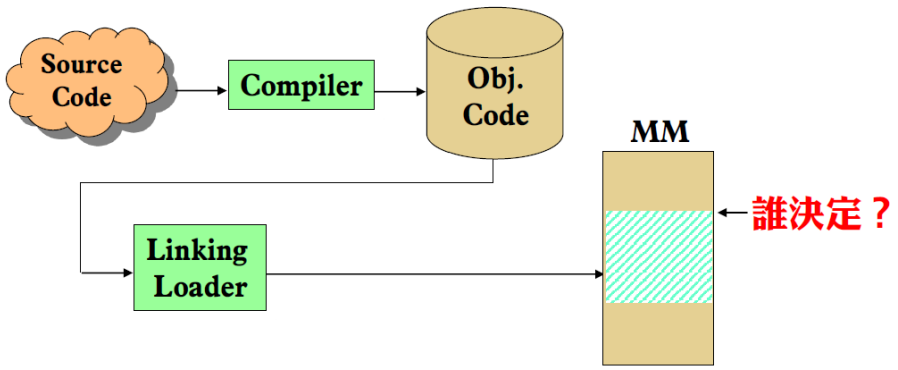
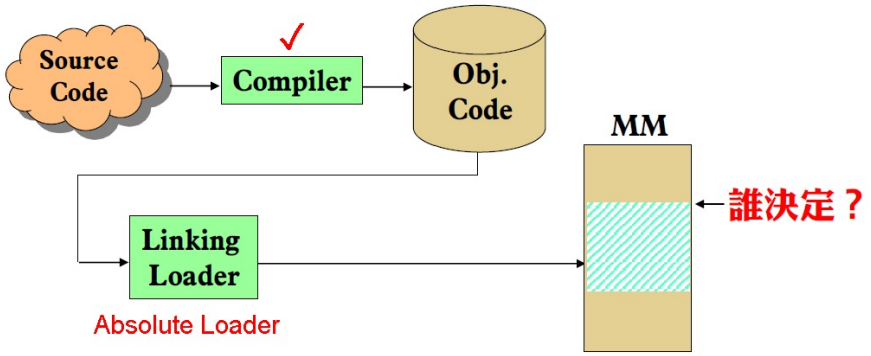
記憶體管理

Binding 及其時期

- 決定程式執行的起始位址
  - 程式要在記憶體哪個地方開始執行
  - 所有程式與 Data 都需要在記憶體中才能被 CPU 使用
  - 程式內宣告的資料和變數的位址一併確定
- Binding 時期
  - Compiling Time
  - Loading Time --> 鏈結副程式
  - Execution Time

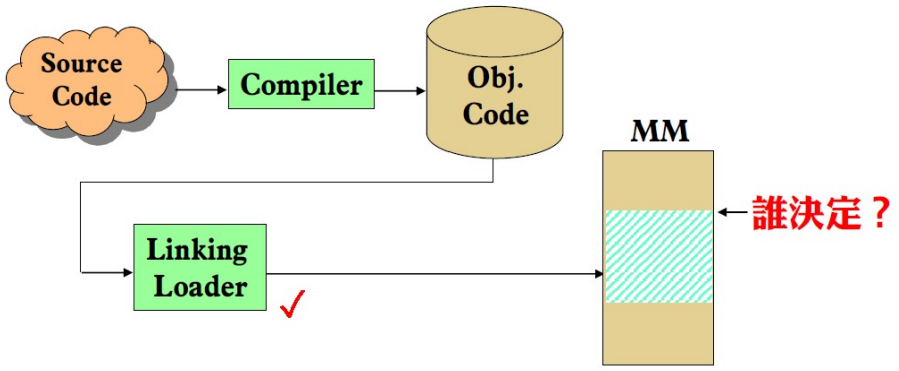


Compiling Time



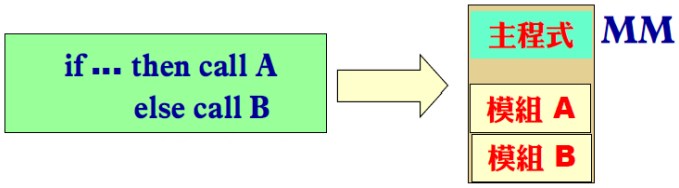
- **由 Compiler 決定**
  - 程式執行的起始位置是**固定**的，不能變更
  - Compile 出來的**目的碼**為 **Absolute Obj. Code(絕對目的碼)**
    - Compiler 產生 Obj code **同時決定了記憶體位址**
    - Linking Loader 變成 Absolute Loader 只負責載入
  - 如果選定的位址內有其他程式在執行，必須 Re-Compiling
  - 彈性太小
    - 一般應用程式不會使用，通常適用於常駐程式
- 不支援 Relocation 重定位
- 若要並更程式執行的起始位只，必須對程式 Re-Compiling

Loading Time



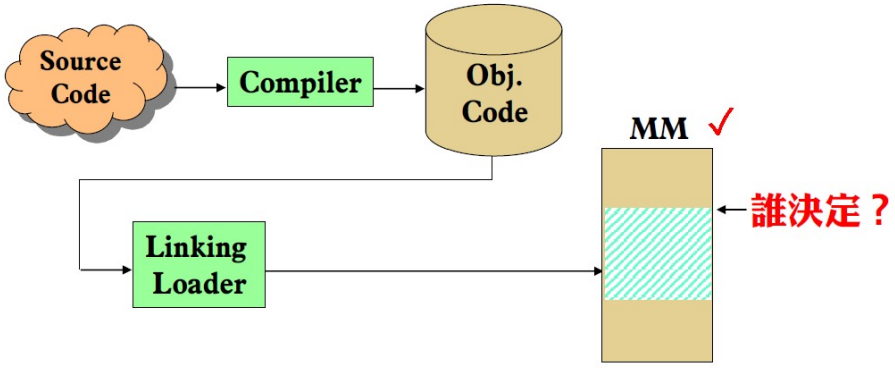
- **由 Linking Loader 決定**
  - Compiler 不知道 Obj. Code 起始位址
- **Linking Loader 通常會做四件事**
  - **Linking** --> 鏈結程式碼和所使用的副程式
  - **Allocation** --> 配置程式碼的起始位址
  - **Loading** --> 載入到記憶體的位址
  - **Relocation** --> 如果記憶體位址有其他程式在執行，需要重定位
- **程式不一定從固定位址開始執行，每一次程式重新執行，只需要再重新 Linking Load**
- 支援 Relocation 重定位

- **缺失**
  - 在 Execution Time 沒有被呼叫的模組仍然需要事先 Linking, Allocation, Loading --> 浪費時間和記憶體
    - **舉例**

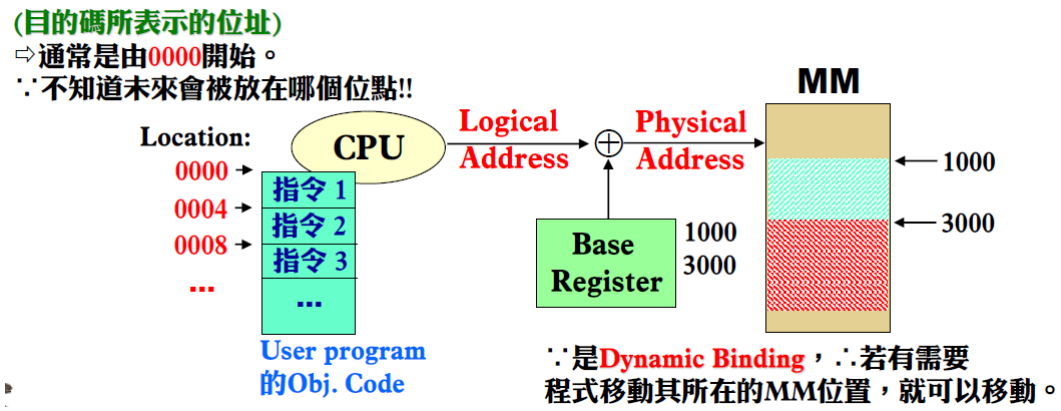


      - A 和 B 兩個模組不可能同時執行，但仍需要事先 Linking, Loading 到記憶體中
      - OS 有很多錯誤處理程序，問題更嚴重 (錯誤處理程序可能有 10000 個)
  - 如果副程式很多，每次重新執行皆須再作 4 項工作 --> 耗時
  - 程式執行期間，仍然**不可以改變起始位址**
    - 如果程式執行期間發現記憶體不夠用，只能終止程式，退回到 Linking Loader 去做 Relocation

Execution Time



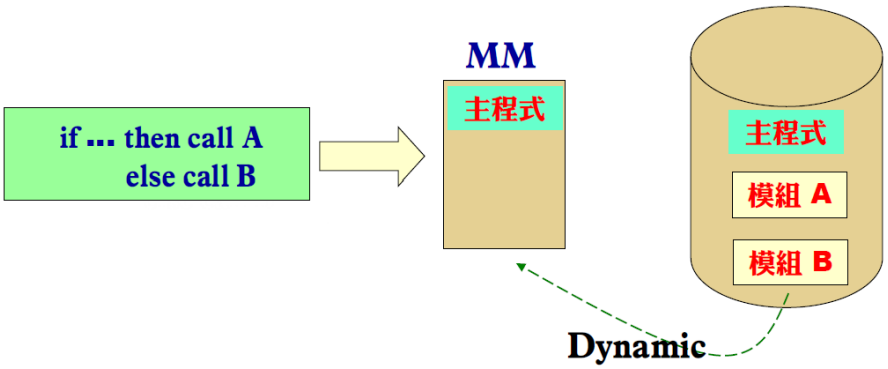
- **由 OS 動態決定**
  - Dynamic Binding (Dynamic --> 程式執行期間的動作)
- **定義** 在執行期間才決定程式執行的起始位址，表示在程式執行期間，可以任意變更起始位址
- **硬體支援**
  - OS 會利用一個 **Base Register 紀錄目前程式的起始位址**
  - 每次 CPU 送出的 **Local Address 都需要和 Base Register 相加**，才會得到 Physical Address，再到記憶體中存取



- **缺點** 程式執行較慢，**效能差**
- **優點** **彈性高**

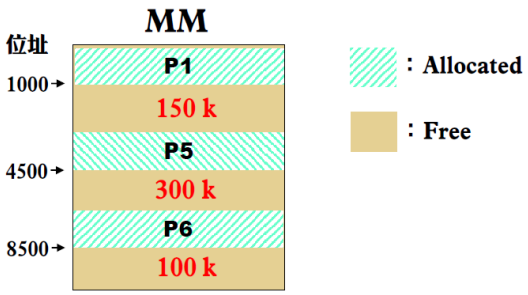
Dynamic Loading

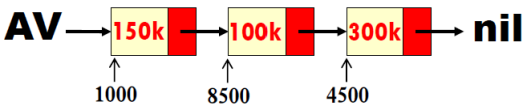
- **定義** 在程式執行期間，某個模組拜真正呼叫到，才將其載入到記憶體中
- **目的** 節省記憶體空間



動態變動分區記憶體管理

- OS 採用 **`Contiguous Allocation`**，依據 `Process` 的大小，找一**塊足夠大的連續可用空間**配置給 `Process` 使用
- OS 會利用 `Link List` 保存管理 `Free Block`，稱為 **`AV-List(Available List)`**





動態分區的 Allocation 方式

First-Fit

- 若所需的記憶體大小為 `n`，從 `AV-List` 的頭部開始搜尋，直到找到第一個 `Free Block Size >= n` 為止
- **缺點**
  - 經過多次配置後，**容易在 `AV-List` 前端產生許多非常小的可用空間 (因為被配置的機率低)**
  - 每次搜尋都需要經過這些區塊，**增加搜尋時間**

- 解決方法：Next-Fit Allocation

Next-Fit

- 從上次配置的下一個 Block 開始搜尋，直到找到第一個 Free Block Size >= n 為止
- 通常 AV-List 會使用 **Circular Link List**

Best-Fit

- 若所需的記憶體大小為 n，從 AV-List 中找到 **Free Block Size >= n 且 (size - n) 值最小者**

Worst-Fit

- 若所需的記憶體大小為 n，從 AV-List 中找到 **Free Block Size >= n 且 (size - n) 值最大者**

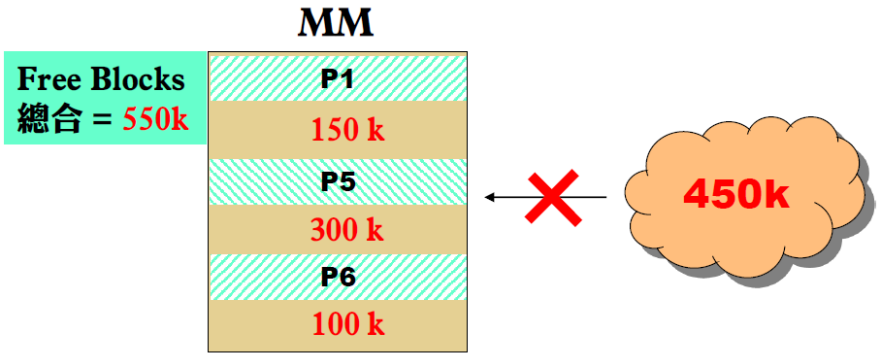
	時間效益	空間利用率
First-Fit	優	~優
Next-Fit	優	優
Best-Fit	差	優
Worst-Fit	差	差

- **共通問題**
  - **External Fragmentation 外部碎裂**
  - **極小 Free Blocks**
    - 增加搜尋成本
    - **解決方法：** OS 規定一個 **ε** 值，若 **(Free Block Size - Process Size) <= ε** 則整個 Free Block 全部配給這個 Process

外部碎裂

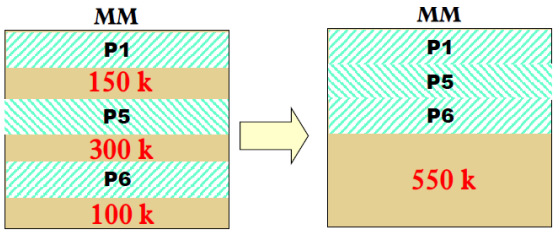
- **定義：** 雖然**所有 Free Block Size 的總和 >= Process 需求**，但是因為這些 Free Block **不連續**，所以無法配置造成記憶體浪費，降低記憶體使用率

- **舉例：** 有一個 Process 需要 450K 的空間



解決外部碎裂問題

- 方法一： 壓縮
  - 移動**執行中的 Process**，使不連續的 Free Block 聚集成一塊連續可用空間



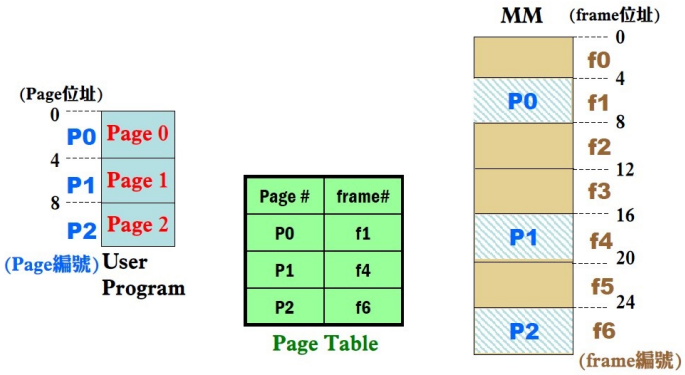
  - 困難
    - 難以在短時間內**決定一個最佳的壓縮策略 (成本最小)**
    - Process 必須是 **Dynamic Binding**
- 方法二： **Page Memory Management**

內部碎裂

- OS **配置給 Process 的記憶體空間於 Process 實際所需空間**，造成其他 Process 也無法使用這個空間，行程記憶體浪費
  - 舉例： 有一個 Process 需要 149K 的空間

Page Memory Management 分頁記憶體管理

- **Physical Memory 實體記憶體：** 是為一組 **Frame (頁框)** 集合，每個頁框的大小均相等
- **Logical Memory 邏輯記憶體：** 即 User Program，視為一組 **Page (頁面)** 的集合，**頁面大小等於頁框大小**
- 配置方式
  - 假設 User Program 的大小為 n 個頁面，OS 只需要在實體記憶體中**找到>=n 個 Free Frame 即可配置**
  - **Non-Contiguous Allocation** 配給 User Program 的頁框**不一定連續**
  - **不需配置連續空間，所以可消除外部碎裂**
- OS 會為每個 User Program 建立 Page Table 紀錄每個 Page 被載入的 **Frame 編號**或**起始位址**
  - 假設 Page Size = 4



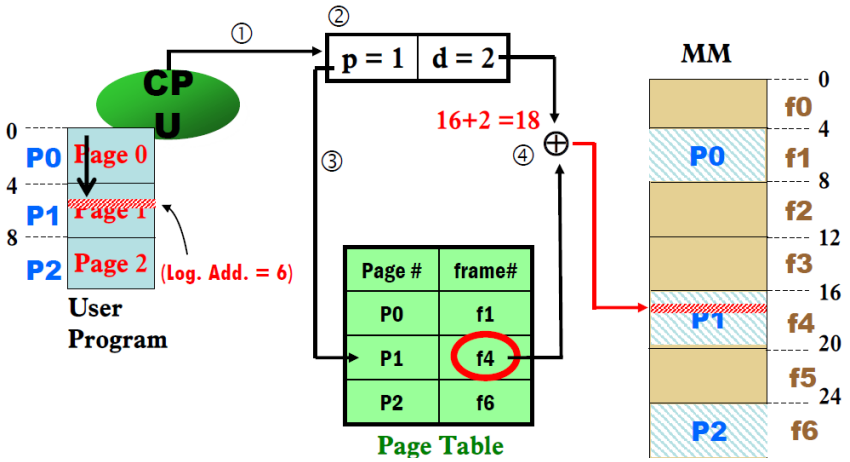
- Logical Address 轉換成 Physical Address 的過程
  - CPU 送出一個**單一值的 Logical Address**
  - Logical Address 會自動被分解為**<p, d>**
    - Page#: **p** = Logical Address / Page Size
    - Page Offset: **d** = Logical Address % Page Size (頁偏移量)
  - 根據 p 去查 Page Table，取得 Page **對應的頁框 f 的起始位址**
    - 起始位址 = 頁框編號 x 頁框大小
  - **f 的起始位址 + d** 即可得到 Physical Address
- 範例
  - 假設 Page Size = 4

    現在要執行 Logical Address = 6 的指令
  - 過程

    p = 6 / 4 = 1 --> Page Table: Page# = P1; Frame# = f4

    d = 6 % 4 = 2

    Physical Address = f4 * Page Size + d = 4 * 4 + 2 = 18

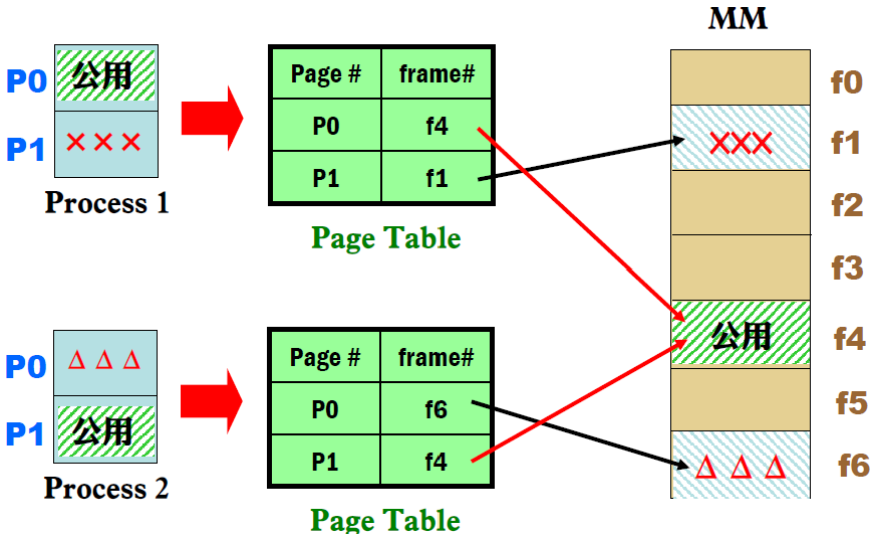


- 優點:
  - 解決外部碎裂問題
  - 支援記憶體共享和保護
  - 支援 Dynamic Loading 和 Virtual Memory 的製作
- 缺點
  - Memory 有效存取時間長

    - Logical Address 轉 Physical Address 過程需要**計算 p，計算 d，查分頁表，計算 f+d**
  - 會有內部破碎問題
    - User Program 大小不一定是 Page Size 整數倍
    - EX1: Page Size = 4K, User Program = 21K
      - 使用 6 frame，產生 24K - 21K = **3K 的內部碎裂**
    - EX2: Page Size = 100K, User Program = 101K
      - 使用 2 frame，**產生 99K 內部碎裂 --> Page Size 越大越嚴重**
  - 需要額外硬體支援
    - Page Table Implementation (每個 Process 都需要 1 個 **Page Table**)
    - Logic Address 轉 Physical Address (需要**搜尋器、加法器**)

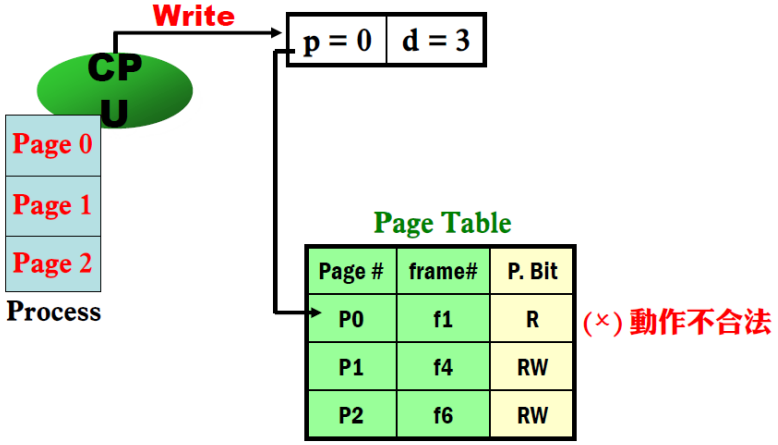
共享記憶體

- Process 透過本身的**分頁表**，對應相同的頁框即可達成共享機制



記憶體保護

- 在分頁表上加上 Protection Bit 欄位
  - R: Read Only
  - RW: Read/Write



Page Table 製作

方法一 使用 Register

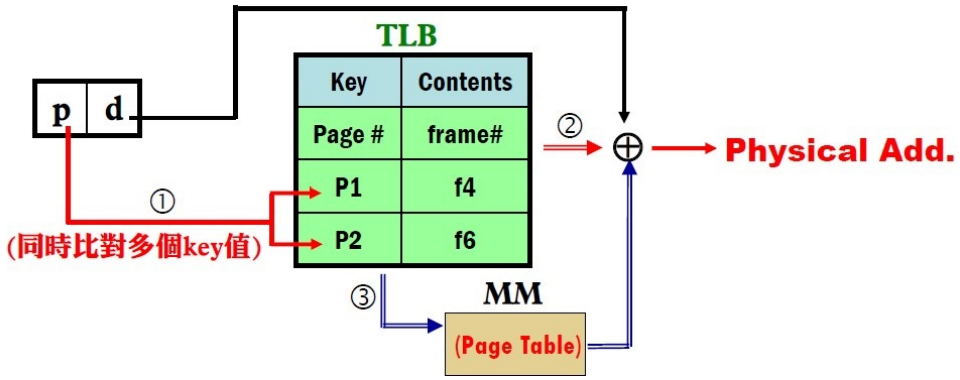
- 使用 **Register** 存分頁表每個項目內容
- 優點：速度快
- 缺點：不適用 Page Table 太大的情況

方法二 Page Table 保存在記憶體中

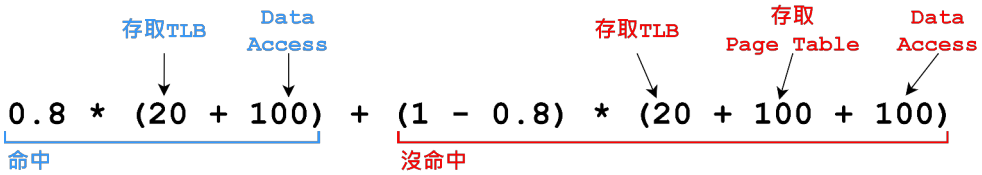
- OS 利用 **Page Table Base Register(PTBR)** 記錄 Page Table 在記憶體的起始位址
- 優點：適用於 Page Table 較大的情況
- 缺點：速度慢，需要 2 次記憶體存取，第一次取出 Page Table，第二次用於存取 Data Access，浪費時間

方法三 使用 TLB 保存部分(常用)Page Table

- 使用 **Transction Lookside Buffer(TLB)** 保存常用的 Page Table，完整的 Page Table 儲存在記憶體中
- 查詢 Page Table
  1. 到 TLB 查詢有無對應的 Page Table
  2. 如果 Hit 則輸出 Frame 的起始位址，速度等同方法一
  3. 如果 Miss 則到 Memory 中取出 Page Table 查詢



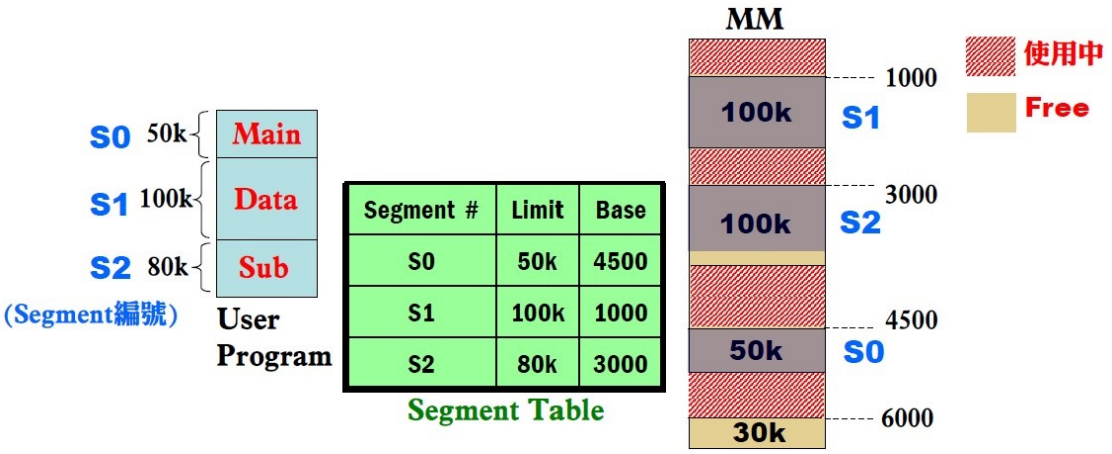
- 範例
  - TLB 存取花費 20ns
    記憶體存取花費 100ns
    TLB Hit Ratia: 80%
    有效記憶體存取時間為和?
  - **Solution**



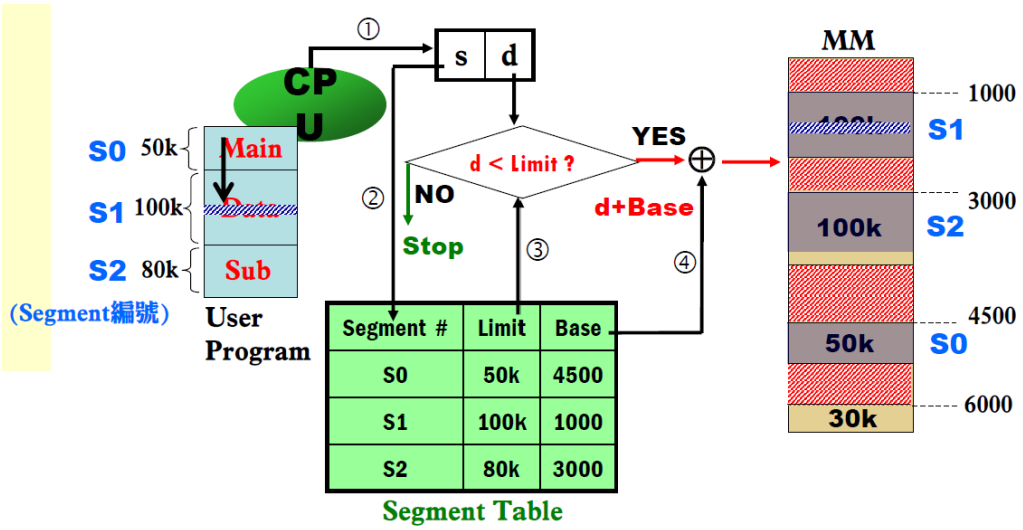

```
= 0.8 * 120 + 0.2 * 220
= 140 (ns)
```

Segment Memory Management 分段記憶體管理

- **Physical Memory**: 不需事先區分記憶體空間，如果記憶體中存在 Free Memory Space >= Process 所需空間，就將該 Space 配置出去
- **Logical Memory**: 即 User Program，是為一組 Segment (段) 的集合，各段大小不一
- **Segment**: Main, Subroutine, Data Section...etc
- 配置方式
  - Segment 之間可採用不連續性配置
  - 單一 Segment 採用連續性配置
- OS 建立 **Segment Table** 給每個 Process 記錄**各段大小 (Limit)** 和**各段載入記憶體的起始位址 (Base)**



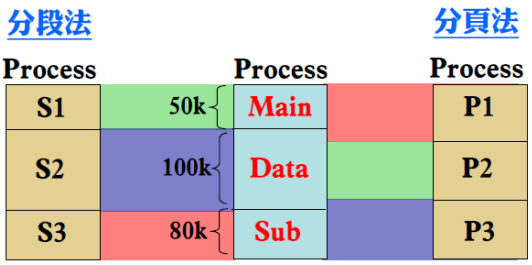
- Logical Address 轉 Physical Address
  1. CPU 送出 Logical Address: <s, d>
     - Segment#: **s**
     - Segment offset: **d**
  2. 根據 s 查詢 Segment Table 取得該段 Limit
  3. 檢查 **d < Limit**
     - true: 合法存取，取出段的 Base，go to 4
     - false: 非法記憶體存取，Stop
  4. d + Base = Physical Address



- 範例
  - 偏移量單位 (K)

    ```
    <s, d>      Physical Address
    --------    ----------------
    <0, 30>  -> 4530
    <1, 80>  -> 1080
    <1, 120> -> Error (larger than Limit 100K)
    <2, 50>  -> 3050
    ```

- 優點
  - 沒有內部破碎問題
  - 可支援記憶體共享和保護
  - 可真原 Dynamic Loading 和 Virtual Memory 製作
- 缺點
  - 可能有外部破碎問題
  - 記憶體存取時間較長
  - 需要額外硬體支援
- **為何分段法比分頁法更容易達成共享和保護**

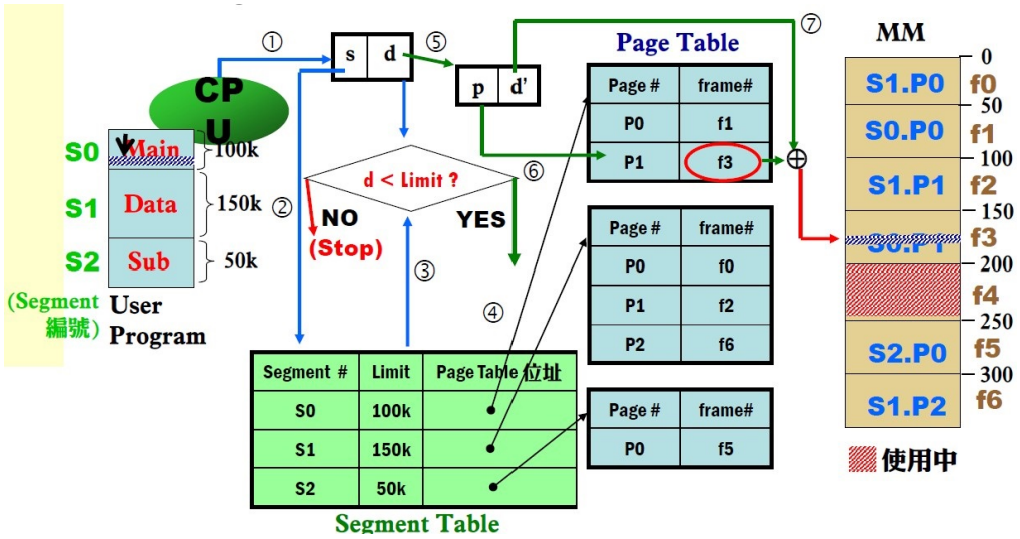


  - 分頁法要求每個 Page Size 相同，有的 Page 可能涵蓋度不同的程式片段，保護不易
  - 分段法不要求 Segment Size 相同，每個 Segment 分別涵蓋不同的程式片段，易於保護

分頁式分段

- 段之中分頁，記憶體還是視為一組 frame

- User Program 由一組 Segment 組成，Segment 由一組 Page 組成
- 每個 Process 會有一個 Segment Table，每個段會有一個 Page Table
- Logical Address 轉 Physical Address
  - CPU 送出 Logical Address <s, d>
  - 根據 s 查詢 Segment Table 取得 Limit
  - 檢查 d < Limit
  - 取出相應的 Page Table
  - 將 d 分解成<p, d'>
  - 根據 p 查詢 Page Table 取得 frame 起始位址 f
  - f + d'得到 Physical Address
- 範例
  - Page Size = 50K
  - 記憶體是為一組 frame 的組合



  - **<s, d>　　Physical Address**

    -------　　----------

    <1, 70>　-> 120

    <2, 80>　-> Error (larger than segment limit 50K)

    <0, 80>　-> 180
- 分析
  - 沒有外部破碎問題
  - 有內部破碎問題，因為分頁
  - Table 數目過多，極佔空間
  - Memory Access Time 更長