作業系統：Main Memory

目錄

1. Background

記憶體是電腦系統的核心，CPU 要從記憶體抓指令與資料。執行指令流程如下：

1. 從記憶體抓指令
2. 解碼
3. 抓取操作數
4. 執行後可能寫回記憶體

作業系統：Main Memory

CPU 只能直接操作暫存器與主記憶體（不能直接操作硬碟）。而主記憶體存取速度慢 → 引入快取 (Cache) 來加速。

每個 CPU 只能直接操作暫存器和主記憶體（例如寄存器的速度通常為一個 CPU 時脈週期）。若主記憶體資料尚未完成存取，CPU 會「停滯（stall）」，除非使用快取或多執行緒技術來減少等待。系統需保護：

- 作業系統不可被使用者程式讀寫
- 使用者程式不可互相干擾

1.1 解法：基底暫存器（base）+ 限制暫存器（limit）

每個 process 的記憶體限制由這兩個暫存器定義：

- base：起始位址
- limit：可使用的記憶體範圍

只有作業系統（kernel mode）可以修改 base/limit 暫存器。若 user 程式違規，就會產生 trap 給作業系統

Figure 1：A base and a limit register define a logical address space

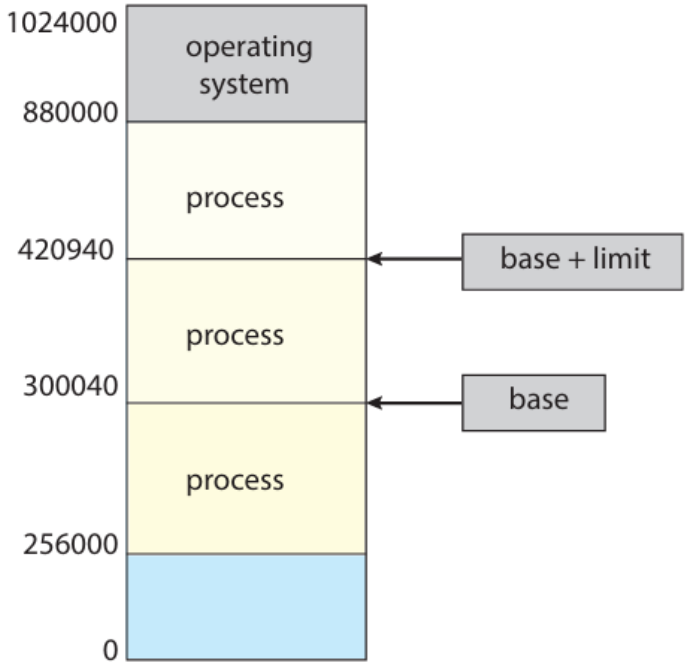
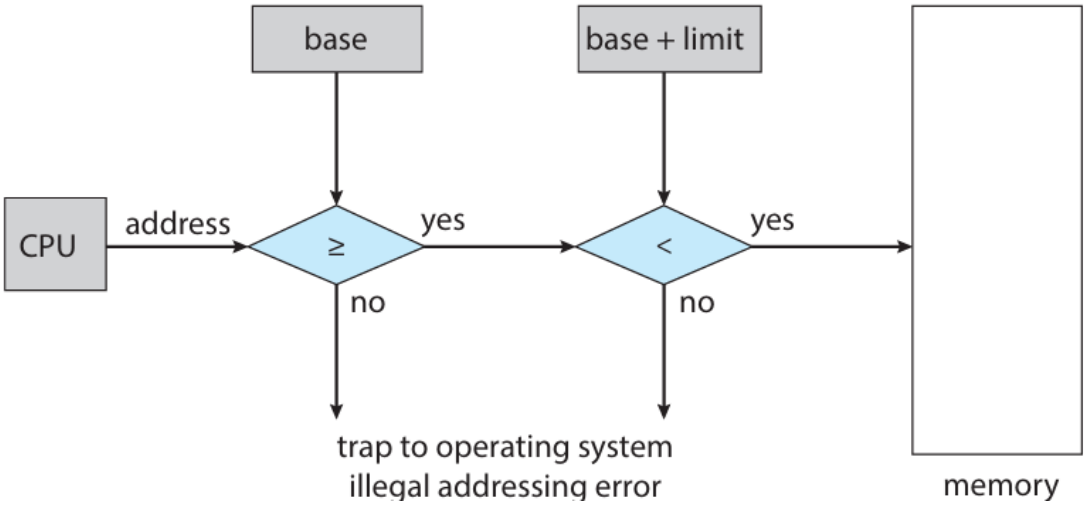


Figure 2：Hardware address protection with base and limit registers

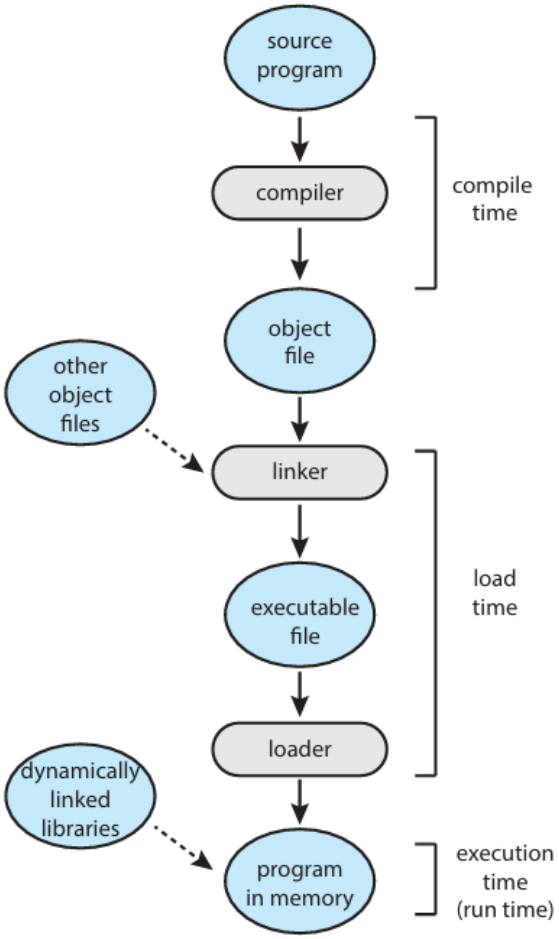


2. 位址繫結：Address Binding

位址繫結是將變數名稱（symbolic address）變成實際位址的過程。有三種繫結方式：

編譯時（Compile time）	載入時（Load time）	執行時（Execution time）
程式位址在編譯時就固定（例如：從位址 100 開始）。若位置改變，必須重新編譯	編譯產生「可重定位（relocatable）」程式碼。真正的實體位址延後到載入到記憶體時決定	現今最常見。程式執行過程中可以搬動到不同位置 需要硬體支援，例如：MMU

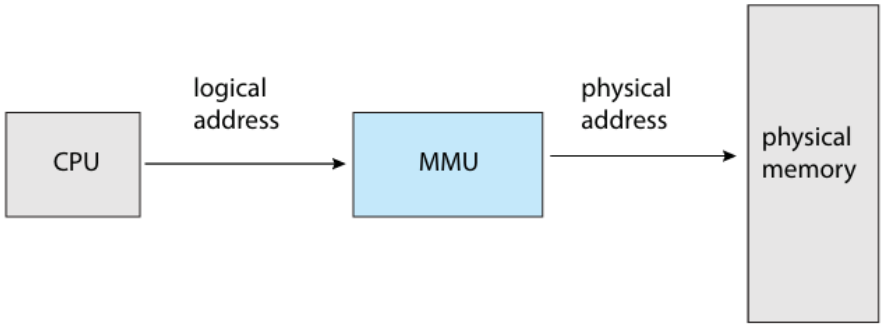
Figure 3：Multistep processing of a user program



3. Logical vs Physical Address Space（邏輯位址 vs 實體位址）

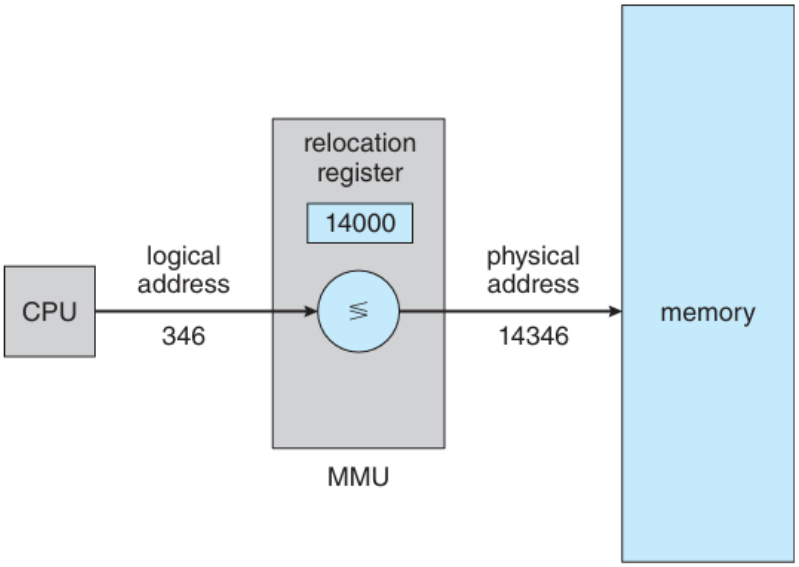
- 邏輯位址（Logical / Virtual Address）：程式內看到的地址（例如你寫 arr[3]）
- 實體位址（Physical Address）：真正送進記憶體的位址（經過硬體轉換）
- 使用 MMU（Memory Management Unit） 做地址轉換：
  - user 程式產生邏輯位址
  - MMU 使用「重定位暫存器（relocation register）」轉成實體位址

Figure 4：Memory management unit (MMU).



範例：如果 user 想存位置 346，重定位暫存器 ＝14000。那實體位址 ＝14346

Figure 5：Dynamic relocation using a relocation register.



4. Dynamic Loading（動態載入）

這是程式設計層面的技巧，不一定需要作業系統支援。

平常的話，都是整個程式都先載入記憶體，而動態載入是只載入主程式，其他函式直到呼叫時才載入。好處是

- 減少記憶體使用
- 稀少使用的函式（如錯誤處理）不用一直佔記憶體

4.1 動態連結與共享函式庫(Dynamic Linking and Shared Libraries)

靜態連結：每個程式都內嵌一份函式庫（浪費空間）

動態連結（DLL）：

- 程式執行時才連結共享函式庫
- 多個程式可共用一份程式碼（記憶體節省）

5. 連續記憶體配置(Contiguous Memory Allocation)

每個 process 都會被分配到一段「連續的」記憶體區塊。整體記憶體空間會分成兩部分：

- 一部分保留給作業系統（通常放在高位址）
- 一部分給使用者程式

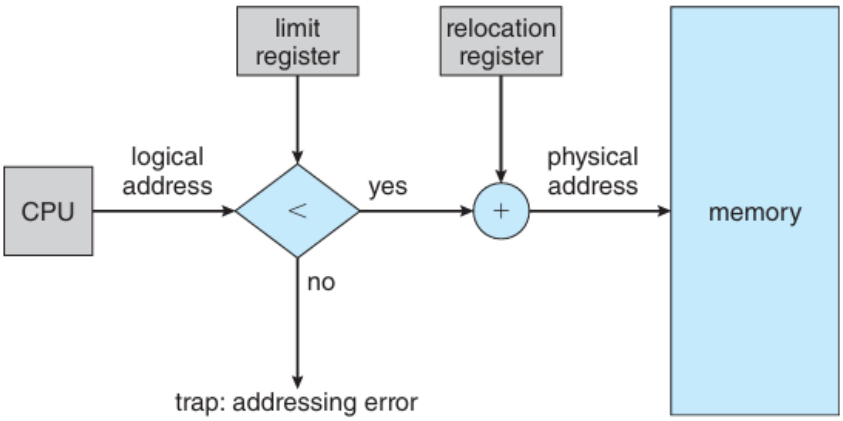
5.1 記憶體保護(Memory Protection)

防止一個 process 去存取不屬於它的記憶體

解法：使用 relocation register（基底暫存器）與 limit register（範圍限制）

- Relocation Register：process 可存取的最小物理位址（也就是它的起點）
- Limit Register：可存取的邏輯記憶體長度

Figure 6：Hardware support for relocation and limit registers.



操作流程：

1. 使用者程式產生邏輯位址（例如 100）
2. MMU 幫它加上 relocation（如 100040 + 100 = 100140）
3. 判斷是否超出 limit（否則觸發 trap）

Context switch 時，OS 會重新載入新的 relocation 與 limit。

6. 記憶體分配(Memory Allocation)

一開始，整個記憶體是個大洞（hole），只有 OS 的部分是固定的，其餘可以被分配給使用者 process。當 process 進入記憶體時：

1. OS 找到一個夠大的洞（hole）
2. 分出一塊給這個 process，其餘保留成新的洞
3. 當 process 結束，釋放記憶體 → 成為新的洞
4. 如果相鄰的洞可以合併 → merge 成一個大洞

6.1 三種洞的分配策略（hole selection）

策略	說明	優缺點
First Fit	找到第一個夠大的洞就用	快速但會留下很多小碎塊
Best Fit	找最剛剛好的小洞	節省空間但要全部掃過，慢
Worst Fit	找最大洞	剩下最大空間，但容易浪費

7. Fragmentation（碎裂）

7.1 External Fragmentation（外部碎裂）

記憶體被切成很多小塊，總量夠但無法連續分配。最壞情況是每兩個 process 間都有一小段無法使用。

有個 50% 規則：當有 N 個已分配區塊時，會產生額外約 0.5N 個無法使用的小碎塊 → 約三分之一記憶體浪費。

7.2 Internal Fragmentation（內部碎裂）

分配比需求稍大的記憶體區塊時，造成內部浪費。

例如：洞有 18464 bytes，process 只要 18462 bytes → 剩 2 bytes 難再利用

7.3 解法 01：Compaction（壓縮）

針對外部碎裂。

把記憶體中的資料集中到一邊，空洞集中成一大塊。

- 需配合動態位址重配置（dynamic relocation）
- 把所有 process 往一側搬，更新 relocation register

缺點：成本高、效率差

7.4 解法 02：非連續分配（noncontiguous allocation）

採用 paging(接下來會提到)。不需要把整個 process 放在連續區域。可以把 process 分散放在不同記憶體空間中

8. 分頁管理(Paging)

傳統記憶體管理方式要求一段連續的實體記憶體空間，這導致外部碎裂（external fragmentation）。Paging 分頁是一種讓「實體記憶體可非連續配置」的技術，透過將邏輯記憶體分成等大小的 pages，實體記憶體分成 frames，解決外部碎裂與壓縮問題。

8.1 分頁基本方法

- pages：將邏輯記憶體分為固定大小的區塊
- frames：將實體記憶體分為相同大小的區塊

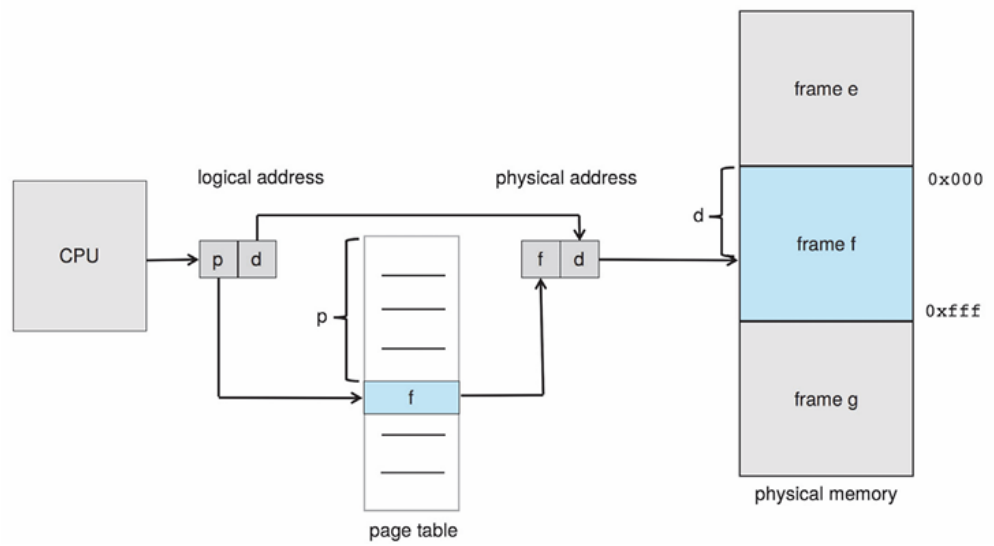
地址轉換（使用 Page Table）： 每個 CPU 產生的邏輯位址被分為兩部分。

- p：Page number 頁號
- d：Page offset 頁內偏移

轉換過程如下：

1. 用 p 去查 page table → 找出對應的 frame number
2. frame number × frame 大小 + d → 得到實體位址

Figure 7：Paging hardware



假設：

- page size = 4 bytes，Logical address = 13
- Page number = 13 / 4 = 3, offset = 1
- 查 page table 得 page 3 → frame 2
- 所以 physical address = (2 × 4) + 1 = 9

優點	缺點
避免外部碎裂	可能產生內部碎裂（最後一頁未滿）
提供動態位址對應（dynamic relocation）	需額外儲存 page table，消耗空間
使用者程式看起來像是連續空間	

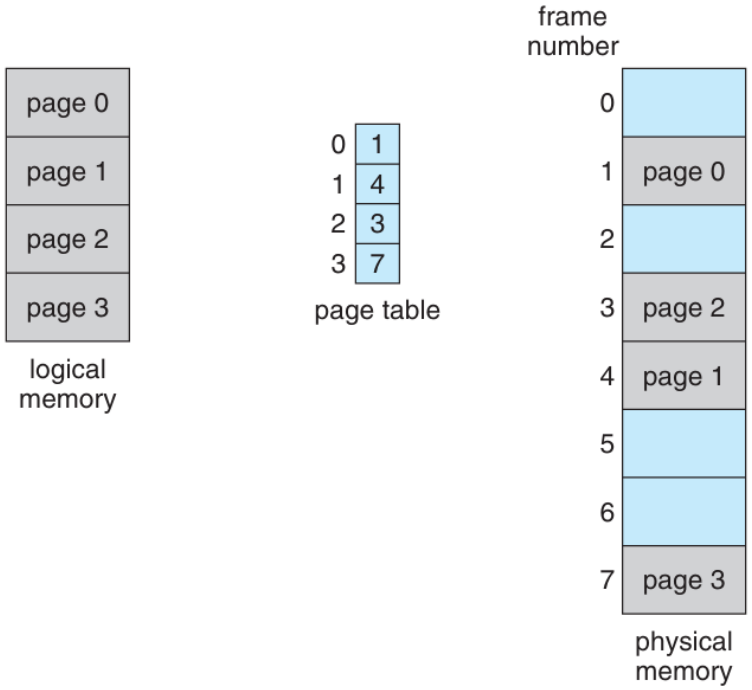
8.2 硬體支援與 TLB

若 page table 存在主記憶體，每次要存取資料都需兩次存取（查 page table + 存取資料），效率低。

TLB（Translation Lookaside Buffer）：是一種小型、高速、可聯想式記憶體。儲存常用的 page table 對應資料（頁號→框號）動作流程：

1. CPU 產生 logical address（p, d）
2. 查 TLB 是否有對應的頁號 p
3. 有命中（hit）：直接取得 frame → 組合成實體位址
4. 未命中（miss）：查 page table → 更新 TLB

Figure 8：Paging model of logical and physical memory



9. 分頁保護機制

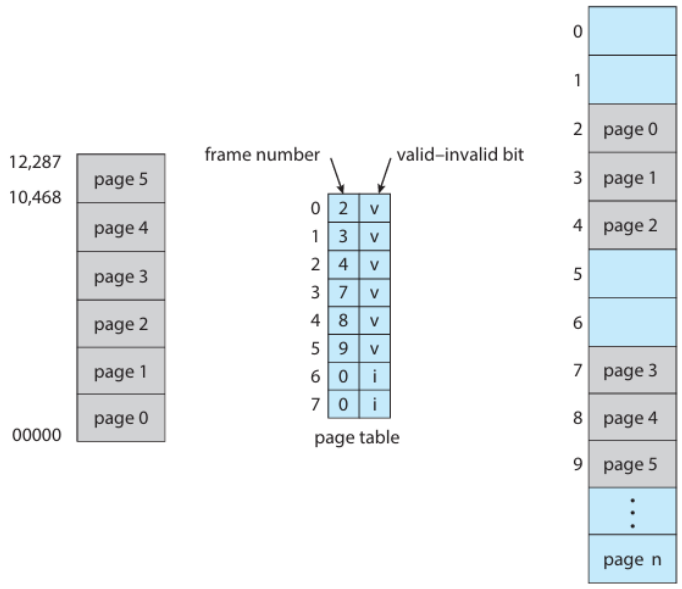
page table 可附加 保護位元（protection bit）：

- 可設定頁面為「唯讀」或「可寫」
- 防止程式誤寫資料（例如共享函式庫）

有效/無效位元（valid/invalid bit）：

- 判斷該頁是否屬於該程序的合法記憶體空間
- 若非法使用，就會發生陷阱（trap），交由作業系統處理

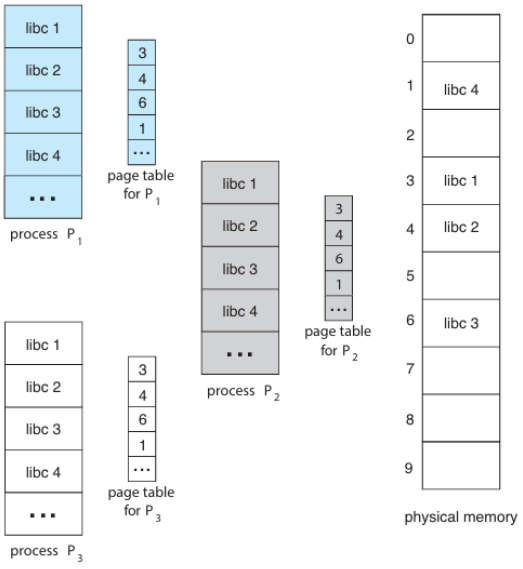
Figure 9：Valid(v) or invalid(i) bit in a page table



10. Shared Pages(共用頁面)

Reentrant code（可重入程式碼）：多個程序可共享一份代碼，例如標準 C 函式庫 libc。好處是節省實體記憶體（如 40 個程式共享 1 份 libc），但每個程式仍有自己的資料區段

Figure 10：Sharing of standard C library in a paging environment



11. 頁表的結構設計(Structure of the Page Table)

當程式邏輯地址空間很大（例如 32-bit 或 64-bit）時，單一平面（linear）的 page table 容易造成記憶體浪費。因此，作業系統採用以下幾種進階的頁表設計策略：

11.1 階層式分頁(Hierarchical Paging)

- 對於 32-bit 系統，每個程式有 4GB 邏輯空間
- page size = 4KB → 每個程式最多需要 2^20 = 1M 頁表項目，每項 4 bytes，總共 4MB
- 若每個 process 都要一份這麼大的 page table，太浪費

解法：將 page table 拆分為多層。

以二階層分頁 (two-level paging)為例：將 20-bit 的 page number 拆成 10-bit p1（外層）與 10-bit p2（內層）

p1 (10 bits)	p2 (10 bits)	offset d (12 bits)

Figure 11：A two-level page-table scheme

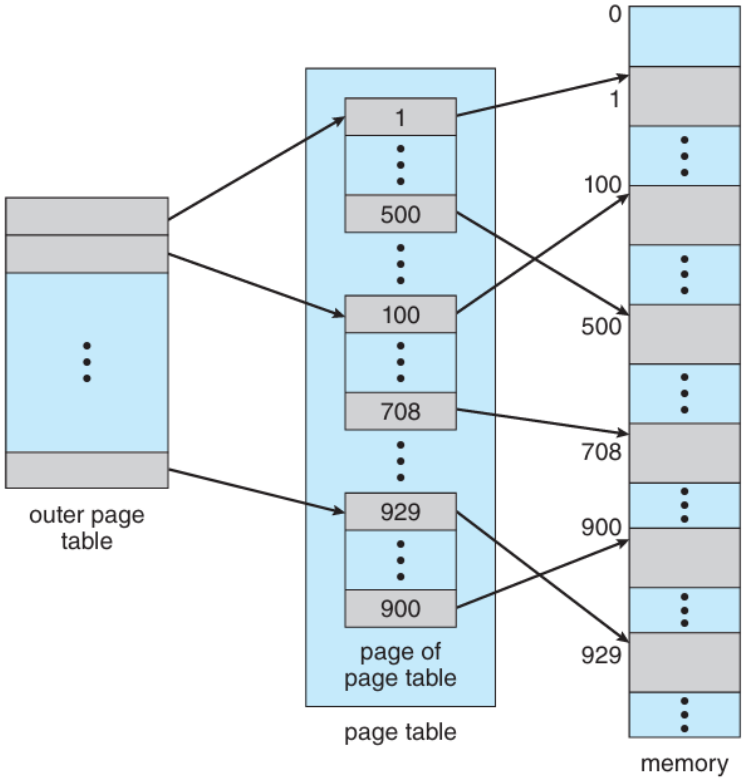
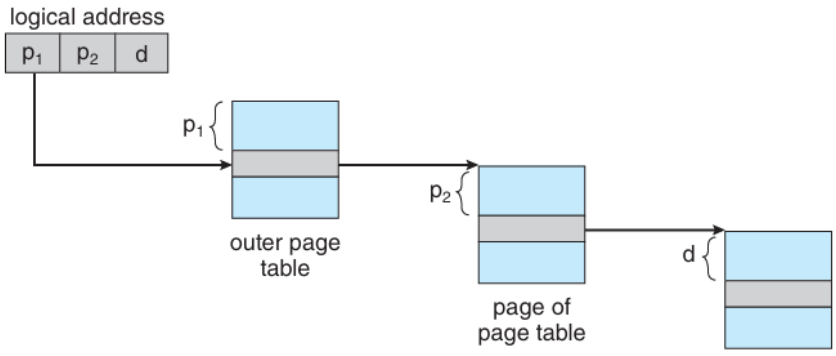


Figure 12：Address translation for a two-level 32-bit paging architecture



11.2 雜湊頁表(Hashed Page Tables)

適用：64-bit address space（巨大且稀疏）

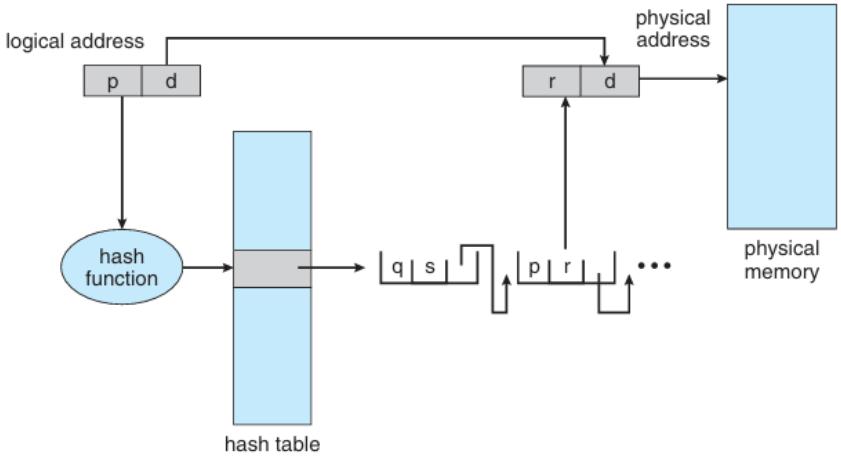
核心概念：

- 使用虛擬頁號 p 做 hash → 得到 hash table index。
- 該 index 對應一個 linked list，搜尋 list 中是否有相同的虛擬頁號。
- 若有對應 → 回傳 frame → 加上 offset 組成 physical address。

每個 list node 包含：虛擬頁號、對應實體頁框號、下一個節點指標。

好處是不需儲存完整的 page table，更適合稀疏分佈的大型位址空間。

Figure 13：Hashed page table



11.3 Inverted Page Table（反向頁表）

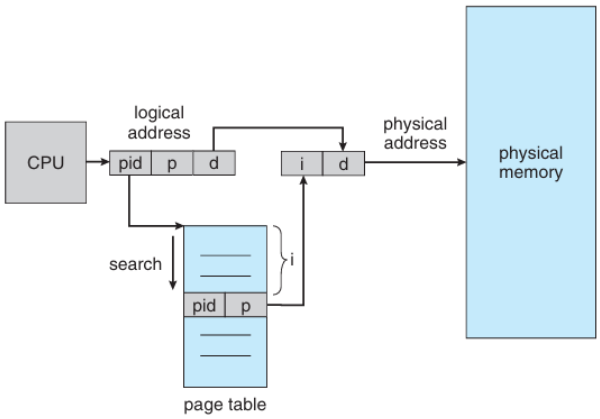
傳統方式的缺點：

- 每個 process 各自持有一份 page table
- 若有很多 process，就需要很多份 page table（即使每份用不到太多）

反向設計：只有一份 page table，對應的是實體記憶體中的每個 frame。每筆資料記錄該 frame 對應的 `<process-id, page number>`。搜尋方式：

- 根據 CPU 輸入的邏輯 `<pid, p>`，找出對應的 frame i
- 結果為 physical address `<i, offset>`

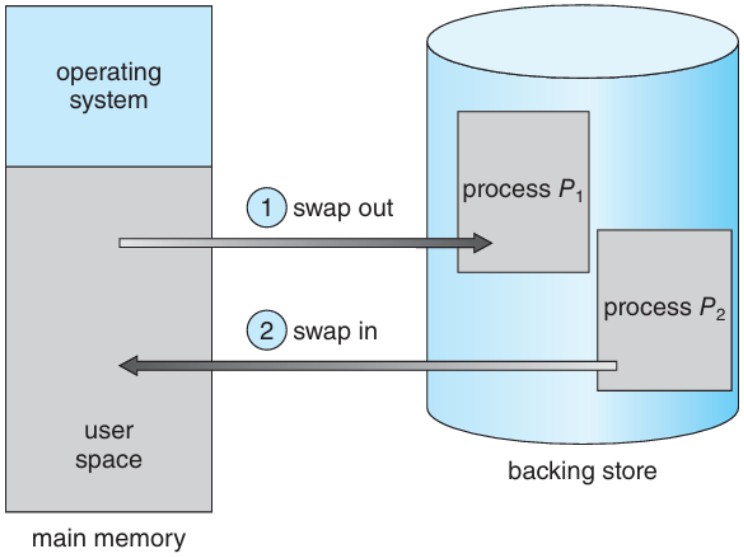
Figure 14： Inverted page table



12. Swapping

Swapping（交換）是指將整個程序或程序的一部分從主記憶體移到備份儲存裝置（如硬碟）以釋放記憶體空間，等到需要時再搬回主記憶體。這項技術的目的是：允許實體記憶體不足的情況下，同時容納更多程序，提升多工程度（degree of multiprogramming）。

Figure 15：Standard swapping of two processes using a disk as a backing store



12.1 Standard Swapping（標準交換）

1. 系統將整個 Process P1 swap-out 到 backing store（例如硬碟）。
2. 騰出的空間可供其他程式如 P2 swap-in 執行。
3. 當 P1 再次活躍時，再把它 swap-in 回來。

儲存內容包括：

- 程式本體與資料
- Thread 的資料結構（若是多執行緒）
- 作業系統追蹤的中介資料（metadata）

優點	缺點
有效利用有限的主記憶體資源	整體速度慢，因為搬移整個程序耗時
非常適合用於「長時間閒置的程序」	在記憶體壓力不大的現代系統中已不常見

12.2 Swapping with Paging（分頁交換）

標準交換是要般整個 process。分頁交換的話是只搬「需要的頁面」(Page-level Swapping)。優勢

- 效率更高 → 僅搬移活躍頁面
- 節省 IO 開銷
- 能與虛擬記憶體完美整合

Figure 16：Swapping with paging