CH7、Memory

記憶體管理

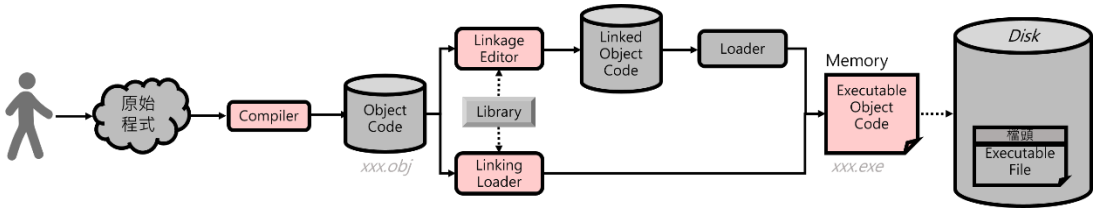
目錄：

Binding

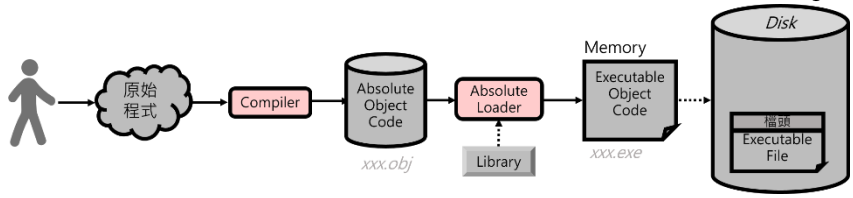
一、Def：決定程式/process 執行的起始位址，此一動作稱之

二、時機點：

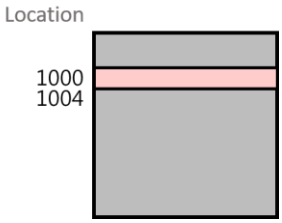
1. Compiling Time：由 Compipler 作 Binding
2. Loading Time 或 Linking Loading Time：Linking Loader 或 Linking Editor
3. Execution Time：由 OS 動態決定，也叫作 Dynamic Binding



1. Compiler 作 Binding：產生之 object code 叫作"Absolute" object code。後面的 loader 叫作"Absolute" loader，主要是作 Allocation 與 Loading only。



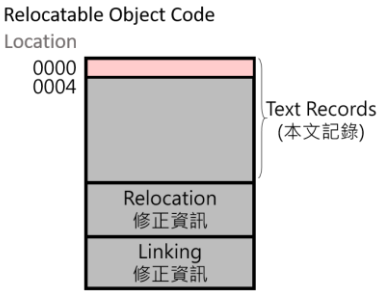
其程式碼起始位址亦為實際 Memory 位置：



缺點：Process 若要改變起始位址，則必須 Re-Compiling，非常不便

(Note：通常用於.com 命令檔)

2. Loading Time 由 Linking Loader 作 Binding：Compiler 所產生的 object code 叫作"Relocatable" object code(可重新定位之目的碼)



(1) 何謂 Relocation 修正？

當程式起始位址改變，某些目的碼的內容必須隨之修正，才能正確執行。

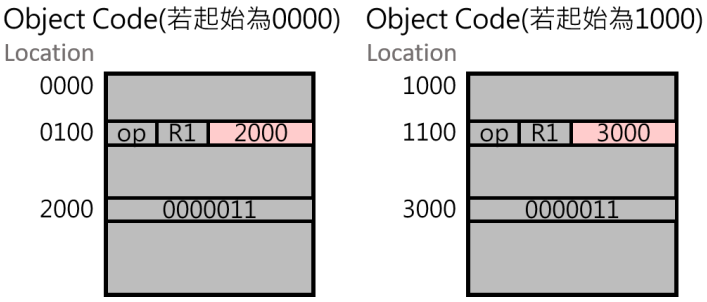
例：採用"直接"定址(Direct Addressing Mode)指令

(CA 的內容，但 CH1 所教的是另外五種定址模式，有間接定址、但並沒有直接定址)

原始程式：

```
Add  R1,   x
x    DB,   "3"
```

則依不同起始位址，會需要修正不同位址

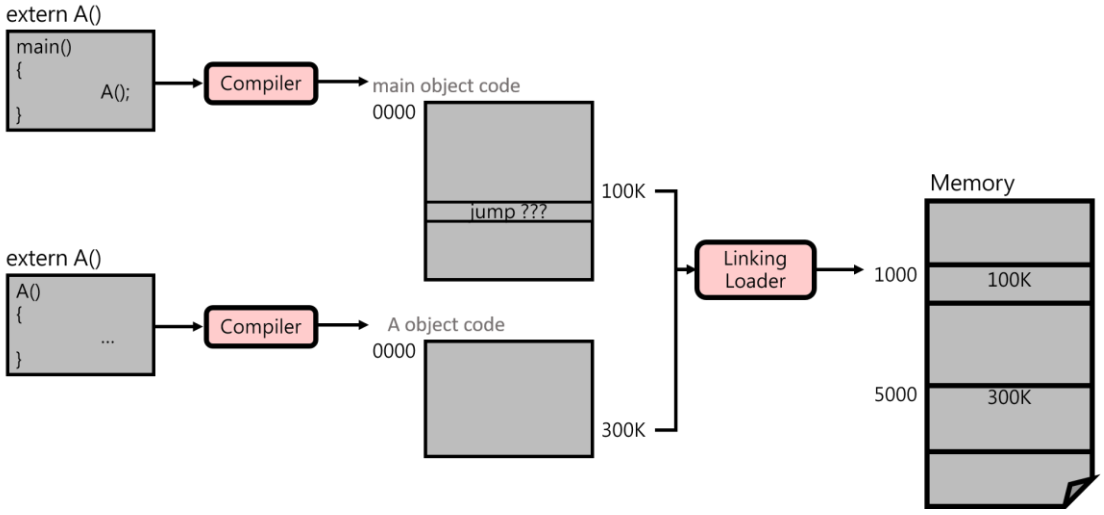


重新定位修正：2000+當時起始位址

(2) 何謂 Linking 修正？

解決 External Symbal Reference 之修正

例：外部符號*(ex：副程式名稱)*、外部變數*(extern)*、Library…等



(3) Linking Loader 主要 4 個工作

I. Allocation：向 OS 要求起始位址

II. Loading：object code 載入到 Memory

III. Linking：依 Compiler 所交辦之 Linking 修正資訊，執行 Linking 修正(???改成 8000)

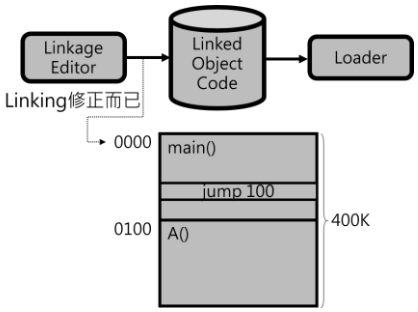
IV. Relocation：作重定位修正

優點：程式的起始位址若要改變，則無需重新 Relocation，直接 Linking 即可

缺點：1.程式重新執行，若 modules 數多，則 Re-linking 很花時間

2.Process 執行期間，不可更改起始位址

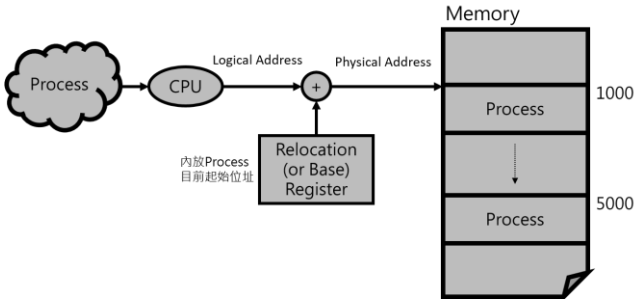
Note：凡是"Static" Binding，皆無法更改



Dynamic Binding

一、Def：決定 Process 起始位址之工作，延到執行時期(Execution Time)才動態執行，即 Process 在執行期間，可任意變更起始位址，且 Process 仍能正確執行

二、需要硬體支援



1. Logical Address：generated by CPU
2. Physcial Address：實際去 Physical Memory(RAM)存取之位址
3. Logical Address = Physical Address，稱為 Static Binding
   Logical Address !=Physical Address，稱為 Dynamic Binding(Page, Segment, Paged Segment)

   若有 Logical Address 轉成 Physical Address 之運作，皆由硬體負責，而該單元叫 MMU(Memory Management Unit)

三、優點： 1.Process 之起始位址可於執行時間任意更動，且能正常執行，有助於 OS 記憶體管理之彈性度*(ex：Compaction 實施，Process swap out 後再 swap in，不一定要相同起始位址)*

缺點： 1.需要硬體支援

2.Process 執行時間較久，效益較差

Dynamic Loading

一、Def：也叫"Load-on-call"，在執行時間，若 module 真正被呼叫到且不在 Memory 中，此時 loader 才將它載入到 Memory 中

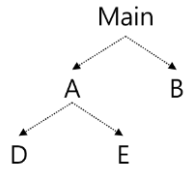
二、優點：節省 Memory Space*(不需要 OS 之額外支援)*

缺點：Process 執行時間較久

Note：(補充)

早期使用"Overlay"技巧，是 Programmer 的責任，OS 無需負責

近期則是 OS 提供 Virtual Memory



Dynamic Linking

一、Def：在 Execution Time，若 Module 被呼叫到，才將之載入，並且與其他 Modules 進行 Linking 修正(外部符號之解決)，適用在 Library Linking *(ex：Dynamic Linking Library, dll)*

二、優點：節省不必要之 Linking Time

(Process A 需要的函數，可能在 Process B 中，故需要 OS 額外支援)

缺點：Process 執行時間較久

Contiguous Memory Allocation(連續性)

(也叫作 Dynamic Varible Partition Memory Mangement 動態變動分區記憶體管理)

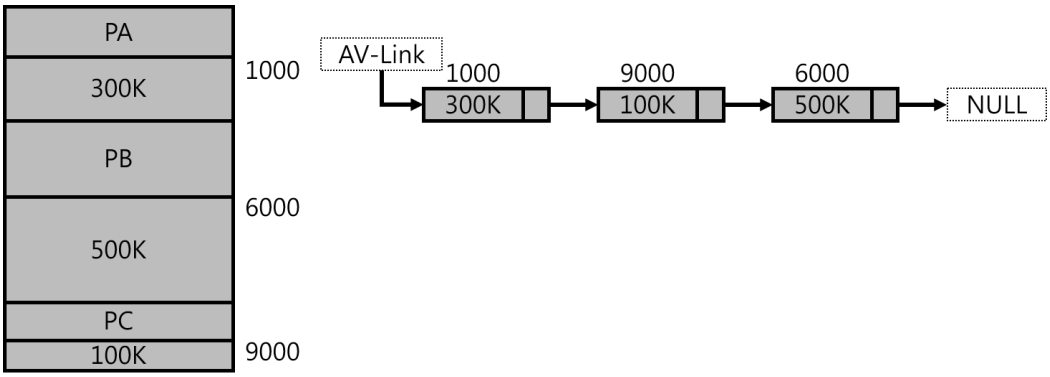
一、OS 必須配置 process 一個連續的 Free Memory Space

二、Partition：

1. process 所佔用的 Memory Space
2. Partition 數目 = process 數目 = multiprogramming degree，由於不同時期，系統內 process 數目不固定，所以 Partition 數目不固定*(Dynamic)*
3. 由於各個 process size 不盡相同，所以各 Partition 就不一定相同*(Variable)*

三、Memory 中會有一些 Free Memory Space (or Block)叫 Hole，通常 OS 會用 Linked-List 觀念管理這些 Holes，叫作 AV-List *(Available List，可用空間串列)*

例：

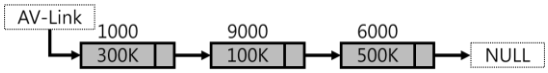


四、配置方式：

1. First Fit
2. Best Fit
3. Worst Fit

1. First Fit：從 AV-List 頭開始找起，直到找到第一個 hole，其 hole size>=process size 為止，即可配置、或找完整條串列，無一夠大為止
2. Best Fit：必須檢查 AV-List 所有 holes，找出一個 hole，其 hole size >= process size，且 hole size 減去 process size 後，差值"最小"的 hole，予以配置
3. Worst Fit：必須檢查 AV-List 所有 holes，找出一個 hole，其 hole size >= process size，且 hole size 減去 process size 後，差值"最大"的 hole，予以配置

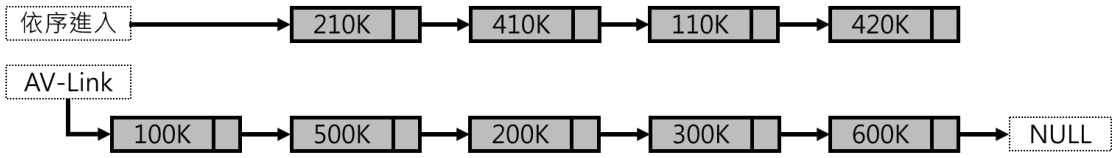
例 1：若 Process=90k，依序 ABC 三個 Block



1. First-Fit 會配置給__Block 之 90K 給 Process，剩下__K
2. Best-Fit 會配置給__Block 之 90K 給 Process，剩下__K
3. Worst-Fit 會配置給__Block 之 90K 給 Process，剩下__K

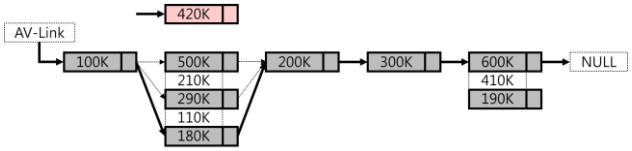
1. A, 210K
2. B, 10K
3. C, 410K

例 2：First/Best/Worst 何者最佳？

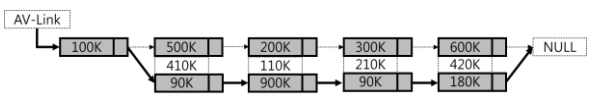


Best Fit 最佳

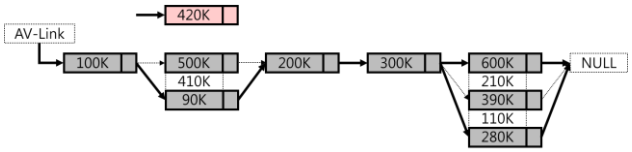
First：



Best：



Worst：



比較表：

	時間效率	空間利用度
First Fit	最佳	近乎”佳”
Best Fit	差	佳
Worst Fit	差	差

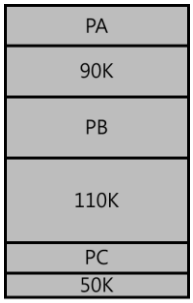
註：整體上來說，First Fit 效能最佳

External Fragmentation(**外部碎裂**)

上述 Contiguous Memory Allocation 方法，均遭遇一個共同問題”外部碎裂”

Def：在 Contiguous Allocation 要求下，目前 AV-List 中任何一個 hole size 均小於 Process size，但這些 Holes size 加總卻>=Process size。然而，因為這些 Holes 並不連續，因此仍無法配置給 Processs，造成空間閒置不用，Memory 利用度低之問題

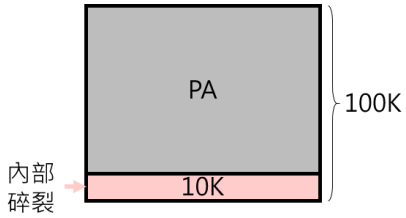
例：若 Process 大小為 220K，這些 Holes 加總 = 250K >= 220K，但這些 Holes 不連續，因此仍無法配置



Note：[恐](補充)(不會考)
一般而言，每配置 N 個大小，平均會有 0.5N 的外部碎裂，因此外部碎裂的比例 = 0.5N/(0.5N+1N) = 1/3。由此可知：外部碎裂是嚴重的問題

Internal Fragmentation(**內部碎裂**)

Def：配置給 Process 之空間，超過 Process size，兩者之差值空間，此 Process 使用不到，其他 Process 亦無法使用，是記憶體空間的浪費

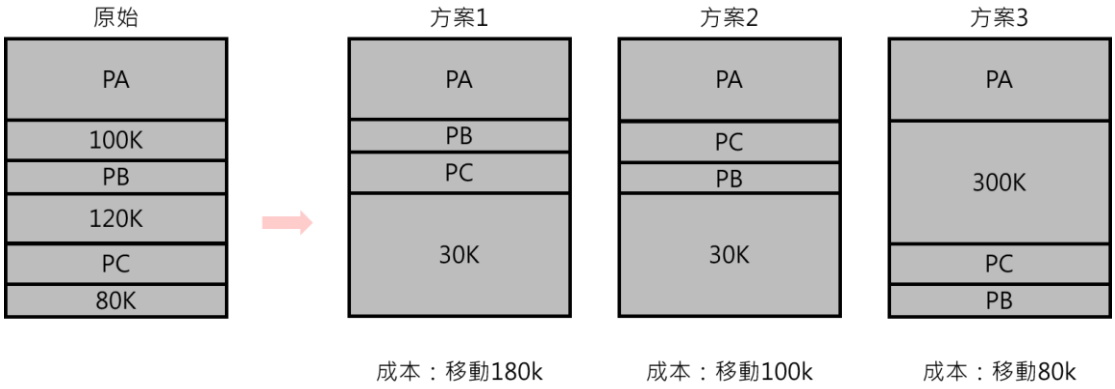


解決外部碎裂之方法

[法一]使用 Compaction(聚集)技術

一、作法：移動執行中的 Process，使得原本非連續的 Holes，得以聚集形成一個夠大的連續 Free Memory Space

二、例：



三、困難點：1.不易制定最佳的 Compaction 策略

2.Process 必須是 Dynamic Binding 才可於執行期間移動

[法二]：使用 Page Memory Management*(後續會提及)*

[法三]*(較不正統)*：將 process 折成 Code Section 與 Data Section 兩部分，分開配置給 hole，以降低外部碎裂發生的機率。因此每個 process 需至少 2 套 Base/Limit Register，分別記錄 Code Section 與 Data Section 的起始位址與大小。[恐]稱之為 Multiple Base/ Limit Register 方法

Page Memory Management(分頁法)

一、Physical Memory(RAM)：視為一組 Frame(頁框)之集合，且各 Frame size 相同。*Note：Frame 是硬體決定，OS 只是配合，分頁是採 Physical Viewpoint*

二、Logical Memory(即 user process 大小)：視為一組 Page(頁面)之集合，且 Page size = Frame size

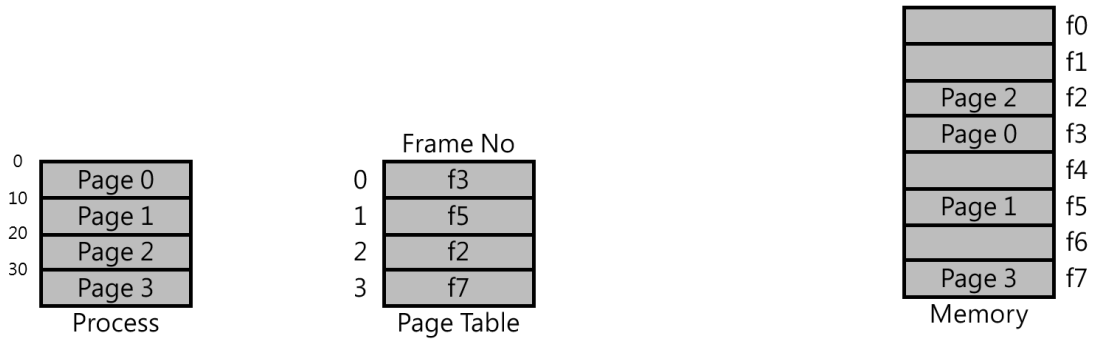
三、配置方式：OS 採"非連續性"配置原則，即若 Process 大小 = n 個 Pages，則 OS 只需依 Physical Memory，挑出 n 個 Free Frames(不一定要連續)，就可以配置給此 Process

四、OS 會替每個 process 建立一個 Page Table(分頁表)，記錄各個 process 配置於哪個 Frame 之 Frame Number，若 Process 大小=n 個 Pages，則它的 Page Table 就會有 n 個 entry

Note：(CH4) Page Table is stored in PCB(6.Memory Mangement Infomation)

1. Process ID	*3. PC*	*5. Scheduler Info*	*7. Accounting info*
2. Process State	*4. CPU Register*	*6. Memory Management info(O)*	*8. IO status info*

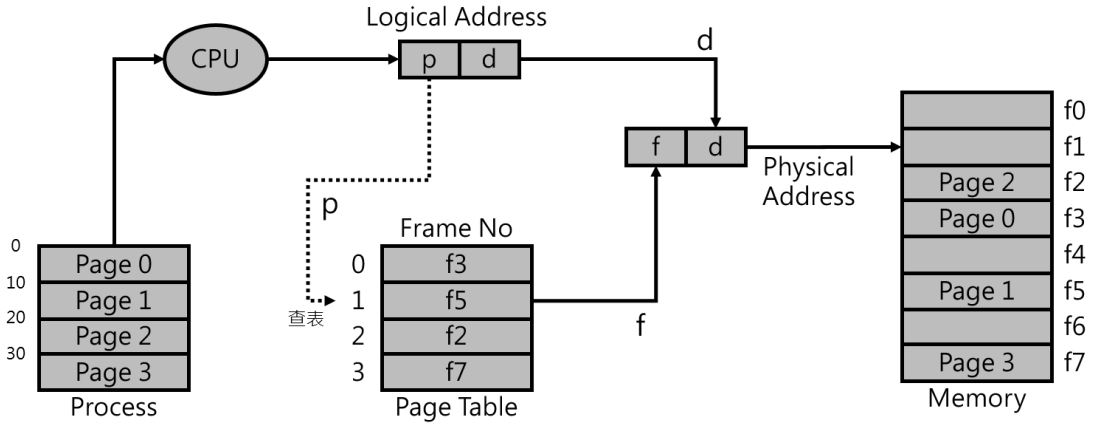
五、圖示



六、Logical Address 轉 Physical address 過程 by MMU(硬體)

假設 Page size = 10

1. Logical Address 初始是單一量，自動拆解成(p, d)，其中 p 代表 Page Number、d 代表 Page offset(頁偏移)
2. 依 p 查詢 Page Table，取得該 Page 的 Frame Number，令為 f
3. f 與 d 合成(f, d)、或 f*Page size + d，即為 Physical Address
   ex：(p, d) = (3, 2)，依 p=3 查表，其 Frame Number = f7 => (7, 2) = Physical Address = 7*10+2 = 72



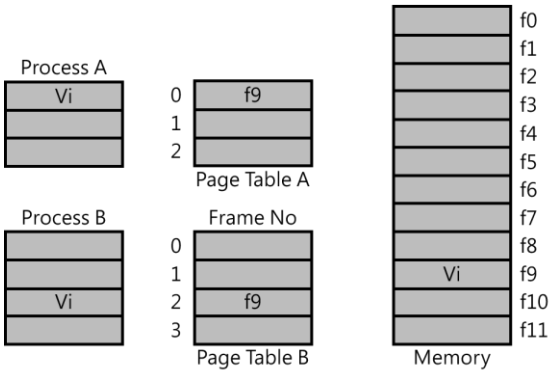
例 2：Logical Address = 17 => (p, d) = (1, 7) => Physical = (5, 7) = 57

七、優點：　1.沒有外部碎裂
　　2.可支援 Memory Sharing 及 Memory Protection 之實施
　　3.可支援 Dynamic loading、Linking 及 Virtual Memory 之實現

Memory Shared(共享)

若多個 Process 彼此具有共享的 Read-only Code/Data Pages，則我們可以藉由 Process 各自的 Page Table 共同 Pages 映射到同一頁框，如此可節省 Memory Space

例：Vi：Read-only Code



Memory Protect(保護)

在 Page Table 中，多加一欄位"Protection Bit"，其值為 R 表 Page 只能 Read-only；W 表 Page 可以讀取與寫入

缺點： 1.有內部碎裂，因為 Process 大小不見得是 Page size 之整數倍

例：Page size = 10K，Process 大小= 32K

因此需配置 4 個 Frames，因此內部碎裂 = 40-32 = 8K

Note：若 Page size 愈大，則內部碎裂的情況愈嚴重；內部碎裂最大空間 = Page size -1

2.需要額外硬體支援。

例：Page Table 之製作、Logical Address 轉 Physical Address by MMU

3.Effective Memory Access Time 較長(than Contiguous)，因為有 Logical Address 轉 Physical Address

Page Table 的製作(保存)

[法一]：使用 Register 保存 Page Table 中每個 entry 內容

優點：查詢 Page Table 無需 Memory Access，速度最快

缺點：不適合用於大型的 Page Table(或大型的 Process)，因為數量有限

[法二]使用 Memory 保存分頁表，且用一個 Register：PTBR(Page Table Base Register)記錄在 Memory 中之位址；或者也有 PTLR(Page Table Length(size) Register)來記錄 Page Table 大小

優點：適用於大型分頁表

缺點：額外一次 Memory Access 來存取 Page Table

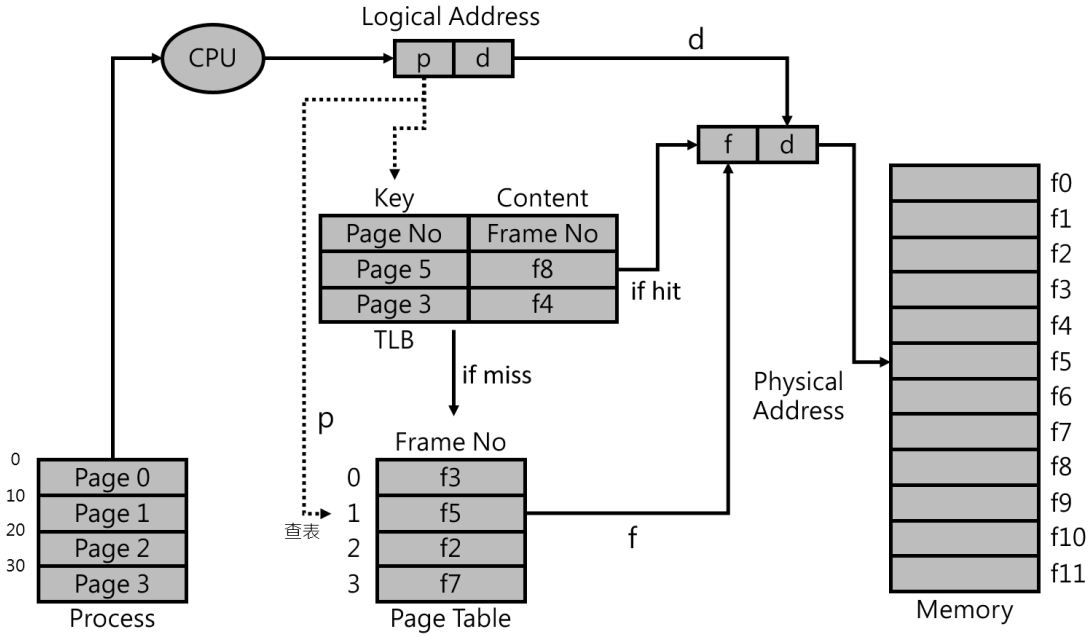
[法三]使用 TLB(Translation Lookaside Buffer)

(或叫關聯式暫存器 Associative Register)

保存 Page Table 中經常被存取之 Page Number 及 Frame Number，且完整的 Page Table 存於 Memory 中

使用 TLB 之 Effective Memory Access Time =

p*(TLB Time + Memory Access Time) + (1-p)*(TLB Time + 2*Memory Access Time),
where p is TLB hit ratio



例：

Register Access Time = 0ns (ignored)

Memory Access Time = 200ns

TLB Time = 100ns

TLB hit ratio = 90%

求 Effective Memory Access Time, if Page Table is stored using：

1. Register
2. Memory
3. TLB

1. *200ns*
2. *400ns*
3. *0.9(100+200) + 0.1(100+2*200) = 320ns*

Paging 相關計算

[型一]使用 TLB 之 Effective Memory Access Time

[型二]計算 Logical Address 與 Physical Address Bit 數目

例：

Page size = 1KB

Process 最大有 8 個 Page

Physical Memory 有 32 Frame

求：1.Logical Address Length、2.Physical Address Length

1. *Logical Address：(p, d)*
   因為 Page size=1KB=2^{10} bytes，所以 d 佔 10bits
   因為 Process 最多 8 個 Page，所以 p 佔 3 個 bits，因此 3+10=13 bits
2. *Physical Address：(f, d)*
   因為 Physical Memory 有 32 個 Frame，所以 f 佔 5 個 bits，因此 5+10=15 bits

[型三]：Page Table size 相關計算

例 1：

Page size=8KB
Process 大小=2MB
Page Table entry 佔 4 bytes
求此 Process 的 Page Table size ?

Process 大小= 2MB/8KB = 2^8 個 Pages
*所以 Page Table 有 2^8 個 entry => size = 2^8*4 bytes = 1KB*

例 2：

Logical Address = 32 bits
Page size = 16KB
Page Table entry 佔 4 bytes
求 MAX Page Table size ?

p+d=32 bits
Page size = 16KB = 2^{14} bytes
因為 d 佔 14 bits
所以 p 佔 32-14 = 18 bits
Process 最大可有 2^{18} 個 Pages
*因此 MAX Page Table size = 2^{18} 個 entry*4 bytes = 1MB*

EX：承上，若 Logical Address = 48 bits 呢 ?

Page 佔 48-14 = 34 bits
*所以 MAX Page Table size = 2^{34} * 4 bytes = 64GB*

Note：Page Table size 太大是個議題，後面會提到

例 3：

Page size = 16KB
Page Table entry 佔 4 bytes
MAX Page Table size 恰為 one page
求 Logical Address Length ?

Page size = 16 KB = 2^{14} bytes
d 佔 14 bits
因此 MAX Page Table size = one page = 16KB
MAX Page Table = 16KB/4bytes = 2^{12} entry
因此 P 佔 12 bits => 12+14 = 26 bits

[型四]Page Table size 太大之解法的相關計算(後緒提到)

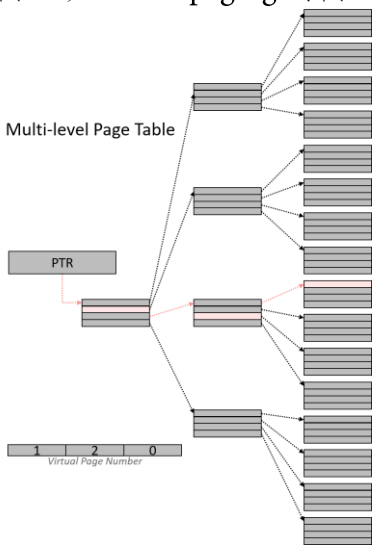
Page Table 太大之解決方案

1. Multilevel Paging (Herarchical Pagin, Paging the Page Table, Forward Mapping)
2. Hashing Page Table
3. Inverted Page Table

Multilevel Paging(多層分頁表)

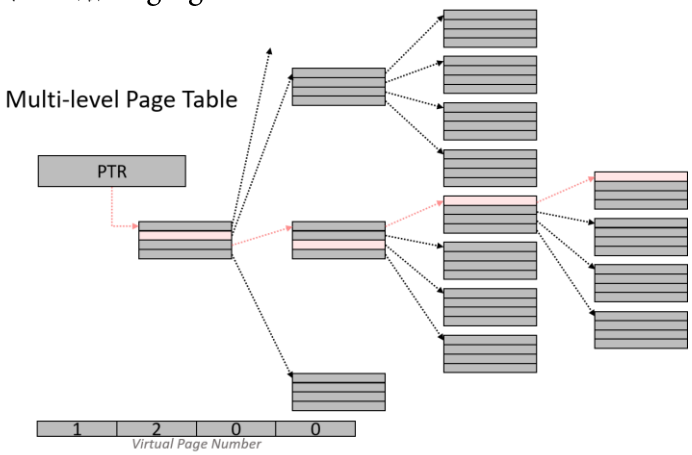
一、Def：藉由此法，不需將整個 Page Table 全部載入 Memory 中，而是載入部分所需要的內容，因此提出多層次的分頁作法

例：以 2-level paging 為例



Level 1 Page Tabe：有 2^x 個 entry，每個 entry 記錄某個 Level 2 Page Table 之 pointer (address)
Level 2 Page Table：有 2^y 個 entry，每個 entry 記錄 Frame Number
Process 在執行時，只需 1 個 Level 1 Page Table 與一個 Level 2 Page Table 在 Memory 中即可。因此可以大幅減少 Page Table 佔用 Memory Space

例：3 層 Paging



缺點：Effective Memory Access Time 更久，因為需要多次 Memory Access

例：2-Level Paging：整個過程有__次 Memory Access
3-Level Paging：整個過程有__次 Memory Access

3、4

二、相關計算

例 1：

TLB Time = 100ns
TLB hit ratio = 80%
Memory Access Time = 200ns
2-level paging 採用，求 Effective Memory Access Time？

*0.8(100+200) + 0.2(100+3*200) = 380ns*

例 2：

Logical Address = 32 bits
Page size = 4KB
Page Table entry 佔 4 bytes
求 MAX Page Table size？
1. 1-level Paging
2. 2-level Paging(level 1 與 level 2 bits 數相等)

1. *(p , d)*
   d = 12
   p = 32-12 = 20 bits
   *MAX Page Table size = 2^{20} * 4 bytes = 4MB*

2. *(p1, p2, d)，又 level 1 與 level 2 bits 數相等*
   d = 12
   p = 32-12 = 20 bits => p1 = p2 = 20/2 = 10 bits
   *1 個 level 1 Page Table MAX size = 2^{10} * 4 bytes = 4KB*
   *1 個 level 2 Page Table MAX size = 2^{10} * 4 bytes = 4KB*
   MAX Page Table size = 4+4 8KB in the memory

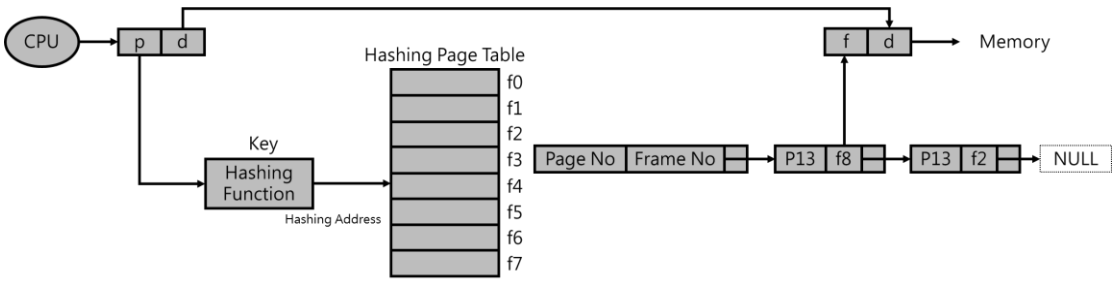
例 3：

Logical Address = 64 bits
Page size = 16KB
Page Table entry 佔 4 bytes
任一 level paging 之 MAX Page Table size 至多 one page(常識)，則至少分幾層？

(p1, p2, …, d) = 64，又 Page size = 16KB = 2^{14} Bits
故 p1+p2+… = 64-14 = 50 bits
因為任一層之 Page Table 至多 16KB/4 bytes = 2^{12}
因此[50/12] = 5 層(2, 12, 12, 12, 12, 14) //inner 最大

Hashing Page Table(雜湊分頁表)

一、Def：利用 Hashing 技巧，將 Page Table 視為 Hash Table 且有相同的 Hashing Address 的 Page Number，及它的 Frame Number 資訊，會置入於同一個 entry(Bucket)中，且以 Linked-List(chain)串接



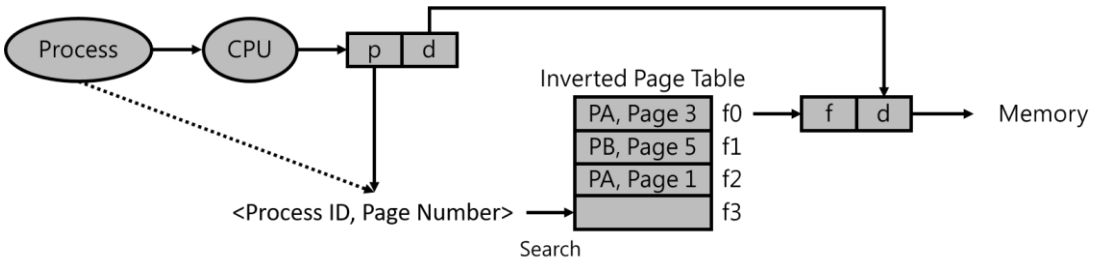
缺點：使用 Linear Search 在 Linked-List 中找符合的 Page No，較為耗時

二、例：H(x) = x%53，Page Table entry 佔 4 bytes，求 Hashing Page Table size？

*有 53 個 entry，所以：53*4 = 212 bytes*

Inverted Page Table(反轉分頁表)

一、Def：是以 Physical Memory(RAM)為記錄對象，並非以 Process 為對象，即若有 n 個 Frames，則此表就有 n 個 entry，每個 entry 記錄<Process ID, Page Number>，配對資訊，代表此 Frame 存放的是哪個 Process 的哪個 Page，整個系統只有一份表格



缺點：
1.必須使用<Process ID, Page Number>資訊一一比對查詢，此舉甚為耗時
2.喪失了"Memory Sharing"之好處，即無法支援其實現

二、例：
Physical Memory = 16GB
Page Table entry 佔 4 bytes
求 Inverted Page Table size？

Inverted Page Table 有 16GB/8KB = 2^{21} 個 Frame
*因此 size = 2^{21} * 4 bytes = 8MB*

Segment Memory Management(分段)

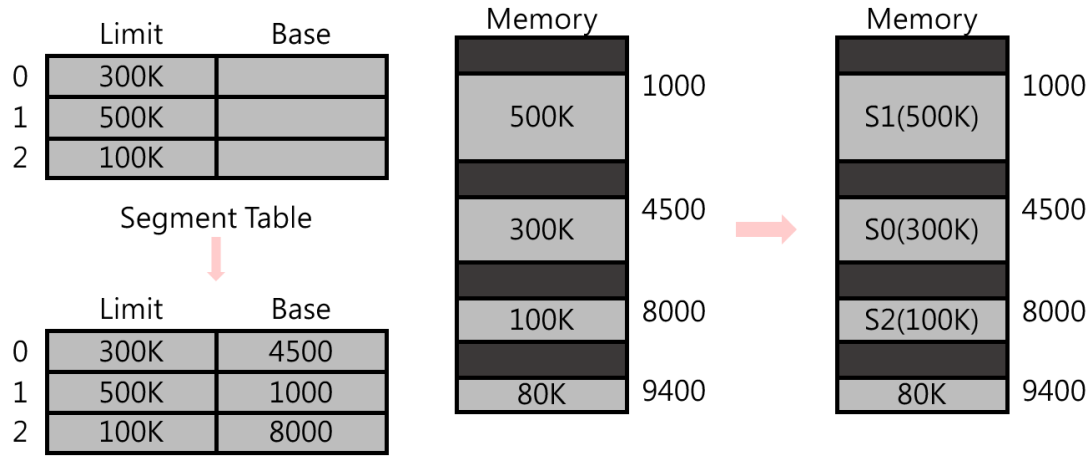
一、Physical Memory 視為一個夠大的連續可用空間

二、Logical Memory(process)視為一組 Segment(分段)之集合，且各段大小不一定相同，段的觀點是採用 Logical viewpoint，與 user 對 Memory 之看法一致
例：Code Segment, Data Segment, Stack Segment…等

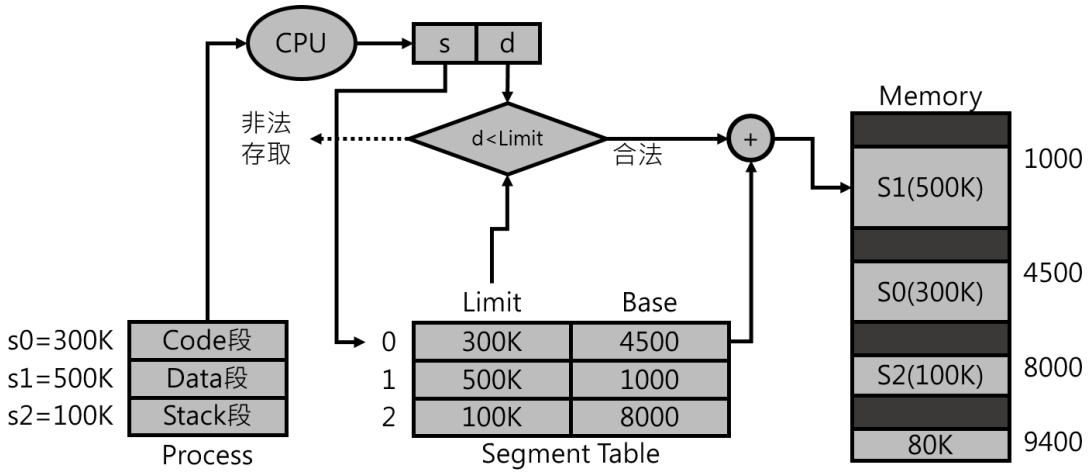
三、配置原則：
1. 段與段之間可以是非連續性配置
2. 但對每一個段而言，必須使用連續的空間

四、OS 會為每一個 Process 建立分段表(Segment Table)，記錄敏個段的 Limit 及 Base(起始位址)

五、圖示：



六、Logical Address 轉 Physical Address：
1. Logical Address initally 是 2 個量(s, d)，其中 s 表段編號；d 表段偏移量
2. 依 s 查分段表，取得其 limit 與 base
3. Check d 是否<limit，若成立，則合法：Physical Address = Base + d
若不成立，則為非法存取



例：分段表如下：求下列 Logical Address 之 Physcial Address(s, d)？

	Limit	Base
0	100	4200
1	500	80
2	830	7300
3	940	1000

1. (0, 90)
2. (1, 380)
3. (2, 900)
4. (3, 940)

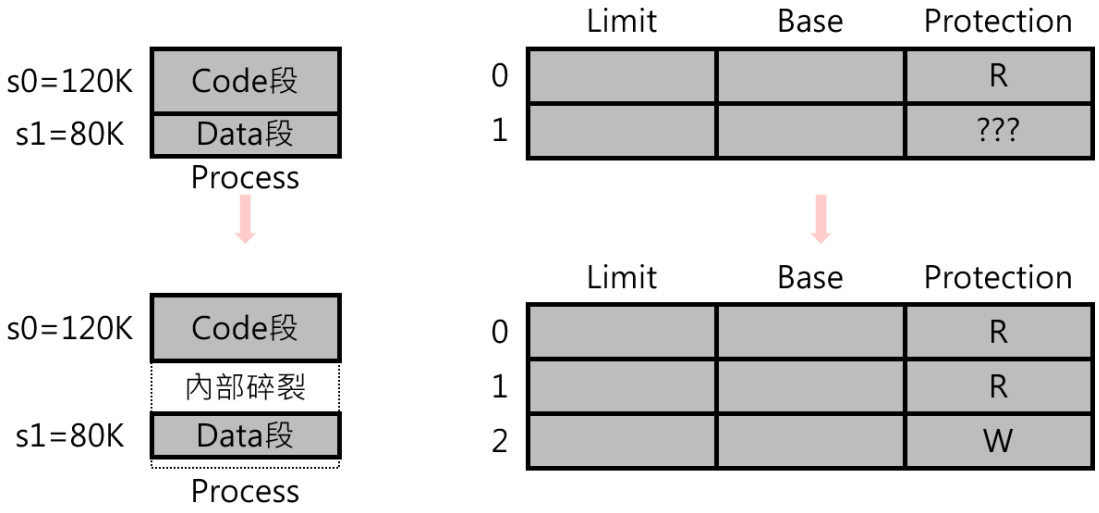
1. *因為 90<100，故合法：4200+90 = 4290*
2. *因為 380<500，故合法：80+380 = 460*
3. *因為 900 !<830，故非法存取*
4. *940 !<940，故非法存取*

七、優點：　1.沒有內部碎裂

2.可支持 Memory Sharing 及 Memory Protection 且比 Page 容易實施，因為分段採用 Logical Viewpoint

3.可支持 Dynamic Loading, Linking 及 Virtual Memory

(優點 2)例：Protection 為例：(分頁 size=100K)



缺點：

1. 有外部碎裂
2. 需有額外硬體支援*(ex：分段表保存，Logical Address 轉 Physical Address)*
3. Effective Memory Access Time 更久

比較表

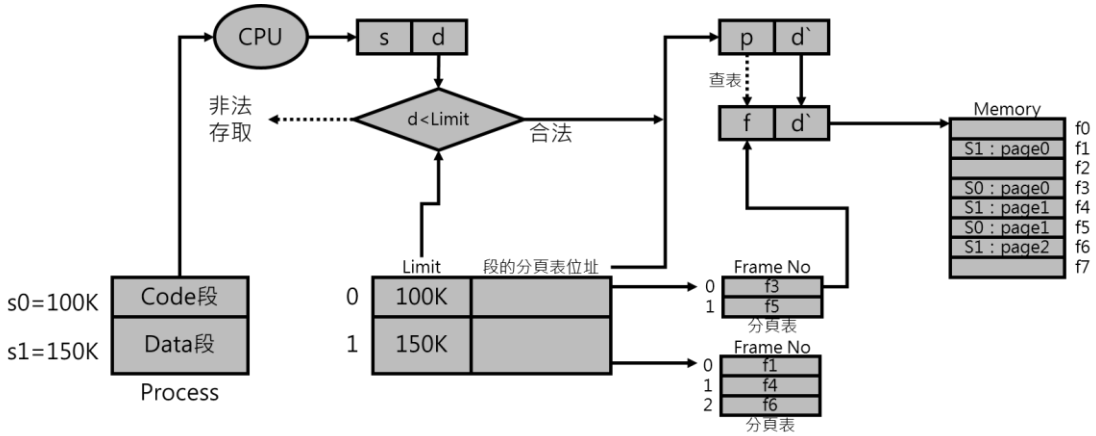
Page	Segment
各分頁大小相同	各分段大小不一定相同
採用 Physical Viewpoint	採用 Logical Viewpoint
無外部碎裂	有外部碎裂
有內部碎裂	無內部碎裂
Memory Protect 及 Sharing 較難實施	Memory Protect 及 Sharing 較易實施
無需 check page offset < page size	需要檢查 offset<分段大小
Logical Address initially 單一量	Logical Address initially 2 個量(s, d)
Page Table 記錄 Frame Number	分段表記錄段 Limit 與 Base

Paged Segment Memory Management

一、觀念：『段再分頁』: Process -> 段組成 -> 分頁組成

二、動機：希望保有分段 Logical Viewpoint 的優點，又要解決外部碎裂的問題

三、圖示：



小結：

	Contiguous Allocation	Page	Segment	Paged Segment
外部碎裂	有	無	有	無
內部碎裂	無	有	無	有