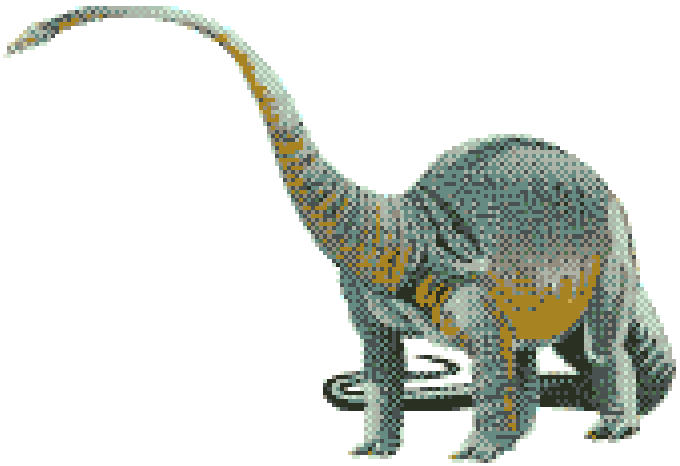
作業系統(Operating Systems)

Course 8: 記憶體管理 (Memory Management)



授課教師：陳士杰

國立聯合大學 資訊管理學系

■本章重點

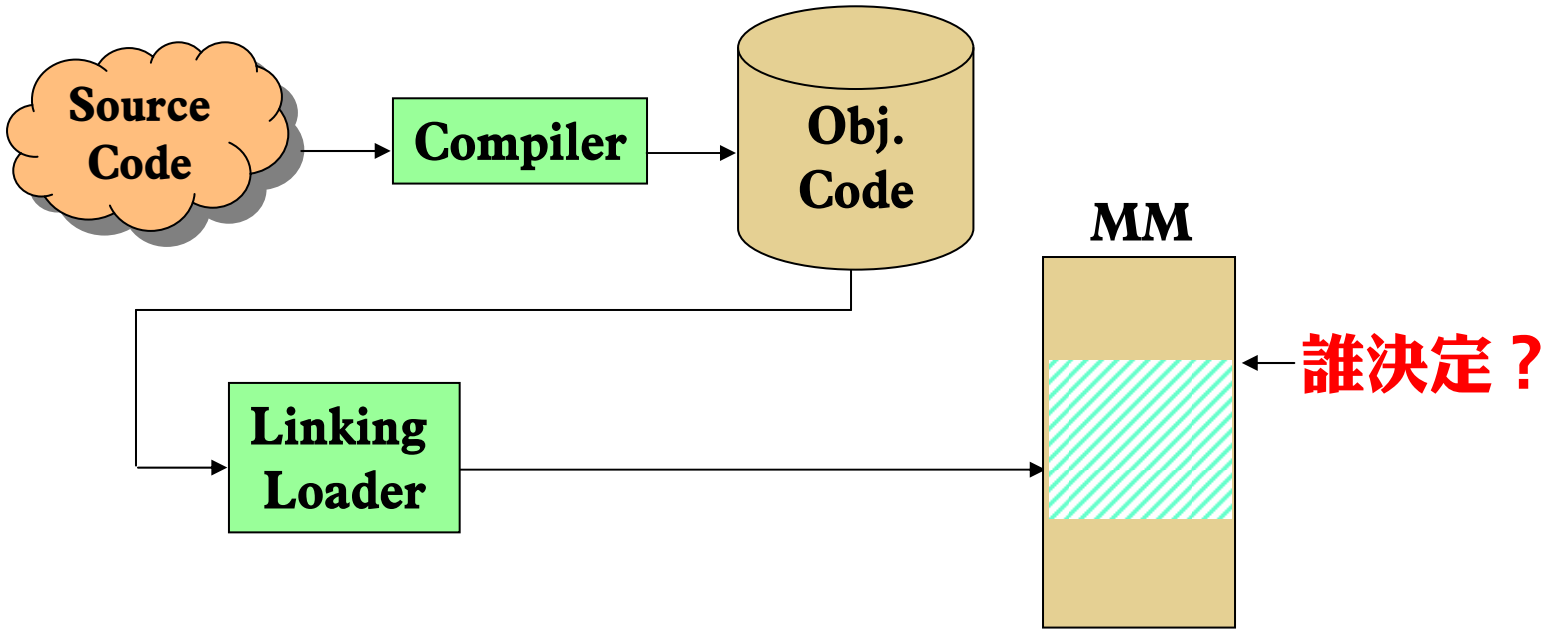
- Binding及其時期
- Dynamic Binding作法
- 動態變動分區記憶體管理
  - First Fit
  - Best Fit
  - Worst Fit
  - Next Fit
- 外部與內部碎裂 (External & Internal Fragmentation)
- Compaction(壓縮)
- Page Memory Management
- Segment Memory Management
- Paged Segment Memory Management

Binding及其時期

- 何謂Binding？
  - Def: 決定程式執行的起始位址。
  - 即：程式要在記憶體的哪個地方開始執行。
    - ∵所有程式與Data均須在MM中方可為CPU使用。
    - 連帶地將程式內所宣告的資料與變數位在MM的什麼地方也一併確定了。

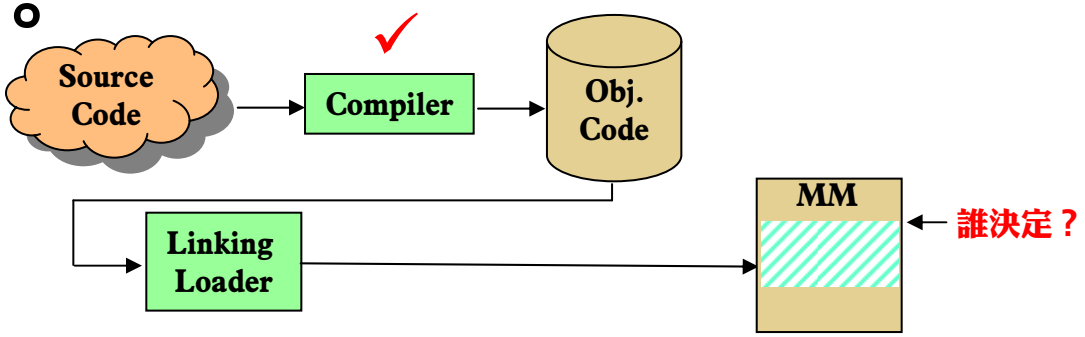
Binding的時期

- 可能的Binding時期有三個：
  1. Compiling Time
  2. Loading Time
  3. Execution Time



Compiling Time

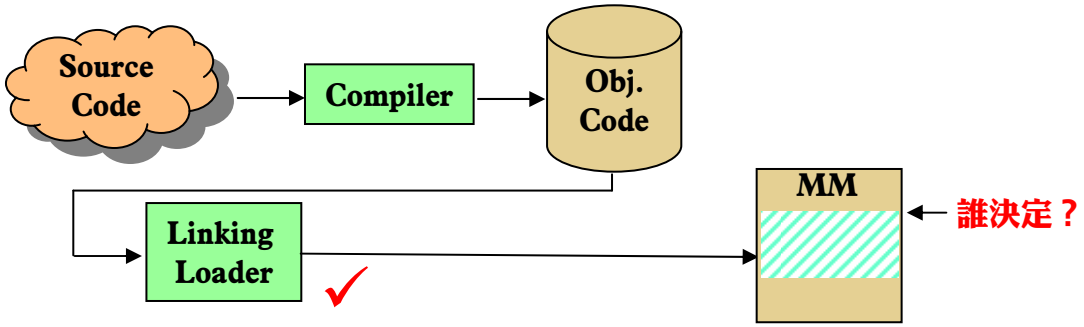
- 由Compiler決定。



  - 將來程式執行的起始位址是固定的，不得變更。
    - 所Compile出來的目的碼為Absolute Obj. Code。若所決定的位址內有其它的程式在執行，則須Re-compiling。
    - 彈性太小
  - 不支援Relocation (重定位)
  - 若要變更程式執行的起始位址，須對程式再Re-compiling。

Loading Time

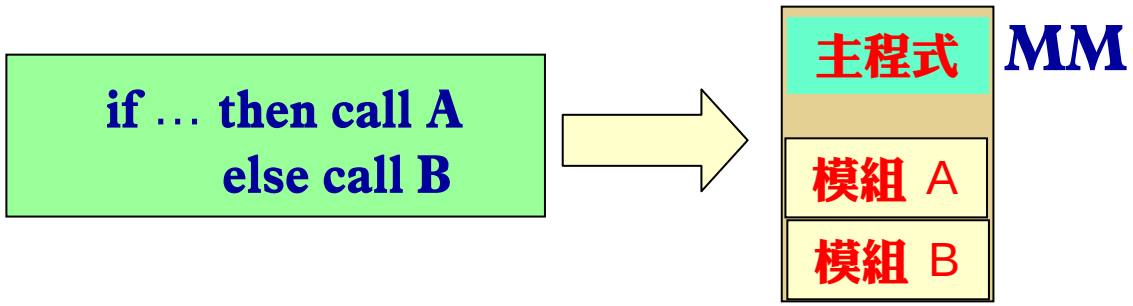
- 由Linking Loader決定。



  - Linking Loader通常會做四件事：Linking, Allocation, Loading, Relocation
  - 程式不一定都由固定位址開始執行!!.∵.每一次程式重執行時，只須再重新Linking Load一次即可。
  - 支援Relocation (重定位)

- **缺失：**
  - 在Execution Time沒有被呼叫到的模組仍需事先Linking, Allocation, Loading，浪費時間，也浪費記憶體。
  - 若副程式很多，每次重新執行皆須再作4項工作，耗時。
  - 程式執行期間，仍不可以改變起始位址。

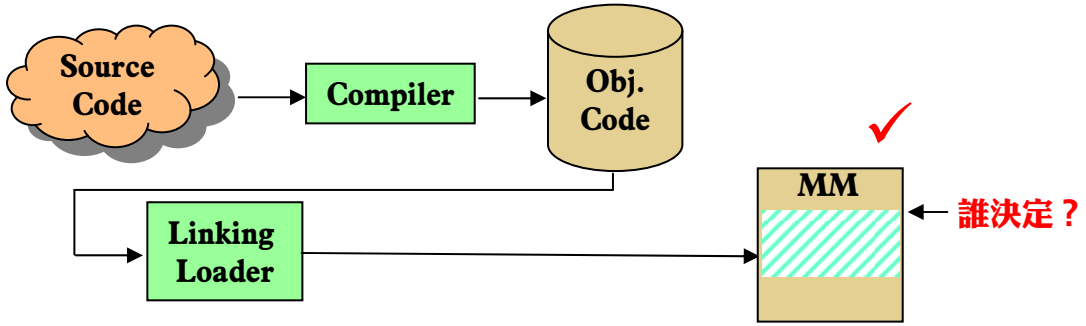
如：



- A和B這兩個模組不可能同時執行到，但仍需要事先Linking Load到MM中。
- 在OS中問題更大條!!因為OS有很多錯誤處理程序!!!

Execution Time

- 由OS動態決定。

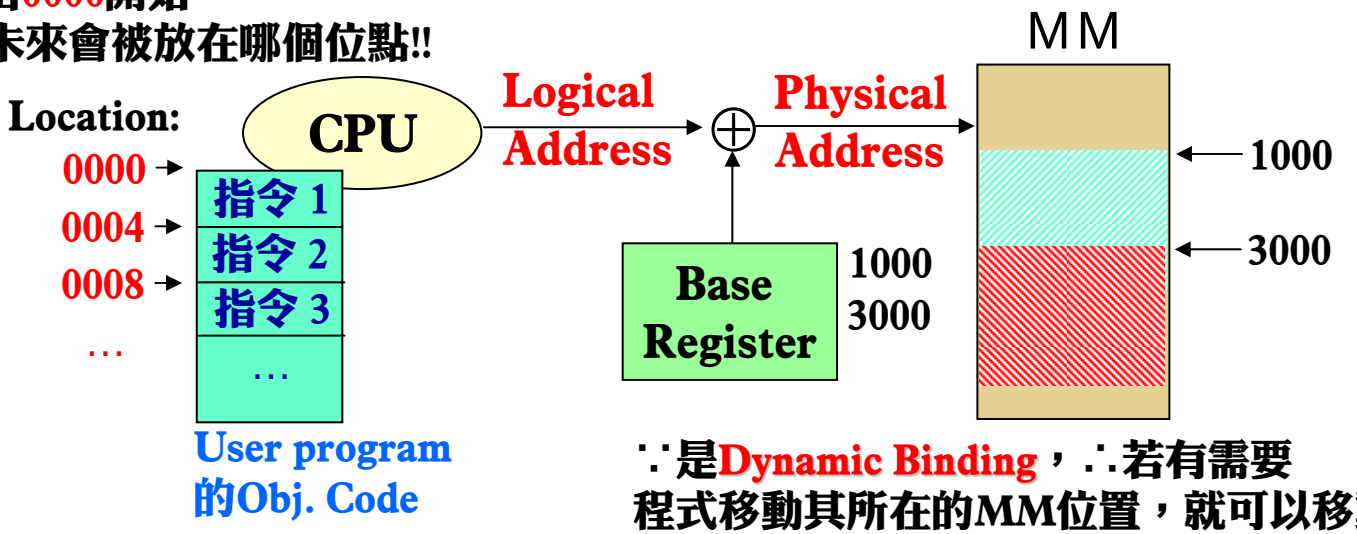


  - 又稱為Dynamic Binding
  - Def: 在程式執行期間 (Execution Time) 才決定程式執行的起始位址。表示程式執行期間，可任意變更其起始位址。

- **所需要的額外硬體支援：**
  - **OS會利用一個Base Register，記錄目前程式的起始位址。**
  - **每次CPU所送出的Local Address皆須與Base Register相加，才會得出Physical Address，再到MM中進行存取。**

圖示：

(目的碼所表示的位址)
⇨通常是由0000開始。
∵不知道未來會被放在哪個位點!!

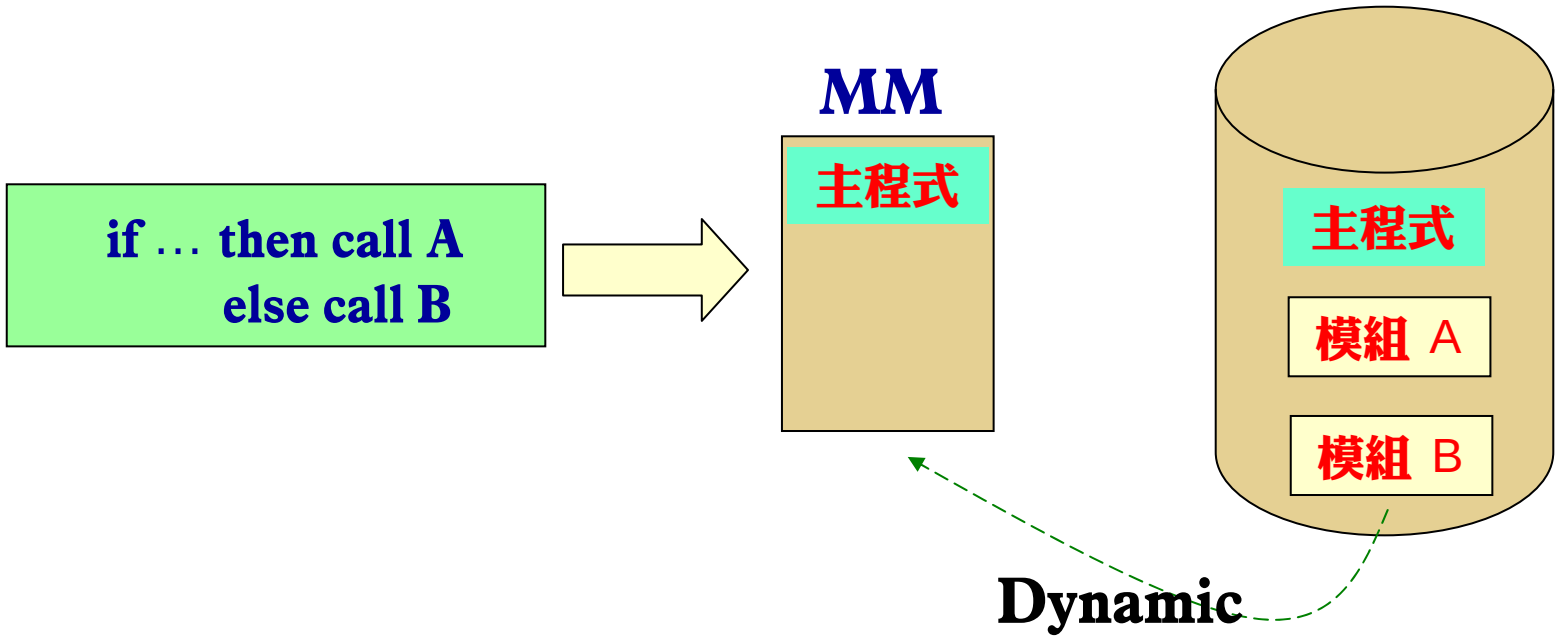


∵是Dynamic Binding，∴若有需要程式移動其所在的MM位置，就可以移動。

- 缺點：程式執行較慢，Performance 差。
- 優點：彈性高。

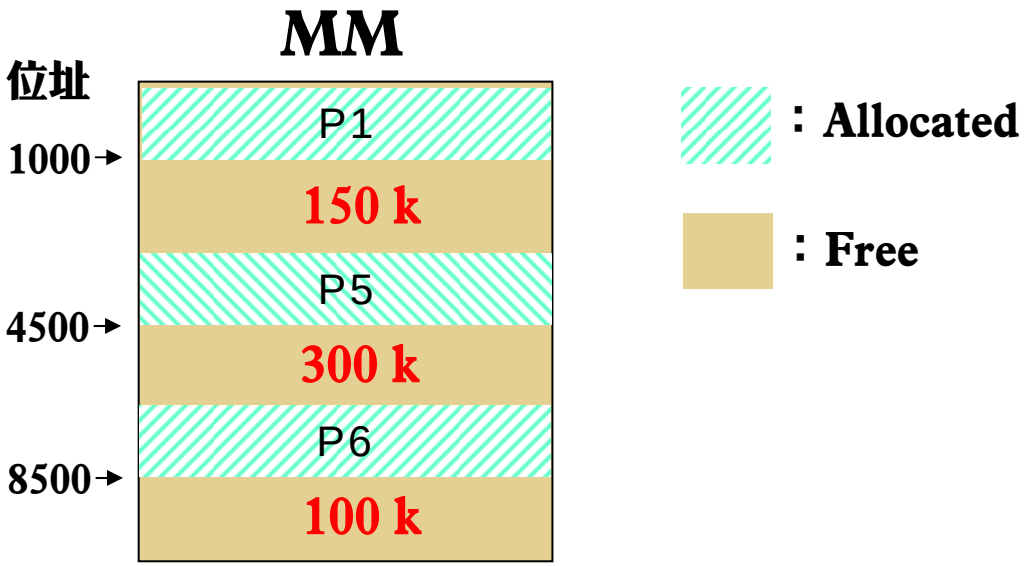
Dynamic Loading

- Def：在程式執行期間，當某個模組被真正呼叫到時，才將其載入到MM中。
- 其主要目的在於想節省MM空間，發揮Memory Utilization

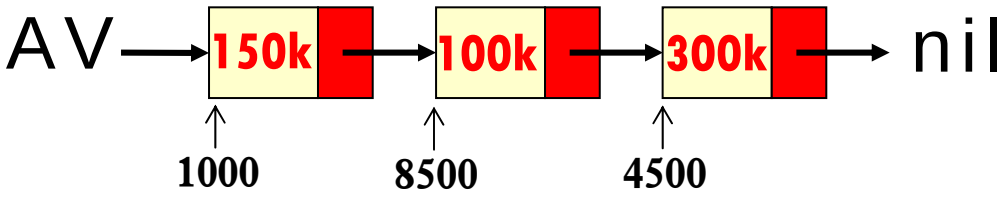


■動態變動分區之Memory Management

- 在Multiprogramming之下，記憶體內存在有多個Process執行，且各個Process的size並不相同，進入系統及完成工作的時間也不盡相同。
- OS採用 "Contiguous Allocation"的方式，依據各個Process的大小，找到一塊夠大的連續可用空間，配置給該Process使用。
- 例：



- OS會利用Link List保存 (管理) Free Blocks，稱為AV-List (Available list)。



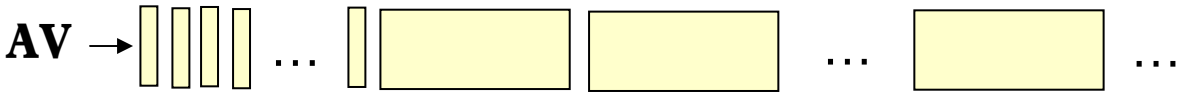
■動態變動分區之Allocation方式

- First-Fit
- Best-Fit
- Worst-Fit
- Next-Fit

- 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。
- Next-Fit
- Best-Fit
- Worst-Fit

First-Fit深入分析

- 缺點：
  - 在經過多次配置後，易在AV-list前端附近產生許多非常小的可用空間 (被配置機率低)。然而每次Search皆要經過這些區塊，徒增Search Time。



- 解法：Next-Fit Allocation

- First-Fit
- Next-Fit
  - 從上次配置後的下一個Block開始搜尋，直到找到第一個free block size ≥ n 為止。
  - 為First-Fit的變形。
  - 通常AV-list會以Circular link list表示。
- Best-Fit
- Worst-Fit

- First-Fit
- Next-Fit
- **Best-Fit**
  - **若所需的記憶體大小為n，從AV-list所有Blocks中，找出 size ≥ n 且(size – n) 值最小者。**
- Worst-Fit

- First-Fit
- Next-Fit
- Best-Fit
- **Worst-Fit**
  - **若所需的記憶體大小為n，從AV-list所有Blocks中，找出 size ≥ n 且(size – n) 值最大者。**

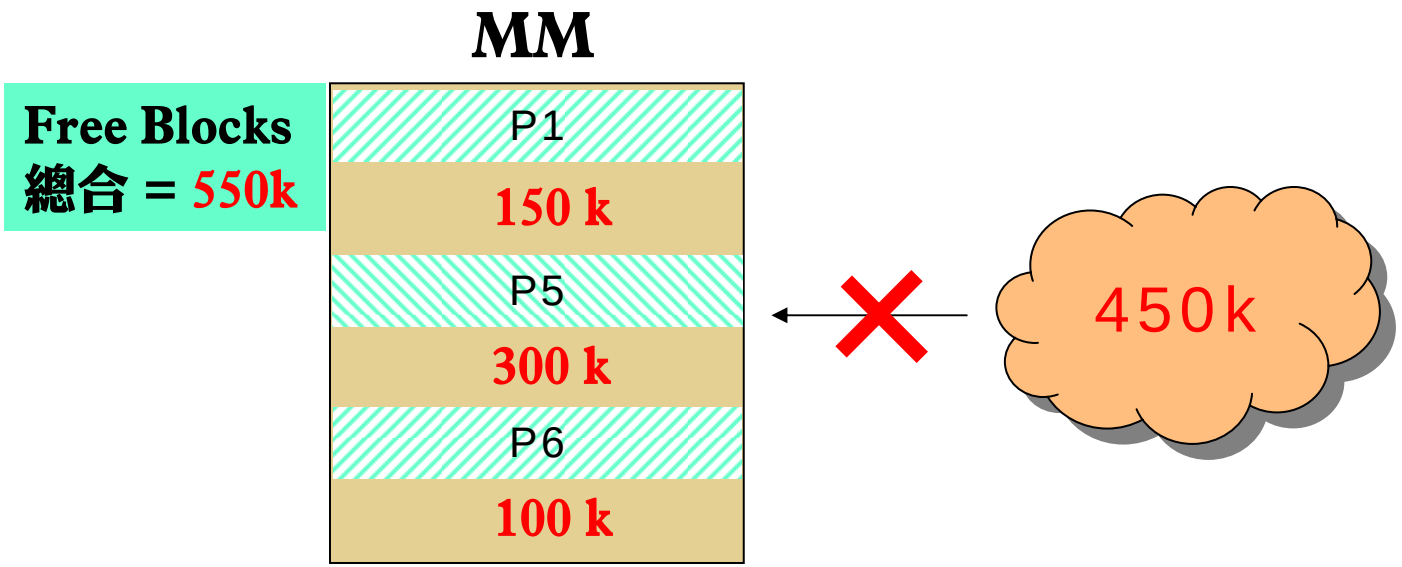
比較

	Time效益	空間利用度
First-Fit	優	≒優
Best-Fit	差	優
Worse-Fit	差	差
Next-Fit	優	優

- 不論是First/Best/Worse/Next Fit Allocation，皆存在下列共通問題：
  - 均有外部碎裂(External Fragmentation)問題
  - 配置完所剩的極小Free Blocks仍會保存在AV-list中，徒增Search Time與記錄成本。
    - 解法：OS規定一個 ε 值，若 (Free Block Size – Process大小) ＜ ε ，則整個Free Block 全配給此Process。

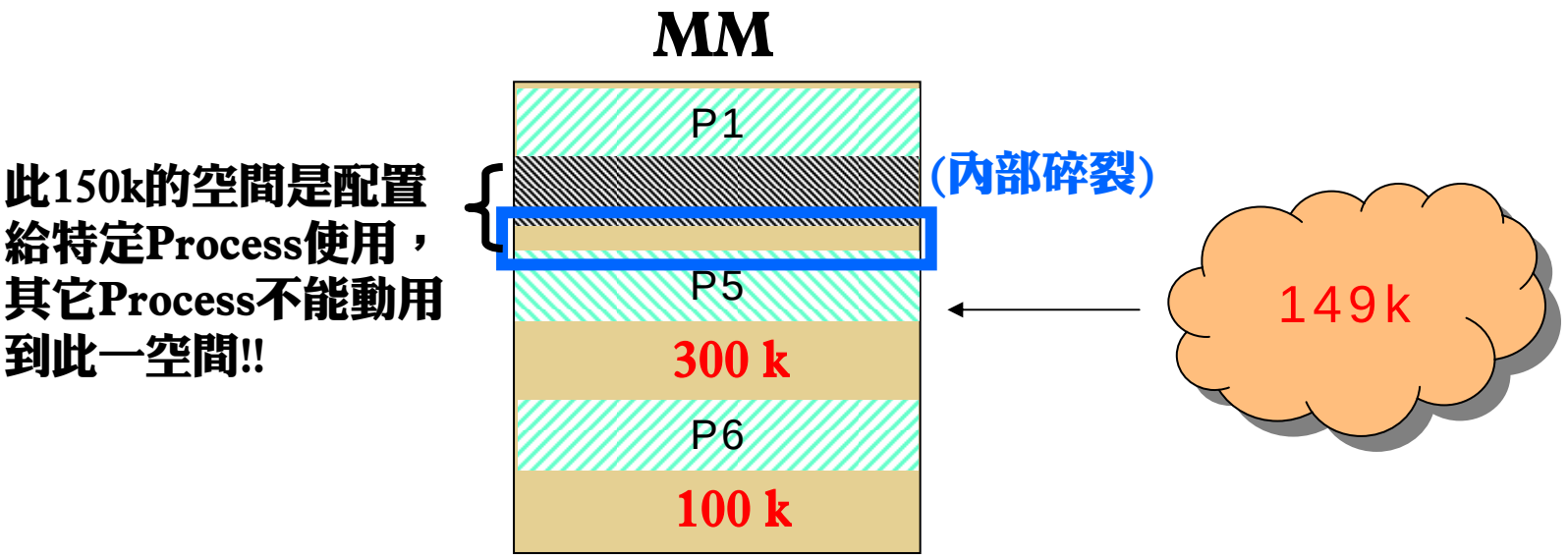
外部碎裂 (External Fragmentation)

- Def: 在連續性配置方式下，可能記憶體中所有Free Block的Size總合 ≥ Process需求大小，但因為這些Free Blocks並不連續，∴仍無法配置，形成Memory空間浪費，降低Memory Utilization。
- 例：有一個Process需要450k的空間：



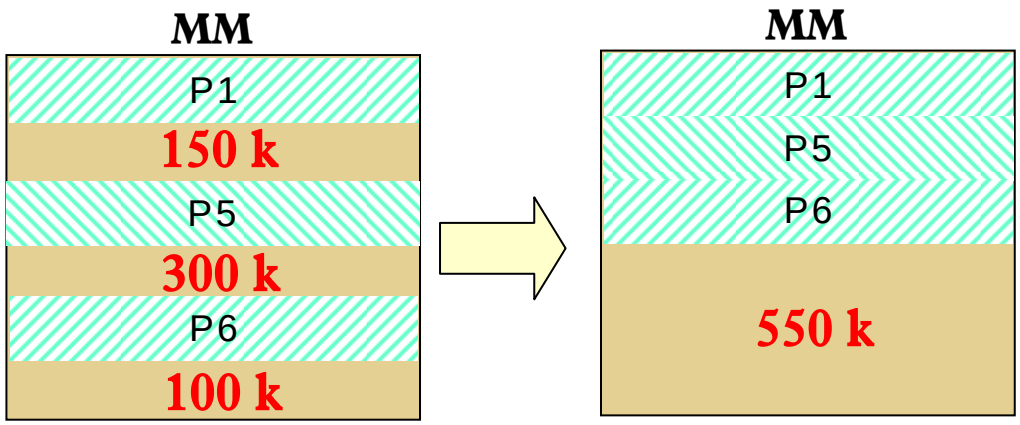
內部碎裂 (Internal Fragmentation)

- Def: OS配置給Process的Memory空間大於Process實際所需空間，此一差值空間該Process用不到，且其它Process也使用不到，形成空間之閒置浪費，此一閒置空間稱之為內部碎裂。
- 例：有一個Process需要149k的空間：



解決外部碎裂問題

- 方法 1：壓縮 (Compaction)
  - 移動 "執行中的Process"，使得不連續的Free Blocks得以聚集成一塊夠大的連續可用空間。
  - 圖示：

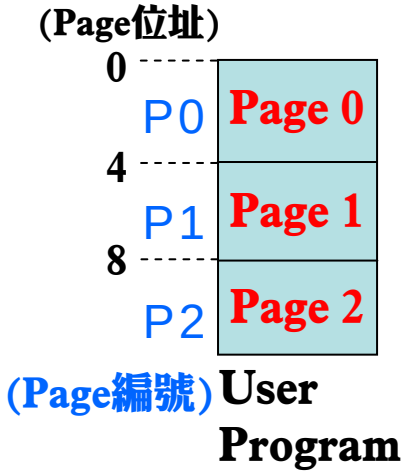


  - 困難：
    - 很難在短時間內決定一個最佳的壓縮策略 (即：成本最小)。
    - Process必須是Dynamic Binding才可以支援。
- 方法 2：利用Page Memory Management。

■ Page Memory Management (分頁記憶體管理)

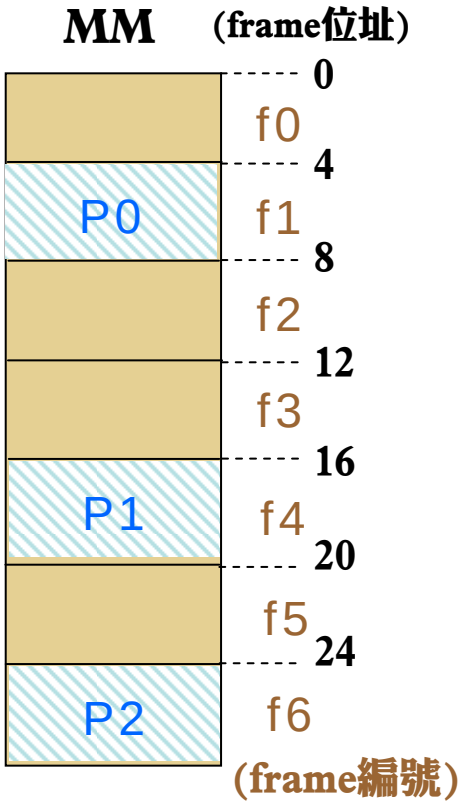
- **實體記憶體 (Physical Memory)**：視為一組**頁框**(Frame)之集合。各頁框的大小均相等。
- **邏輯記憶體 (Logical Memory)**：即**User Program**。視為一組**頁面**(Page)的集合。頁面大小等同於頁框之大小。
- **配置方式**：
  - 假設User Program的大小為n個Pages，則OS只要在Physical Memory中找到≥n個**Free Frame**即可配置。
  - 採取 "**Non-contiguous**" Allocation (不連續性配置)，即配置給User Program使用的多個頁框不一定要連續。
  - ∵不需連續配置可用空間，∴可消除外部碎裂的問題。

- OS會為每一個Process建立一個稱為 “Page Table” 的表格，記錄每一個Page被載入的Frame編號或起始位址。
- 圖示：
  - 假設Page Size = 4



Page #	frame#
P0	f1
P1	f4
P2	f6

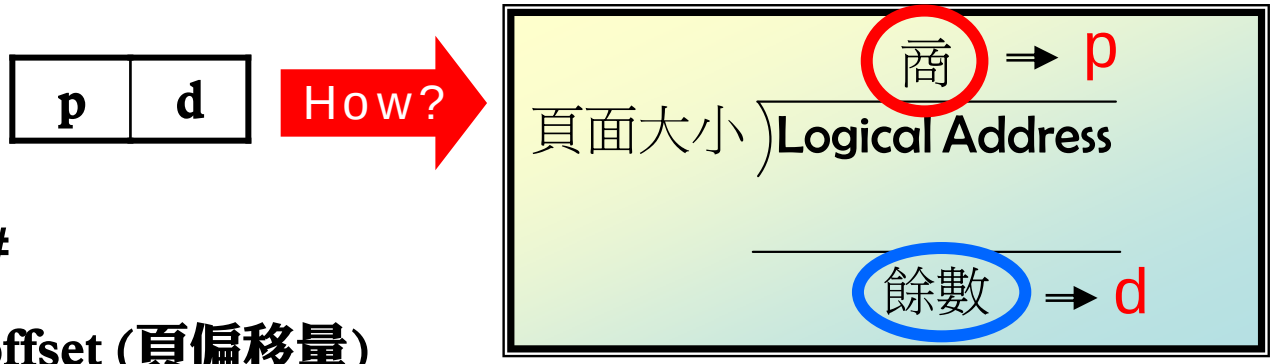
Page Table



- **Logical Address轉換成Physical Address之過程：**

① CPU送出一個**單一值的Logical Address**

② 此Logical Address會自動被拆解成下面**兩個值**：



p: Page #

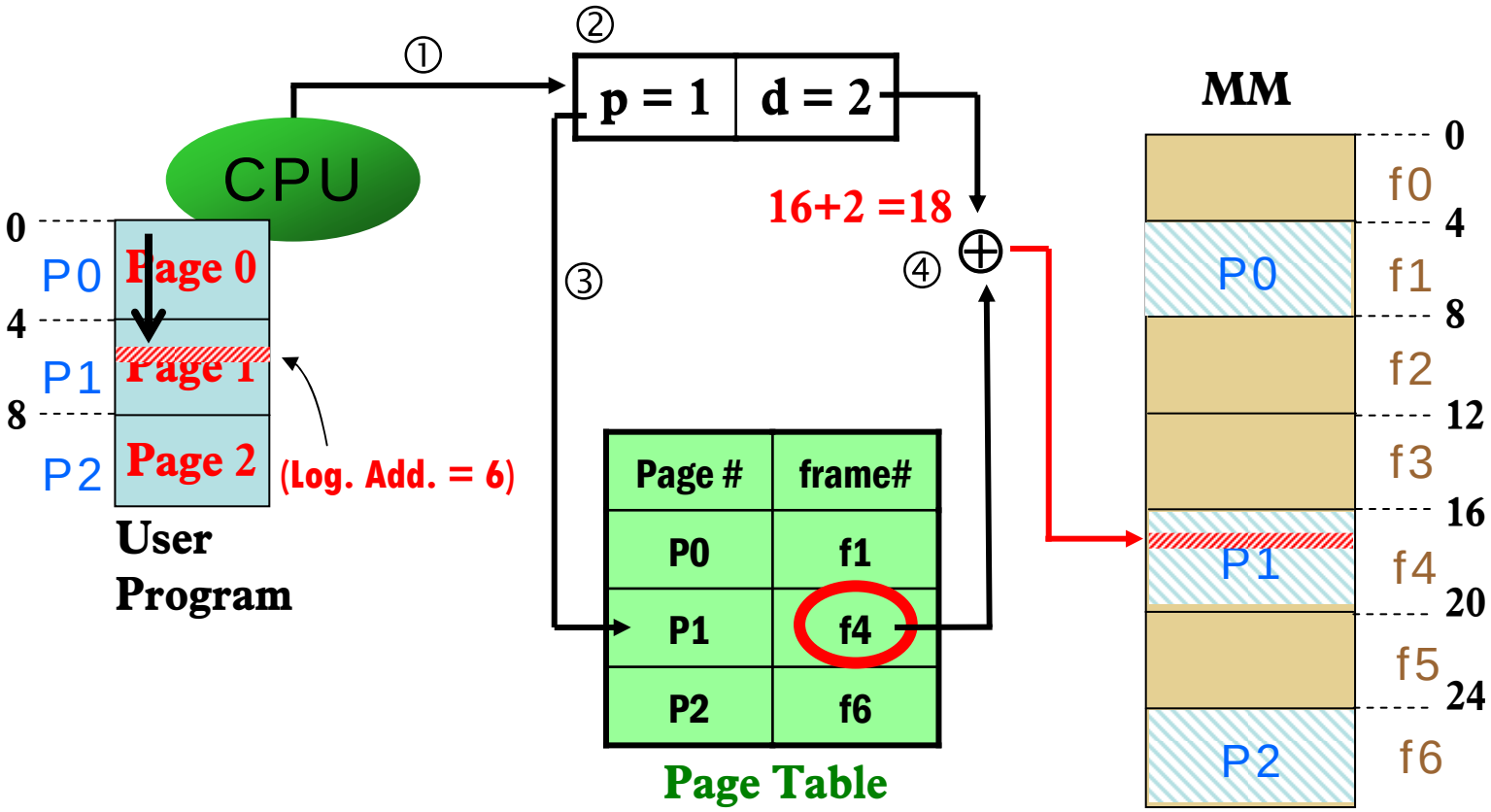
d: Page offset (頁偏移量)

③ 根據 p 去查Page Table，取得該Page之**對應頁框 f 的起始位址**或編號

- 起始位址 = 頁框編號 × 頁框大小

④ **f 的起始位址 + d** 即得Physical Address

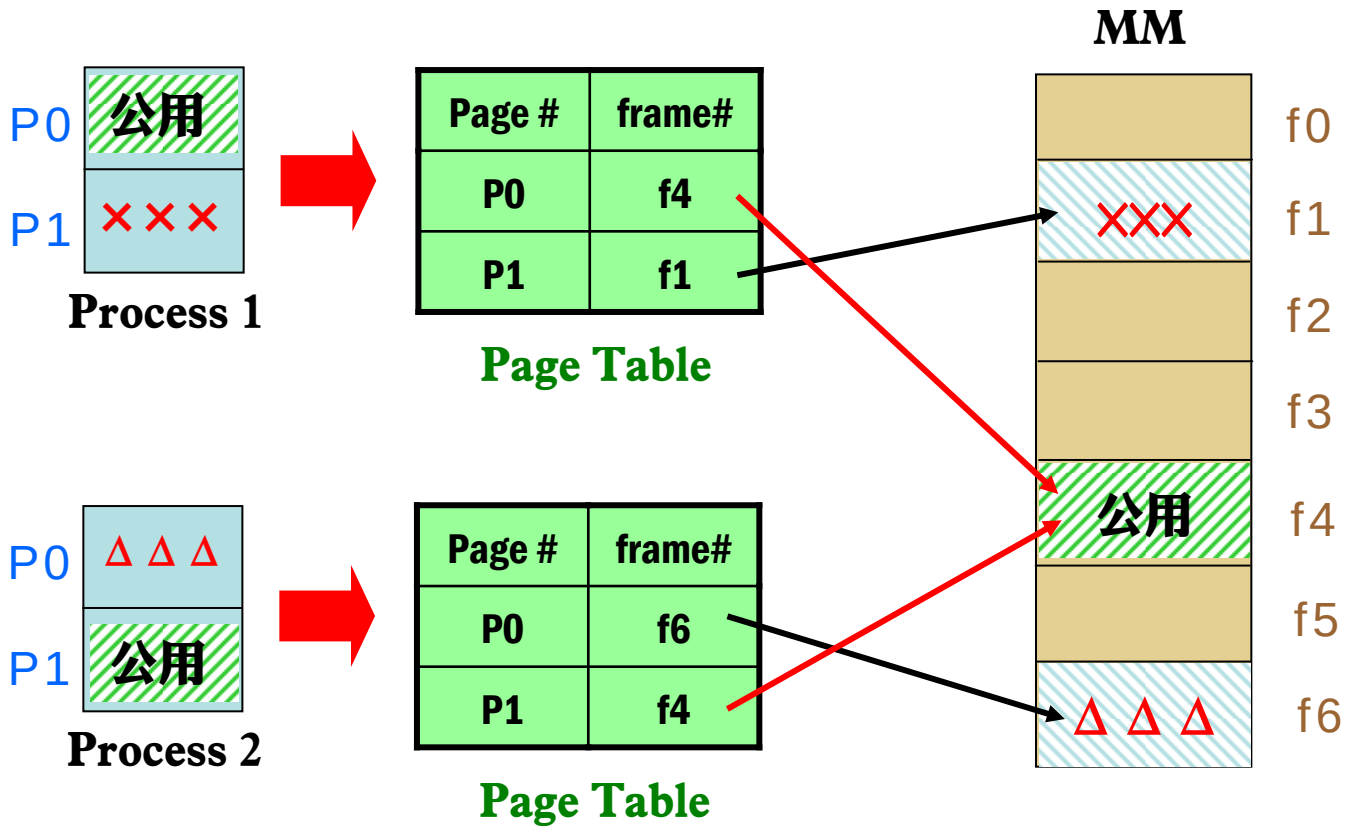
- 例：
  - 假設每個Page的Size = 4
  - 目前要執行邏輯位址為6的指令



- 優點：
  - 解決External Fragmentation問題
  - 可以支援記憶體的共享(Sharing)及保護(Protection)
  - 支援Dynamic Loading及Virtual Memory的製作

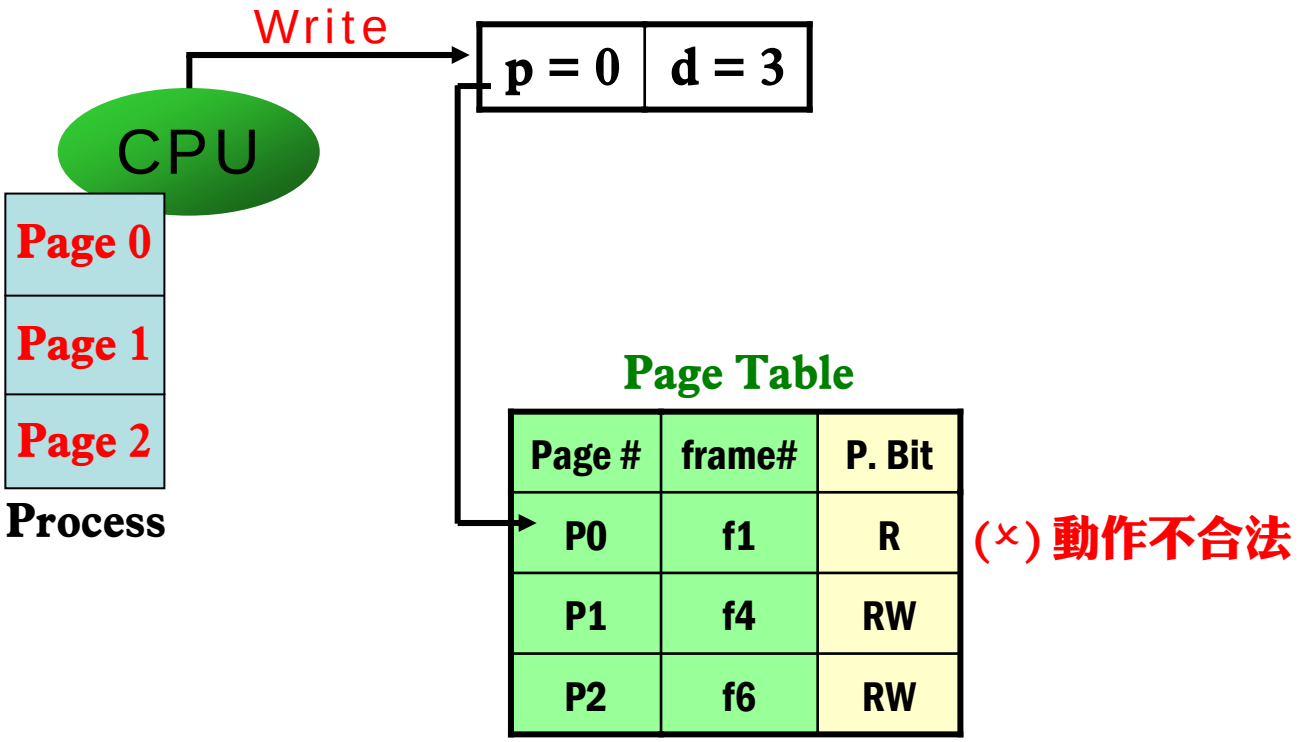
記憶體的共享(Sharing)

- 各Process透過本身分頁表，對應相同的頁框即可達成。



記憶體的保護(Protection)

- 在分頁表上多加一個 "Protection Bit" 欄位，值為：
  - R：表示Read only
  - RW：表示Read/Write皆可



缺點：

- **Memory有效存取時間較長**
  - 因為Logical Add.轉Physical Add.的過程中，需計算 p 與 d，且有查分頁表、f 及 d 相加動作，耗時。
- **會有Internal Fragmentation問題**
  - User Program大小不見得是Page Size的整數倍。
  - Ex 1: Page Size = 4k, User Program = 21k，需配置 6 個frame，且會產生 24k - 21k = 3k的內部碎裂!!
  - Ex 2: Page Size = 100k, User Program = 101k ，需配置 2 個frame，且會產生99k的內部碎裂!! (∴Page Size愈大，則愈嚴重)
- **需要額外的硬體支援**
  - Page Table Implementation (每個Process皆有1個Page Table)
  - Logic Address → Physical Address (用到搜尋器、加法器)

■Page Table的製作

【方法 1】使用Register保存分頁表每個項目的內容

- 優點：速度快
- 缺點：僅適用於Page Table Size較小的情況；太大的Page Table則不適用。

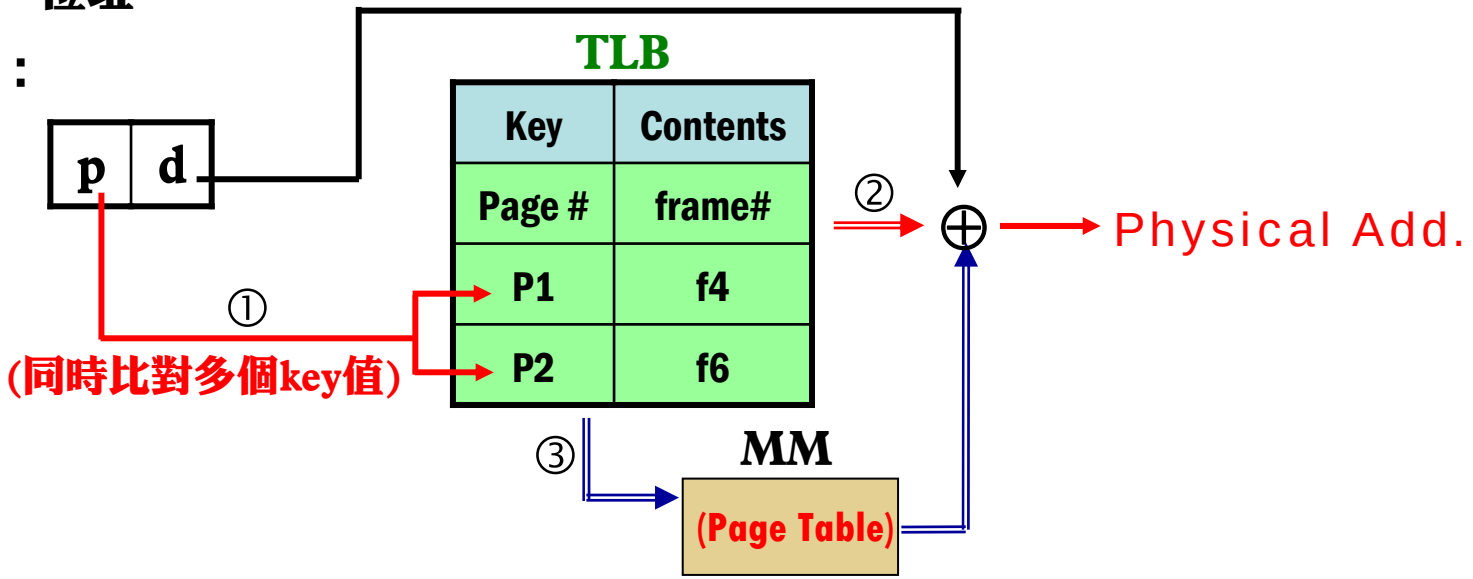
【方法 2】Page Table保存在Memory中，OS利用一個PTBR (Page Table Base Register)來記錄此Page Table在Memory中之起始位址。

- 優點：適用於Page Table Size較大之情況
- 缺點：速度慢。
  - ∵需要兩次的Memory Access (一次用於由Memory取出Page Table，一次用於真正的Data Access)，∴浪費時間。

【方法 3】使用TLB (Transaction Lookaside Buffer)來保存部份 (常用) 的Page Table。完整的Page Table在MM 中。

- 查詢Page Table的過程：
  - ① 首先，到TLB查詢有無對應的Page Number存在
  - ② 若Hit，則輸出Frame的起始位址，速度等同於【方法 1】
  - ③ 若Miss，則到Memory中取出Page Table查詢，以取得Frame的起始位址。

圖示：



- 例：假設
  - TLB存取花費20 ns
  - Memory存取花費100ns
  - TLB Hit Ratio為80%

則有效的記憶體存取時間為何？

Sol:

存取TLB　Data Access　存取TLB　存取 Page Table　Data Access

$$0.8*(20ns+100ns)+(1-0.8)*(20ns+100ns+100ns)$$

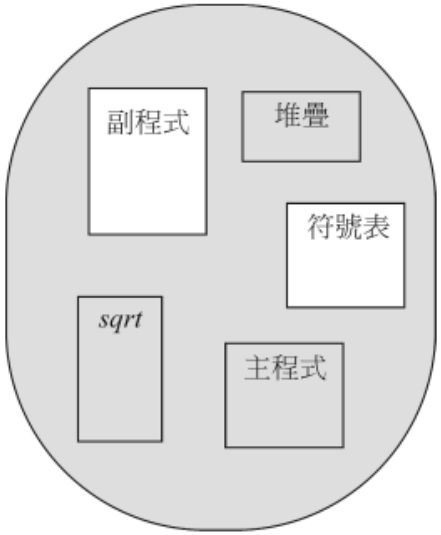
(命中)　(沒命中)

$$=0.8*120ns+0.2*220ns$$

$$=140ns$$

■ Segment Memory Management (分段記憶體管理)

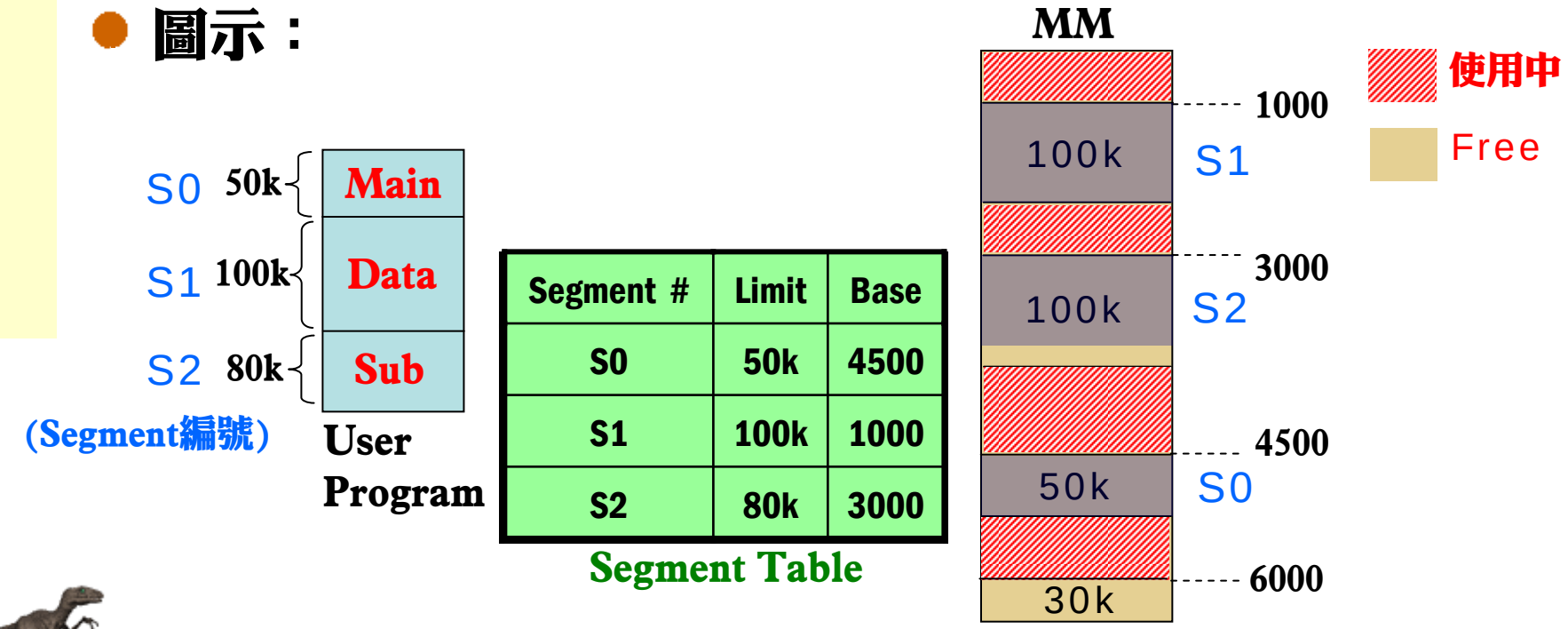
- **實體記憶體 (Physical Memory)**：不需事先區分記憶體空間。若Memory中存在Process所需的連續夠大之Free Memory Space，即將該Space配置出去。
- **邏輯記憶體 (Logical Memory)**：即User Program。視為一組段(Segment)的集合，且各段大小不一。
- 何謂Segment：
  - Main
  - Subroutine
  - Data Section
  - ...
- 分段對Memory的看法與使用者一致。



邏輯位址空間

圖 8.18 使用者對一個程式的看法

- 配置方式：
  - 段與段之間可以採用 “不連續性” 配置。
  - 就 “單一段” 而言，是採連續性配置。
- OS會替每個Process準備Segment Table，來記錄各段的大小 (Limit)及各段載入M.M.的起始位址(Base)。
- 圖示：



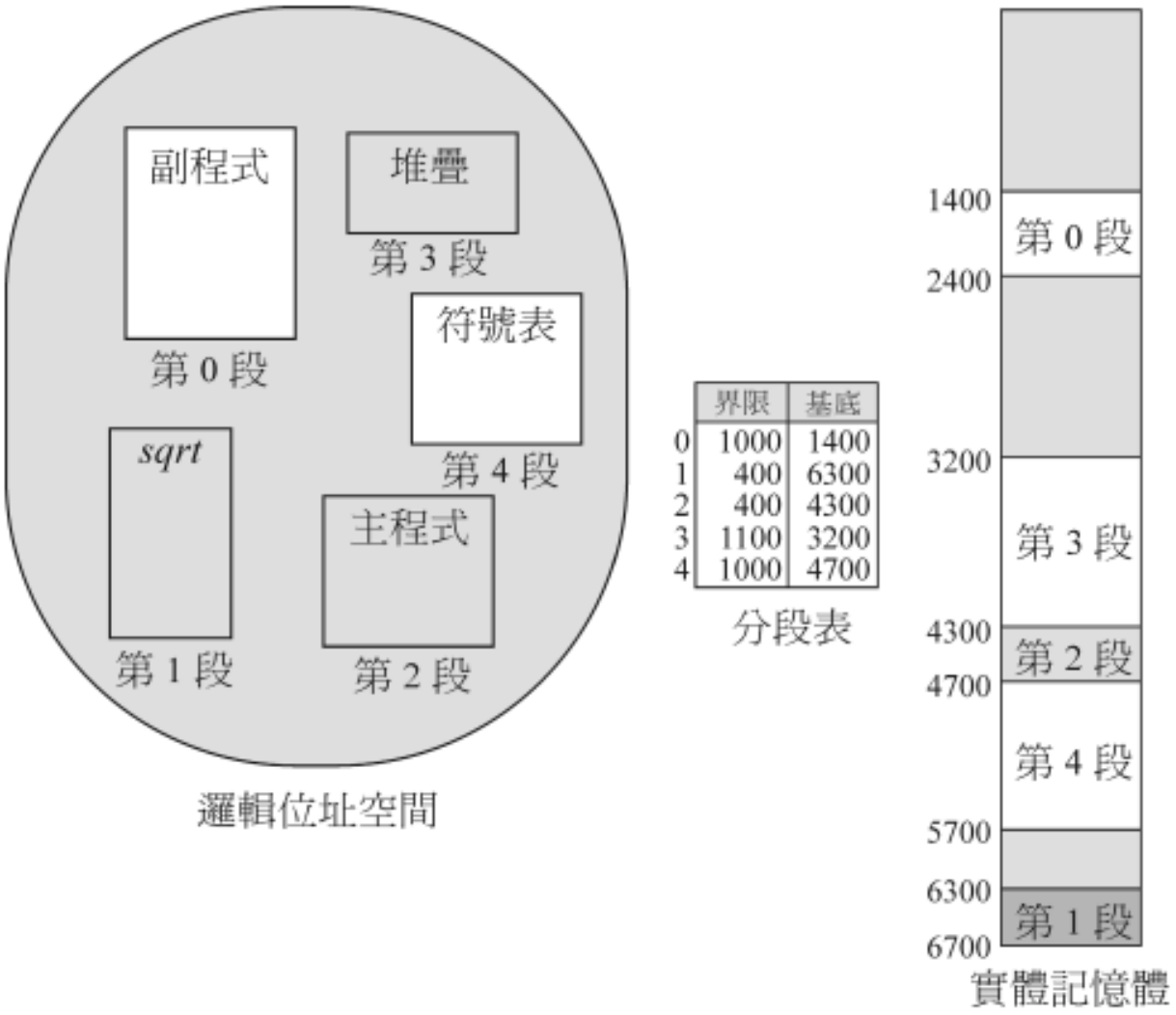
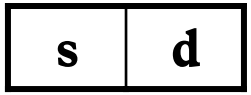

副程式
堆疊
第 3 段
符號表
第 0 段
sqrt
第 4 段
主程式
第 1 段
第 2 段
邏輯位址空間
界限
基底
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
分段表
1400
第 0 段
2400
3200
第 3 段
4300
第 2 段
4700
第 4 段
5700
6300
第 1 段
6700
實體記憶體

圖 8.20 分段法的例子

- Logical Address轉換成Physical Address過程：

① CPU送出兩個值的Logical Address



s: Segment #

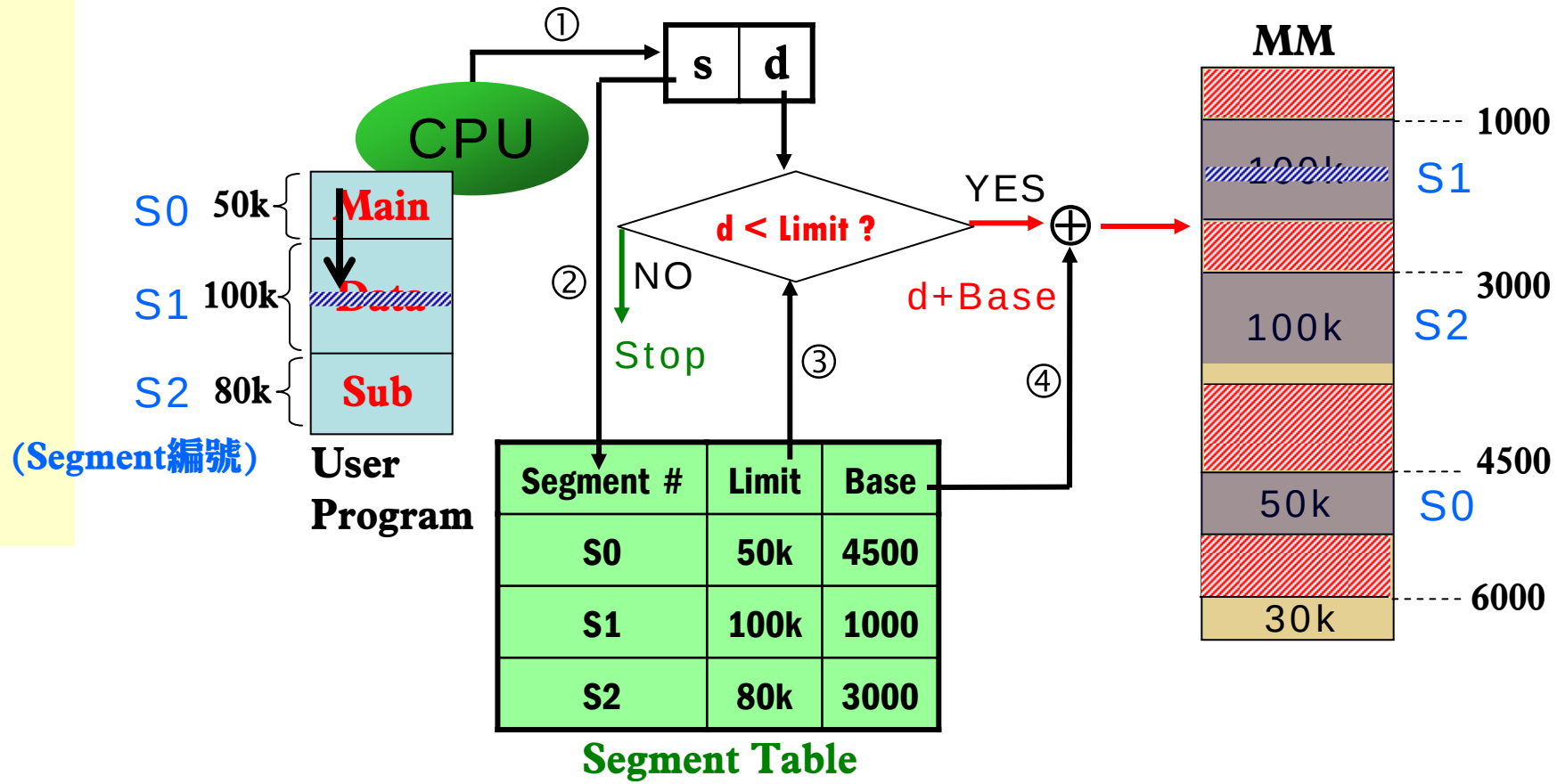
d: Segment offset (段偏移量)

② 根據 s 去查Segment Table，取得該段之Limit

③ 檢查 d < Limit成立，則表示合法存取，取出段的Base，goto ④；否則表示illegal Memory Access，Stop。

④ d + Base即得出Physical Address。

圖示：

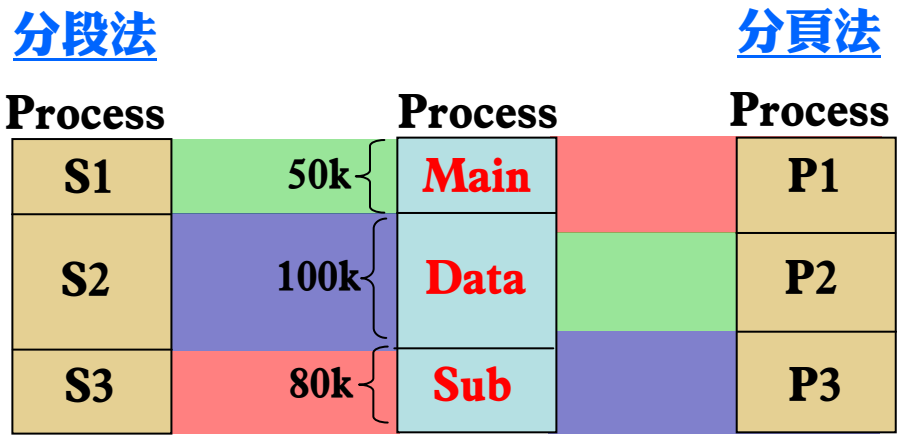

①
s
d
CPU
S0 50k
Main
S1 100k
Data
S2 80k
Sub
(Segment編號)
User Program
②
NO
Stop
d < Limit ?
YES
d+Base
③
④
Segment # | Limit | Base
S0 | 50k | 4500
S1 | 100k | 1000
S2 | 80k | 3000
Segment Table
MM
1000
100k
S1
3000
100k
S2
4500
50k
S0
6000
30k

- 範例：
  - 偏移量 d 以 k 為單位。

<s, d>		Physical Address
<0, 30>	→	4530
<1, 80>	→	1080
<1, 120>	→	Error
<2, 50>	→	3050

- 優點：
  - 沒有Internal Fragmentation問題!!
  - 可支援Memory之Sharing 與Protection。
    - 比分頁法更容易，若Sharing和保護的要求較高時，以分段實作較好。
  - 可支援Dynamic Loading及Virtual Memory的製作。
- 缺點：
  - 可能有External Fragmentation問題。
  - 記憶體存取時間較長。
  - 需要額外硬體的支援。

- 為何分段法比分頁法更容易達成共享與保護？
  - 假設主程式(Main)與副程式(Sub)設成Read-only，Data為R/W。



  - **分頁法**：分頁法要求每個Page Size相同。∴有的Page可能會涵蓋到不同需求的程式片段，保護不易。
  - **分段法**：分段法不要求每個Segment Size相同。∴每個Segment可分別涵蓋不同需求的程式片段，易於保護。

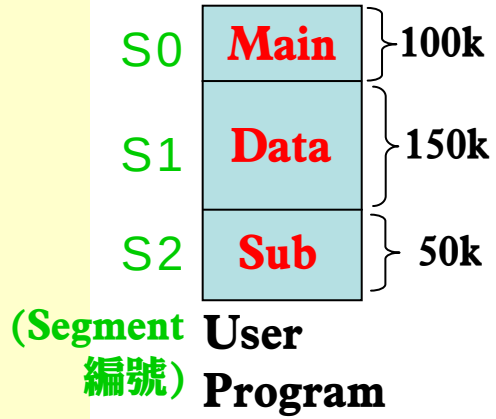
Paging vs. Segment

Paging	Segment
每個Page Size相同	各Segment大小不一定相同
有Internal Fragmentation問題	有External Fragmentation問題
與User對Memory看法不一致	與User對Memory看法一致
Logical Address為單一值	Logical Address為兩個值 (s, d)
無需Check d < Page Size	需Check d < Segment Limit
對Memory Sharing及Protection實作上較為困難	較分頁法容易
分頁表不需存在Page Size	分段表需存在Segment Limit

■Paged Segment Memory Management (分頁式分段)

- 觀念：
  - 段再分頁。
  - User Program由一組Segment所組成，而每個段由一組Page所組成。
  - 每個Process會有一個Segment Table，而每個段會有一個Page Table。
- 目的：
  - 保有分段法的 “與User對Memory看法一致” 及 “Sharing/Protection易實作” 之優點。
  - 避免分段法的External Fragmentation問題。

- 圖示：
  - 設 Page Size = 50k
  - MM視為一組frame的組合。



Segment #	Limit	Page Table 位址
S0	100k	
S1	150k	
S2	50k	

Segment Table

Page Table

Page #	frame#
P0	f1
P1	f3

Page #	frame#
P0	f0
P1	f2
P2	f6

Page #	frame#
P0	f5

MM

S1.P0	f0
S0.P0	f1
S1.P1	f2
S0.P1	f3
	f4
S2.P0	f5
S1.P2	f6

使用中

- **Logical Address轉換成Physical Address過程：**

① CPU送出兩個值的Logical Address (同Segment)

s	d

② 根據 s 去查Segment Table，取得該段之Limit

③ 檢查 d < Limit成立，則表示合法存取，goto ④；否則表示illegal Memory Access，Stop。

④ 取出相對應的Page Table。

⑤ 將 d 拆成兩個值：p 與 d' (p: Page #, d': Page offset)

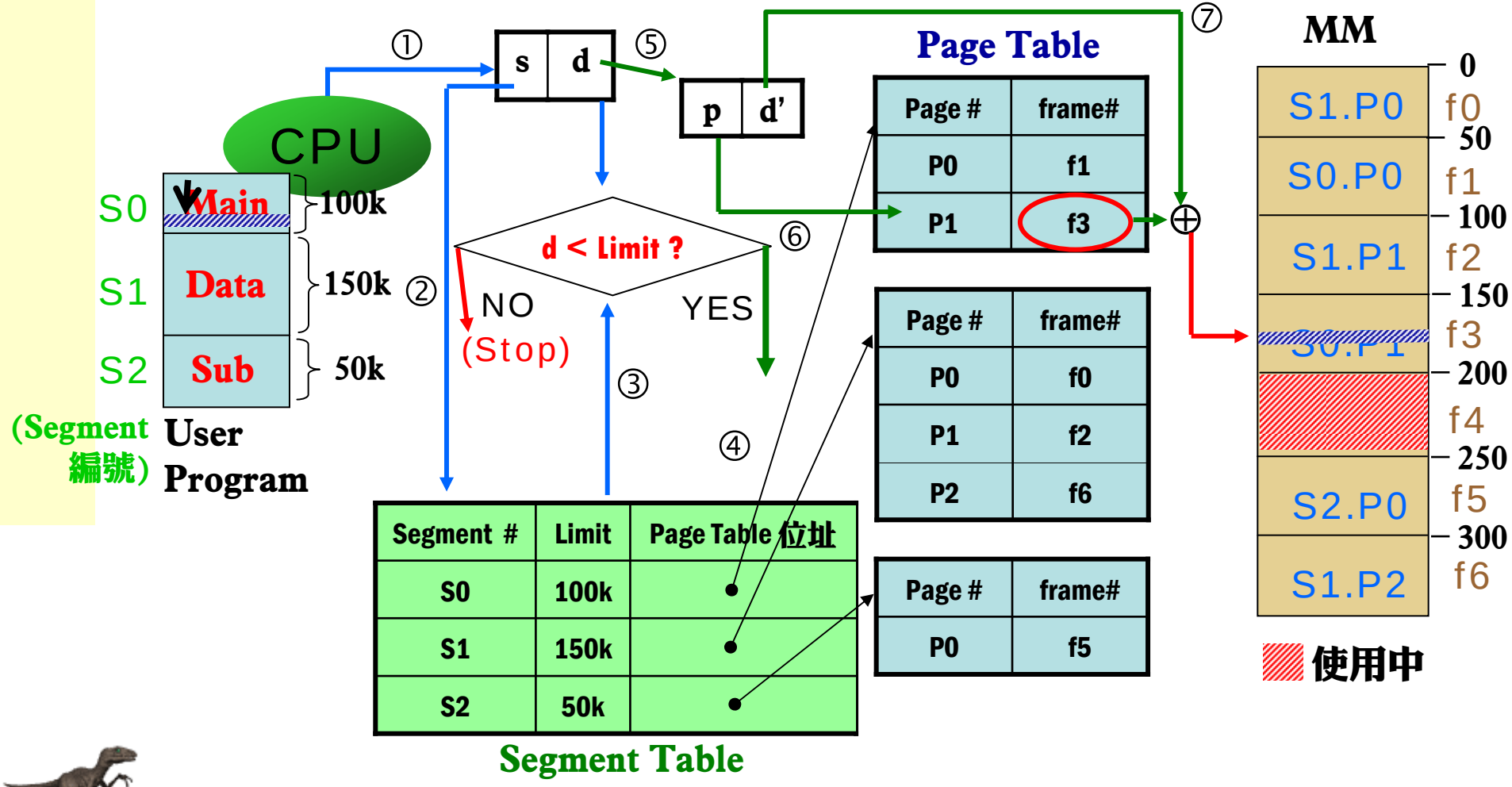
- 即：d 除以Page Size，所得到的商為 p，餘數為 d'

⑥ 依據 p 到Page Table查詢，取得frame的起始位址 f

⑦ f + d' 即得Physical Address

圖示：

- 設 Page Size = 50k，MM視為一組frame的組合。



- 範例：
  - 偏移量 d 以 k 為單位。

<s, d>		Physical Address
<1, 70>	→	120
<2, 80>	→	Error
<0, 80>	→	180

- 分析
  - 沒有External Fragmentation Problem
  - 仍有Internal Fragmentation Problem (∵最終仍是分頁)
  - Table數目太多，極佔空間
  - Memory Access Time更長

補　充

■分頁記憶體管理相關計算題

- Page Size為1024 bytes，User Program至多8 pages，Physical Memory有32個frames，求Logical Address和Physical Address各佔多少bits？

Ans:

- Logical Address: | p | d |

  ∵程式最多8個page，∴ p (Page #) 佔3個bits

  ∵Page Size = 1024 = 2^{10}，∴ d (Page offset) 佔10個bits

  故，共13個bits。

- Physical Address: | f | d |

  ∵記憶體最多32個page，∴ f (frame #) 佔5個bits

  ∵frame size = Page Size = 2^{10}，∴ d 佔10個bits

  故，共15個bits。

- Page Size = 256bytes，若User Program大小為16MB，且Page Table Entry (即：Page Table的每一格)佔4bytes，則其Page Table Size為何？

Ans:

- 此Program有 16MB/256B = 2^{24} / 2^{8} = 2^{16} 個Pages。
- Page Table Size = 2^{16} × 4 bytes = 2^{18} bytes = 256kB。