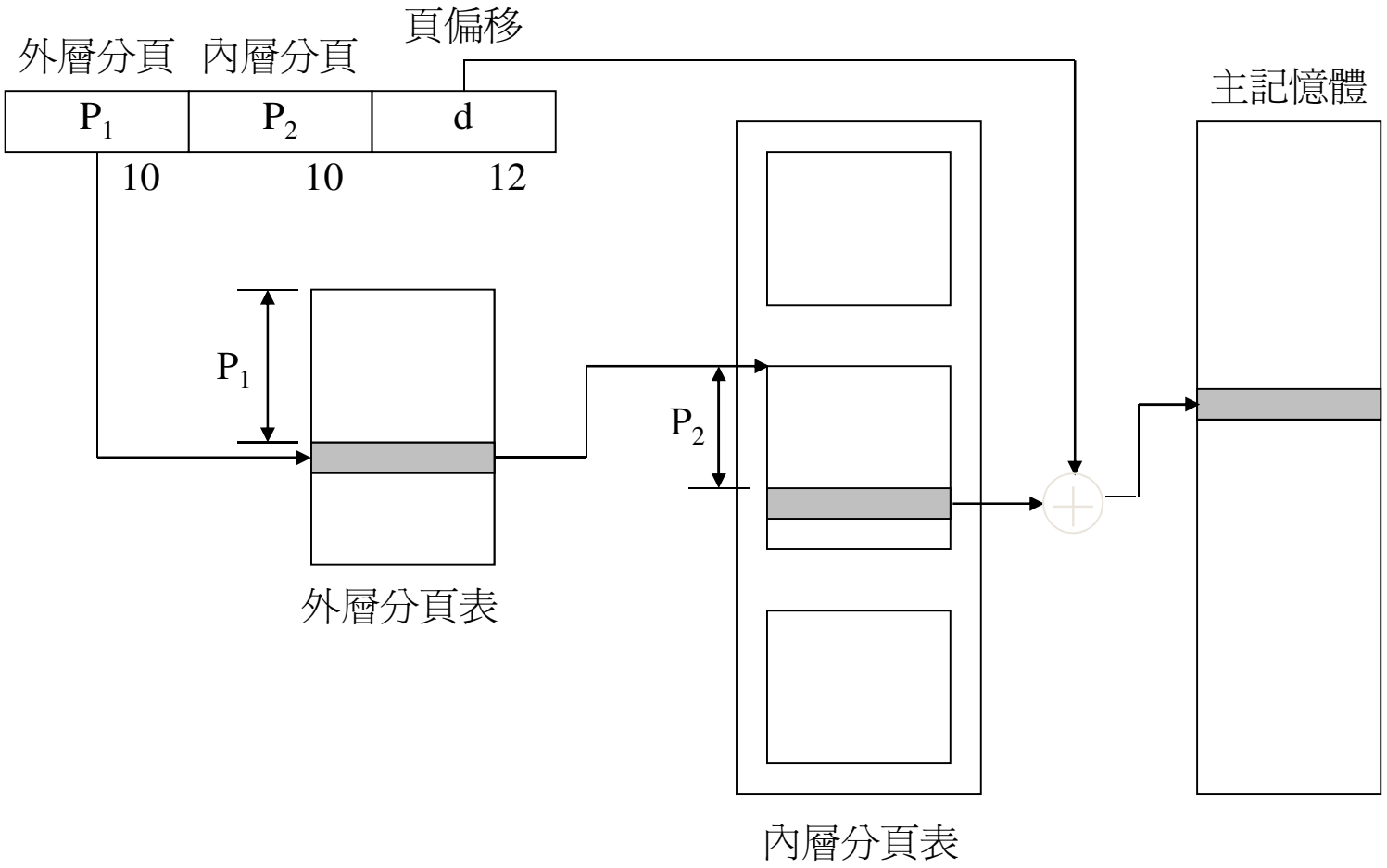
- 使用分頁法另一項好處
  - 簡單達到程式碼共用



多層分頁法（1）

- 不希望分頁表佔用連續的記憶體空間→想把分頁表分成較小的單位儲存
- 多層式分頁將分頁表也進行分頁
  - 以兩層式的分頁來說，在由邏輯位址轉換成實體位址時；
  - 先以 P_1 爲索引到外層分頁表中找尋，所找到的記錄會指向一個相對應的內層分頁表
  - 然後再以 P_2 爲索引到該內層分頁表中找尋，所找到的記錄會指向一個頁框
  - 最後和頁偏移 d 相加成爲實體記憶體位址

兩層式分頁法



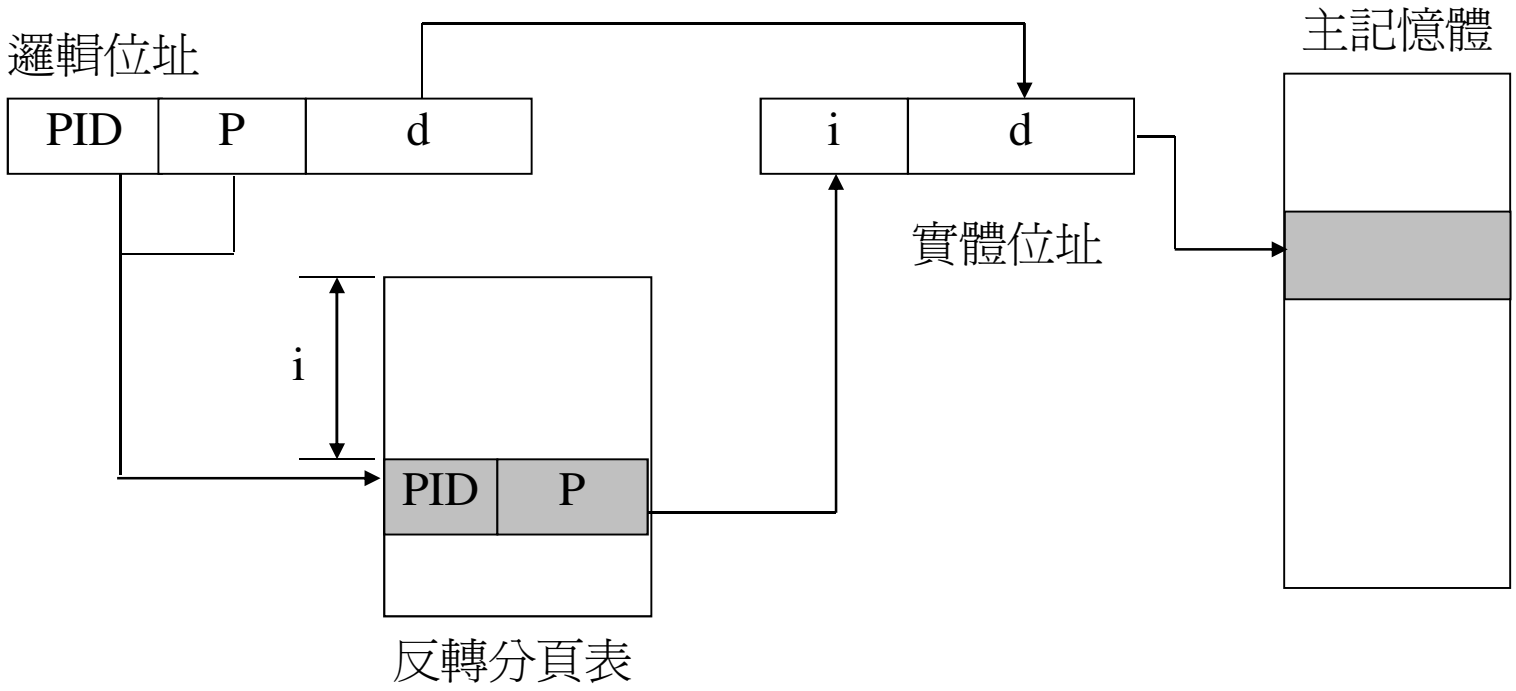
多層分頁法（２）

- 64 位元的架構不適合階層式的分頁方式（外層分頁表將佔用太大的連續記憶體空間）
- 對於邏輯位址大於 32 位元的硬體架構，可使用雜湊分頁表
  - 利用雜湊函數將邏輯位址對應到實體位址
  - 可能發生雜湊碰撞而降低效率，但是可有效減少分頁表空間

反轉分頁表（1）

- 解決分頁表空間太大的問題：整個系統僅用一個分頁表
  - 行程辨識碼加上邏輯分頁與實體頁框是一對一的對應
  - 反轉分頁表的記錄數目要與頁框一致
- 反轉分頁表的架構下
  - 一個邏輯位址包含三個欄位：行程辨識碼（PID）、分頁碼、與頁偏移
  - 反轉分頁表中每個記錄包含：行程辨識碼與分頁碼兩欄位
  - 當一個行程要進行位址轉換，必須以 PID 與分頁碼當索引，到反轉分頁表中找尋所屬的頁框編號，再與頁偏移相加就可以得到實體位址

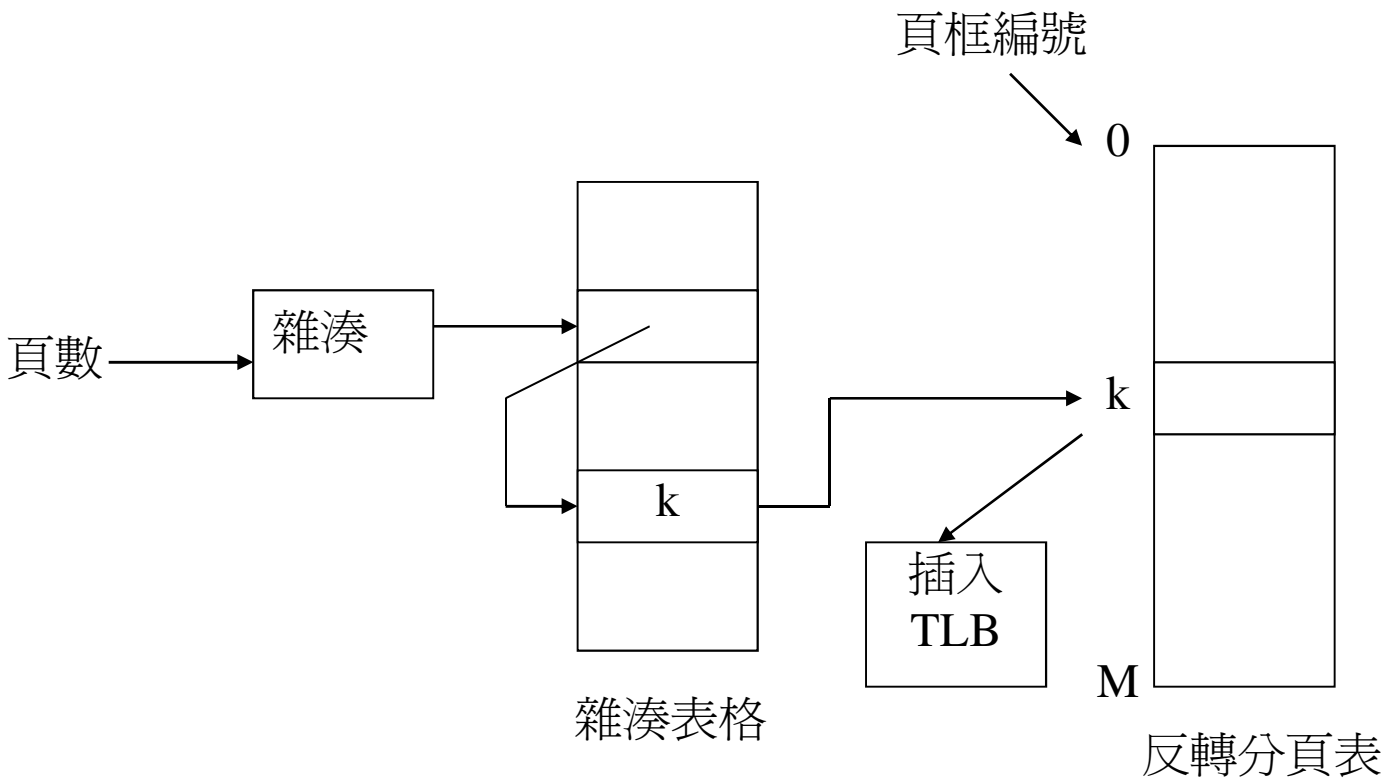
反轉分頁表（2）



反轉分頁表（3）

- 反轉分頁表可以降低儲存每個分頁表所需要的空間，但是搜尋分頁表所用的時間卻大量增加
  - 減輕此問題：使用雜湊表格
- 從取得邏輯位址開始到在記憶體中存得資料，至少需要兩次的記憶體讀取：一次是雜湊表格，另一次是反轉分頁表
  - 解決的辦法是利用類似關聯式記憶體的方式來加快搜尋
- 在一個使用反轉分頁表的系統中，無法達到分頁共用的目的

反轉分頁表 - 雜湊表格



第八章 記憶體管理

- 背景介紹
- 連續配置
- 分頁
- 分段
  - 基本方法
  - 分頁式分段
- 摘要

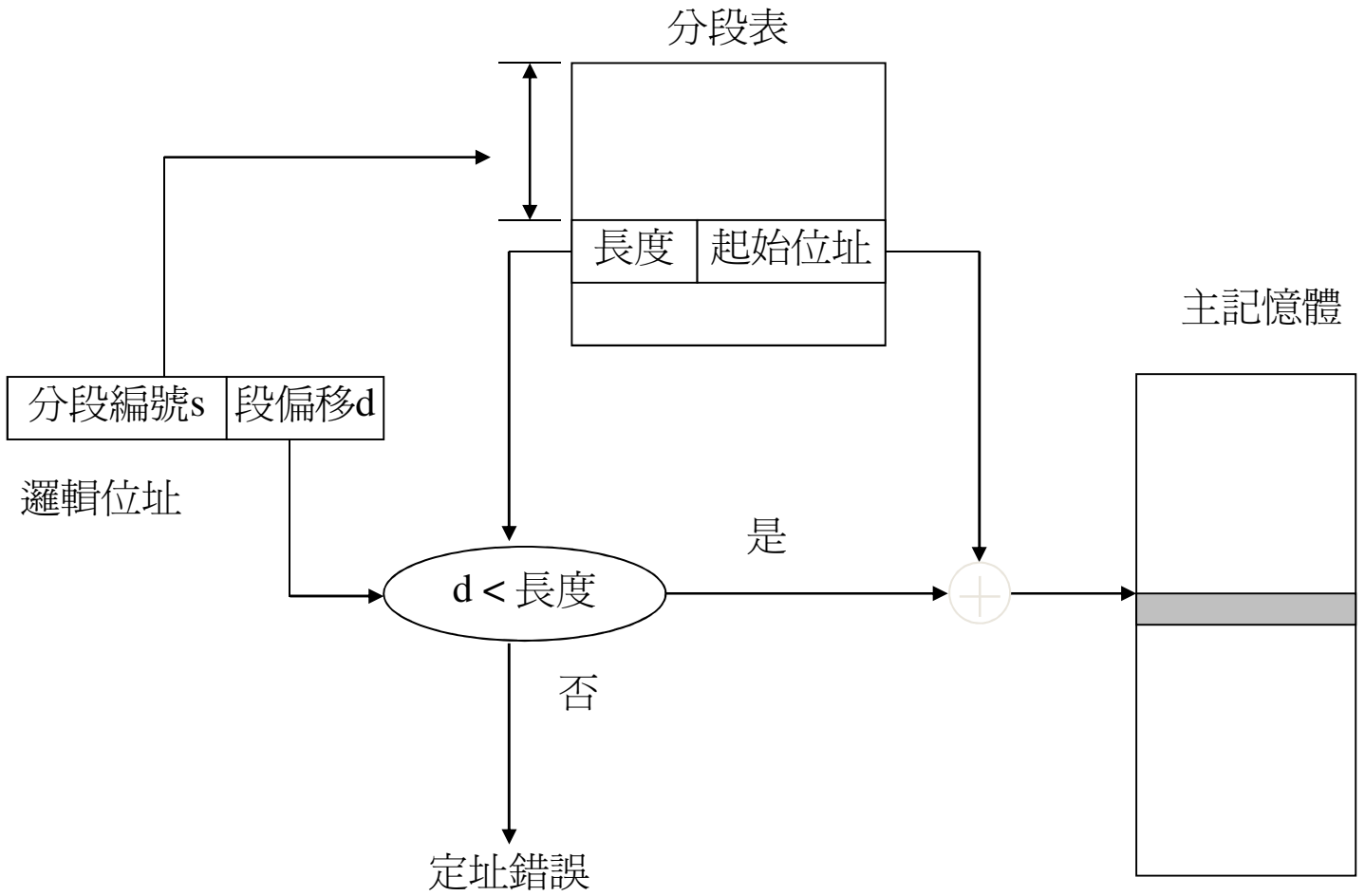
分段

- 依照程式的邏輯功能將邏輯位址空間切割成許多分段
- 每個分段的長度都不盡相同，長度是由此分段中程式的大小來決定
- 若要參考分段中某一位元組，可以藉由此分段的起始位址配合段偏移來決定

基本方法（1）

- 一個行程的邏輯位址空間被分成許多分段，每個分段都有一個名稱和長度，也有一個分段編號
  - 一個行程的邏輯位址被分成兩個欄位：分段編號（當作分段表的索引）與段偏移
- 每個程式均有一個分段表，其中每一記錄都有：
  - 分段基底值：記錄分段在主記憶體中實際開始的位址
  - 分段界限值：記錄該分段的大小，以避免程式執行時超過該分段界限

分段法的使用

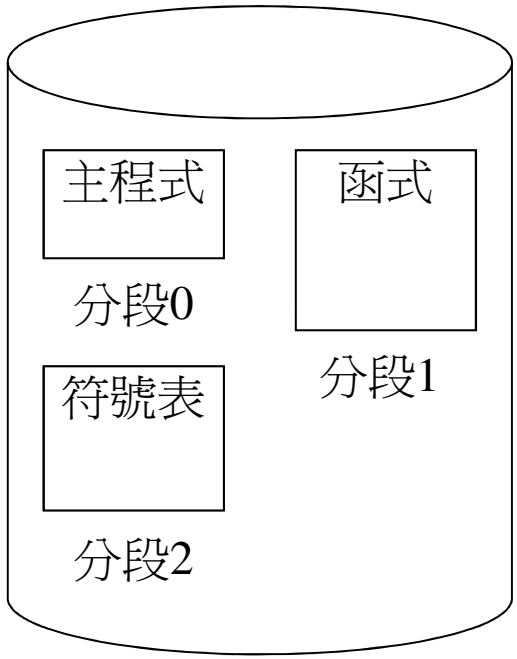


分段法

主記憶體

	位址	長度
0	5100	200
1	1500	300
2	3200	300

分段表

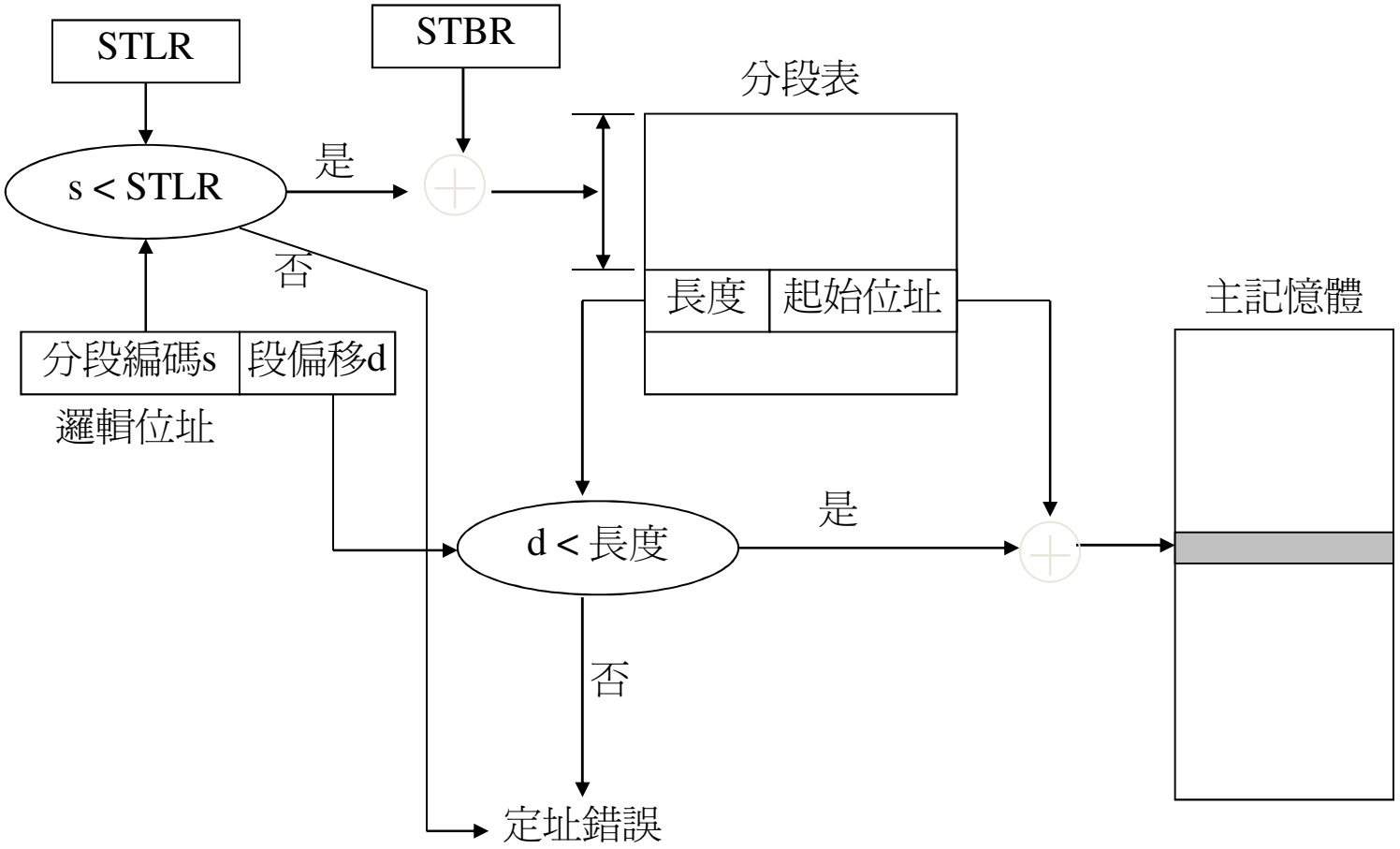


邏輯位址空間

基本方法（2）

- 分段表可以存放在
  - 快速的暫存器：節省對界限暫存器相加與基底暫存器比較的時間
  - 主記憶體：對於有大量分段的程式而言，不宜放在快速暫存器中
    - 利用一個分段表基底暫存器來指向分段表；使用一個分段表長度基底暫存器記錄分段大小
    - 將分段基底值與段偏移相加，取得要存取位元組的實體位址
    - 缺點：每個邏輯位址需要對實際記憶體存取兩次，所以速度比較慢
    - 解決方法：也可利用關聯式暫存器儲存經常會被使用的分段表記錄

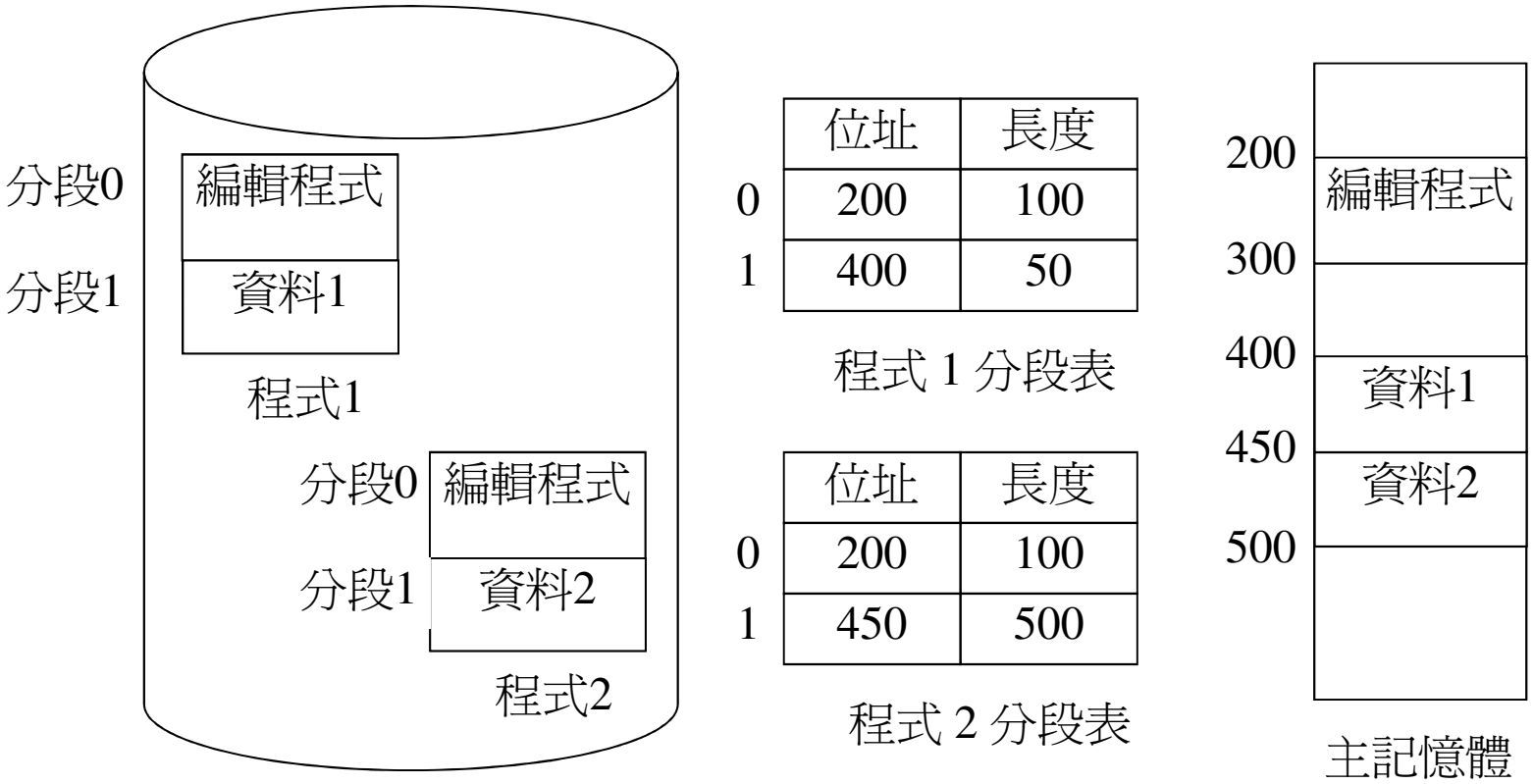
配合 STLR 與 STBR 的分段法



基本方法（3）

- 分段法可達到分段保護的功能
  - 可以指定指令分段是唯讀或是唯執行的方式
  - 分段表中每個項目有一組保護位元，以避免不正當的存取
- 分段法可達到資料或程式碼共用的功能
  - 兩個分段表中有共同的項目指向同一個實體位址
- 問題：行程可能跳到本身以外的記憶體位址執行程式碼
  - 如果此記憶體位址的程式碼共用，勢必所有共用行程所屬共用分段的分段編號都要相同
  - 該編輯程式如何參考本身的程式？且共用的行程增多後，想要找到一個可共用的編號將會更加地困難

分段共用


分段0
分段1
編輯程式
資料1
程式1
分段0
分段1
編輯程式
資料2
程式2
位址
長度
0
200
100
1
400
50
程式 1 分段表
位址
長度
0
200
100
1
450
500
程式 2 分段表
200
300
400
450
500
編輯程式
資料1
資料2
主記憶體

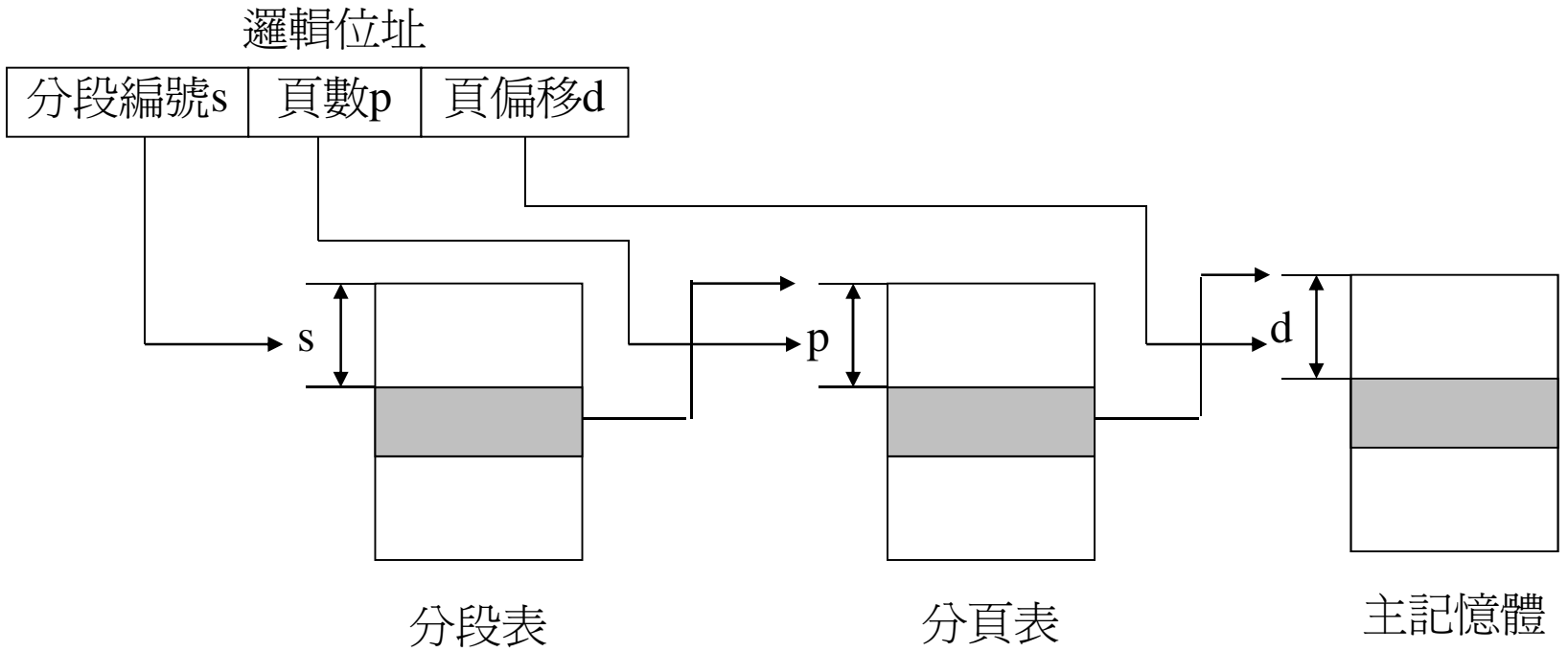
基本方法（4）

- 分段法中分段長度是變動的，若有外部斷裂，便可能導致記憶體中根本找不到足夠空間供一個分段載入
  - 該行程可以等待別的行程釋放記憶體空間，或是利用聚集法取得夠大記憶體空間以供載入
  - 若選擇等待其他行程釋放空間，CPU 排程程式可以考慮讓優先權較高、需要記憶體空間較小的行程先行，以增進效率

分頁式分段

- 在分頁式分段的方式下
  - 每個行程會依據邏輯功能切成分段，而每個分段會再被分割成分頁，所以在這種架構下外層爲分段表，內層爲分頁表
  - 一個邏輯位址包含了分段編號 s、分頁碼 p 與頁偏移 d
  - 系統會先透過 s 在分段表中找到分頁表的起始位址，再透過 p 在分頁表中找到分頁的頁框位置，最後再與 d 相加成爲實體記憶體位址
- 平均每個分段也會有 1/2 分頁的內部斷裂
  - 所以分頁式分段中每個行程會有較多的內部斷裂
  - 著名的 OS/2 作業系統便是使用分頁式分段法，使用的分頁大小爲 4 KB

分頁式分段（２）



摘要（1）

- 記憶體管理的主要目的：
  - 有效地運用與維護記憶體空間
- 程式以二進位可執行檔的型態儲存於磁碟中
- 程式在執行時透過記憶體管理單元將邏輯位址轉換成實體位址
- 程式位址連結的工作可在編譯、載入或執行階段完成
- 當行程所需的執行空間大於配置給行程的記憶體時：
  - 利用重疊技術
  - 置換

摘要（2）

- 記憶體管理的方式可分爲
  - 連續配置：採用的方式有
    - 單一分割配置：單一使用者的系統
    - 多重分割配置：多元程式的環境
      - 最先、最佳、或最差符合法來分配記憶體
      - 不當的配置方式將會加重記憶體外部斷裂與內部斷裂的問題
  - 非連續配置
    - 分頁法，用來解決記憶體外部斷裂的問題，需要硬體的支援
      - 將載入的程式分割成固定大小的分頁，主記憶體也分割成固定大小的頁框，頁框與分頁大小相同
      - 執行程式時，把程式所有的分頁載入記憶體任何可用的頁框中，不需相鄰

摘要（3）

- 多層次分頁
- 反轉分頁表
- 分段法
  - 依照程式的邏輯功能將邏輯位址空間切割成許多分段
  - 達到資料或程式碼共用的目的
- 分頁式分段：每個行程會依據邏輯功能切成分段，而每個分段再分割成分頁