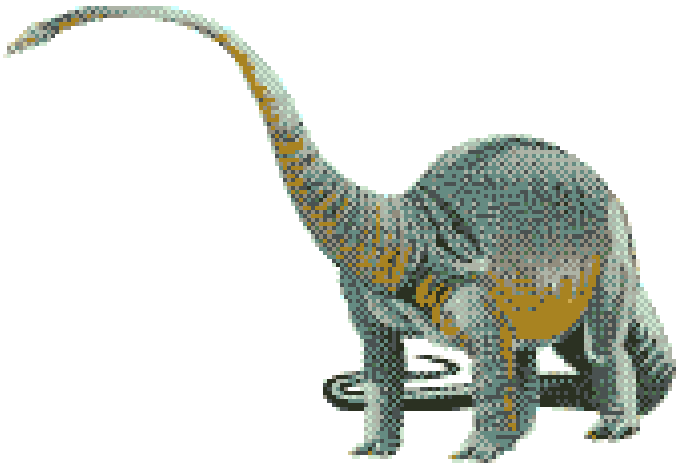
作業系統(Operating Systems)

Course 9: 虛擬記憶體 (Virtual Memory)



授課教師：陳士杰

國立聯合大學 資訊管理學系

■本章重點

- 目的、好處
- Demand Paging技術
- Page Fault處理
- Effective Memory Access Time
- 影響Page Fault Ratio因子
- Page Replacement Algo.
- Frame Allocation Limitation
- Thrashing及其解決方式
- Page Size之影響
- Program Structure之影響

Virtual Memory (虛擬記憶體)

- 目的：允許Program Size大於Physical Memory Size情況下，Program仍然能執行。
- 達成策略：採用Partial Loading的概念 --
  - Dynamic Loading
    - 在執行期間，當副程式真正被Call到，才將之載入MM中，不會浪費MM空間。
    - Programmar的負擔。∵OS只提供Loader，只有Programmar知道哪些副程式是互斥的。
  - Virtual Memory
    - OS的負擔，Programmar無負擔。

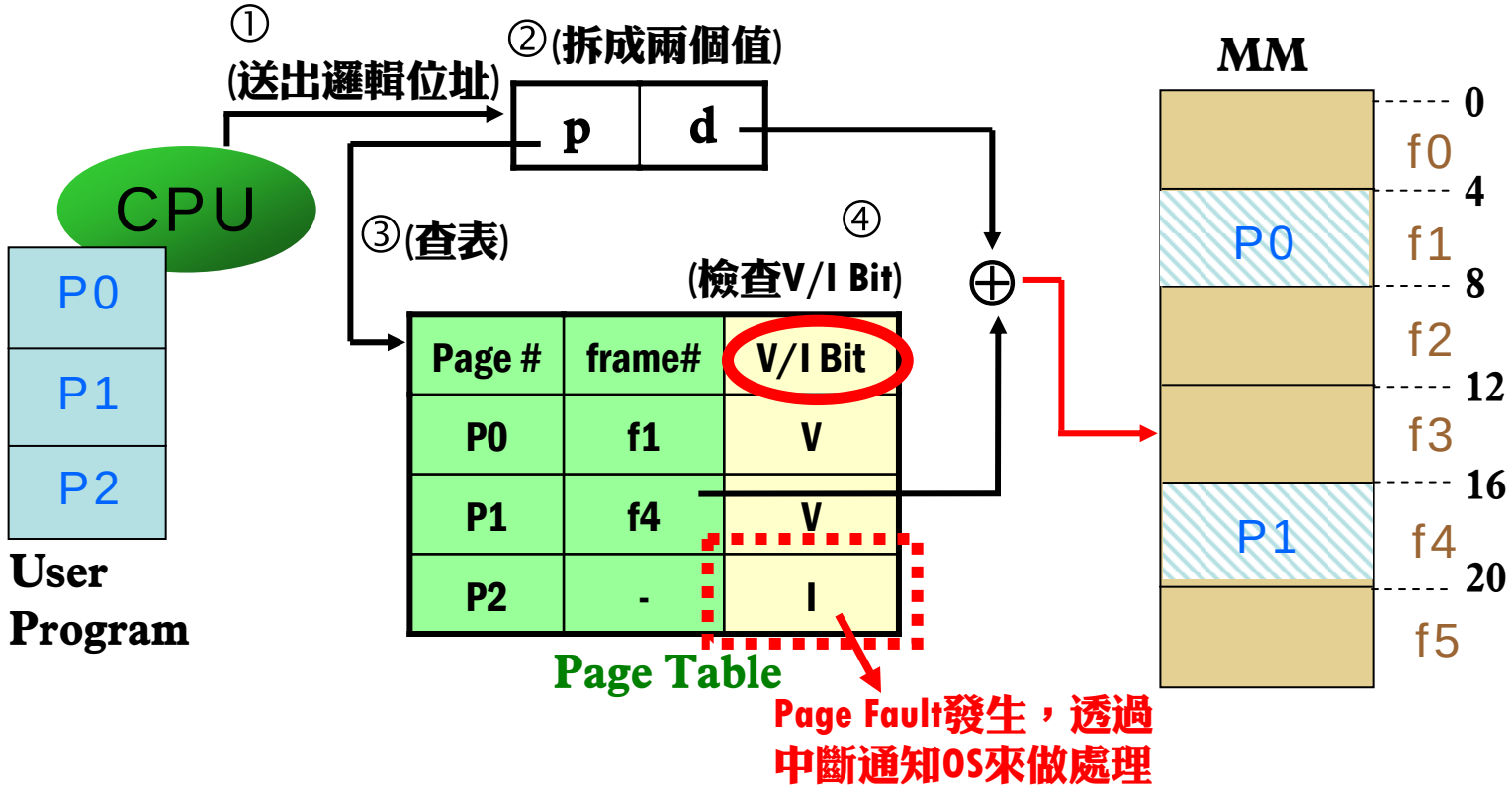
- 好處：
  - 允許Program Size大於Physical Memory Size而Program仍能執行。
  - Programmar專心負責寫好程式，無須考量Program Size太大的問題。
  - 記憶體的各個小空間皆有機會被利用到，Memory Utilization ↑。
  - 儘可能提高Multiprogramming Degree，提昇CPU Utilization。
    - ∵程式不用將所有的Page搬入MM才能執行。∴原本系統只能執行5個程式，現可能執行20個。
  - 每一次的I/O Transfer Time↓(∵不用將整個程式的所有Page載入)
    - 然而，載入整個程式很耗費I/O Transfer Time!! (∵總傳輸次數變多!!)

■實現Virtual Memory之技術

- 使用Demand Paging (需求分頁) 技術
  - 為實現Virtual Memory的技術之一，以Paging Memory Management為基礎所發展出來。不同的是，其採用 "Lazy Swapper" 技巧。
    - 即：程式執行之初，並不將全部的Pages全數載入MM中，而是僅載入執行所須的Pages到MM中，其餘的Pages全數置於Blocking Store中。若執行所需的Pages都在MM，則一切正常執行；反之，則要處理Page Fault問題，由OS另行處理

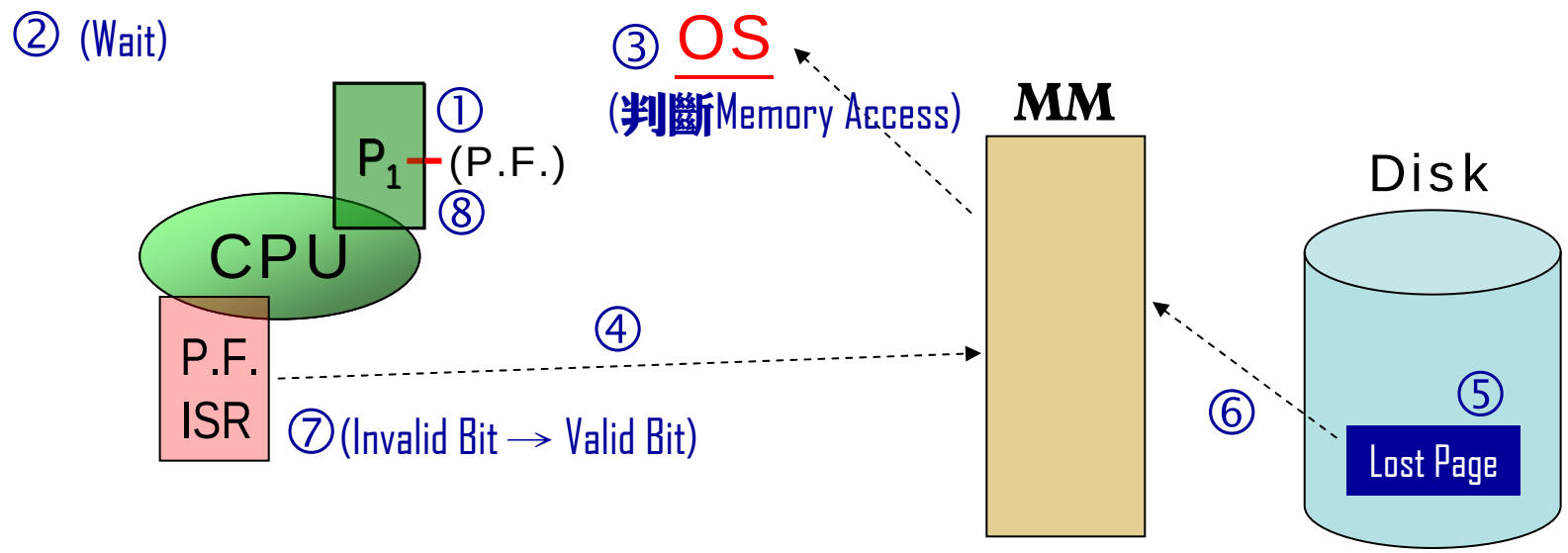
分頁表 (Page Table) 之配合修正

- 多加一個 "Valid/Invalid Bit" 欄位，用以指示Page是否在Memory中。
  - Valid：表示Page在Memory中
  - Invalid：表示Page不在Memory中



Page Fault定義及其處理程序

- 在Demand Paging之Virtual Memory系統中，若執行中的Process試圖存取不在記憶體中的Page時，則發生Page Fault。



- 處理程序：
  - ① OS收到由Page Fault所引起的Interrupt (中斷)
    - 由Memory Management Unit發出。
  - ② OS暫停目前Process的執行，並保存此Process的狀態
  - ③ OS會去判斷此Memory Access位址是否合法。若非法則終止此Process，否則表示是由Page Fault所引起!!
  - ④ OS去Memory檢查有無Free Frame。若沒有，則必須執行 “Page Replacement” 以空出一個可用頁框
  - ⑤ OS去Disk中找到Lost Page之所在位置
  - ⑥ 將此Lost Page載入到可用頁框
  - ⑦ 修改Page Table，指明此Page所在頁框，並將Invalid Bit改成Valid Bit
  - ⑧ 恢復原先Process中斷前之執行

■Virtual Memory之效益評估

- 由Effective Memory Access Time決定，愈短則效益愈好，愈接近MM存取時間。
- Effective Memory Access Time公式：

  (1-p)×ma + p×(Page Fault Processing Time)

  其中
  - p：表示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之時間總和
- 結論：要有效降低Effective Memory Access Time，則必須降低Page Fault Ratio
  - ∵ma與Page Fault Processing Time是固定的!!

- 例：假設
  - ma = 20ns
  - P.F. Processing Time = 2ms
  - p = 20%

求Effective Memory Access Time = ? μs

Ans:

$(1-0.2)\times 200ns+0.2\times 2ms$

$=0.8\times 200ns+0.2\times 2\times 10^{+6}ns$

$=160ns+4\times 10^{+5}ns$

$=400160ns$

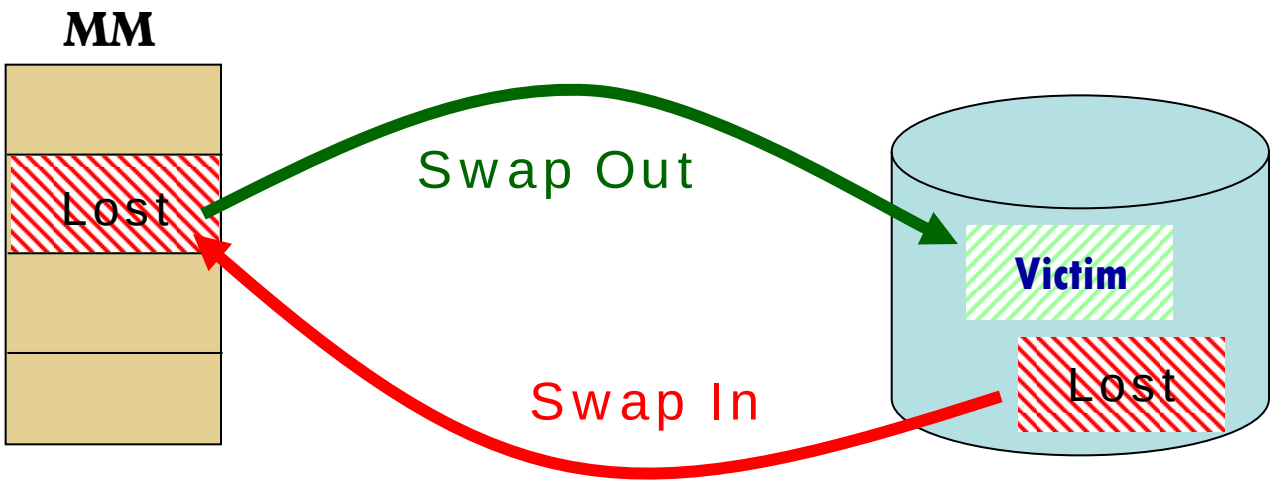
$=400.16\ \mu s$

■影響Page Fault Ratio的因素

- Page Replacement Algo.之選擇
- 頁框 (Frame) 數的分配多寡
- Page Size大小
- Program Structure

■Page Replacement

- 當Page Fault發生且Memory無可用頁框時，則OS必須執行此工作。
  - OS必須選擇一個犧牲者(Victim Page)，將其Swap Out到Blocking Store (e.g. Disk) 以空出一個可用頁框，再將Lost Page載入 (Swap In) 到此頁框。
  - 圖示：



  - Swap Out 和Swap In分別是2個Disk I/O的動作。
    - Swap In是必要的，∵一定要將Lost Page置入到MM中
    - Swap Out呢？不盡然是必要的!!需視Victim Page是否曾被修改

是否可將非必要的Swap Out省掉？

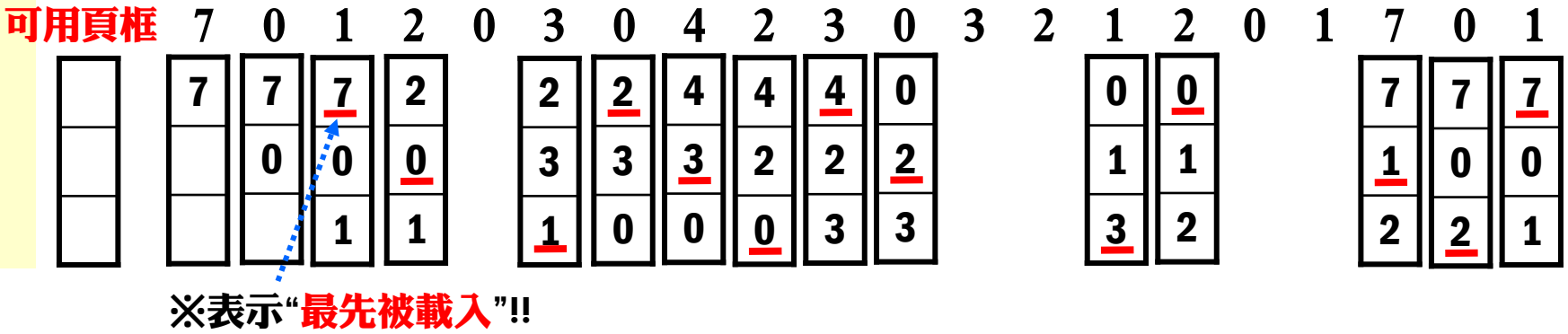
- **判斷依據**：Victim Page從**執行之初**一直到**準備被替換**之前是否曾被寫入或修改?
  - (Yes: 須回存至H.D.；No: 可刪除或覆蓋)
- 利用 "**Modification Bit**" (或Dirty Bit) 來實行上述判斷依據，以節省Victim Page之Swap Out I/O 動作。
  - Modification Bit = 0，表示此頁面的內容未曾被修改過。
  - Modification Bit = 1，表示此頁面的內容曾被修改過。
- 此Bit初始值設成0。若Victim Page的Modification Bit值為0，則**不用Swap Out**，否則必須Swap Out到Blocking Store。故可節省不必要的 I/O 動作。

Page Replacement Algorithms

- FIFO
- OPT
- LRU
- LRU近似法則
- Frequency

FIFO Algorithm

- **最先載入的Page** (即：Loading Time最小者)，優先視為Victim Page。
  - 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" 及 "FIFO Replacement Algo."，求Page Fault次數。



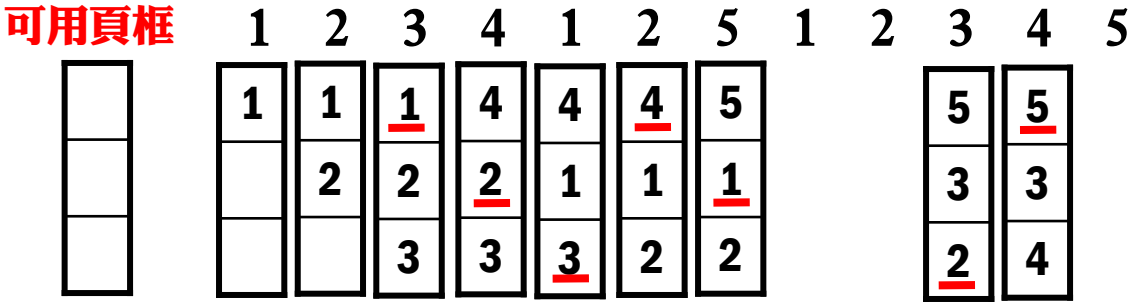
有15次的Page Faults，Page Fault Ratio = 15/20 = 75%。

- **Pure Demand Paging**
  - 程式執行之初，不預先載入任何Page。
  - 缺點：執行之初，會產生大量的Page Fault
  - 優點：載入的Page皆是Process所需的頁面，故後續的Page Fault Ratio會下降至合理值 (趨於穩定)。
- **Prepaging(預先分頁)**
  - 事先猜測程式執行之初會使用哪些Page，並預先將這些Pages載入。
  - 優點：若猜得準確，則可以避免程式執行之初大量Page Fault發生。
  - 缺點：若猜測錯誤，則先前載入Page的I/O動作白白浪費。

- 特質：
  - 簡單，易於實作
  - 效益差 (∵Page fault ratio最高)
  - 可能有Belady異常現象
    - 當Process分配到較多的Frame數目，有時其Page Fault Ratio卻不降反升。

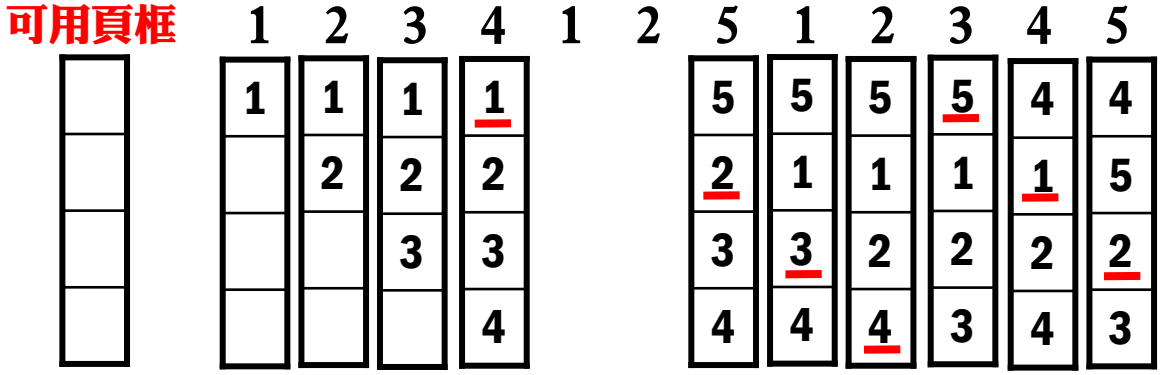
Belady Anomaly範例 (in FIFO Algo.)

- 例：給予下列的Page Reference String 及可用頁框有3個：



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

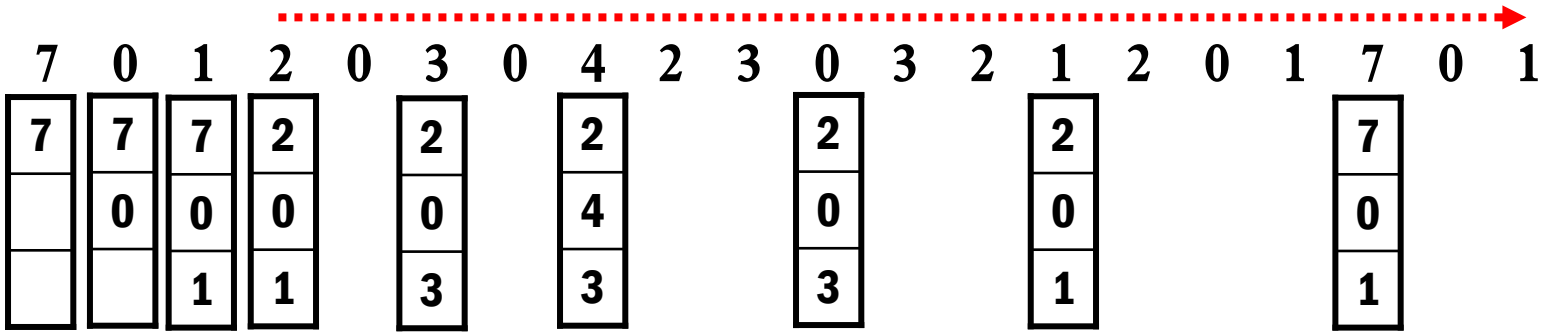
- 例：給予下列的Page Reference String 及可用頁框有4個：



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

OPT (Optimal) Algorithm

- 以 “將來長期不會使用的Page” 視為Victim Page。
  - 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” 及 “OPT Algo.”，求Page Fault次數。

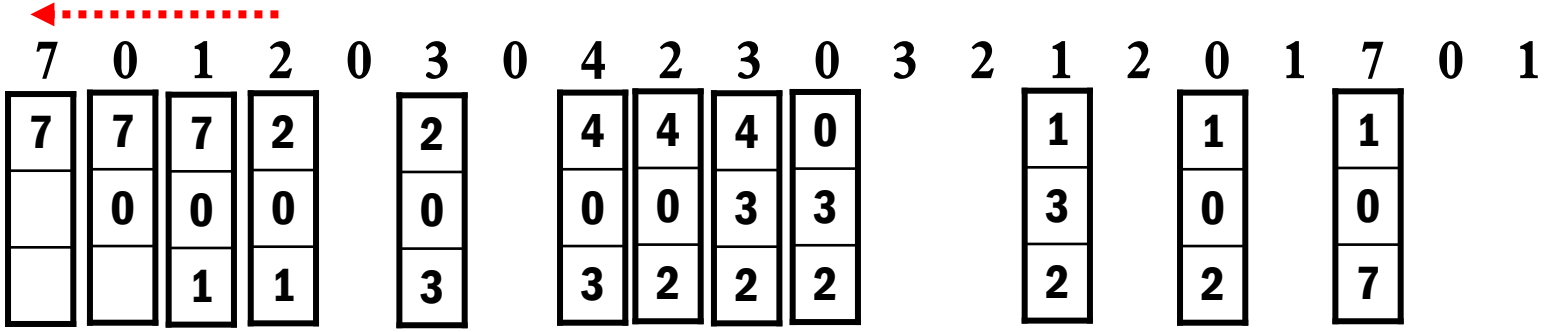


有9次的Page Faults，Page Fault Ratio = 9/20 = 45%。

- 特質：
  - 效果最佳 (∵Page fault ratio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

- 以 "最近不常使用的Page" 視為Victim Page。
  - 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" 及 "LRU Algo."，求Page Fault次數。

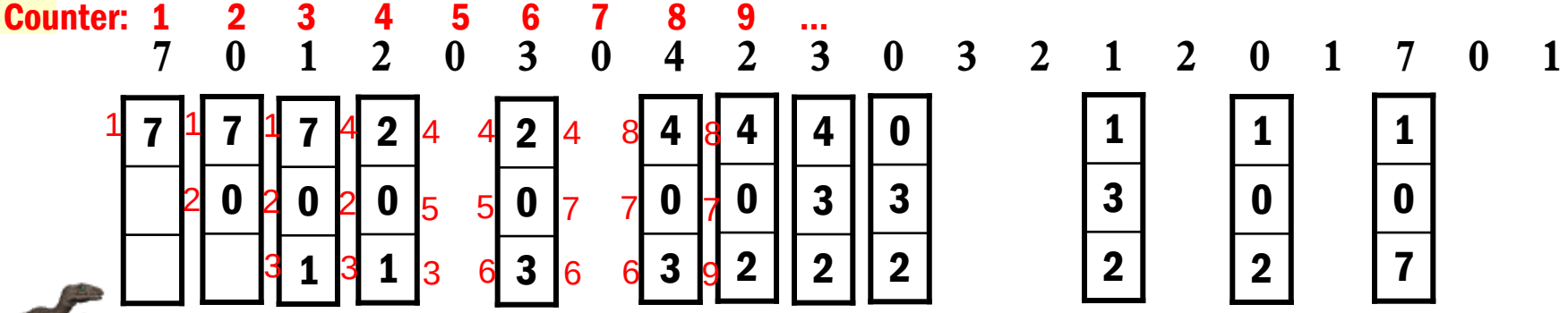


有12次的Page Faults，Page Fault Ratio = 12/20 = 60%。

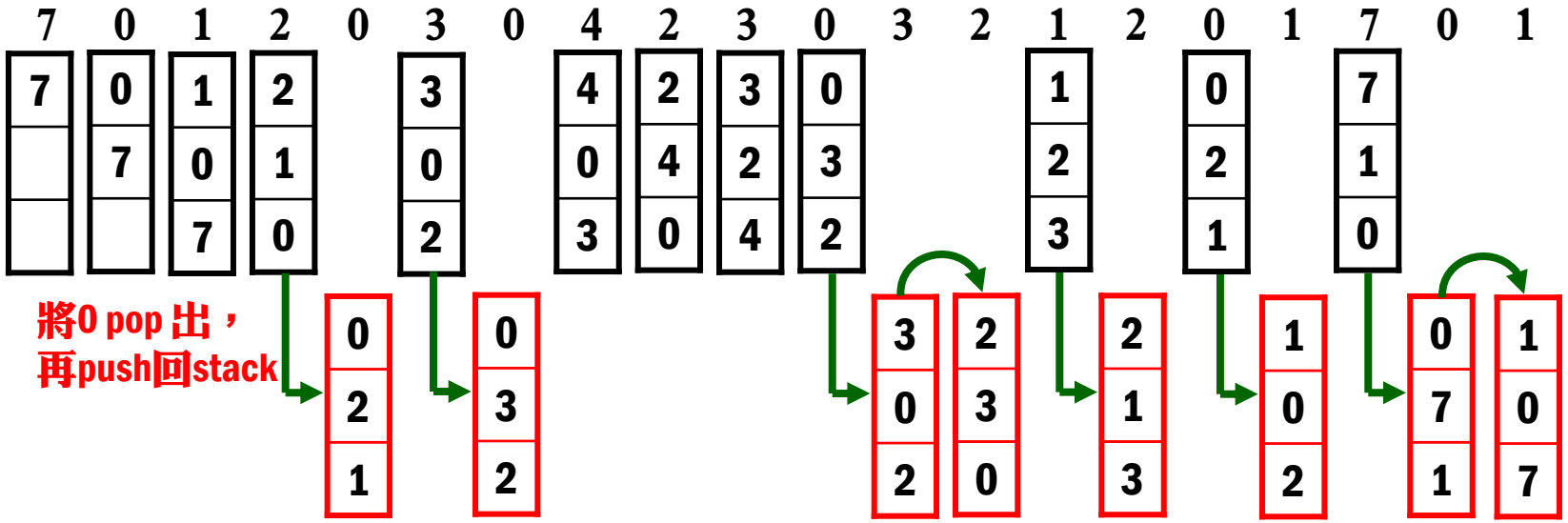
- 特質：
  - 效果不錯 (Page fault ratio尚可接受)
  - 不會有Belady異常現象
  - 製作成本高，需要大量硬體支援 (如：需要Counter或Stack)

LRU製作方式

- 使用Counter (計數器)，做為Logical Timer
  - 切確記錄每個Page最後一次的參考時間
  - 參考時間最小者即是LRU Page
  - 作法：
    - Counter初值為0
    - 每個Page皆有一個附屬的Register，用以存Counter的值。
    - 當有Memory Access動作發生，則Counter值加1，並且將Counter值設定給被存取頁面的附屬Register。
    - LRU Page即是Register值最小的Page。
  - 例：



- 利用Stack (堆疊)
  - 頁框數目即為此Stack的大小!!
  - 作法：
    - Stack頂端放的是最後 (最近) 參考的Page
    - Stack的底端即為LRU Page。
    - 當某Page被參考到，則會將此Page從Stack中取出，並Push回Stack成為頂端的Page。
  - 例：

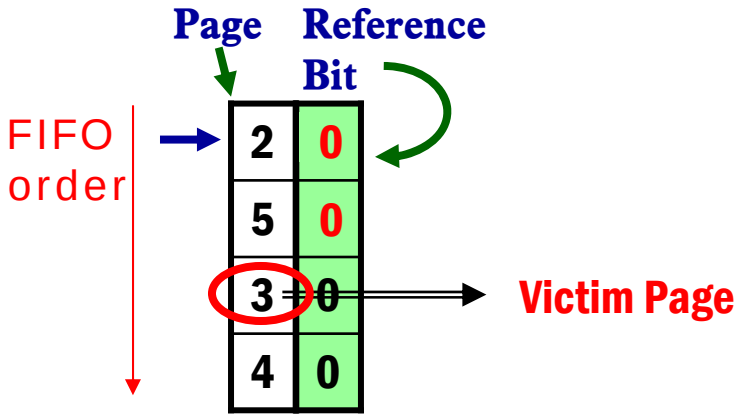


LRU 近似法則

- 緣由：∵LRU製作成本過高
- 作法：
  - Second Chance (二次機會法則)
  - Enhance Second Chance (加強式二次機會法則)
- 都有可能退化成FIFO，∴會遇到Belady異常情況

Second Chance (二次機會法則)

- 以FIFO法則為基礎，搭配Reference Bit使用。
- 程序：
  - ① 先以FIFO挑出Page
  - ② 檢查此Page的Reference Bit：
    - 若為 1，表示最近有被參考過，則放棄此Page作為Victim Page，並將Reference Bit值重設為0，goto ①。
    - 若為 0，表示最近沒有被參考過，則選擇此Page作為Victim Page。
- 例：



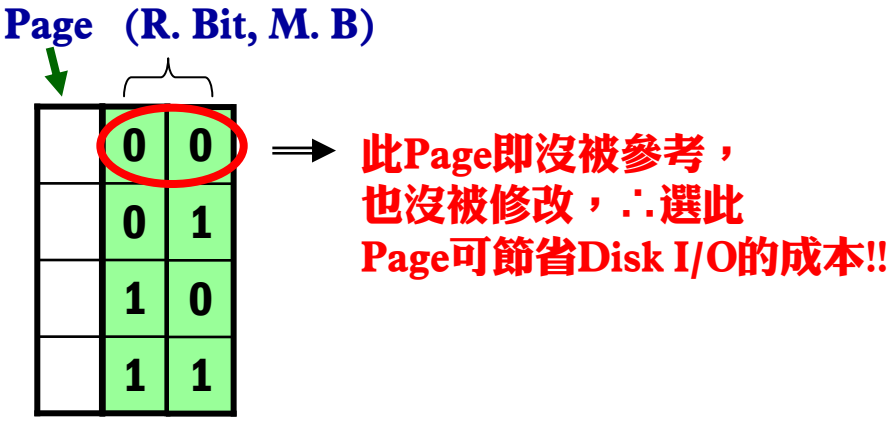
- 所有Page的Reference Bit皆為 0 或 1 ，則退化成FIFO Algo。∴可能有Belady Anomaly。

Enhance Second Chance (加強式二次機會法則)

- 使用 (Reference Bit, Modification Bit) 配對值作為挑選 Victim Page的依據。
  - 以二進位無號數來使用，取最小值做為Victim Page
  - 若有 ≥ 2個Page具有相同最小值，則以FIFO為準
- 例：

R: 最近有無被參考到。
(以此Bit為最主要)

M: 最近有無被修改。



- 有可能退化成FIFO Algo，∴可能有Belady Anomaly。

Reference Frequency Base Algo.

- 以頁面之參考次數作為挑選Victim Page的依據
- 作法：
  - MFU (Most Frequently Used): 參考次數最多的Page作為Victim Page
  - LFU (Least Frequently Used): 參考次數最少的Page作為Victim Page
- 若有多個Pages具有相同值，則以FIFO為準。
- 特性：
  - Page Fault Ratio太高 (∴不常使用)
  - 會有Belady Anomaly
  - 製作Cost也高 (∵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

則Victim Page各為何？

- 針對FIFO, LRU, Second Chance, Enhanced Second Chance。

Ans:

- FIFO = P3 (只看Load Time)
- LRU = P1 (只看Reference Time)
- Second Chance = P2 (Load Time + Reference Time)
- Enhanced Second Chance = P4 (Reference Bit + Modification Bit)

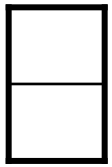
■頁框數 (Frames) 的分配多寡對Page fault ratio之影響

- 一般來說，Process所分配到的頁框數愈多，則Page Fault Ratio愈低。
- Process所分配的頁框數目有最少數目與最大數目的限制，此兩類數目限制均取決於硬體因素。
  - 最大數目的限制：由實際記憶體大小 (Physical Memory Size) 決定
  - 最少數目的限制：由“機器指令結構”決定
    - 必須能讓任何一個機器指令順利執行完成
    - 即：機器指令執行過程中，Memory Access可能之最多次數

- 最少分配頁框數，必須能讓任何一條機器指令順利完成。
  - 即：機器指令執行過程中，Memory Access可能之最多次數
  - 機器指令執行過程中，若Page Fault Interrupt發生，此指令必須從頭開始執行。
- 機器指令的執行週期有5個Stages:
  - IF: Instruction Fetch
  - DE: Decode
  - FO: Fetch Operand
  - EX: Execution
  - WM: Write Result to Memory

- 哪些Stage會做Memory Access:
  - 一定會：IF Stage
  - 不一定會：FO Stage, WM Stage

- 假設IF、FO、WM 三個Stages頂多各做一次Memory Access：
  - ∴最少的頁框分配數目為 3
  - 若小於 3，則機器指令可能永遠無法執行完成!!
  - 例： 可用頁框



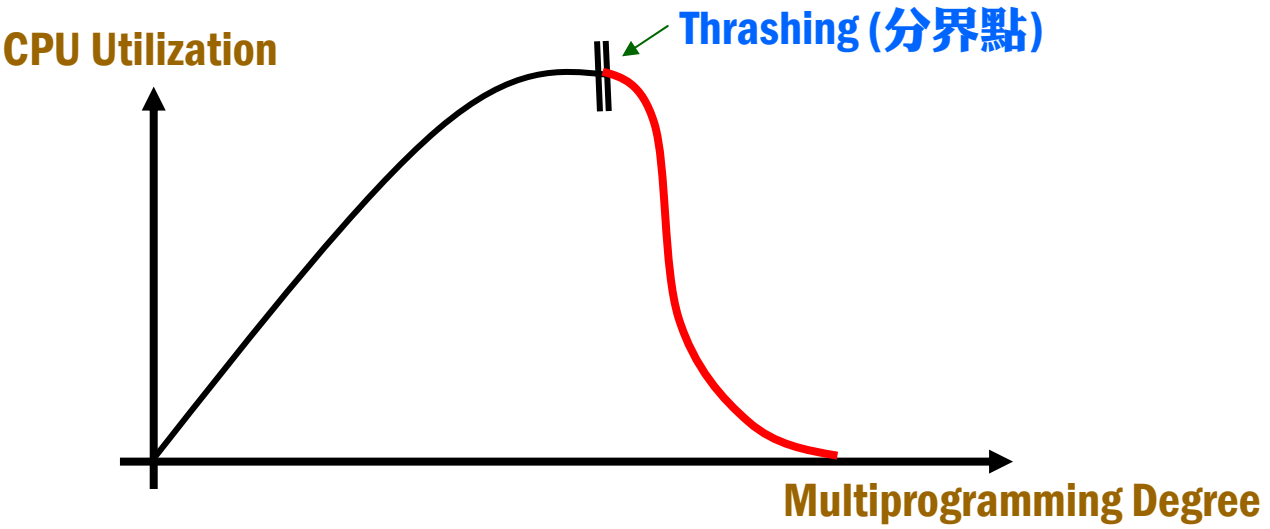
    - 指令中需要做Memory Access的Pages一直相互踢來踢去，該指令永遠做不完!!
- 假設IF、FO、WM 三個Stages頂多各做 2 次Memory Access
  - ∴最少的頁框分配數目為 6

頁框數分配不足所引發的問題

- **Thrashing (振盪)**
  - 在Multiprogramming且為Demand Paging的環境中，若Process分配到的頁框數不足，則其會經常發生Page Fault，此時必須執行Page Replacement。
  - 若採用 "Global Replacement Policy"，則此Process會替換 (搶奪) 其它Process的Page以空出頁框，造成其它Process也會發生Page Fault，而這些Process也會去搶其它Process之頁框，如此一再循環發生，造成所有Process皆在處理Page Faults。
    - Global Replacement Policy：選Victim時可選別人的頁面替換出去。
    - Local Replacement Policy：選Victim時只可選自己的頁面。
  - 此時，所有Process皆忙於Page之Swap In/Out，造成CPU經常idle，CPU Utilization下降。此時OS (Multiprogramming Sys.)會企圖引進更多的Processes進入系統。

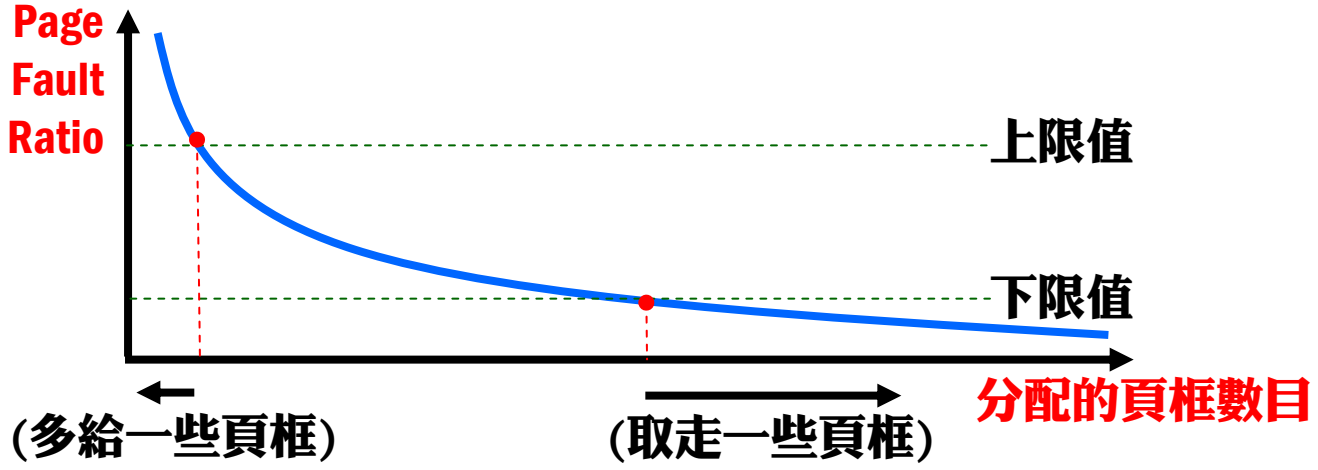
- 這些新進入的Processes馬上又會因為頁框數不足，而又發生Page Faults，又與其它Process搶奪頁框，所以CPU又再次idle。但是OS又企圖拉高Multiprogramming Degree。
- 如此現象一再發生，造成CPU Utilization急速下降，Throughput ↓，所有Process花在處理Page Fault的時間遠大於正常執行時間，稱之為Thrashing。

- 圖示：



Thrashing的解決方法

- 【方法1】降低Multiprogramming Degree
- 【方法2】利用Page Fault Frequency (Ratio)控制來防止Thrashing
  - 作法：OS規定合理的Page Fault Ratio之上限與下限值，把該ratio控制在一個合理範圍內。(∵有振盪發生時， Page Fault Ratio一定是處於高檔)
    - Case 1: 若某Process的Page Fault Ratio > 上限值，則OS應多分配額外的頁框給該Process。
    - Case 2: 若某Process的Page Fault Ratio < 下限值，則OS應從該Process取走多餘的頁框，以分配給其它有需要的Process。
  - 圖示：



- 【方法3】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數，以防止Thrashing。
  - 緣由：Process執行時，對於Memory的存取區域並非是均勻(Uniform)的，而是具有區域性(Locality)的特質。
  - 區域性(Locality)模式分成兩種：
    - Temporal Locality (時間區域性)
      - Process目前所存取的記憶體區域，過不久後會再度被存取。
      - Ex: Loop, Subroutine, Counter, Stack
    - Spatial Locality (空間區域性)
      - Process目前所存取的記憶體區域，其鄰近區域極有可能也會被存取。
      - Ex: Array, Sequential Code Execution, Global Data Area

- 作法：
  - OS設定一個Working Set Window (工作集視窗；Δ) 大小，以Δ次記憶體存取中，所存取到的不同Page之集合，此一集合稱為Working Set。而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Processes，令：
  - WSS_i為$Process_i$在某時期的Working Set Size。
  - D為某時期所有Processes之頁框總需求量，即：$D = \sum_{i=1}^{n} WSS_i$。
  - M為Physical Memory大小 (可用頁框總數)

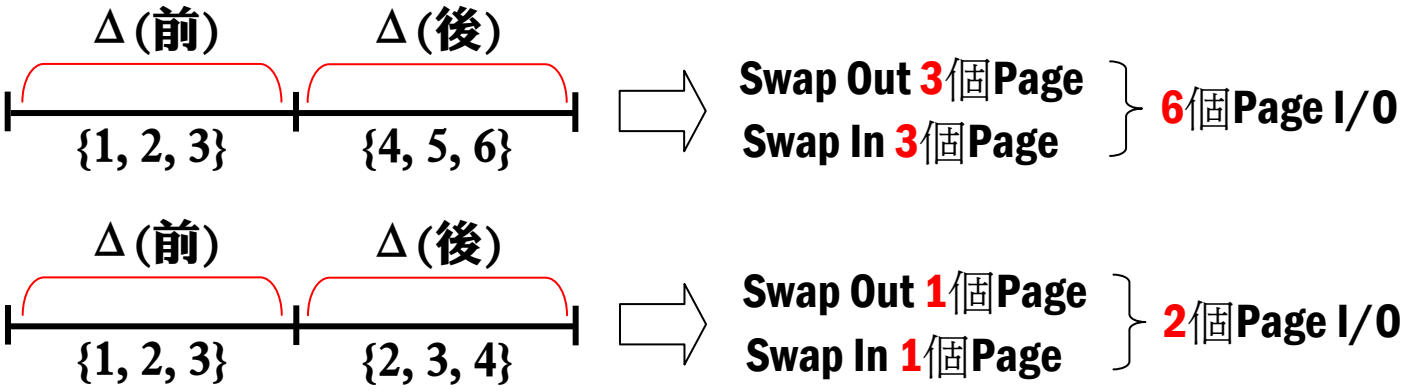
Case 1:

- $D \leq M$，則OS會依據WSS_i，分配足夠的頁框數給$Process_i$，則可防止Thrashing。

Case 2:

- $D > M$，則OS會選擇Process暫停執行，直到 $D \leq M$ 為止，此時即回到Case 1來做處理。等到未來記憶體頁框足夠時再恢復原先的Suspended Process

- 優點：
  - 可以防止Thrashing產生
  - 對於Prepaging亦有幫助
- 缺點：
  - 以上一次的Working Set來預估下一次的Working Set，不易制定精確的Working Set。
  - 若前、後的Working Set內容差異太大，則I/O Transfer Time會拉長。
  - Ex:



■Page Size大小對Page fault ratio之影響

- Page Size愈小，則：
  - Page fault ratio愈高
  - Page Table Size愈大
  - I/O Time (執行整個Process的I/O Time) 愈大

  缺點

  - 內部碎裂愈小
  - Total I/O Time (單一Page的Transfer Time) 愈小
  - Locality愈集中

  優點
- 趨勢：
  - 傾向Larger頁面

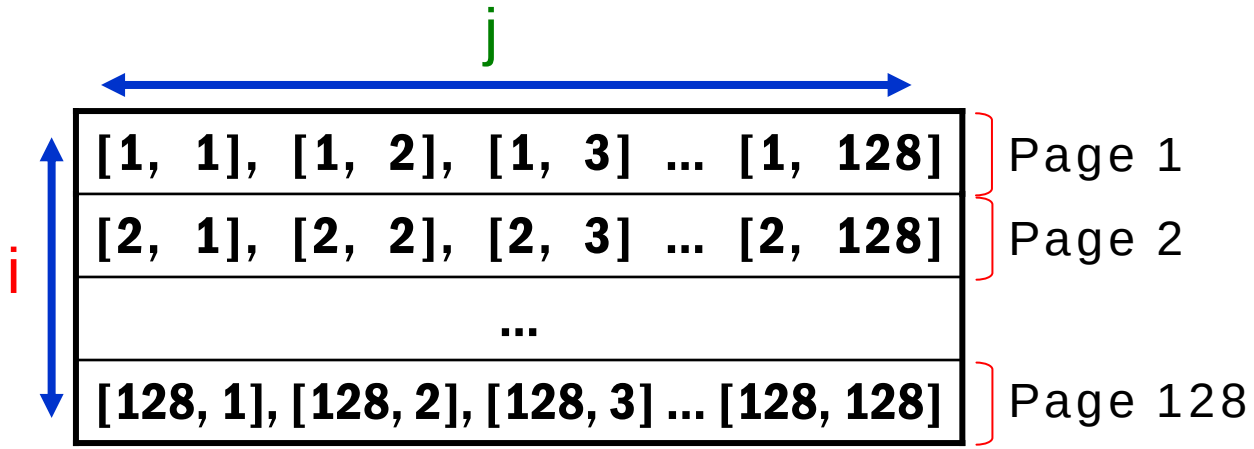
■Page Structure對Page fault ratio之影響

- 所使用的Data Structure與Algorithm
  - 判斷好或不好在於符不符合Locality Model。若符合，則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的相關處理程式，最好與Array在Memory中的儲存方式一致 (即：Row-major, Column-major)

- 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方式存在於Memory中
  - 每個char佔 1 個Byte
  - Page Size: 128 Bytes

(a)
```
For i = 1 to 128 do
    For j = 1 to 128 do
        A[i, j] = 0;
```

(b)
```
For j = 1 to 128 do
    For i = 1 to 128 do
        A[i, j] = 0;
```



Ans：

(a)

- 採Row-major處理Array
- ∵每設定完一列就會發生一次Page fault ，且共有128列
- ∴共128次Page fault

(b)

- 採Column-major處理Array
- ∵每設定完一行就會發生128次Page fault (跨頁面處理)，且共有128行
- ∴共128×128次Page fault