Chapter 8:記憶體管理策略
Memory-Management Strategies

Chapter 8:記憶體管理策略

- 背景說明
- 置換(Swapping)
- 連續記憶體配置(Contiguous Memory Allocation)
- 分段式(Segmentation)
- 分頁式(Paging)
- 分頁表的結構
- 範例：Intel 32和64位元架構

章節目標

- 詳細描述各種記憶體硬體組織。
- 探討各種記憶體管理技術。
- 詳細討論Intel Pentium處理器所支援的純分頁及分段式分頁作法

背景說明

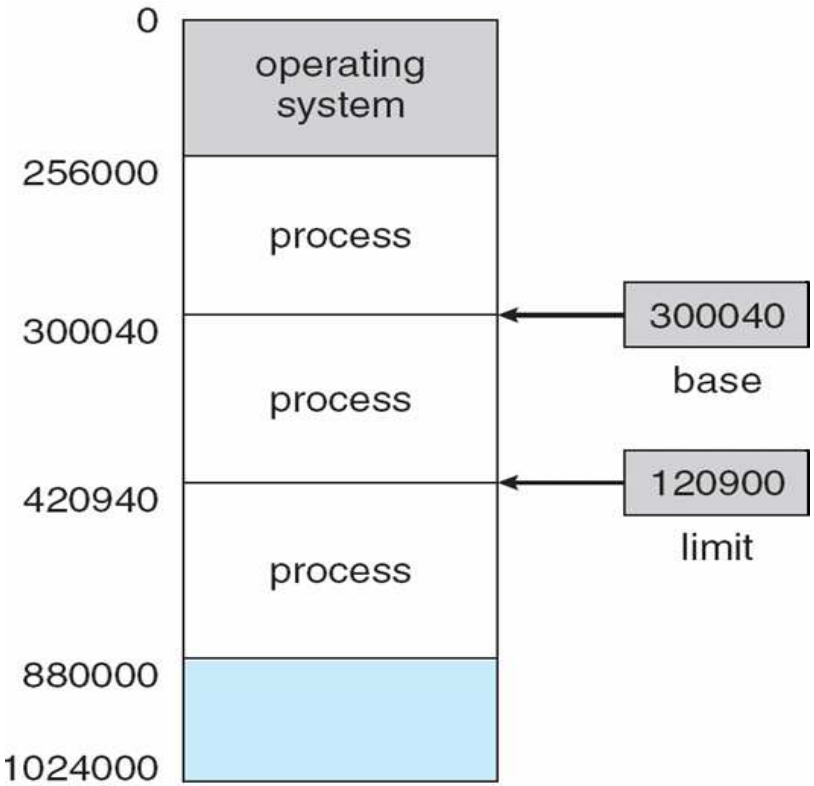
- 因CPU僅可以直接存取主記憶體和暫存器，程式必須先從磁碟移到記憶體才能開始執行
- 主記憶體和暫存器存取速度會影響執行時間
  - 暫存器的存取時間在一個CPU時脈週期(或更短)完成
  - 主記憶體存取時間需要好幾個CPU時脈週期，這會造成CPU停滯(stall)
  - 可透過快取記憶體(Cache)改善存取時間
- 記憶體單元(Memory unit)只看到一序列的記憶位址+讀取請求，或是記憶位址+資料和寫入請求

背景說明(cont.)

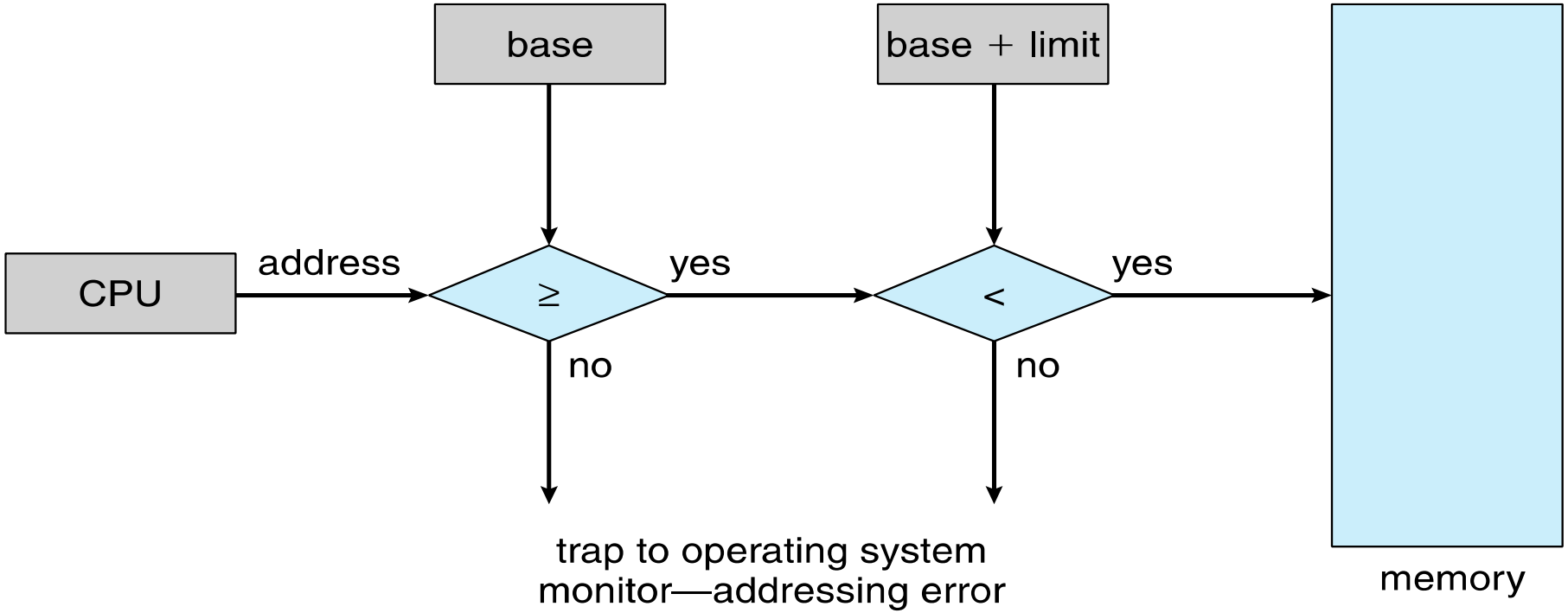
- 為了確保系統正確的運作
  - 應該避免使用者行程不當存取到作業系統的記憶體位址
  - 在多使用者系統，不同使用者的行程必須受到保護
  - ➔ 每一行程需有個別獨立的記憶體空間
    - ➔ 可使用一組基底和界限暫存器來進行

基底和界限暫存器(Base and Limit Registers)

- 邏輯位址空間由一組基底(base)暫存器和界限(limit)暫存器決定
  CPU 必須檢查使用者模式產生的每一個記憶體存取，以確定存取的地址位於該使用者的基底值和限制值之間。



使用基底及界限暫存器的硬體位址保護



位址連結(Address Binding)

- 程式將來要在記憶體的那個address執行
- 在程式的不同階段，位址表示方式會有所不同
  - 原始碼的位址通常是用符號表示，如變數名稱。
  - 編譯過的可重定位程式碼，位址則是以數字表示相對位移量，如距離這個模組的開始處14個位元組
  - 鏈結(linker)或載入程式(loader)再將這些可重新定位的位址連結到絕對位址，如74014
  - 每一次的位址連結工作都是將一個位址空間對應到另一個位址空間

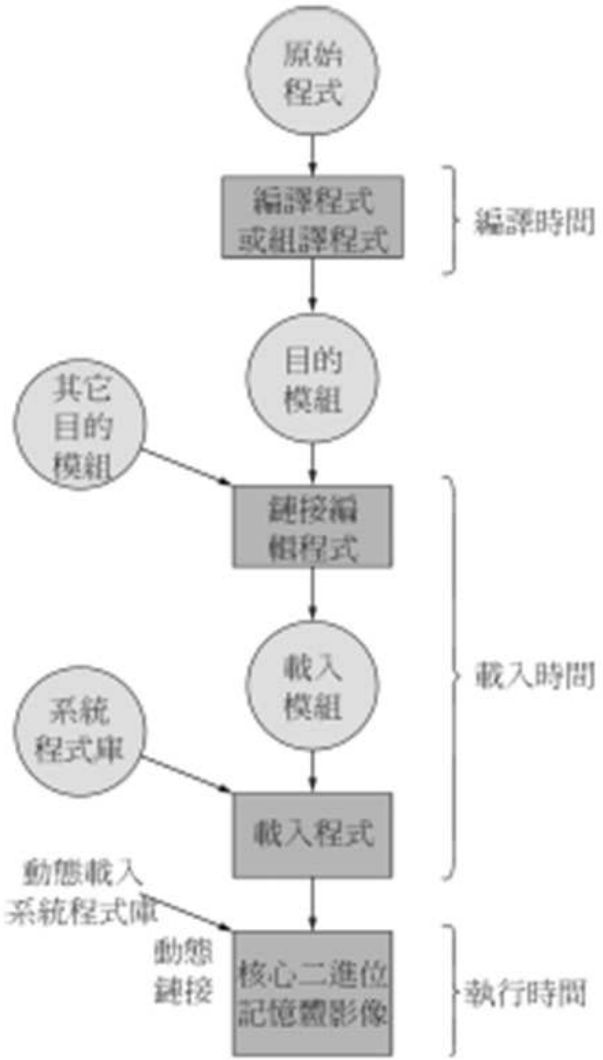
位址連結(cont.)

- 何時可以進行位址連結？
  - 編譯時期(compile time):在程式編譯時預先決定這支程式要放在實體記憶體的哪個位置執行，會產生絕對碼(absolute code)，邏輯位址會等於實體位址。
    - 缺點在於程式碼只能放在記憶體的固定位置，如果要改變執行位置，就必須重新編譯程式。
  - 載入時期(load time): 若在編譯時期不知process 會在記憶體的什麼地方執行，則編譯程式必須產生可重定位碼 (relocatable code)。而載入時才完成最後位址定位。
    - 編譯時是從0開始配置位址，然後在載入時才決定要放在哪個位置。
    - 必須透過邏輯位址加上重定址暫存器的內容，才能取得行程的實體位址
    - 如果需要改變行程在實體記憶體中的位置時，只要在搬移行程之後改變重定址暫存器的值就可以了

位址連結(cont.)

- 執行時期(Execution time):如果行程在執行時能夠從一個記憶體區段移到另一段，那麼連結必須延遲到執行時才做
  - 需要硬體支援 (例如基底和界限暫存器)
  - 大部分通用型作業系統採此做法

從原始檔編譯成可執行程式碼的過程



邏輯位址空間和實體位址空間

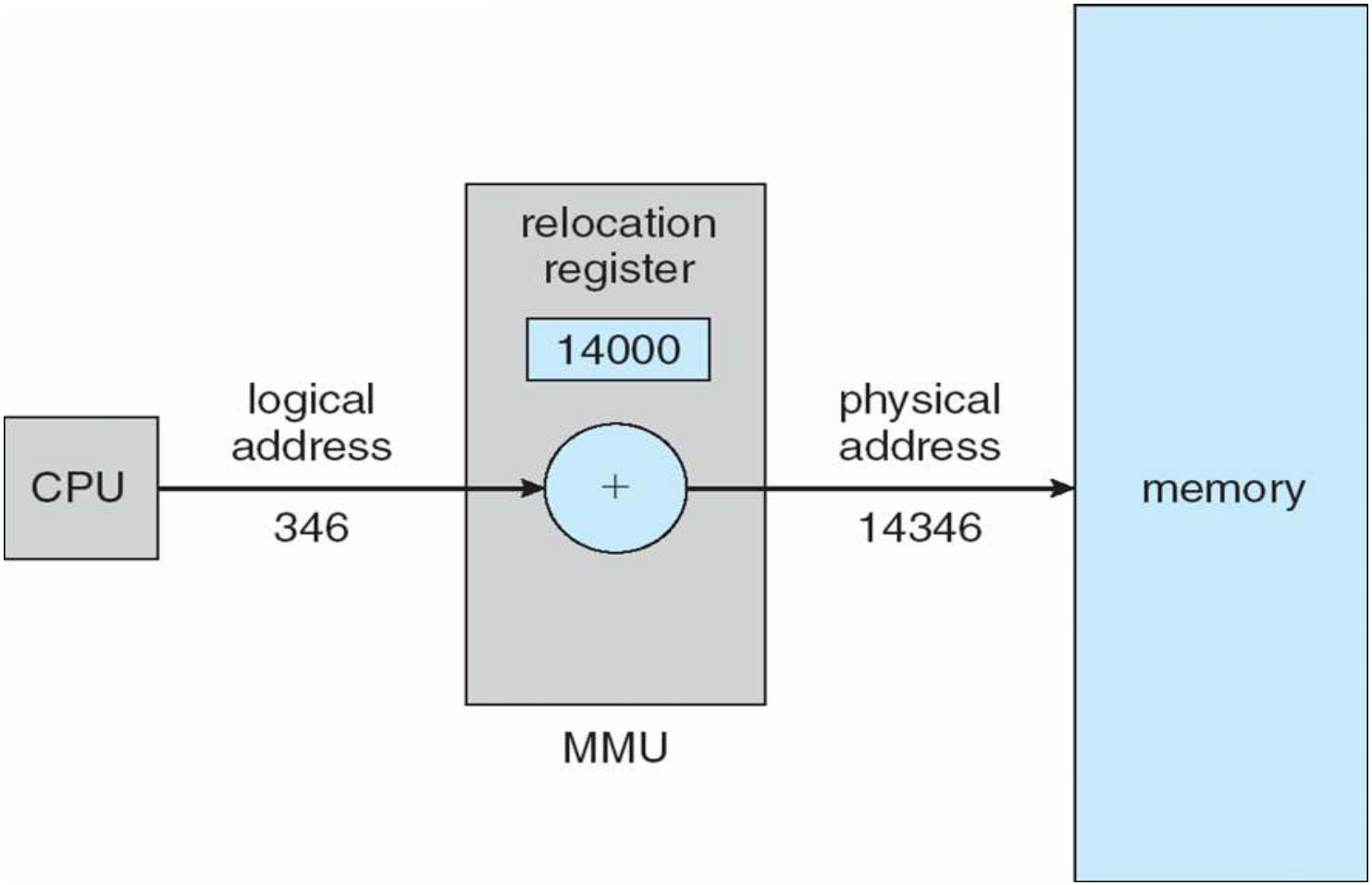
- 邏輯位址(logical address) – CPU所產生的位址;又稱為虛擬位址(virtual address)
- 實體位址(physical address) –記憶體單元所看到的位址
- 編譯時期和載入時期的位址連結方式造成邏輯位址和實體位址相同；而執行時間的位址鏈結方式造成邏輯位址(虛擬位址)和實體位址不相同
- 邏輯位址空間(logical address space)是一個程式產生的所有邏輯位址所形成的集合
- 實體位址空間(physical address space)是一個程式產生的所有實體位址所形成的集合

記憶管理單元(Memory-Management Unit)

- 在行程執行時，負責將虛擬位址轉換成實體位址的硬體裝置
- 主要做法：
  - 連續式記憶體配置(contiguous memory allocation)
    - 使用重定址暫存器：將使用者行程所產生的每一個邏輯位址與重定址暫存器之內容相加得到實體位址
  - 分段式(segmentation)
  - 分頁式(paging)

使用重新定位暫存器作動態重定位

- Kernel loads relocation register when scheduling a process



動態載入(Dynamic Loading)

- 一個程式中的常式(routine)一開始並不會載入記憶體中，直到被呼叫時才載入記憶體，稱為動態載入。
- 常式(routine)需以可重定位的載入格式放在磁碟
- 先載入主程式並開始執行，當呼叫一常式時，呼叫的程式會檢查被呼叫的常式是否已載入，若尚未載入，則由可重定位的鏈結載入程式(relocatable linking loader)負責載入該常式，然後繼續執行。
- 優點：
  - 更有效率的記憶體空間使用
  - 當程式內含有許多不常發生的處理程式(如:錯誤處理程式)時特別有用
- 不需作業系統的特別支援
  - 透過程式設計來規劃
  - OS僅提供程式庫來協助程式設計者

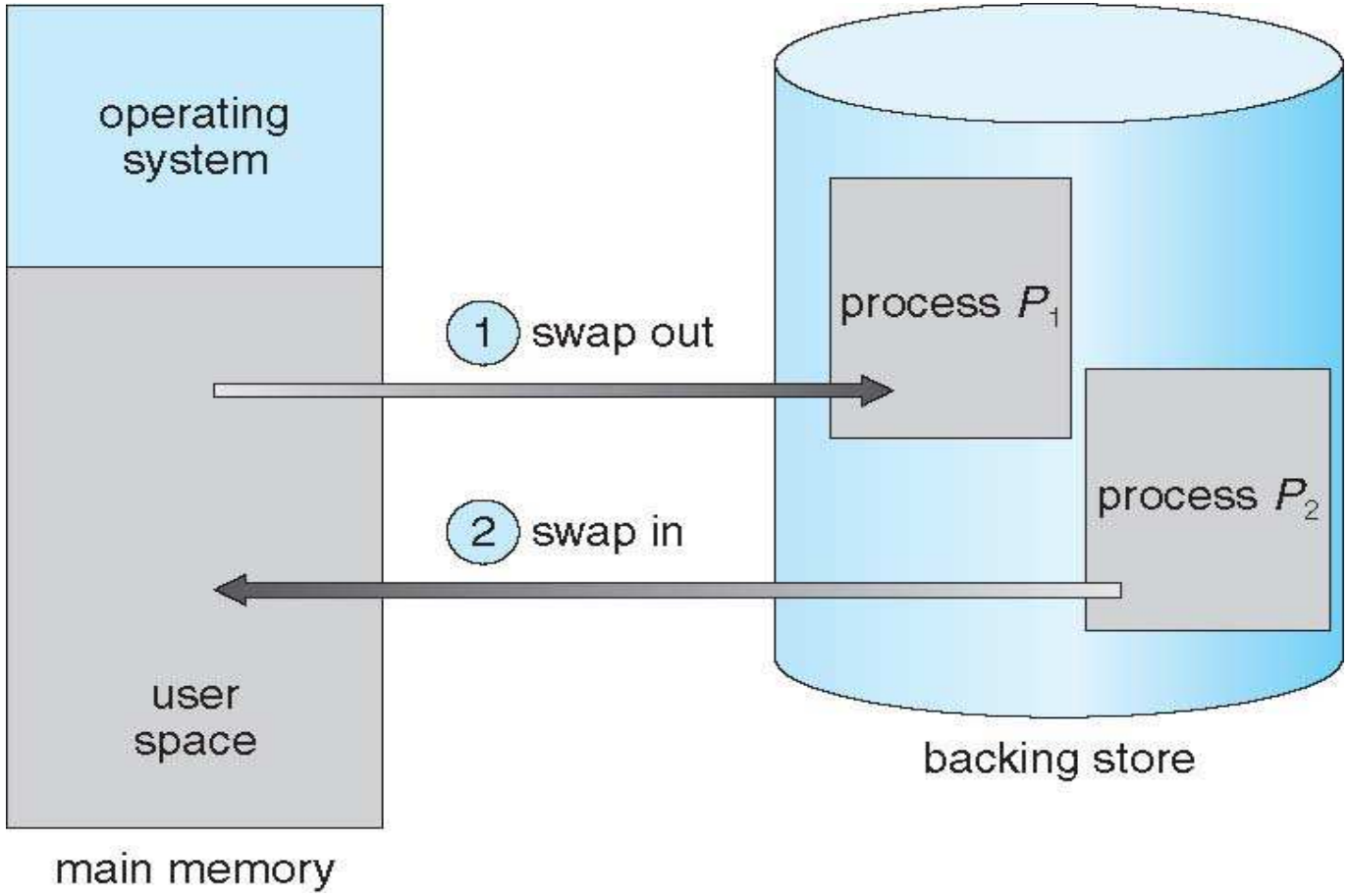
動態鏈結(Dynamic Linking)

- 靜態連結(Static Linking)會在程式編譯鏈結後將程式所需要的函式庫加入程式的執行檔
  - 執行檔會較大
  - 可以保證到不同機器環境下執行時，也不會因為少了這個函式庫導致無法執行檔案
- 動態連結(Dynamic Linking)當程式執行時才鏈結所需使用的系統函式庫(system libraries)的做法，而不是直接加入執行檔中
  - 執行檔會較小
  - 可以與其他程式共用同一份函式庫
  - 當動態連結函式庫更新時，無需重新編譯程式
- 在動態連結的情形下，執行檔中每一個參考到動態連結函式的地方都會有一小段Stub程式
  - 若該函式庫尚未載入則先載入
  - Stub程式用來找到該函式於記憶體中的位址，然後本身再以該函式的位址取代並執行該函式

置換(Swapping) (1)

- 一個行程可能會暫時的被置換(swapped)出記憶體到備份儲存體，而後又再移回到記憶體繼續執行
  - 所有行程所需的實體位址空間可以超過系統記憶體空間，使得系統的多元程式規劃的程度可以提高
- 系統將所有載入記憶體中或置換到磁碟中已準備好執行的行程，利用一就緒佇列(ready queue)進行排班
- 影響置換時間(swapping time) 的因素
  - 置換的記憶體量
  - 內容轉換(context switch)所需時間
- 正在進行I/O等待的行程，其置換要有所限制
  - 正在進行I/O等待的行程不能被置換出去
  - 或是 I/O傳送僅會在核心空間進行，等行程置換回記憶體後再把結果傳給該行程

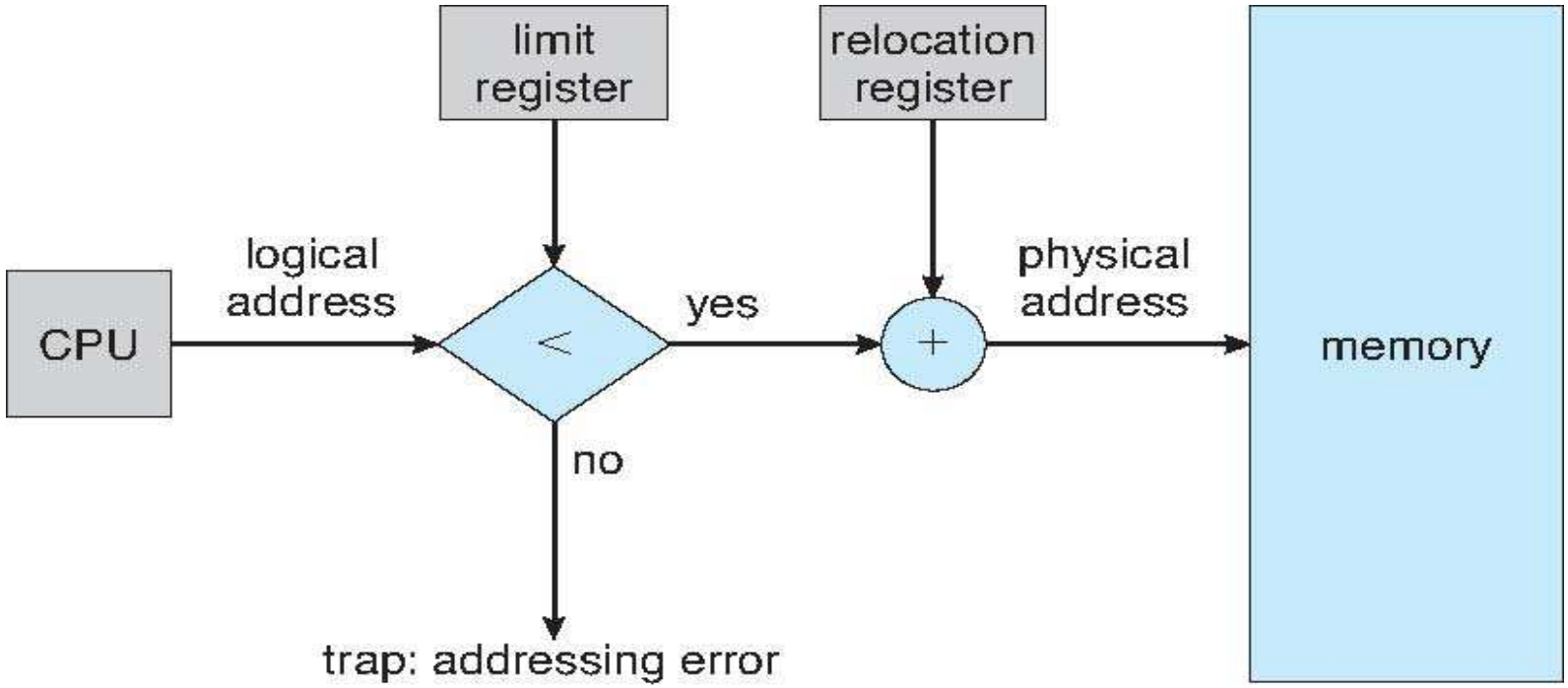
置換示意圖


operating system
process P_1
1 swap out
process P_2
2 swap in
user space
main memory
backing store

連續記憶體配置(Contiguous Memory Allocation)

- 將可供使用者行程使用的記憶體空間視為一大塊連續的空間
- 僅有單一個行程來使用，問題會較單純。 若同時有很多個行程在使用記憶體的情況下，問題會較複雜。
- 單一分割配置(Single-Partition Allocation)
  - 將記憶體空間分成兩塊，一塊給作業系統使用，另一塊則給行程使用，通常作業系統和中斷向量放在低記憶體部份，使用者行程放在高記憶體部份
  - 每一個行程包含在記憶體中一個單獨連續區段
  - 可以搭配重定位暫存器(relocation register)、界限暫存器(limit register)來完成。
  - 好處:
    - 提供記憶體保護 。
    - 可以動態地改變常駐作業系統所占的記憶體空間大小

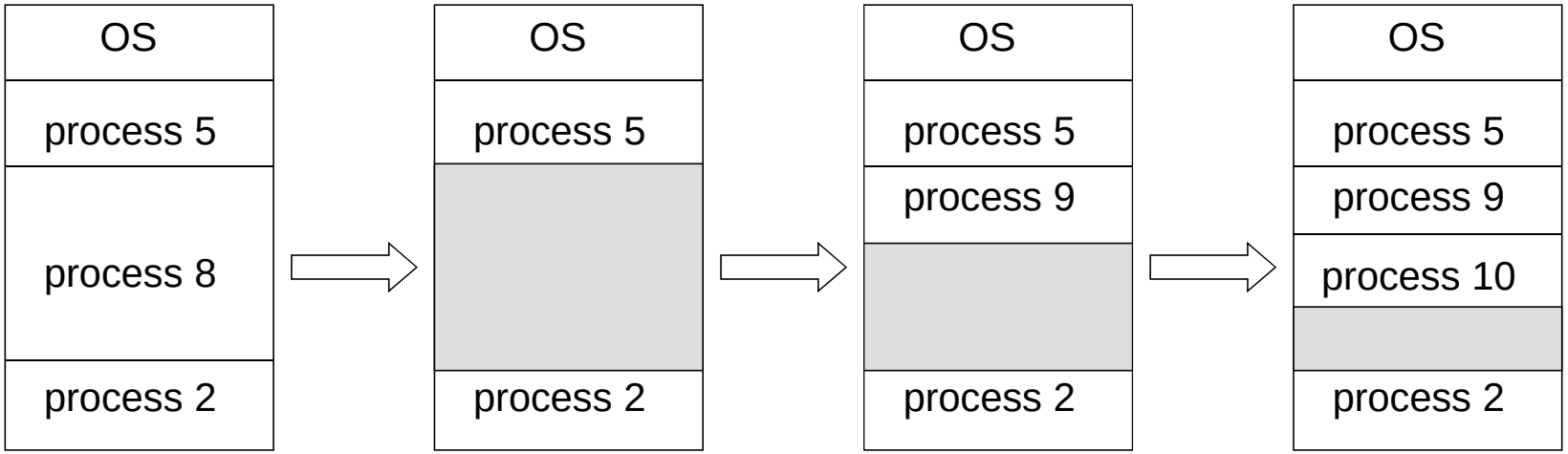
重定位和界限暫存器的硬體支援



連續記憶體配置(Cont.)

- 多重分割配置(Multiple-Partition Allocation)
  - 把整個記憶體空間劃分為一些固定或變動大小的分割(partition)，每個分割只載入一個行程。
  - 多元程式規劃的程度受到分割數目的限制
  - 洞(hole) – 未被使用的分割
  - 當一個行程到達時，它會配置一個足夠大的分割以放入程式
  - 等到行程執行結束之後，其佔用的記憶體分割則被釋放出來，並和鄰接的未被使用分割合併成較大分割
  - 作業系統需維護以下資訊：
    a) 配置的分割區b)未使用分割區(洞)

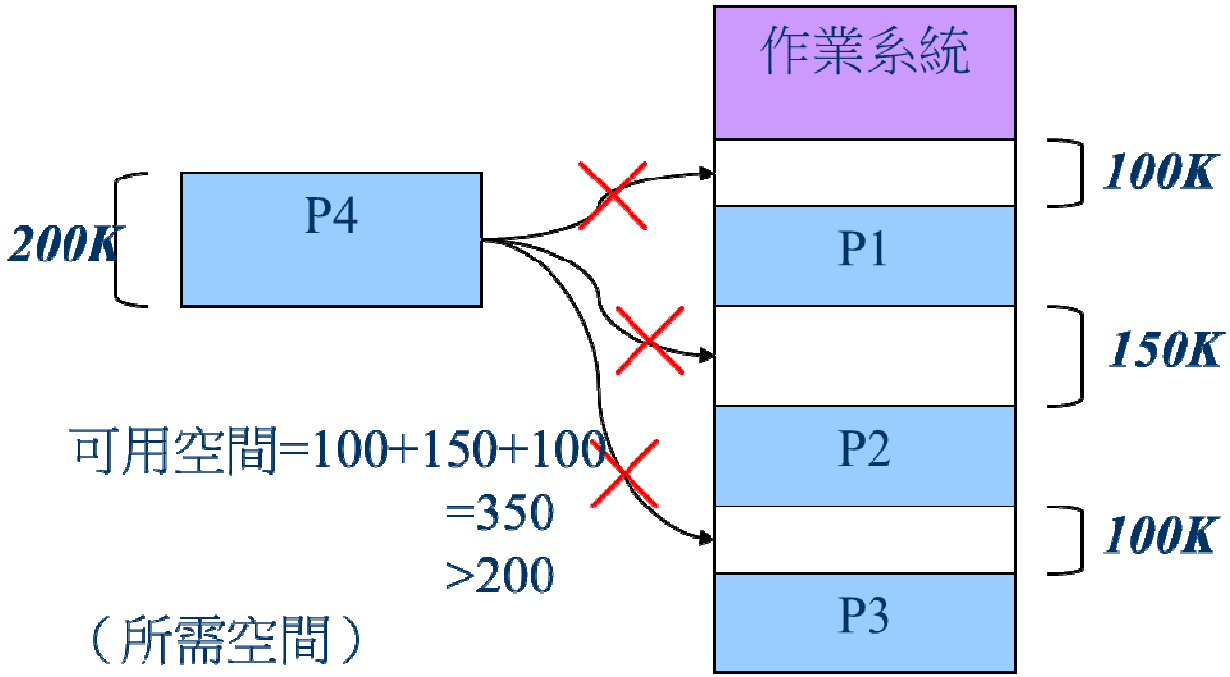
多重分割記憶體配置範例



斷裂 (Fragmentation)

- 外部斷裂 (External Fragmentation) –無任一分割區比現在請求的記憶體空間要大，但所有未被使用的分割區總容量比現在請求的空間要大，而使得行程不能載入的狀況。
  - 會浪費記憶體。
  - 浪費的記憶體多寡，與配置的演算法有關。
  - 所造成的記憶體浪費程度，主要是和記憶體空間總量以及行程的平均大小有關。
  - 百分之五十法則(50-percent rule)
    - 在最佳的狀況下，當配置了n單位的記憶體空間時，外部斷裂平均浪費n/2單位的空間。
- 內部斷裂(Internal Fragmentation) – 配置的記憶體可能稍大於要求的記憶體，使得多餘空間未被使用到
- 可藉由聚集(compaction)降低外部斷裂
  - 收集記憶體內零散的可用空間，成為一大洞

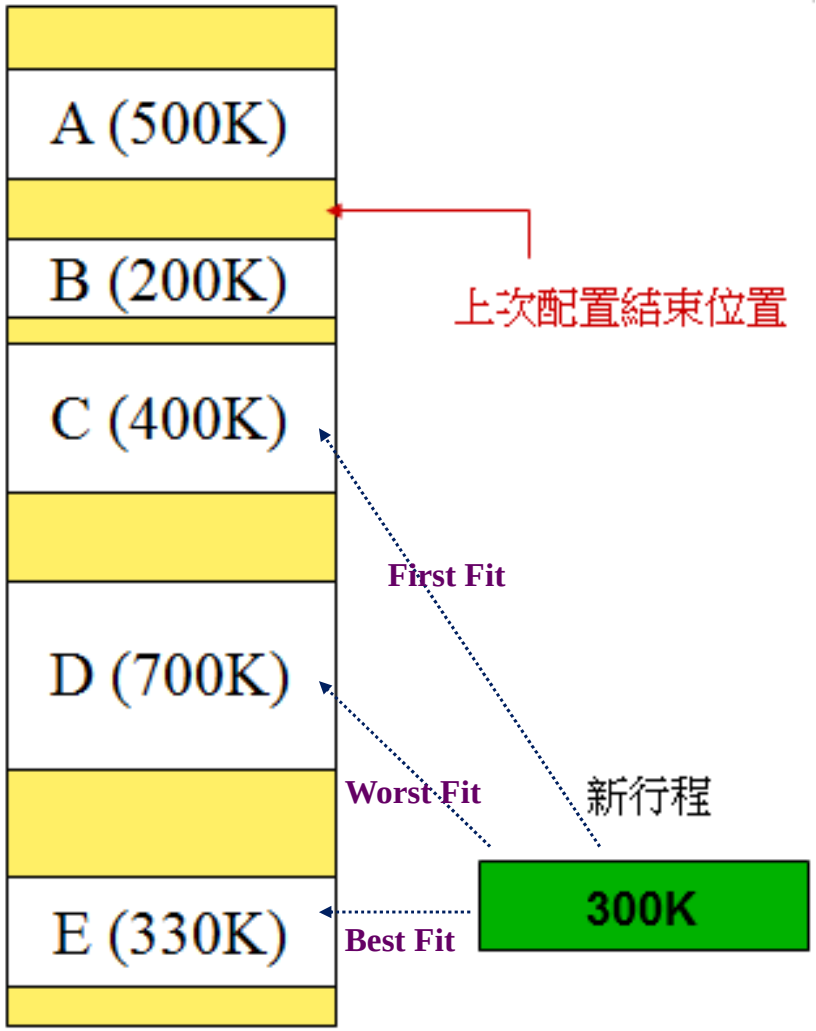
外部斷裂之範例


作業系統
P4
200K
P1
P2
P3
100K
150K
100K
可用空間=100+150+100
=350
>200
（所需空間）

配置記憶體的策略

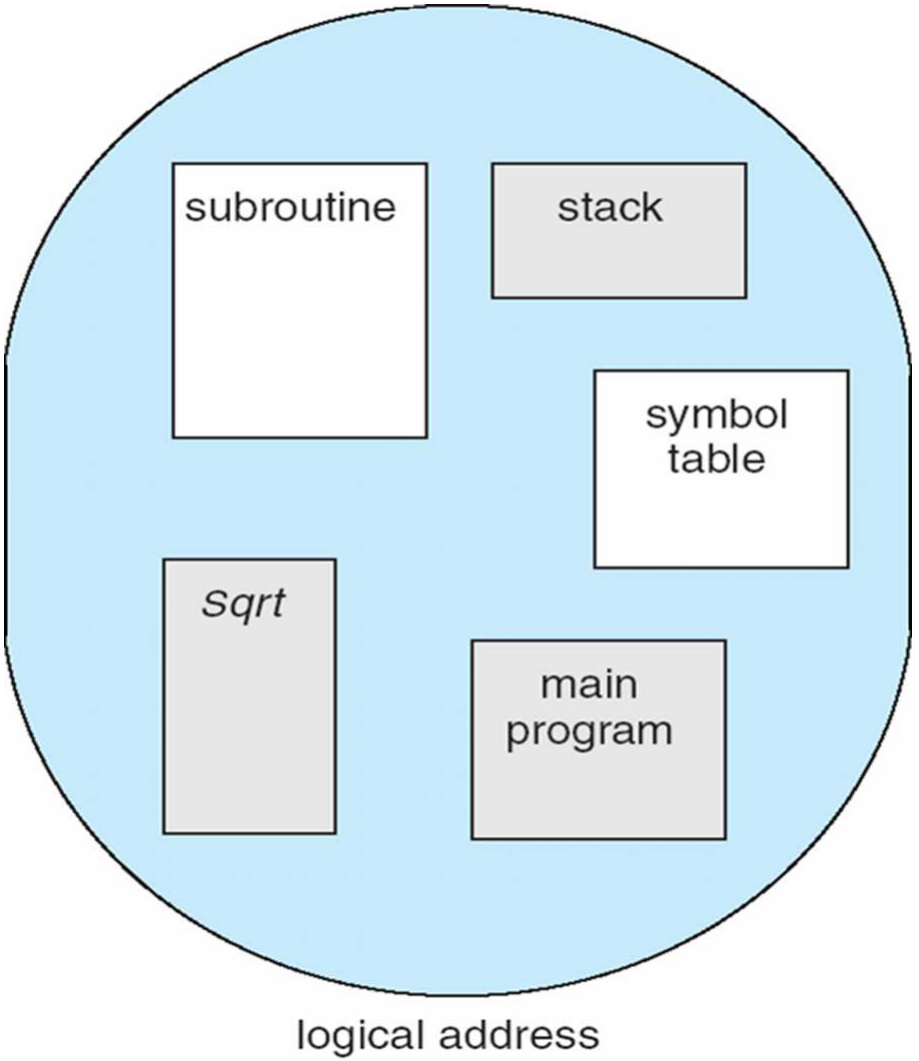
- 如何從可用的洞(hole)去滿足大小為n的記憶體請求?
  - 最先合適(First-Fit)：從第一個夠大的洞配置
  - 最佳合適(Best-Fit) ：從第一個夠大但最小的洞配置
    - 會得到最小的剩餘空間
  - 最差合適(Worst-fit) ：從夠大且最大的洞配置
    - 會得到最大的剩餘空間
- 效能比較
  - 在時間和記憶空間的使用率方面，最先合適與最佳合適都優於最差合適法
  - 在記憶體空間使用率上，最先合適與最佳合適很接近。
  - 最先合適的執行時間通常會比較短。

配置記憶體的策略

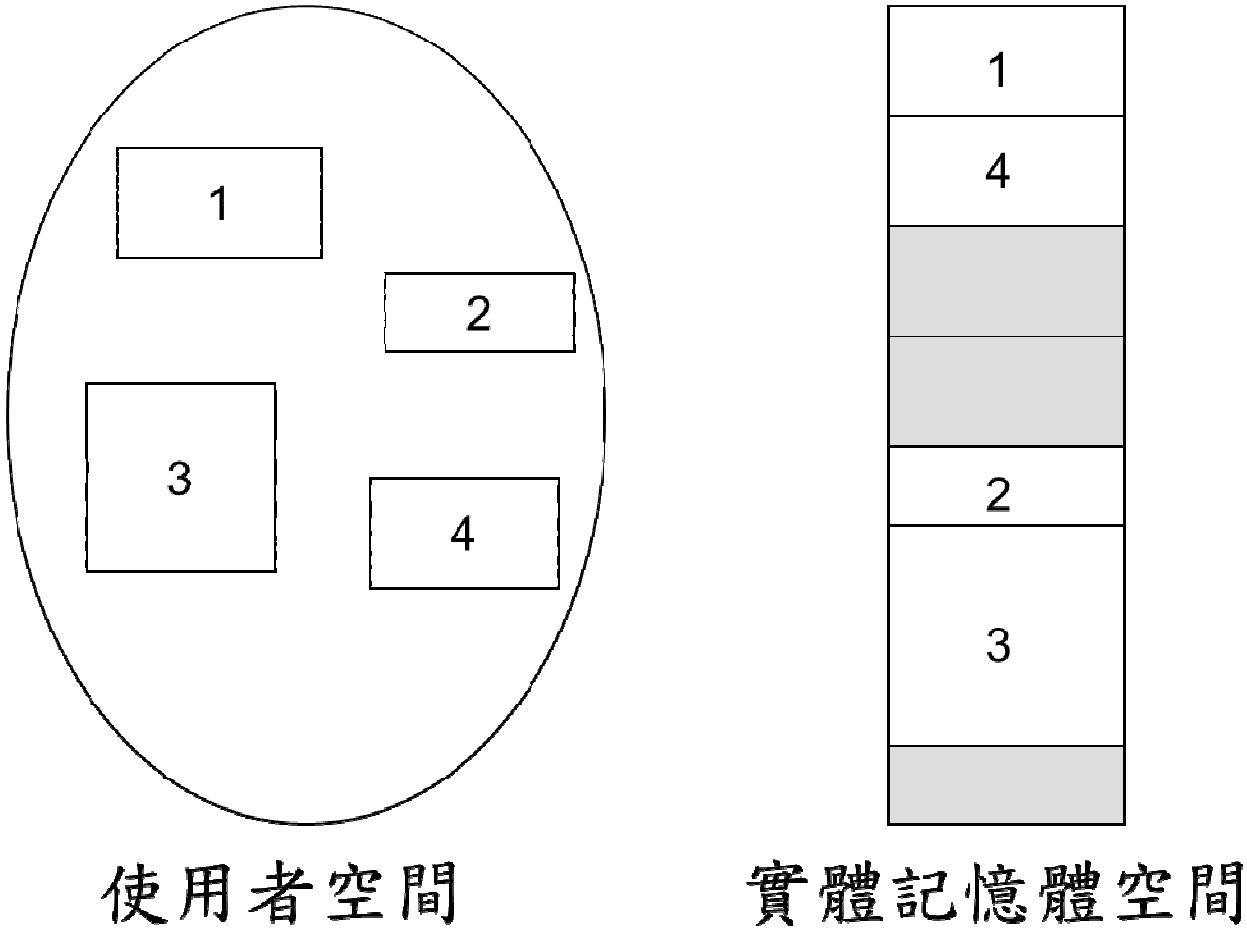

A (500K)
B (200K)
C (400K)
D (700K)
E (330K)
上次配置結束位置
First Fit
Worst Fit
新行程
300K
Best Fit

分段法(Segmentation)

- 從使用者的觀點來看，記憶體的模樣是由許多不同大小的分段(segment)所組成，分段之間並沒有先後順序的關係。
  - 一個分段是一個邏輯單元，例如:主程式、程序、函數、方法、物件、區域 變數、整體變數、堆疊、符號表、陣列等
- 由此觀點衍生出來管理記憶體的方法稱為分段法(Segmentation)。
- 分段因為涉及對程式的解讀，所以責任會落在程式設計師或編譯程式的肩上



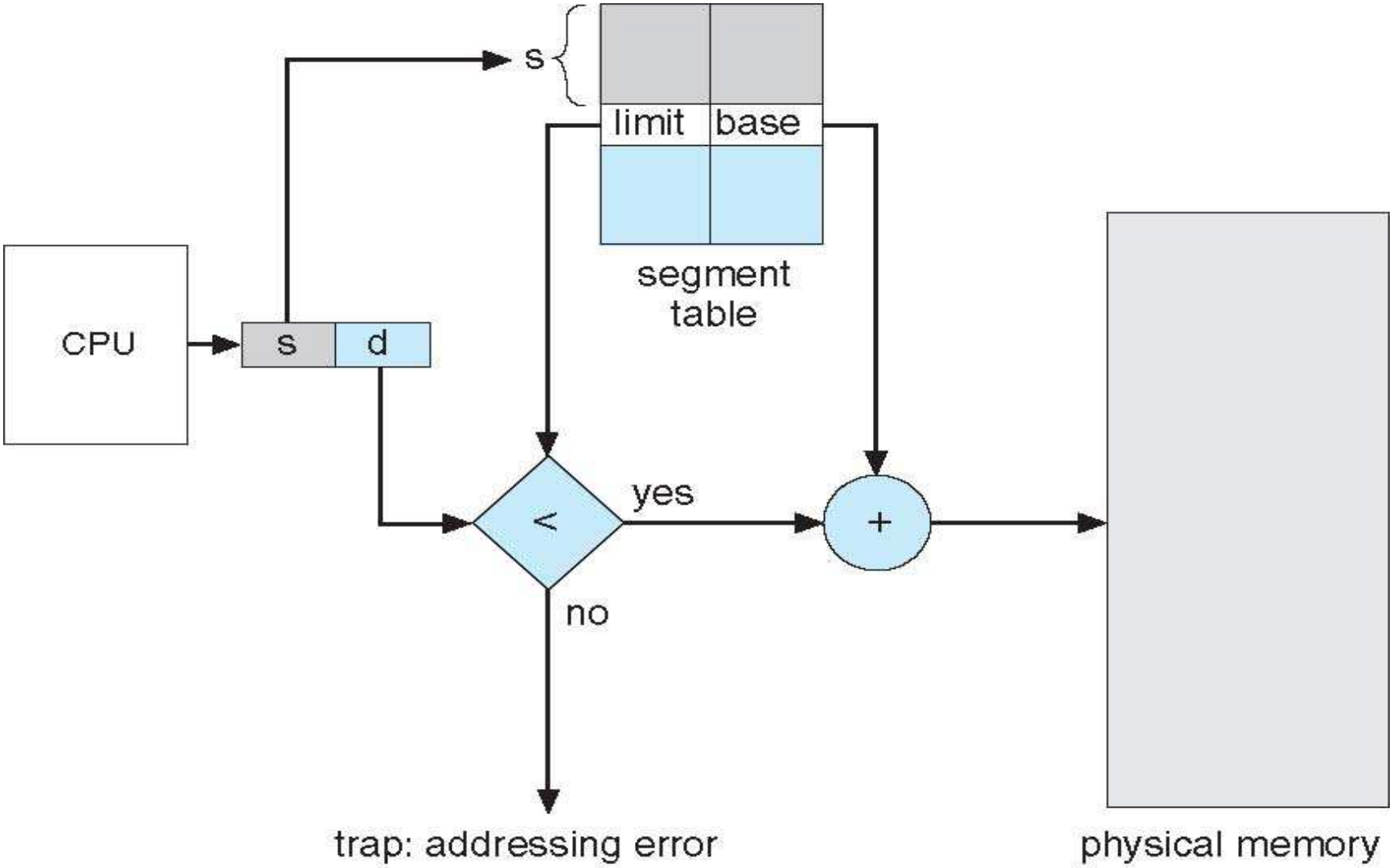
分段法的邏輯觀點



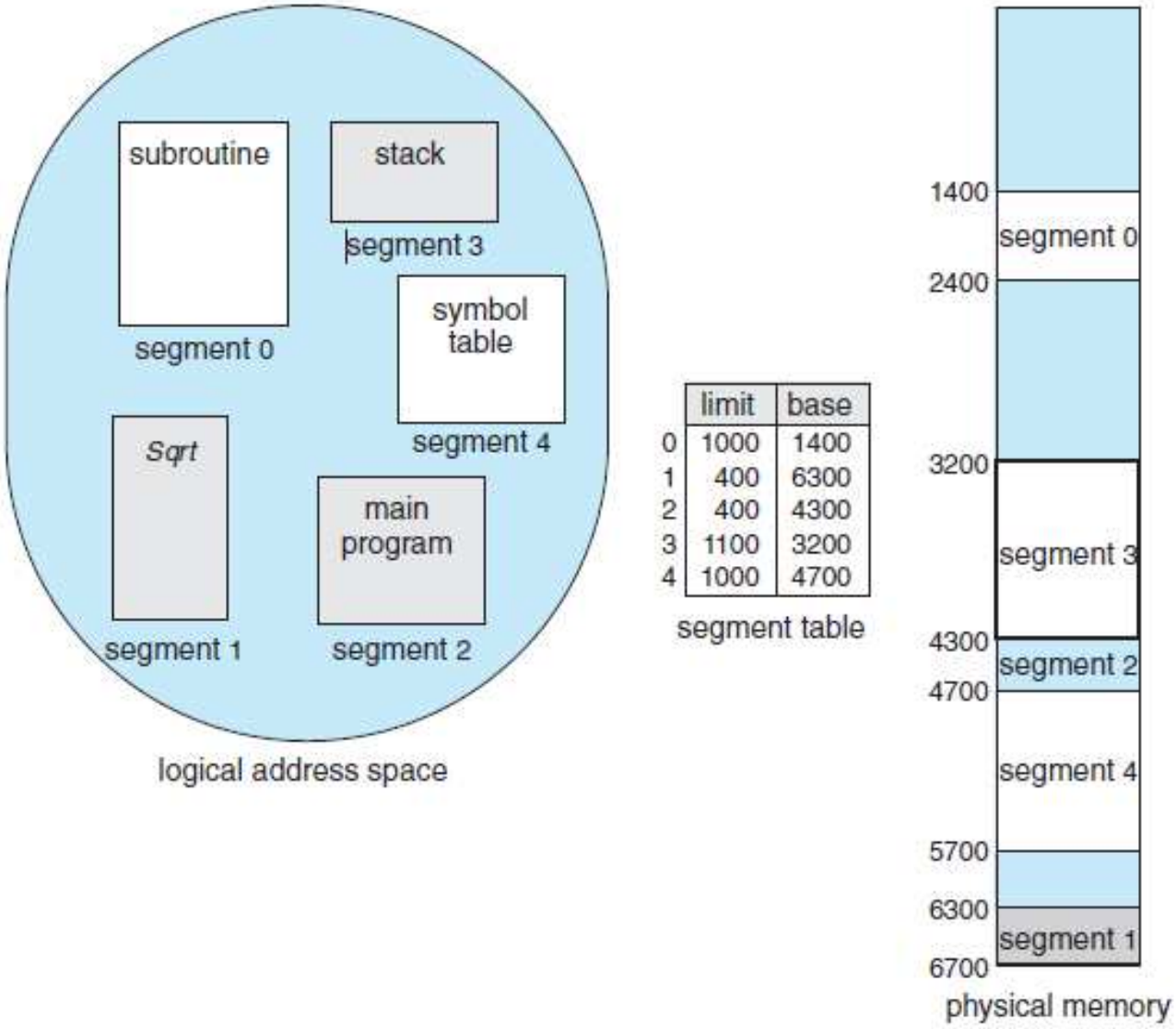
分段法(2)

- 邏輯位址即由＜分段號碼，偏移量＞這樣的組合所表示
- 作業系統必須為每個行程管理一份分段表(Segment table)
- 分段表中每一項紀錄包含:
  - 基底值(base) –分段在主記憶體中的起始位址
  - 界限值(limit) –分段的長度
  - 保護用途欄位
    - validation bit = 0 → 不合法分段
    - read/write/execute 存取權限
- 用分段表基底暫存器(Segment Table Base Register，STBR)儲存分段表的起始位置。
- 用分段表長度暫存器(Segment Table Length Register，STLR)記錄分段表的長度。

分段法位址轉換


s
limit
base
segment table
CPU
s
d
<
yes
+
no
trap: addressing error
physical memory

分段法的實例

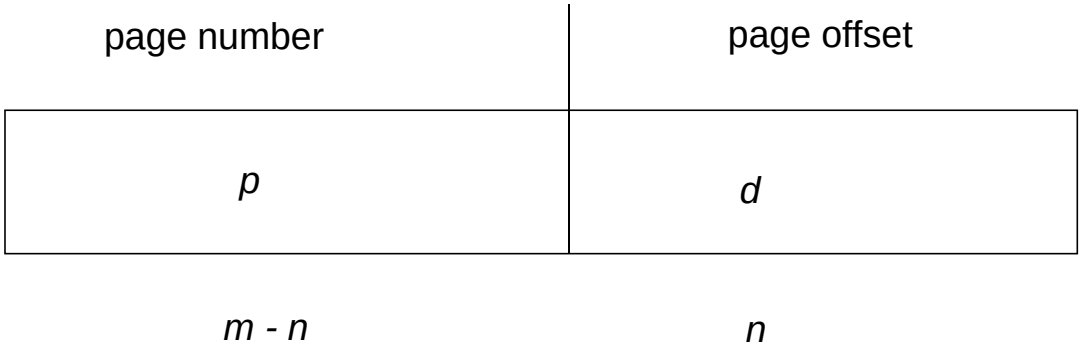

subroutine
stack
segment 3
symbol table
segment 0
segment 4
Sqrt
main program
segment 1
segment 2
logical address space
limit base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory

分頁法(Paging)

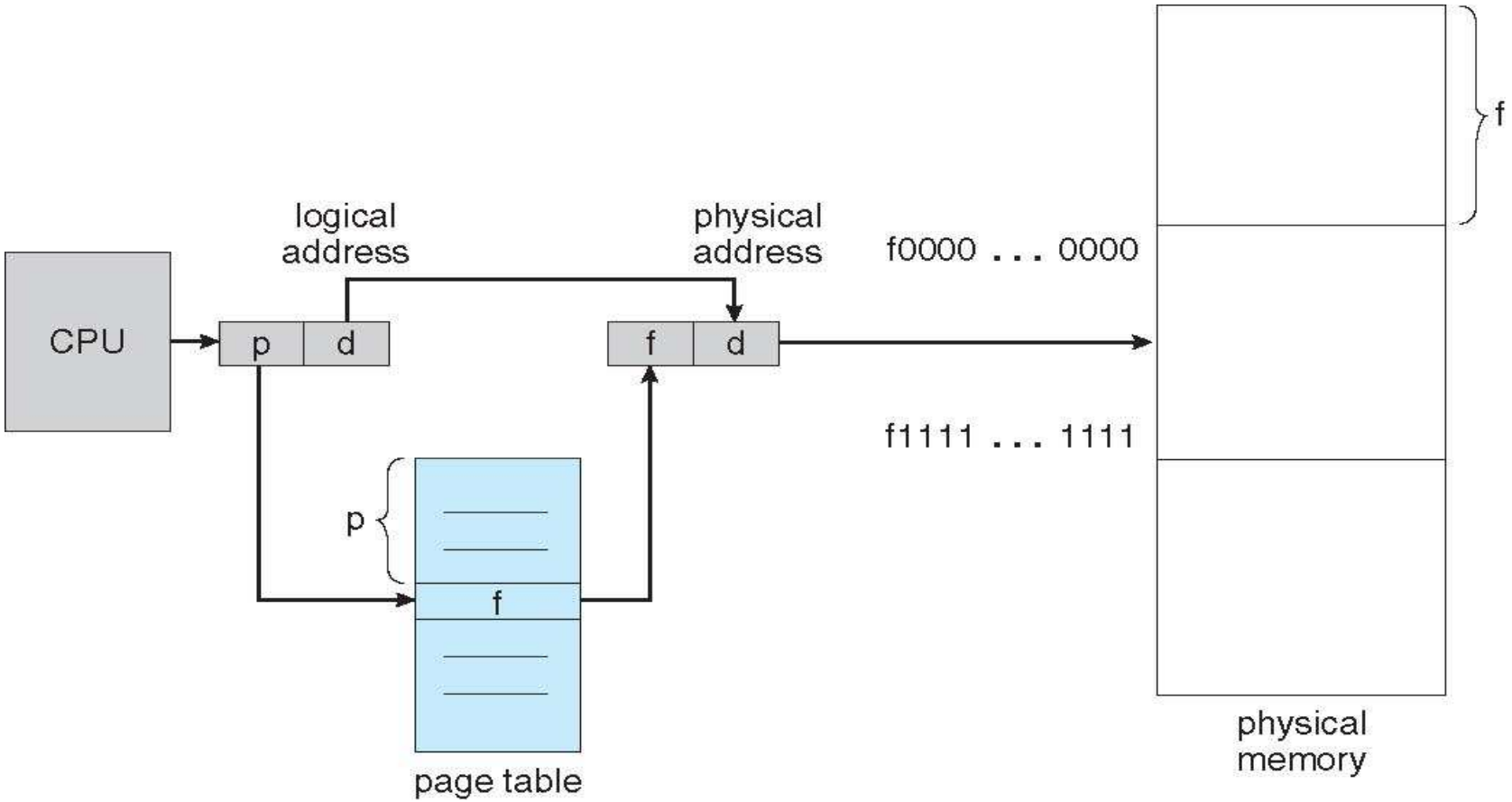
- 可以解決外部斷裂的問題。
- 允許行程內的邏輯位址空間可以不連續。
- 只要實體記憶體有足夠的可用空間，整個行程就一定可以載入記憶體。
- 把行程的邏輯位址空間切割成同樣大小的區塊，稱為分頁(page)
  - 大小是2的次方bytes
- 實體位址空間也切割為相同大小的單位，稱為頁框(frame)
- 在行程還沒載入之前，所有的分頁都儲存在備份儲存體上。
- 行程的任何一個分頁只要找到某個未被使用的頁框，就可以放到實體記憶體中
- 每個行程都有一個分頁表(page table)，用來儲存每個分頁所在的頁框編號
- 內部斷裂的情形依然存在

邏輯與實體位址空間的轉換

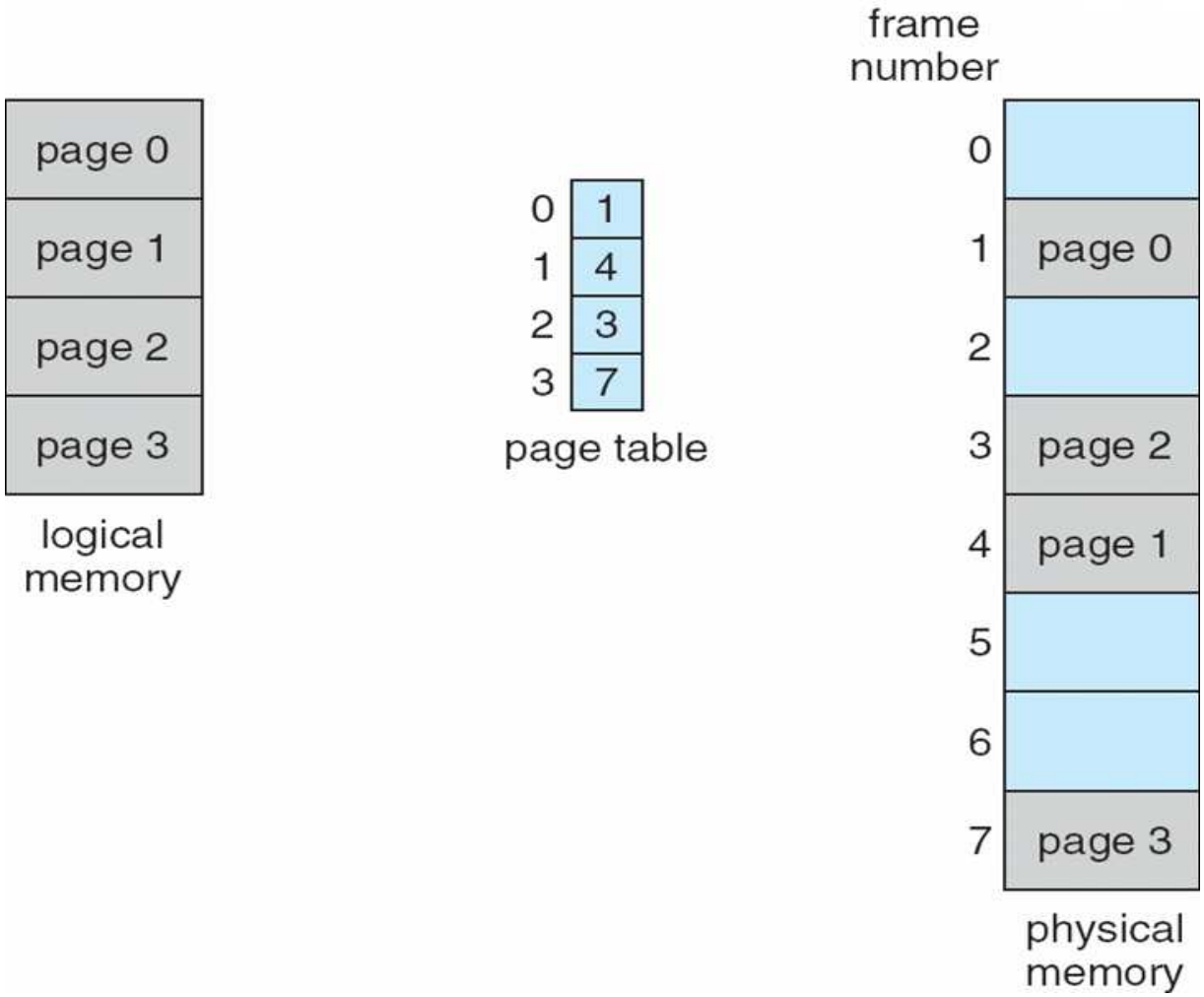
- CPU產生的邏輯位址被分成：
  - 頁編號(Page number , p)
  - 頁偏移量(Page offset , d)



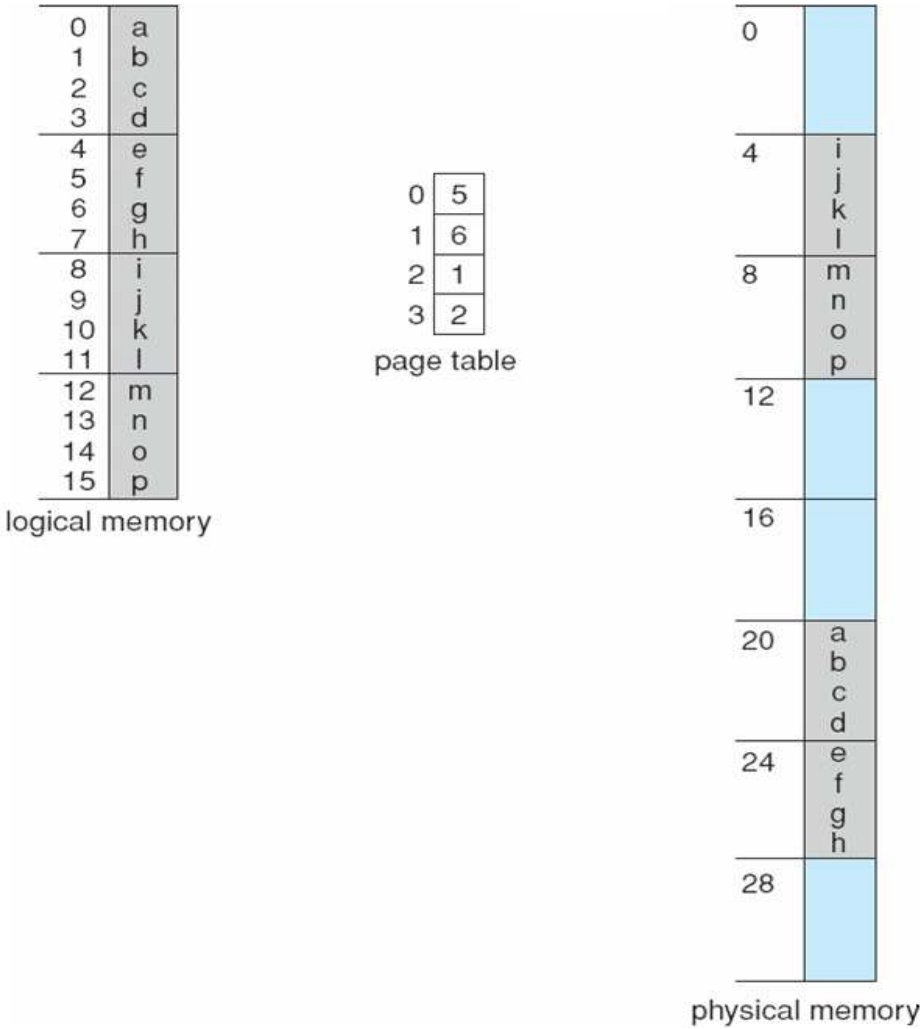
分頁法位址轉換


CPU
logical address
p
d
physical address
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
page table
physical memory

邏輯與實體位址空間的分頁範例



邏輯與實體位址空間的分頁範例

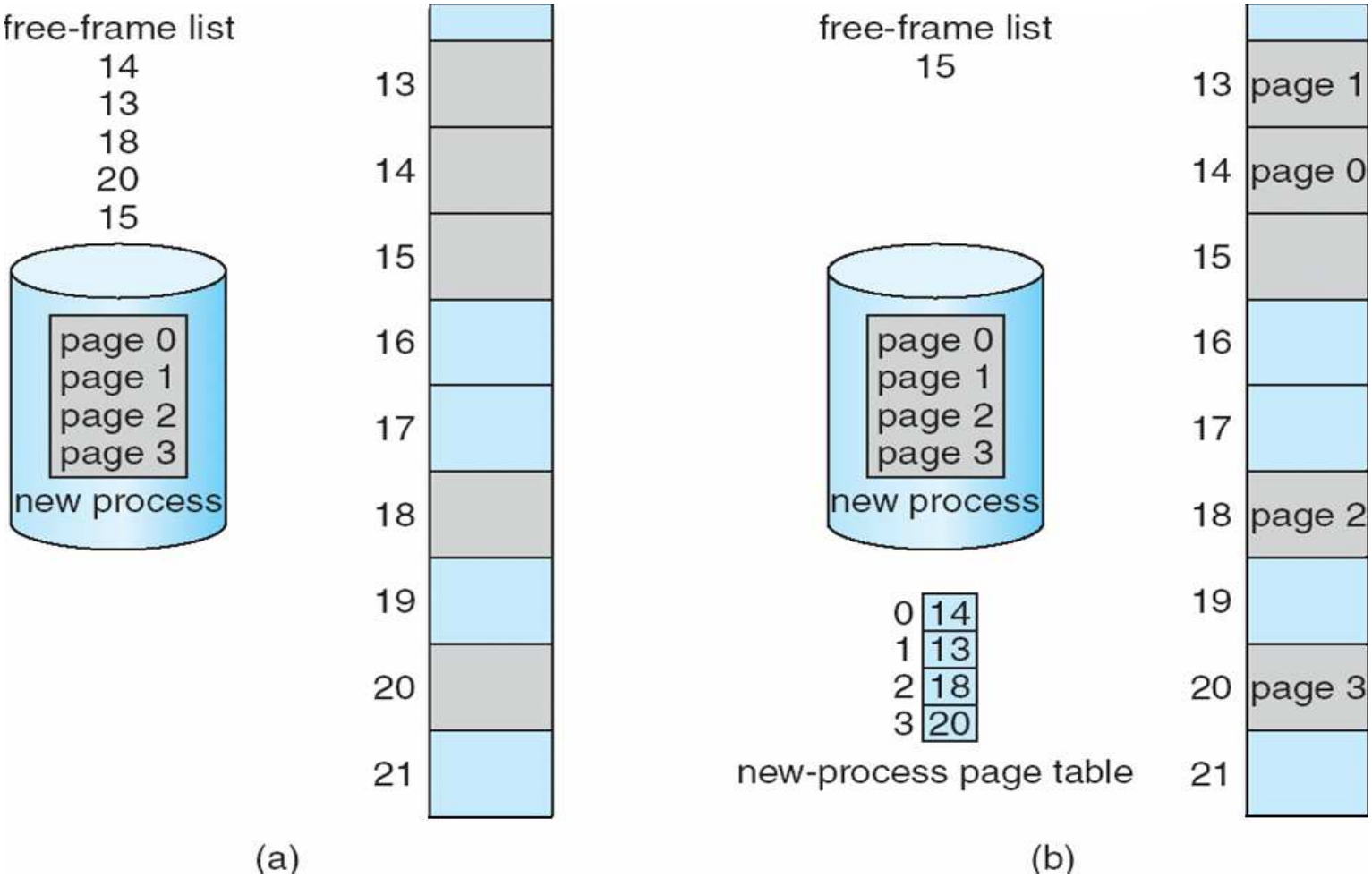


n=2 and *m*=4 32-byte memory and 4-byte pages

分頁法的內部斷裂情況

- 內部斷裂大小的平均值為分頁長度的一半
- 頁面越小，內部碎片所造成的浪費也就越小，但會造成儲存與管理分頁表上的負擔，以及降低磁碟輸入/輸出的效率
- 計算內部斷裂範例
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte

可用頁框(Free Frames)的管理



Before allocation

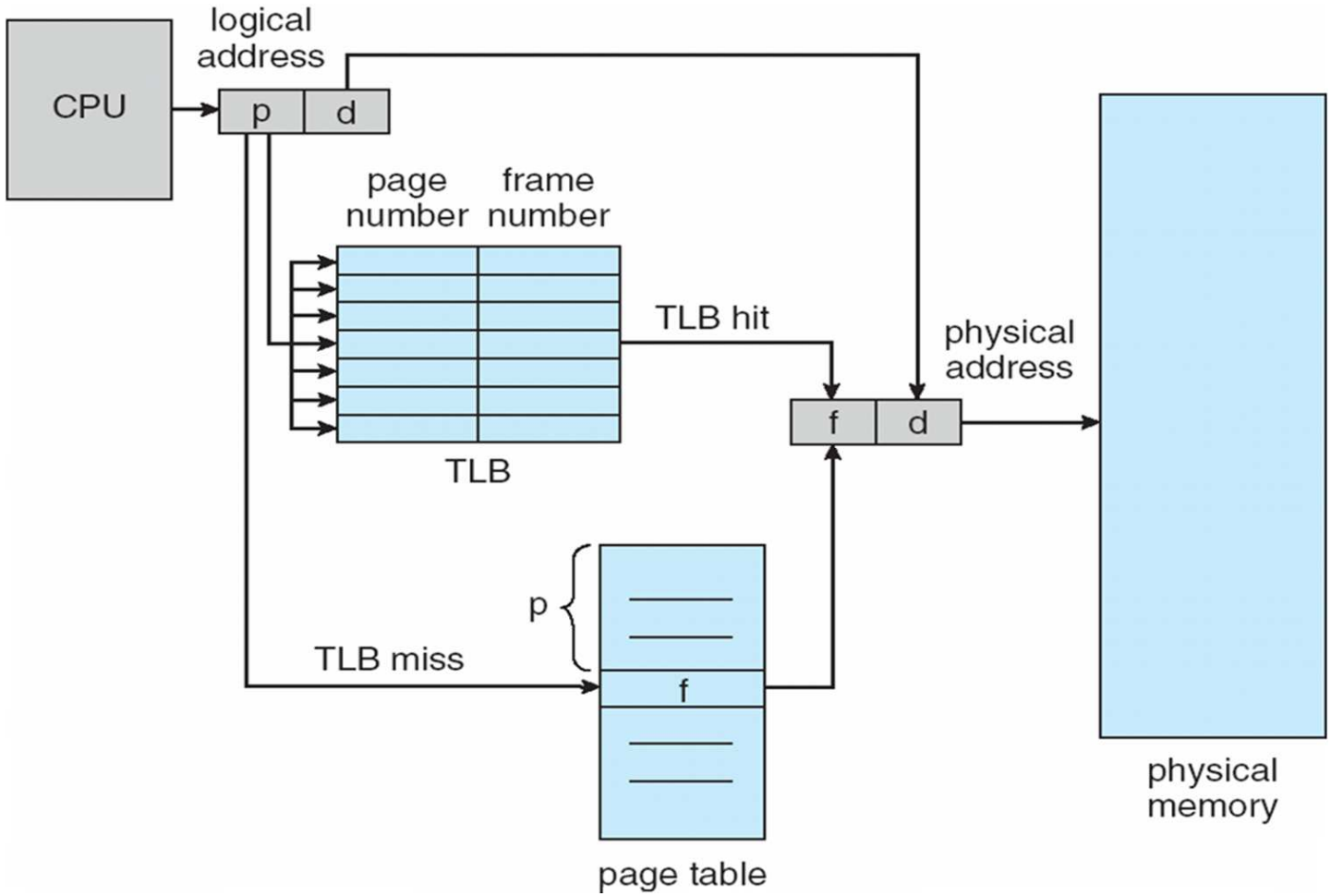
After allocation

分頁表的實作

- 一般的作業系統都是為每一個行程準備一個分頁表，並儲存於主記憶體。
- 用分頁表基底暫存器(Page-table base register , PTBR)來儲存分頁表的基底位址 。
- 用分頁表長度暫存器(Page-table length register , PTLR)來儲存分頁表的的大小。
- 每個資料/指令的存取需要兩次記憶體存取
  - 一次是分頁表，另一次是資料/指令的存取
- 兩次記憶體存取的問題可以使用關聯式記憶體或轉譯旁觀緩衝區(translation look-aside buffer, TLB) 來解決

關連式記憶體(Associative Memory)

- TLB是由特別的高速記憶體所組成，可進行平行資料搜尋比對，加快搜尋速度



加入TLB的分頁法效能的評估

- 兩種評估指標
  - TLB的命中率(hit ratio):
    - 要存取的分頁資料可以在TLB中找到的機率 。
  - 有效記憶體存取時間(effective memory access time) :

    [有效記憶體存取時間] =

    [命中率] × [在TLB可以找到分頁的記憶體存取總時間] +

    (1-[命中率]) × [在TLB找不到分頁的記憶體存取總時間]

有效存取時間(Effective Access Time)

- TLB關聯記憶體查詢 = ε 時間單元
  - 可能小於記憶體存取時間的 10%
- 擊中率(Hit ratio) = α
  - 擊中率 – 頁編號可以在關連記憶體找到的比例;
- 有效存取時間 (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

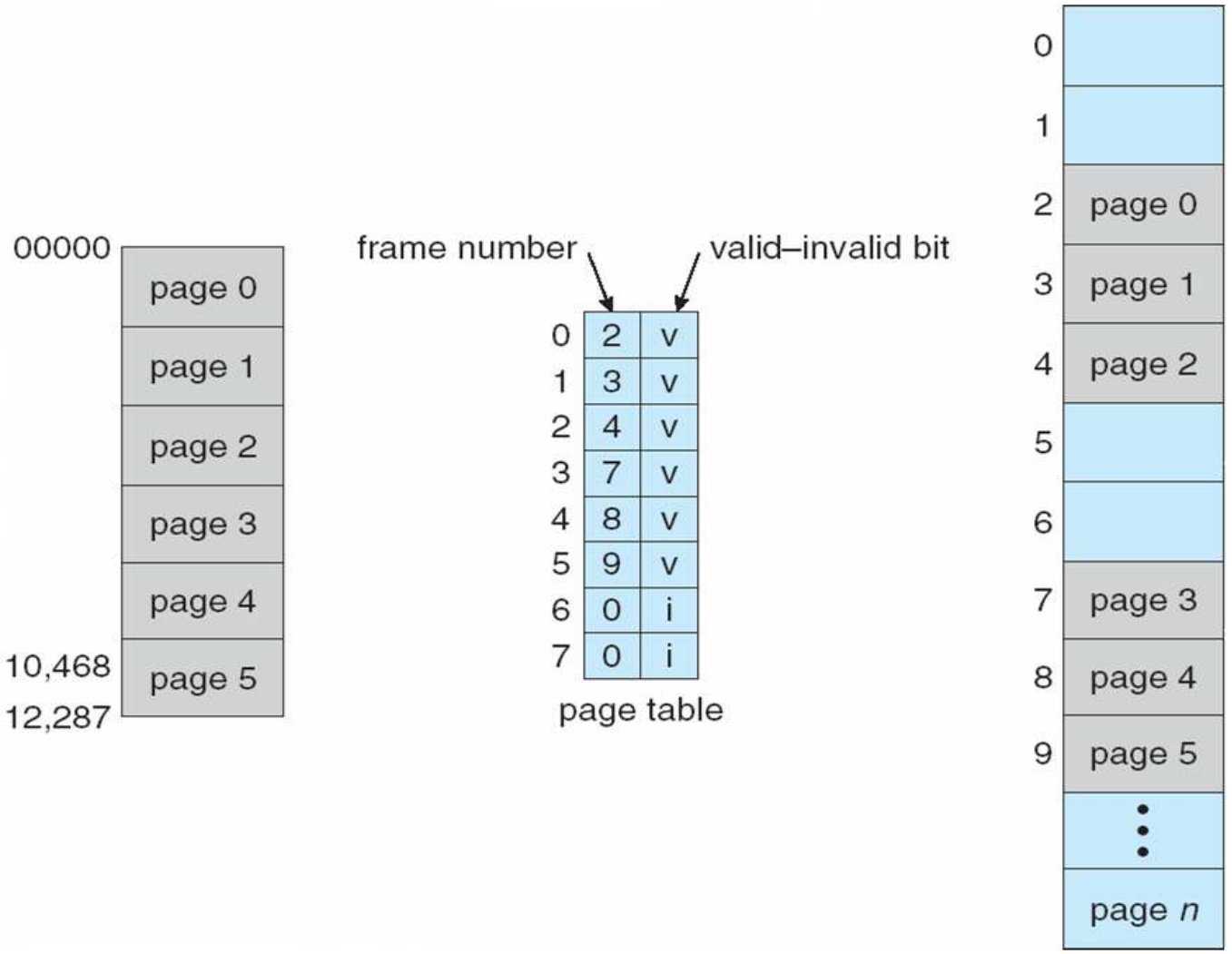
$$= 2 + \varepsilon - \alpha$$

- 考慮α=80%, ε=20ns, 記憶體存取時間=100ns
  - EAT = 0.80 x 100 + 0.20 x 200 = 120ns
- 考慮α=99%, ε=20ns, 記憶體存取時間=100ns
  - EAT = 0.99 x 100 + 0.01 x 200 = 101ns

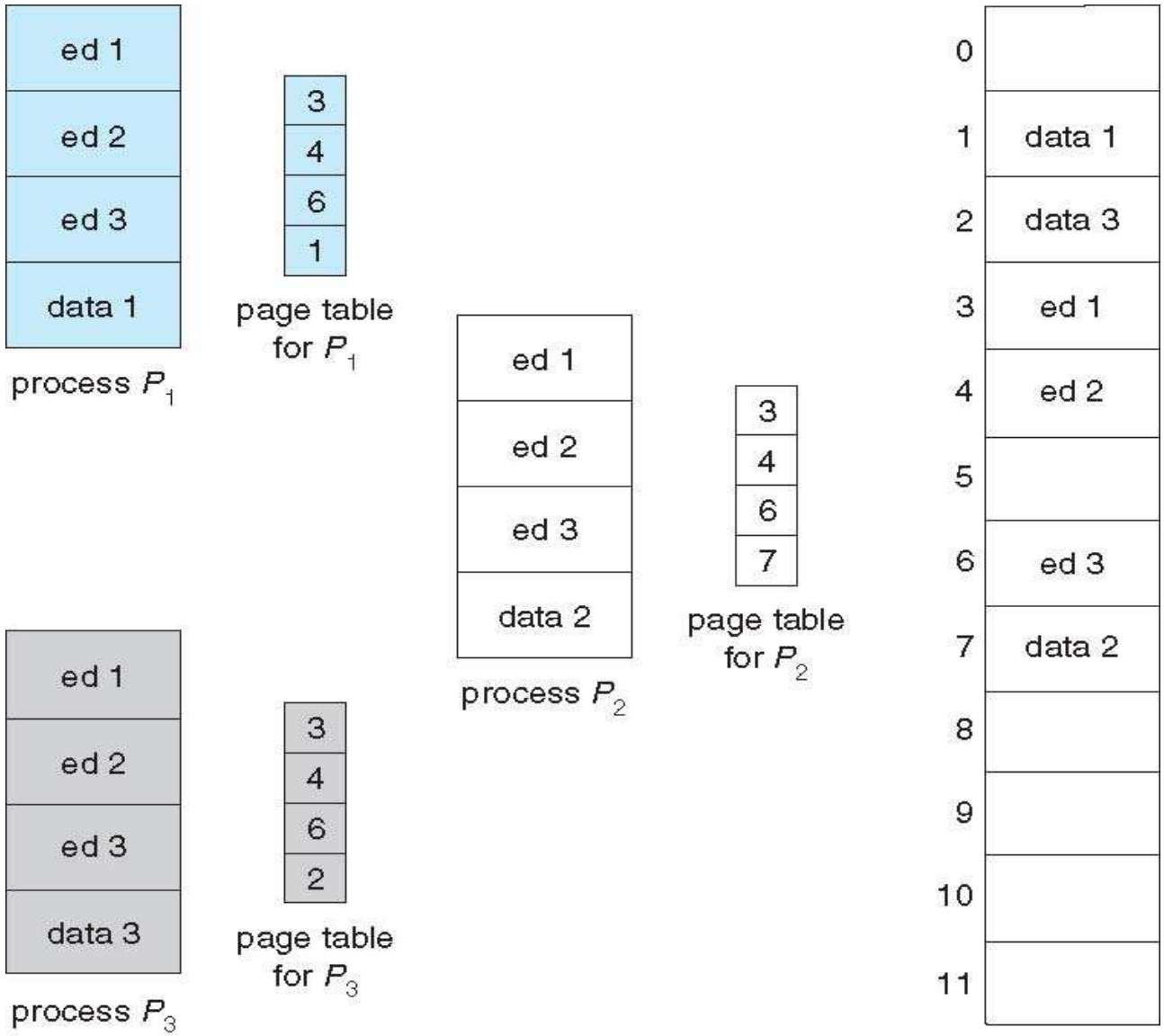
記憶體保護(Memory Protection)

- 藉由每一個頁框的保護位元來表示是唯讀或是可讀-寫存取
  - 也可以加入更多的保護位元來支援更多種存取權限，如：execute-only等 。
- 在分頁表的每一項加入一有效(valid)- 無效(invalid) 位元：
  - “有效”指出該分頁是在行程的邏輯地址空間 ，因此是一個有效分頁
  - “無效” 指出該分頁不在行程的邏輯地址空間，所以無效
  - 或使用page-table length register (PTLR)來判斷亦可
- 任何違反行為都會引發陷阱(trap)而由作業系統接管

分頁表的有效 (v)/無效 (i) 位元



共用分頁範例


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1 data 1
2 data 3
3 ed 1
4 ed 2
5
6 ed 3
7 data 2
8
9
10
11

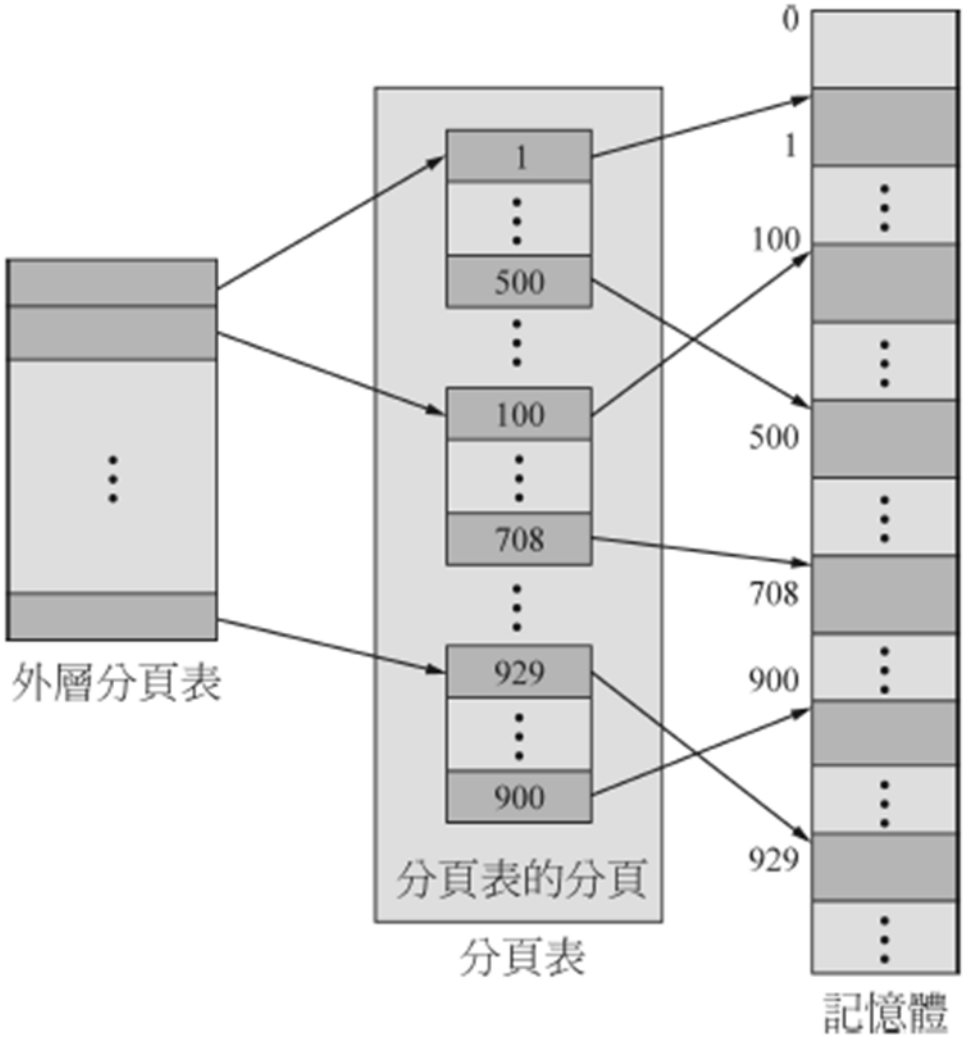
分頁表的結構

- 如下情形，所需的分頁表有可能會非常大
  - 使用32位元的邏輯位址空間其大小為2^{32} Bytes
  - 分頁大小是 4 KB (2^{12})
  - 分頁表會有2^{20}項　($2^{32}/2^{12}$)
  - 如果每一項長度是4 Bytes，分頁表就需要4MB的實體記憶體
    - 使用的記憶體相當大
    - 不希望把分頁表擺在連續的記憶體空間上，其結構可有以下幾種做法：
      - 階層式分頁表(Hierarchical Paging)
      - 雜湊分頁表(Hashed Page Tables)
      - 反轉分頁表(Inverted Page Tables)

階層式分頁表(Hierarchical Page Tables)

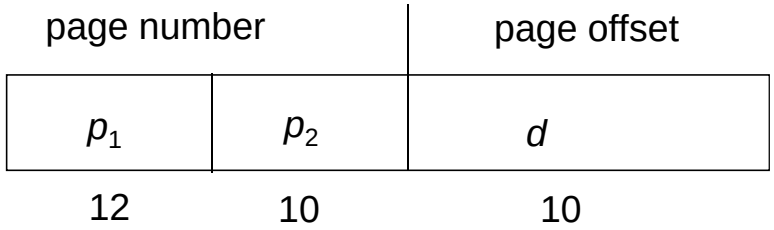
- 把邏輯地址空間分成幾個分頁表
- 分頁表本身也以分頁的方式存放
  - 最簡單的方法就是使用雙層分頁表

雙層分頁表範例


0
1
100
500
708
900
929
1
500
100
708
929
900
外層分頁表
分頁表的分頁
分頁表
記憶體

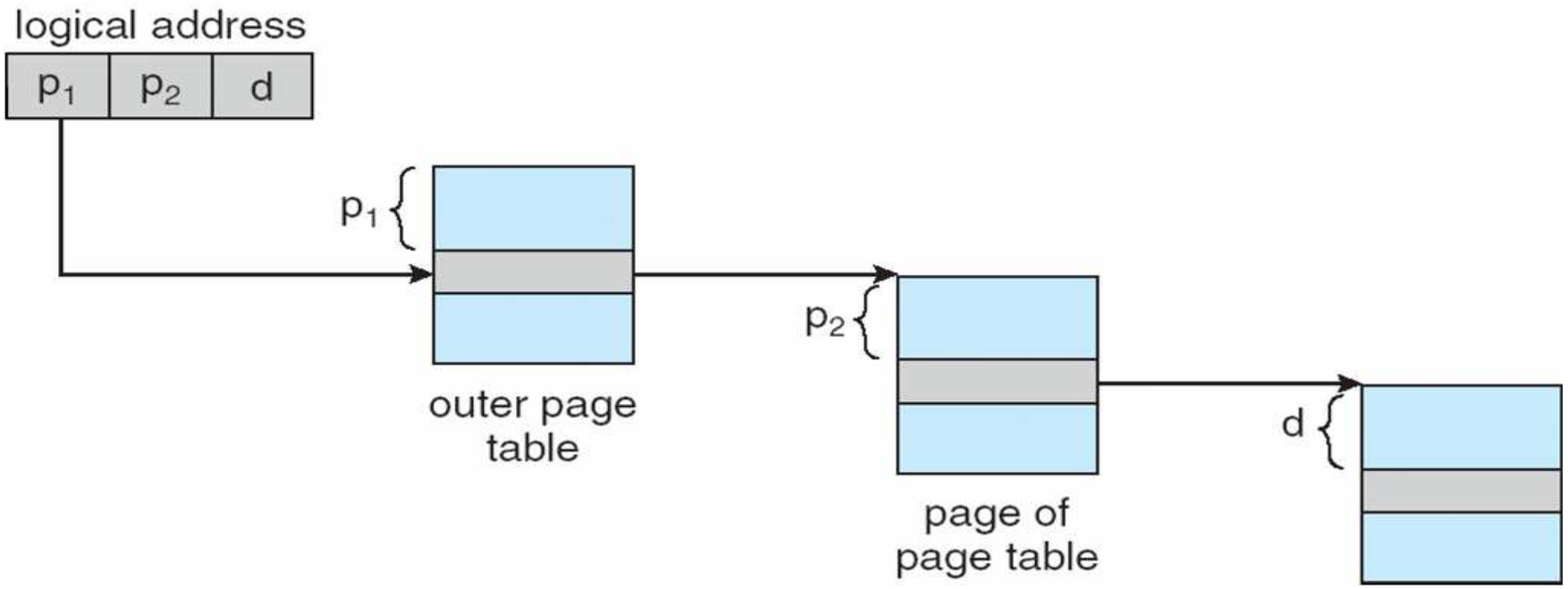
雙層分頁表範例

- 32位元邏輯地址被分成兩欄(分頁大小是1KB)：
  - 分頁號碼佔22位元
  - 分頁偏移量佔10位元
- 因分頁表被分頁，22位元的分頁號碼進一步被分成兩部份:
  - 12位元的分頁號碼(page number), p_1
  - 10位元的分頁偏移量(page offset), p_2
- 因此邏輯位址可以如圖所示：



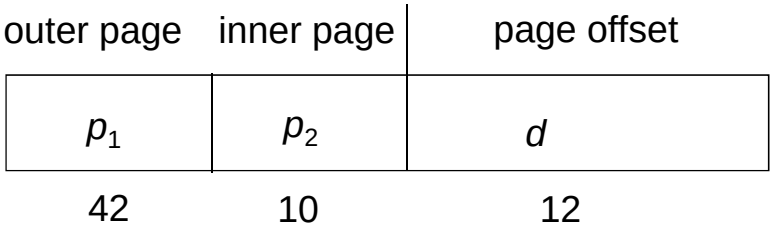
- 其中p_1是指向外層分頁表(outer page)的索引值，而p_2則是內層分頁表(inner page)所指的分頁表之偏移量

雙層分頁表的位址轉換


logical address
p1
p2
d
p1
outer page table
p2
page of page table
d

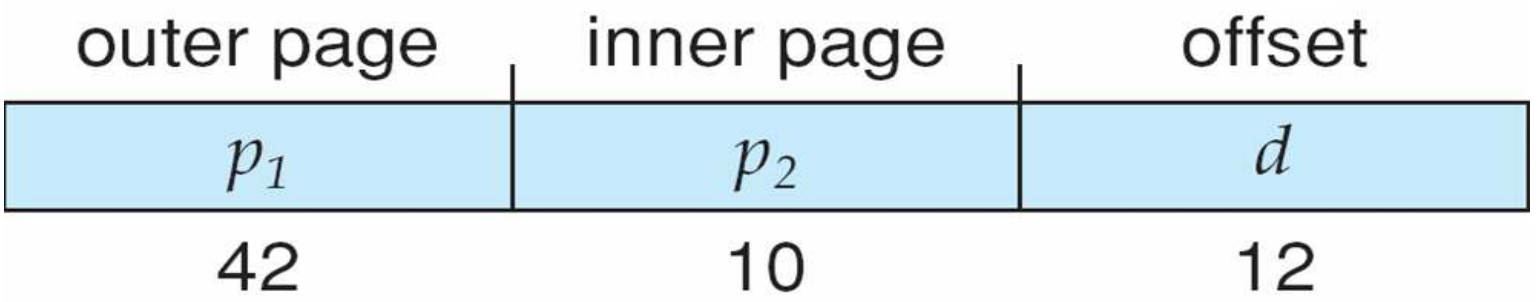
64位元邏輯地址空間

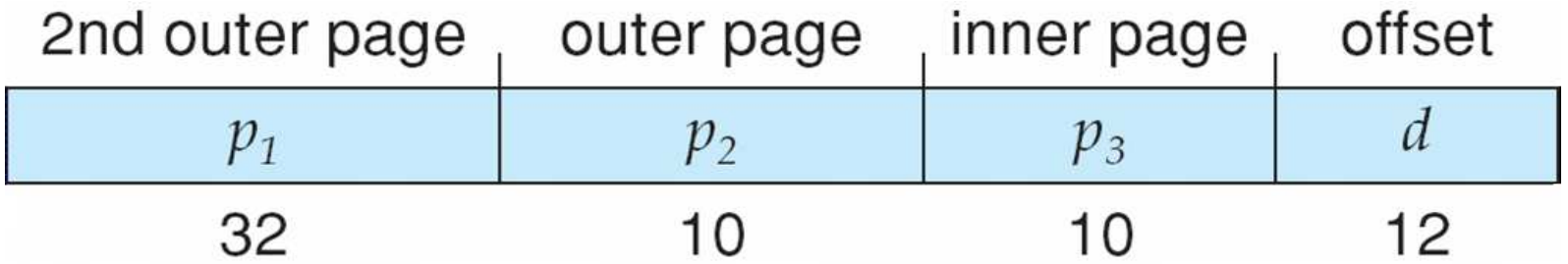
- 當是64位元的邏輯地址空間，雙層分頁方法也無法符合需求，如下情形：
  - 分頁的大小是 4 KB (2^{12})
    - 分頁表有2^{52} 項
    - 內層分頁表 可容納2^{10} 項目，每一項目4 Bytes
    - 位址將變成



    - 外層分頁表有2^{42} 項
    - 一個解決方法是加入第2個外層分頁表
    - 但是在這例子，第2個外層分頁表依然有2^{34} 位元組
      - 而且可能需要4次記憶體存取才能獲得實體記憶體的位置

三層分頁表情形

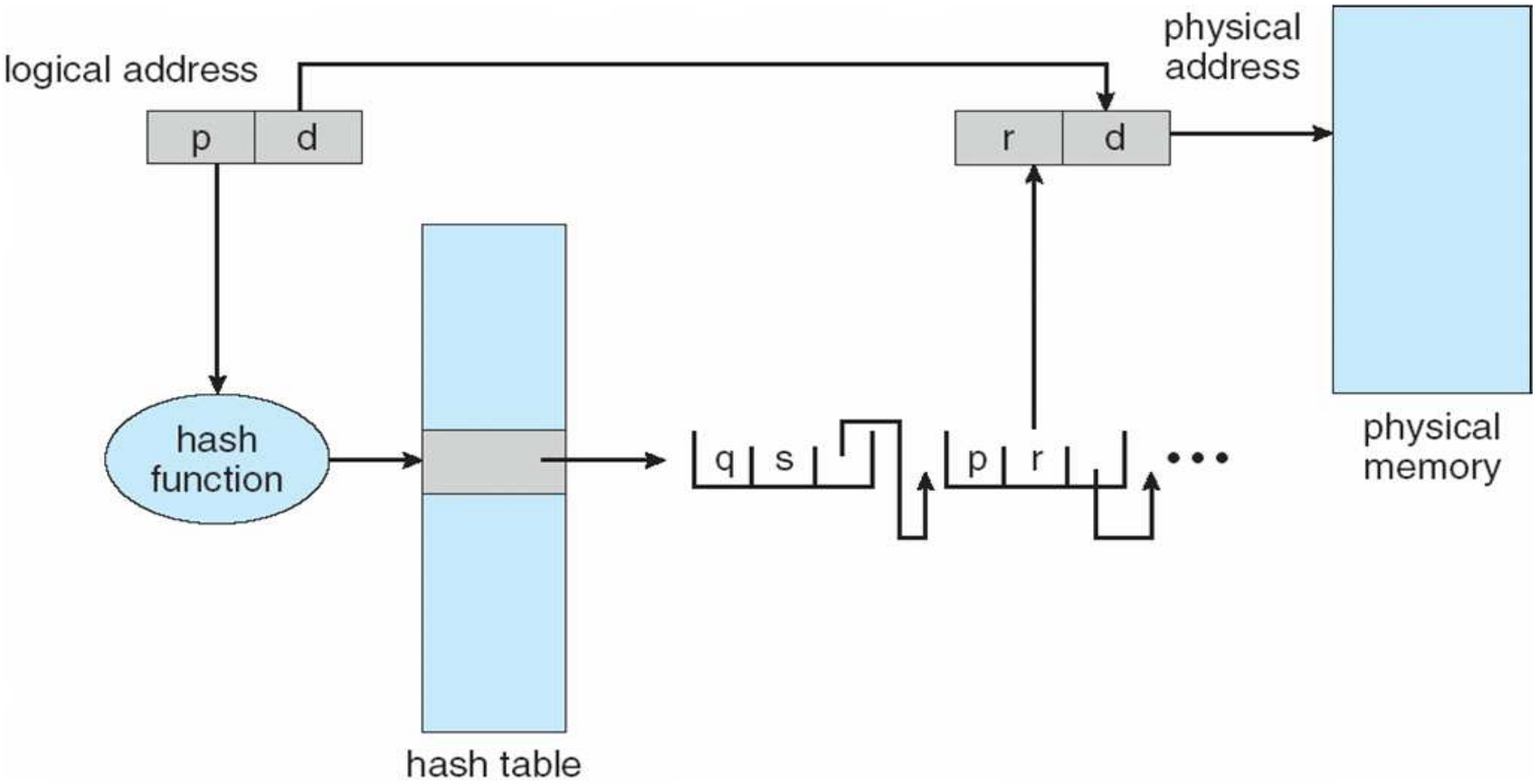

outer page
inner page
offset
p_1
p_2
d
42
10
12


2nd outer page
outer page
inner page
offset
p_1
p_2
p_3
d
32
10
10
12

雜湊分頁表(Hashed Page Tables)

- 常見於位址空間 > 32位元
- 虛擬分頁編號(virtual page number)進行雜湊計算得到分頁表中該項存放位置
  - 分頁表中每一項包括了雜湊到相同位置的分頁串列
- 分頁串列的每一項包括
  - 虛擬分頁編號
  - 頁框編號(frame number)
  - 指向鏈結串列下一節點的指標
- 將虛擬分頁編號與雜湊計算得到的鏈結串列中的虛擬分頁編號做比較
  - 如果兩者吻合，相關的頁框編號就被取出來

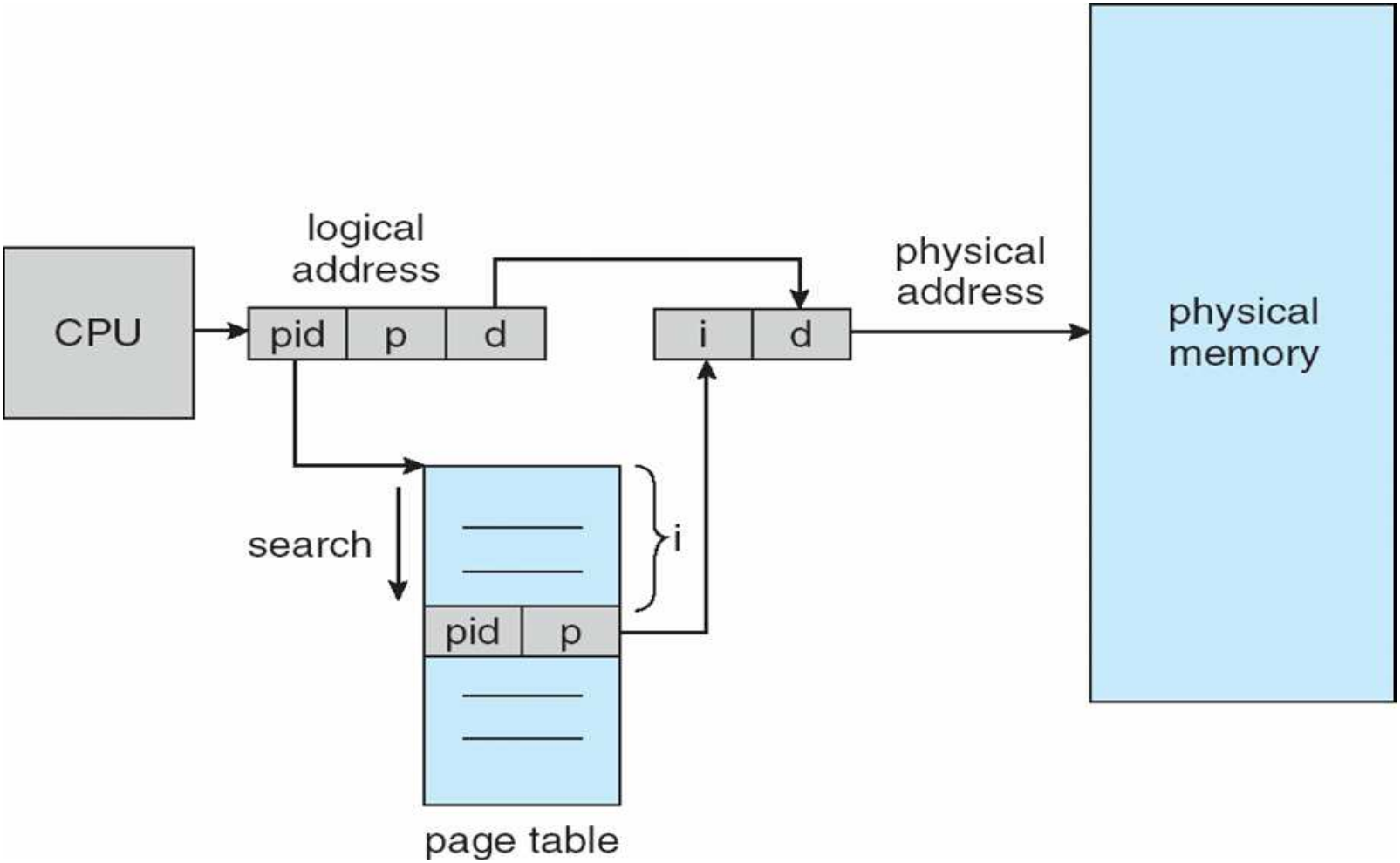
雜湊分頁表的位址轉換


logical address
p
d
physical address
r
d
hash function
hash table
q
s
p
r
physical memory

反轉分頁表(Inverted Page Table)

- 反轉分頁表是記錄所有實體頁框被使用情形，一個頁框一個項目
- 全部行程僅有一個反轉分頁表，不是每一個行程都有自己的一份分頁表
- 每一項中包含了此實體頁框所儲存的虛擬分頁，以及擁有該頁的行程資訊。
- 可降低儲存每一份分頁表所需的記憶體空間
- 當發生分頁參考(page reference)時，因需要搜尋整個反轉分頁表，處理時間會增加
- 無法支援共用記憶體

Inverted Page Table Architecture



範例: Intel 32 和 64-bit 架構

- Pentium CPUs 是32位元，稱為IA-32架構
- 目前Intel CPUs是64位元，稱為IA-64架構

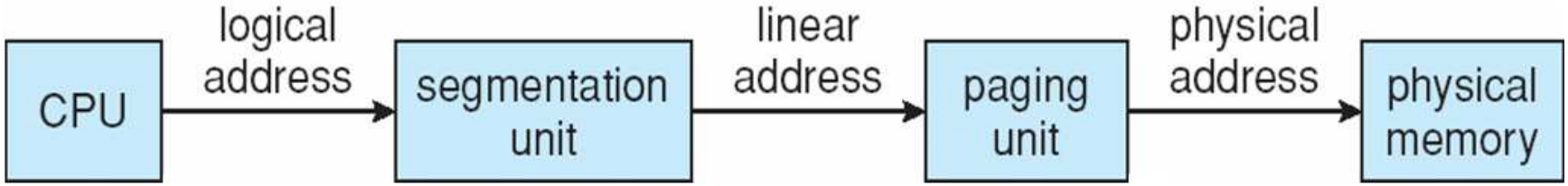
Intel 32 架構

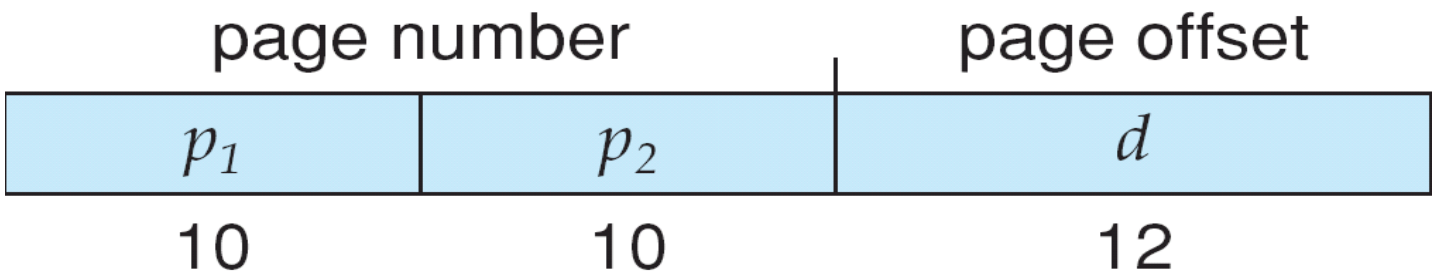
- 支援分段(segmentation)與分頁式分段(segmentation with paging)
  - 每一個分段可以到達 4 GB
  - 每一個行程可以到達16 K個分段
  - 分成兩部分
    - 第一部分可以到達8 K 個分段，是行程私有 (存放在區域描述表(local descriptor table, LDT中)
    - 第二部分可以到達8 K 個分段，是所有行程共用 (存放在全域描述表**(global descriptor table, GDT** 中)

Intel 32 架構(繼續)

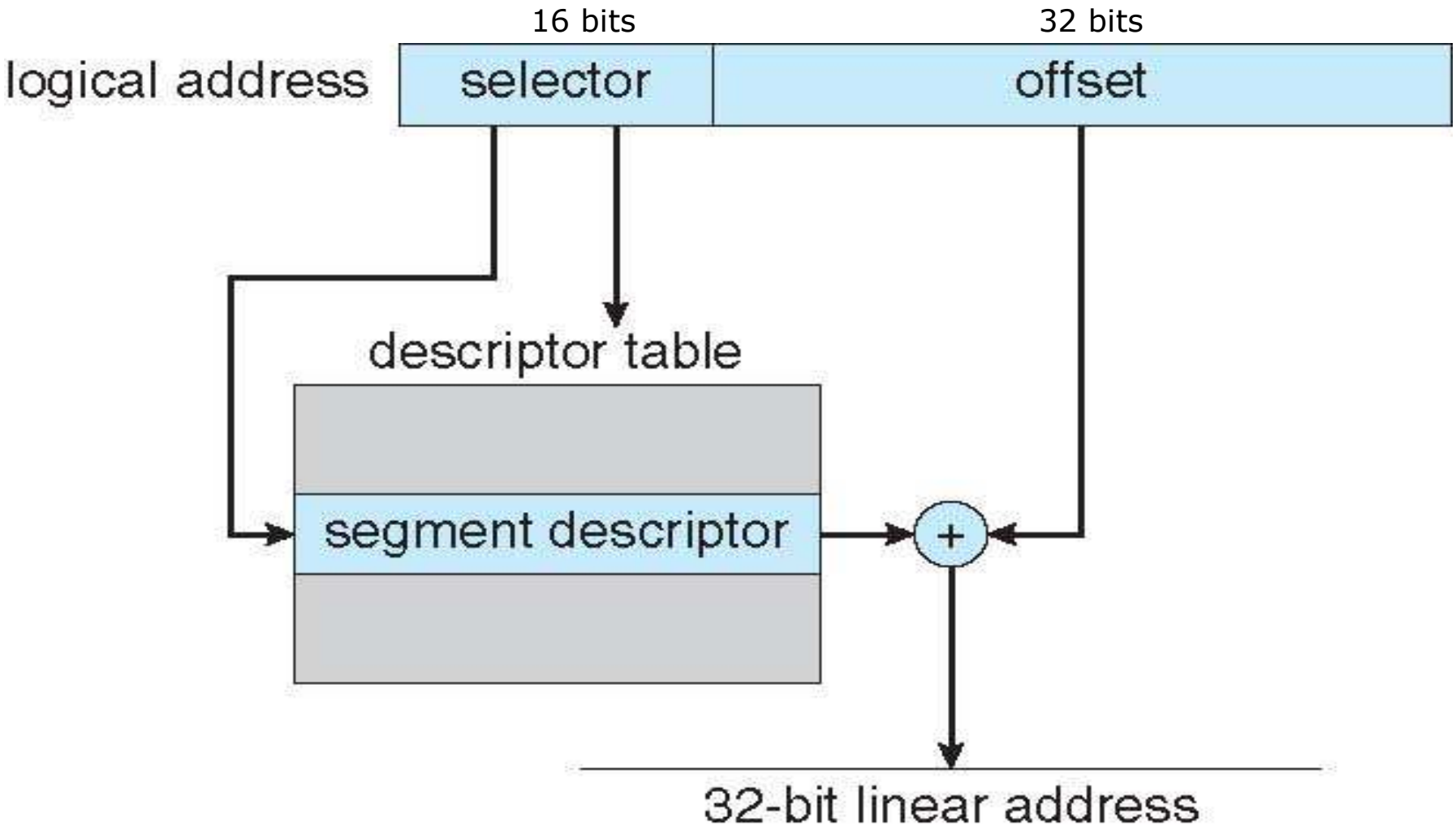
- CPU 產生邏輯地址
  - 選擇器送給分段單元
    - 用來產生線性地址(linear addresses)
  - 線性地址送給分頁單元
    - 產生主記憶體的實體地址
    - 分頁單元形成對應的MMU
    - 分頁大小可以是4 KB或4 MB

IA-32中邏輯對實體位址轉換

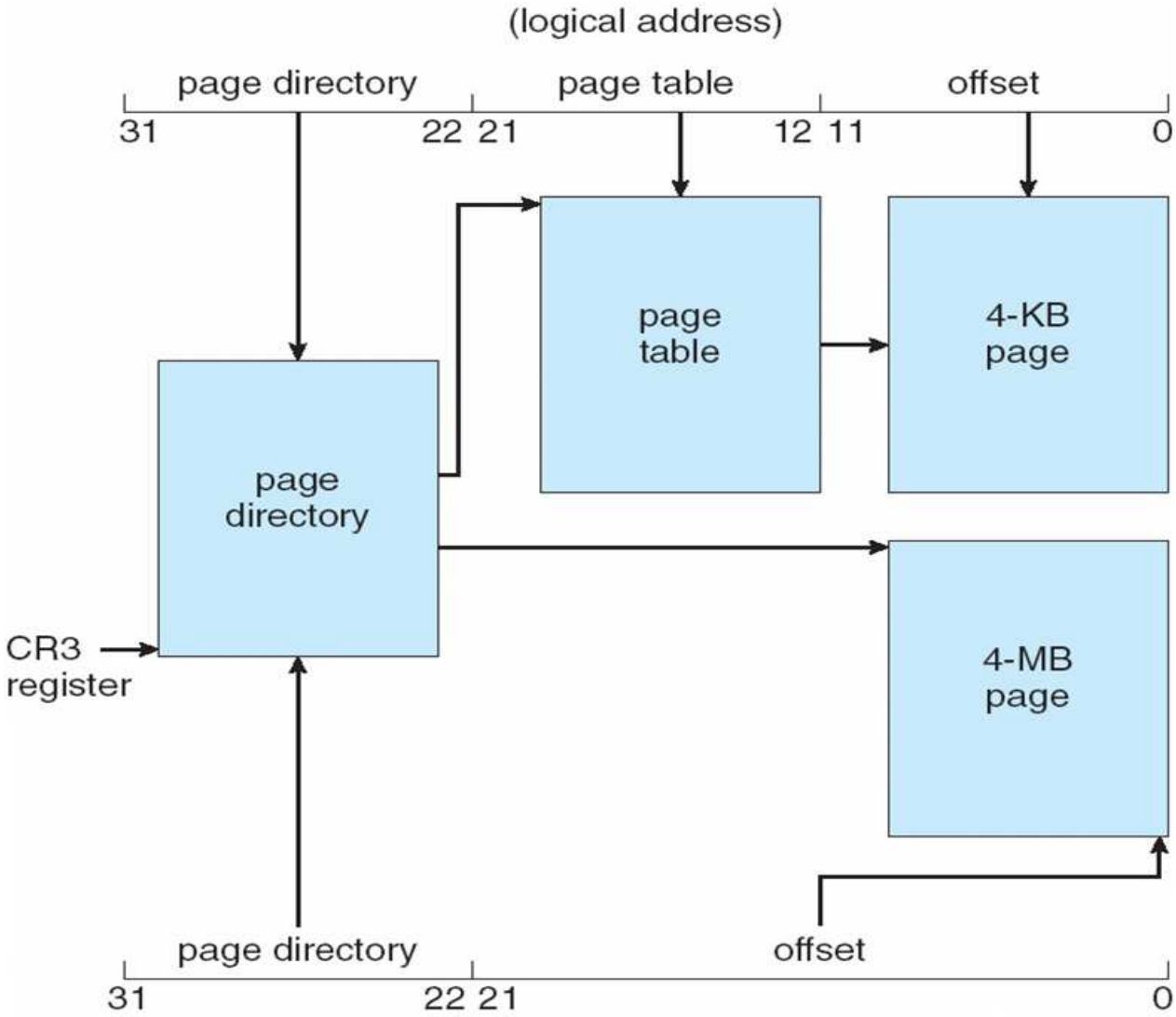




Intel IA-32 Segmentation

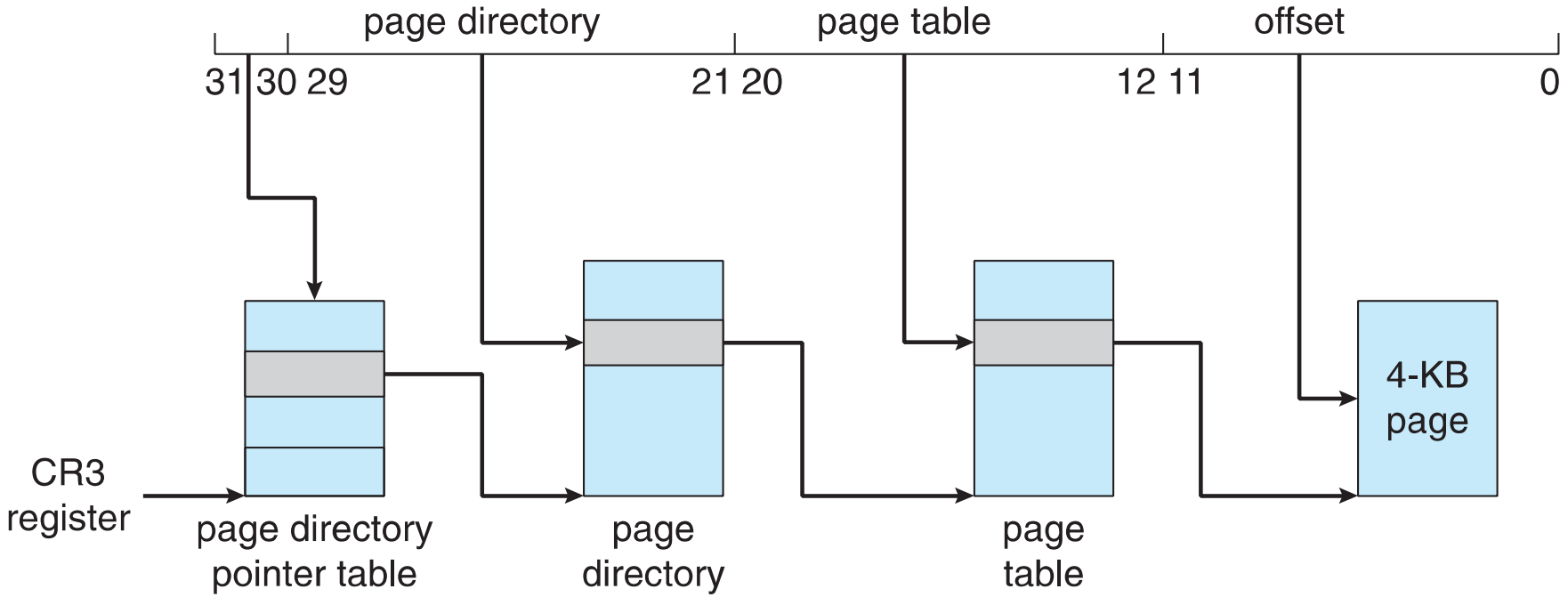


Intel IA-32分頁架構



Intel IA-32 Page Address Extensions

- 32位元地址的限制導致Intel創造了page address extension (PAE), 讓32位元應用程式存取超過4GB的記憶體空間
  - 分頁變成3層的技術
  - 最前面2位元參考到分頁目錄指標表格(page directory pointer table)
  - 分頁目錄和分頁表格變成64位元大小
  - 形成36位元位址實體位址空間增加為64GB



Intel x86-64

- 目前Intel x86 架構的版本
- 64 位元（> 16 exabytes）
- 實際上只製作了48 bit 定址
  - 分頁大小是4 KB, 2 MB, 1 GB
  - 四層的分頁架構
- 也可以使用 PAE，所以虛擬地址是 48 位元，而實體地址是52位元