虛擬記憶體

虛擬記憶體

- 目的: 允許 Program Size 大於 Physical Memory Size 仍然可以執行
- 達成策略: 採用 Partial Loading
  - Dynamic Loading
    - 執行期間當副程式被呼叫到，才將他載入記憶體，不浪費記憶體空間
    - **Programmer 負擔:** Programmer 需要知道哪些副程式互斥，OS 只提供 Loader
  - Virtual Memory
    - OS 負擔，Programmer 無負擔
- 優點
  - **允許 Program Size 大於 Physical Memory Size** 仍然可以執行
  - Programmer 不需考慮 Program Size
  - 記憶體各個小空間皆有機會被用到，**記憶體利用率上升**
  - 盡可能提供 Multiprogramming Degree，**CPU 利用率上升**
    - 程式不用將所有 Page 搬入記憶體才能執行
    - 原本系統只能執行 5 個程式，現在可以執行 20 個

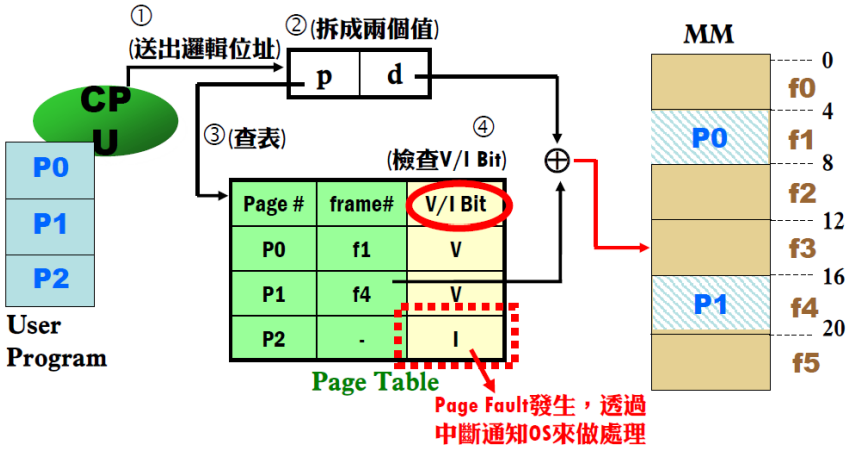
- 每一次 I/O Transfer Time 下降，
  - 只需要載入程式的部分 Page，不用載入程式的所有 Page
  - **如過要載入整個程式**反而很耗費 I/O Transfer Time，因為充數次數變多

實現 Virtual Memory 技術

- 使用 **Demand Paging(需求分頁)** 技術
- 以 **Page Memory Management 為基礎**發展出來，不同的是採用 **Lazy Swapper**
  - 程式執行之初，不將全部 Page 載入記憶體，**僅載入執行所需的 Page** 到記憶體，其餘的 Page 置於 Blocking Store，若所需的 Page 不在記憶體，OS 需處理 **Page Fault 問題**

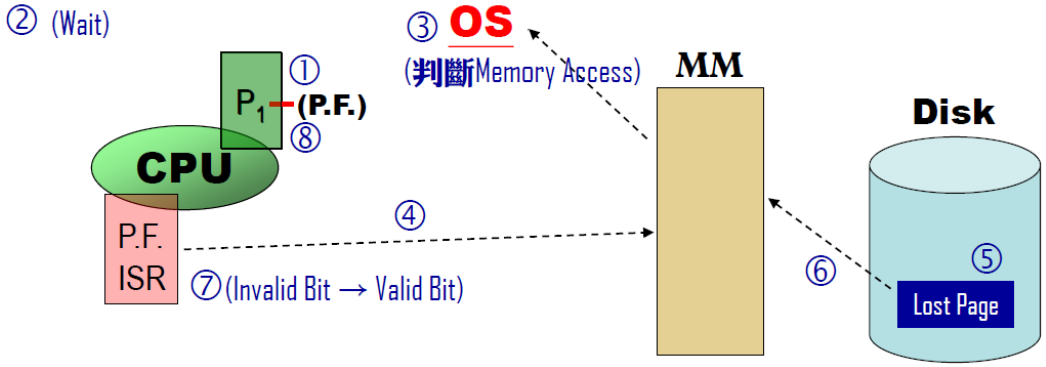
分頁表配合修正

- 新增一欄 **Valid/Invalid Bit**，表示 Page 是否在記憶體中
  - **Valid** 表示 Page **在記憶體中**
  - **Invalid** 表示 Page **不在記憶體中**



Page Fault 定義和處理程序

- 在 Demand Paging 的 Virtual Memory 系統中，若執行中的 Process 存取不在記憶體的 Page 則發生 Page Fault



- 處理程序

  1. OS 收到 Page Fault 所引起的中斷
     - 由 **Memory Management Unit** 發出

2. OS 暫停目前 Process 執行，保存此 Process 狀態
3. OS 判斷此 **Memory Access 是否合法**
   - 合法-->Page Fault
   - 非法-->終止 Process
4. OS 去 Memory 檢查**有沒有 Free Frame**(Page Fault 中斷服務程式 ISR)
   - 沒有-->必須執行 Page Replacement 空出一個可用頁框
5. OS 去 Disk 中**找到 Lost Page 所在的位置**
6. 將此 Lost Page **載入到可用頁框**
7. **更新 Page Table**，指明此 Page 所在的頁框，並將 Invalid Bit 改成 Valid Bit (Page Fault ISR)
8. **恢復**原先 Process 中段前的執行

Virtual Memory 效益評估

- Effective Memory Access Time 決定
  - 越短效益越好，越接近記憶體存取時間
- 公式: **(1 - p) * ma + p * (Page Fault Processing Time)**
  - p: Page Fault Ratio
  - ma: 正常的 Memory Access Time
  - Page Fault Processing Time: **Page Fault 處理程序的所有工作時間總和**
- 結論: 要降低 Effective Memory Access Time 必須**降低 Page Fault Ratio (p)**
  - ma 和 Page Fault Processing Time 是固定的
- 範例
  - ma = 200ns

    Page Fault Processing Time - 2ms

    p = 20%

    求 Effective Memory Access Time?
  - **Solution**
    - (1 - 0.2) * 200ns + 0.2 * 2ms

      = 0.8 * 200ns + 0.2 * 2 * 10^6ns

      = 160ns + 4 * 10^5ns

      = 400160ns

      = 400.16us

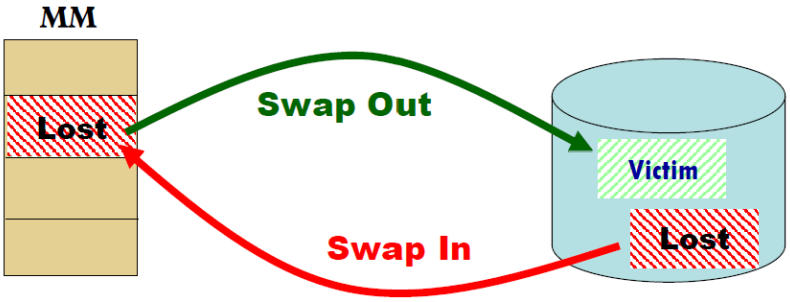
影響 Page Fault Ratio 的因素

- Page Fault Algorithm 的選擇
- 頁框數的分配多寡
- Page Size 大小

- Program Structure

Page Replacement

- 當 **Page Fault 發生**，且記憶體**沒有可用的頁框**，OS 必須執行 Page Replacement
  - OS 必須選擇一個犧牲者 Swap Out 到 Blocking Store 以空出一個頁框，再將 Lost Page 載入到此頁框



- Swap Out 和 Swap In 分別是兩個 I/O 動作
  - Swap In **是必要的**
  - Swap Out **不一定必要**，Victim Page **是否曾被修改**

是否可省略非必要的 Swap Out

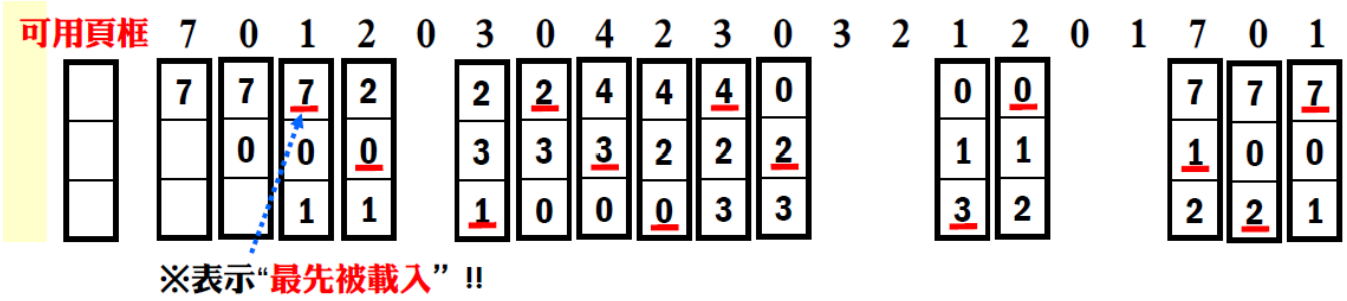
- **判斷依據：** Victim Page 從執行之初到準備被替換，是否曾被寫入或修改
  - Yes：需回存到 HD
  - No：可刪除或覆蓋 (和硬碟的資料內容相同，硬碟的資料不需要更新)
- 利用 Modification Bit (Dirty Bit) 來判斷，節省 Victim Page 的 Swap Out I/O 動作
  - Modification Bit = 0 表示 Page 未被修改-->**不用 Swap Out**
  - Modification Bit = 1 表示 Page 曾被修改-->**必須 Swap Out**
  - Modification Bit **初始值為 0**

Page Replacement Algorithms

- **挑選 Victim Page**

FIFO Algorithm

- **最先載入的 Page** (Loading Time 最小)，優先視為 Victim Page
- 範例：給予下列的 Page Reference String (頁面參考字串)，記憶體中的可用頁框有 3 個，OS 採用 **Pure Demand Paging** 和 **FIFO Replacement Algo.**，求 Page Fault 次數



- Page Fault 次數 = 15-->Page Fault Ratio = 15/20 = 75%

- **Pure Demand Paging**
  - 程式執行之初，**不預先載入任何 Page**
  - 缺點：執行之初會產生大量 Page Fault
  - 優點：載入的 Page 都是 Process 所需的頁面，因此後面的 Page Fault Ratio 會降低(趨於穩定)
- **Prepaging 預先分頁**
  - 猜測程式執行之初會用到哪些 Page，預先載入這些 Page
  - 優點：猜測精準，可以避免程式執行之初大量 Page Fault
  - 缺點：猜測錯誤，預先載入 Page 的 I/O 動作浪費
- 特質
  - 簡單易於操作
  - 效益差，Page Fault Ratio 最高
  - 可能有 **Belady 異常現象**
    - Process 分配到較多的 Frame 數目，Page Fault Ratio 反而上升

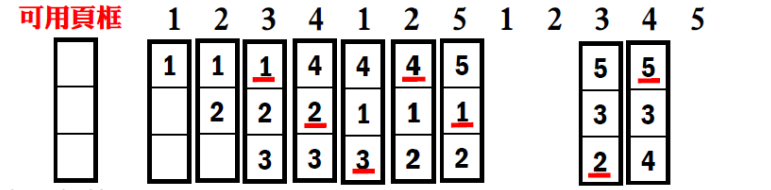
Belady Anomaly 範例(in FIFO Algo.)

- 案例

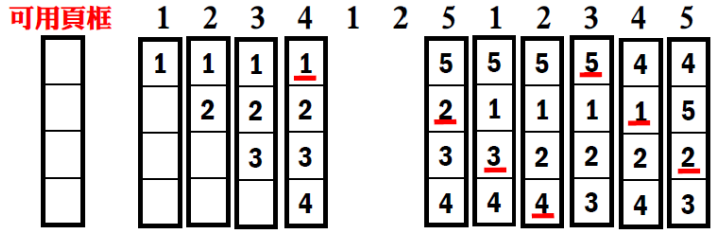
  Page Reference String :1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

  Page Frame = 3, Page Fault Ratio = 75%

  Page Frame = 4, Page Fault Ratio = 83.3%



有9次的Page Faults，Page Fault Ratio = 9/12 = 75%。



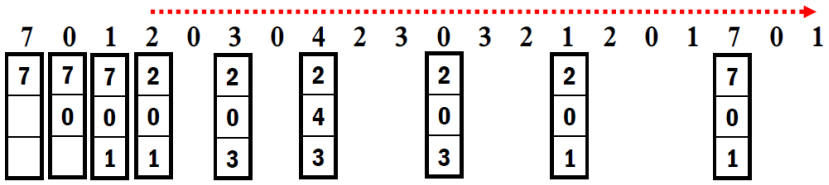
有10次的Page Faults，Page Fault Ratio = 10/12 = 83.3%。

- 原因：**FIFO 不具備 Stack 性質**

- **Stack 性質：n+1 個頁框所包含**的 Page 包含於**(n+1) 個頁框所包含的 Page**

OPT (Optimal) Algorithm

- **未來長期不使用**的 Page 視為 Victim Page
- 範例：OS 採用 Pure Demand Paging 和 OPT Algo，求 Page Fault Ratio
  - 以第四次 Page Fault 為例：7 下一次用到的時間比 0, 1 還久 ---> Swap Out 7 然後 Swap In 2

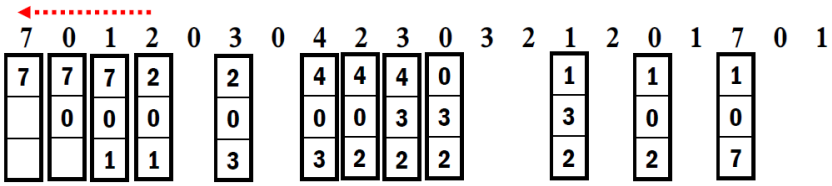


有9次的Page Faults，Page Fault Ratio = 9/20 = 45%。

- 特質
  - 效果最佳 Page Fault Ration 最低
  - 不會有 Belady 異常現象
  - **很難製作**，通常做為理論研究與比較之用

LRU (Least Recently Used) Algorithm

- **最近不常使用 (過去)** 的 Page 視為 Victim Page
- 範例：OS 採用 Pure Demand Paging 和 LRU Algo，求 Page Fault Ratio



有12次的Page Faults，Page Fault Ratio = 12/20 = 60%。

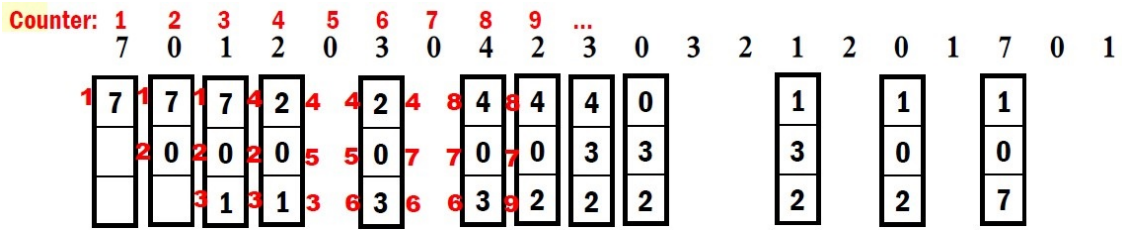
- 特質
  - 效果不錯 Page Fault Ration 尚可接受
  - 不會有 Belady 異常現象
  - 製作成本高，需要大量硬體支援 (需要 Counter 或 Stack)

LRU 製作方式

使用 Counter(計數器) 作為 Logical Timer

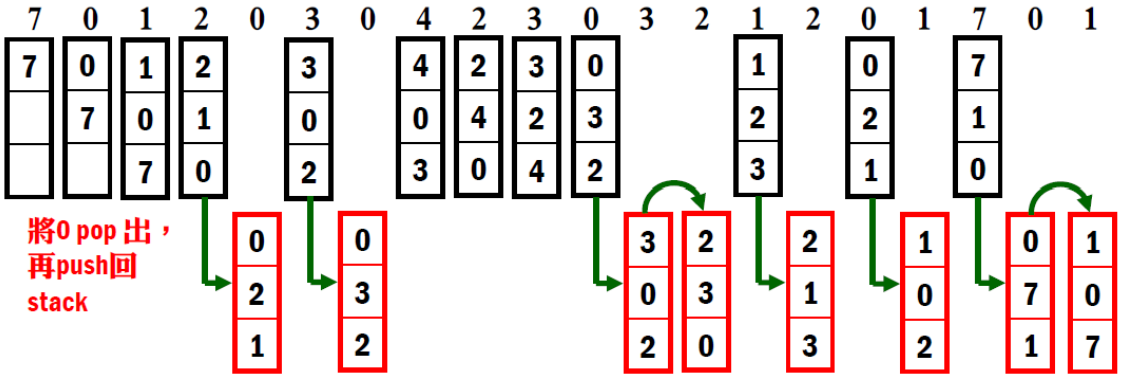
- 確切記錄**每個 Page 最後一次的參考時間**
- 參考時間**最小者**即是 LRU Page
- 作法
  - Counter **初始值 0**
  - 每個 Page 都有一個**附屬的 Register** 儲存 Counter 的值
  - 當有 Memory Access 動作發生，被存取頁面的附屬 Register = Counter + 1

- LRU Page 即是 Register 值最小的 Page



利用 Stack(比較像 Queue)

- 頁框數目極為此 Stack 的大小
- 作法
  - Stack 頂端放的是最後參考的 Page
  - Stack 底端是 LRU Page
  - 當某個 Page 被參考到，將 Page 從 Stack 中取出，並 Push 回 Stack 成為頂端的 Page



LRU 近似法則

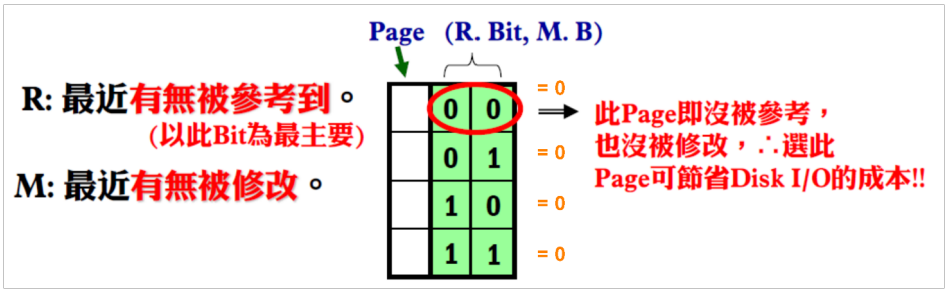
- 動機：LRU 製作成本過高
- 作法
  - Second Chance 二次機會法則
  - Enhance Second Chance 加強二次機會法則
- 都有可能退化成 FIFO，會有 Belady 異常情況

Second Chance

- 以 FIFO 為基礎，搭配 Reference Bit
- 過程
  1. 先以 FIFO 挑出 Page
  2. 檢查 Page 的 Reference Bit
     - = 1：表示最近有被參考過，放棄作為 Victim Page，並將 Reference Bit 設為 0，go to 1
     - = 0：表示最近沒有被參考過，選擇作為 Victim Page
- 所有 Page 的 Reference Bit 都為 0 或都為 1 時，退化為 FIFO --> 可能有 Belady Anomaly

Enhance Second Chance

- 使用(Reference Bit, Modification Bit)
  - 視為二進位無號數，取最小值作為 Victim Page
  - 值相同以 FIFO 為準



- 有可能退化為 FIFO --> 可能有 Belady Anomaly

Reference Frequency Base Algo.

- 以頁面參考次數為挑選 Victim Page 的依據
- 作法
  - **Most Frequency Used(MFU)** 參考次數最多的 Page 作為 Victim Page
  - **Least Frequency Used(LFU)** 參考次數最少的 Page 作為 Victim Page
- Page 參考次數相同則以 FIFO 為準
- 特性
  - **Page Fault Ratio 太高**，不常使用
  - **會有 Belady Anomaly**
  - **製作成本高**，需要硬體支援

練習範例

Page #	Load Time	Last Reference Time	Reference Bit	Modification Bit
1	164	480	1	1
2	300	500	0	1
3	128	700	1	1
4	400	600	0	0

- 求 Victim Page?
  - FIFO, LRU, Second Chance, Enhanced Second Chance
- **Solution**
  - FIFO = P3(Min. **Load Time**)
  - LRU = P1(Min. **Ref. Time**)
  - Second Chance = P2(Min. **Ref. Bit**; Search Order: FIFO)
  - Enhanced Second Chance = P4(Min. **Ref. Bit + Mod. Bit**)

頁框數的分配多寡對 Page Fault Ratio 的影響

- 一般來說 Process **分配到的頁框數越多，Page Fault Ratio 越低**
- Process 分配到的頁框數目有**最小數目**和**最大數目**的限制，取決於**硬體因素**
  - **最大數目：** 由 Physical Memory Size **實際記憶體大小**決定
  - **最小數目：** 由**機器指令結構**決定
    - 必須讓任何一個機器指令順利執行完成，即執行過程中 Memory Access 可能得最多次數

最小分配頁框數

- 機器指令執行過程中 Memory Access 可能得最多次數
- 機器指令執行過程中**若 Page Fault 中斷發生，此指令必須從頭開始執行**
- 機器指令執行週期的 5 個階段
  - IF: Instruction Fetch
  - DE: Decode
  - FO: Fetch Operand
  - EX: Execution
  - WM: Write Results to Memory
  - 案例:

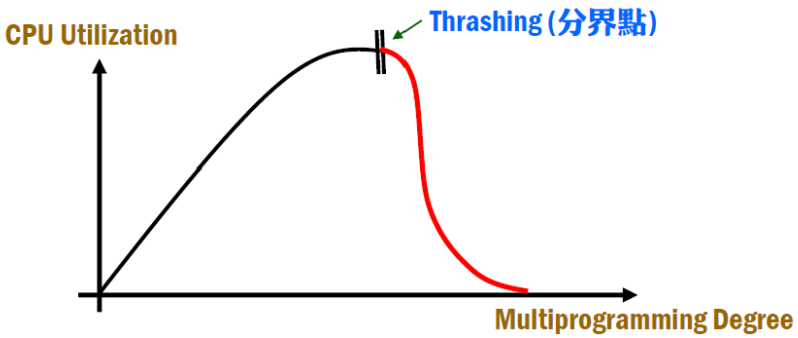
	1+2	**jump**
IF:	指令提取"+"	指令提取
DE:	對"+"解碼 --> 加法	解碼
FO:	提取運算元"1", "2"	x(無運算元)
EX:	執行	執行
WM:	結果寫回記憶體	x(不需要將結果寫回記憶體)

- **Memory Access**
  - 一定會: IF
  - 不一定會: FO, WM
- 舉例
  - 假設 IF, FO, WM 三個階段最多各做 1 次 Memory Access
    - 最小頁框數= 3，**若小於 3 機器指令永遠無法完成執行**
    - 假設頁框數為 2，**第 3 次 Memory Access 時發生 Page Fault，必須從頭執行 --> 永遠無法執行完成**
  - 假設 IF, FO, WM 三個階段最多各做 2 次 Memory Access
    - 最小頁框數= 6

頁框數分配不足所引發的問題

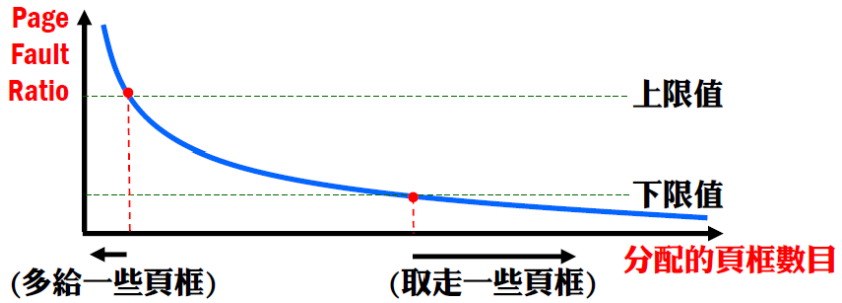
Thrashing 震盪

- **Multiprogramming & Demand Paging** 環境中，如果 Process 分配到的頁框數不足，會經常發生 Page Fault，必須執行 Page Replacement
- 如果採用 **Globle Replacement Policy**，允許 Process 去**搶奪其他 Process 的頁面以空出頁框**，造成其他 Process 也發生 Page Fault，**最後所有 Process 都在處理 Page Fault**
  - **Globle Replacement Policy** 可以選擇別人的頁面作為 Victim 替換出去
  - **Local Replacement Policy** 只能選擇自己的頁面作為 Victim
- 此時所有的 Process 都忙於 Swap In/Swap Out，造成 **CPU 利用率下降**，OS 會**企圖拉高 Multiprogramming Degree 讓更多 Process 進入系統，馬上又會因為頁框數不足發生 Page Fault**，導致惡性循環
- **CPU 利用率急速下降，Throughput 下降，所有 Process 處理 Page Fault 的時間遠大於正常時間，稱之為 Thrashing**
  - OS 沒有辦法判斷 CPU Idle 是什麼原因造成



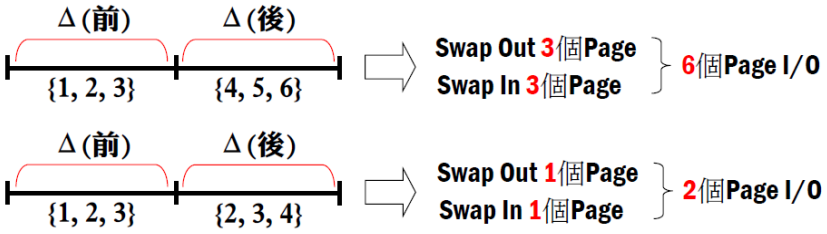
Thrashing 的解決方法

- 方法一：降低 Multiprograming Degree
- 方法二：利用 Page Fault Ratio 控制來防止 Thrashing
  - 作法：OS 規定 Page Fault Ratio 的**上限值**和**下限值**
  - Case 1：Page Fault Ratio > 上限值，OS 分配額外的頁框給該 Process
  - Case 2：Page Fault Ratio < 下限值，OS 抽走該 Process 多餘的頁框，分配給其他 Process



  - 當所有的 Process 的 Page Fault Ratio 都大於上限值，則使用方法一

- 方法三：利用**Working Set Model**預估各 Process 在不同執行時期所需的頁框數，並依此提供足夠的頁框，以防止 Thrashing
  - Process 執行時，記憶體的存取區域並非均勻，而是有**區域性(Locality)**
  - **區域性**模式分類
    - **Temporal Locality 時間區域性**
      - 目前存取的記憶體區域，**過不久後**會再度被存取
      - ex. Loop, Subroutine, Counter, Stack
    - **Spatial Locality 空間區域性**
      - 目前存取的記憶體區域，鄰近的區域即有可能也會被存取
      - ex. Array, Sequential Code Execution, Global Data Area
  - 作法：OS 設定一個**Working Wet Window**(工作即視窗 Δ)大小，以 Δ 次記憶體存取中，所存取到不同 Page 之集合，此一即和稱為**Working Set**，Working Set 中的 Process 個數稱為**Working Set Size(工作集大小 WSS)**
    - 不同時期 Δ 可能不一樣
    - **Ex: Pages** $\underbrace{1, 1, 2, 1, 3, 1, 1, 1, 2}_{\Delta=9}, \underbrace{1, 1, 1, 7, 6, 4, 1, 1}_{\Delta=8}$
      Working Set = {1, 2, 3} Working Set = {1, 4, 6, 7}
      Working Set Size= 3 Working Set Size= 4
    - 假設有 n 個 Process
      - **WSS_i** 為 $Process_i$ 在某個時期的 Working Set Size
      - **D** 為某時期所有 Process 之頁框總需求量 $D=\sum_{i=1}^{n} WSS_i$
      - **M** 為 Physical Memory 大小(可用頁框總數)
    - Case 1: D <= M 則 OS 會依據 WSSi 分配足夠頁框給 Processi
    - Case 2: D > M 則 OS 會選擇暫停 Processi 執行，值到 D <= M，等到未來記憶體足夠時再恢復原先的 Suspend Process
  - 優點：
    - 可防止 Thrashing 產生
    - 對於 Prepaging 有幫助
  - 缺點
    - 用上一次的 Working Set 預估下一次的 Working Set，**不易制定精準的 Working Set**
    - 如果前後的 Working Set 內容差異太大，I/O Transfer Time 會拉長

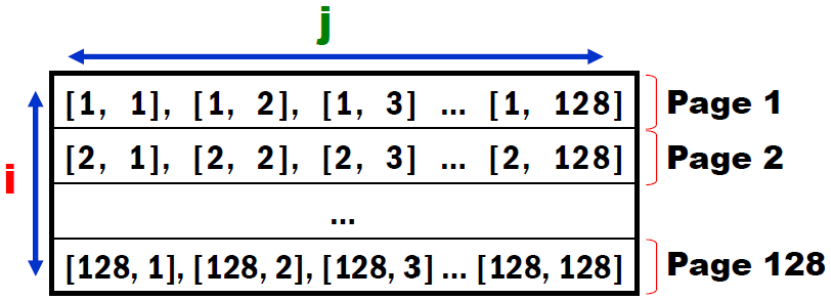


Page Size 大小對 Page Fault Ratio 的影響

- **Page Size 越小 (優點) (缺點)**
  - Page Fault Ratio 越**高**
  - Page Table Size 越**大**
  - I/O Time 越**大** (執行整個 Process 的 I/O Time)
  - 內部碎裂越**小**
  - Total I/O Time 越**小** (單一 Page 的 Transfer Time)
  - Locality 越**集中**
- 趨勢：**傾向 Larger 頁面**

Page Structure 對 Page Fault Ratio 的影響

- 判斷**演算法**和**資料結構**：是否符合 **Locality Model**
  - 符合 --> Page Fault Ratio 下降，對 Virtual Memory 有利
  - **Good:** Loop, Subroutine, Counter, Stack, Array, Sequential Code Execution, Global Data Area, Sequential Search
  - **Bad:** Link List, Hashing Binary Search, goto, jump
- Array 的處理程式，最好與 Array 在 Memory 中的儲存方式一致 (Row-Major, Column-Major) --> 必須符合資料結構的使用特性
- **範例**：求下列兩個程式最多的 Page Fault 次數
  - Array A[1...128, 1...128] of char
  - Array 是以 Row-Major 方式儲存在記憶體中
  - 每個 char 佔 1 bytes
  - Page Size = 128 bytes
  - ```
    for i = 1 to 128 do
        for j = 1 to 128 do
            A[i, j] = 0;
    ```
  - ```
    for j = 1 to 128 do
        for i = 1 to 128 do
            A[i, j] = 0;
    ```



- Solution
  - Row-Major 處理 Array
    每設定完一列發生 1 次 Page Fault，共 128 列
    128 次 Page Fault
  - Column-Major 處理 Array
    每設定完一欄發生 128 次 Page Fault，共 128 欄
    128 * 128 次 Page Fault