作業系統：Virtual Memory

**目錄**

[作業系統：Virtual Memory 1](#_Toc204864529)

[1. Basic Concepts 1](#_Toc204864530)

[1.1 虛擬記憶體的好處 2](#_Toc204864531)

[1.2 虛擬地址空間（Virtual Address Space, VAS） 2](#_Toc204864532)

[1.3 虛擬記憶體的共享機制 3](#_Toc204864533)

[2. Demand Paging（需求分頁） 4](#_Toc204864534)

[2.1 需求分頁的主要特性 4](#_Toc204864535)

[2.2 分頁錯誤處理流程（Page Fault Handling） 5](#_Toc204864536)

[2.3 指令重啟問題（Instruction Restart Challenge） 6](#_Toc204864537)

[2.4 Free-Frame List（空框架清單） 6](#_Toc204864538)

[3. Page Replacement（頁面替換） 7](#_Toc204864539)

[3.1 頁面替換的處理流程 7](#_Toc204864540)

[3.2 Dirty Bit（修改位元）的作用 8](#_Toc204864541)

[4. 常見的頁面替換演算法 8](#_Toc204864542)

[4.1 FIFO（First-In-First-Out，先進先出） 8](#_Toc204864543)

[4.2 OPT（Optimal Page Replacement，最佳演算法） 9](#_Toc204864544)

[4.3 LRU（Least Recently Used，最久未使用） 9](#_Toc204864545)

[4.4 LRU Approximation 10](#_Toc204864546)

[5. Frame Allocation（框架分配） 10](#_Toc204864547)

[5.1 系統 Frame 資源分配概念 10](#_Toc204864548)

[5.2 Minimum Number of Frames（最小 Frame 數量） 10](#_Toc204864549)

[6. Thrashing（抖動） 11](#_Toc204864550)

[6.1 Thrashing 的成因：系統誤以為 CPU 利用率低，增加多工處理數量 11](#_Toc204864551)

[6.2 Page-Fault Frequency (PFF) 策略 11](#_Toc204864552)

[7. Memory Compression（記憶體壓縮） 12](#_Toc204864553)

[7.1 傳統虛擬記憶體管理中： 12](#_Toc204864554)

[7.2 運作原理： 12](#_Toc204864555)

[8. Prepaging（預先載入） 13](#_Toc204864556)

[9. Page Size（頁面大小） 13](#_Toc204864557)

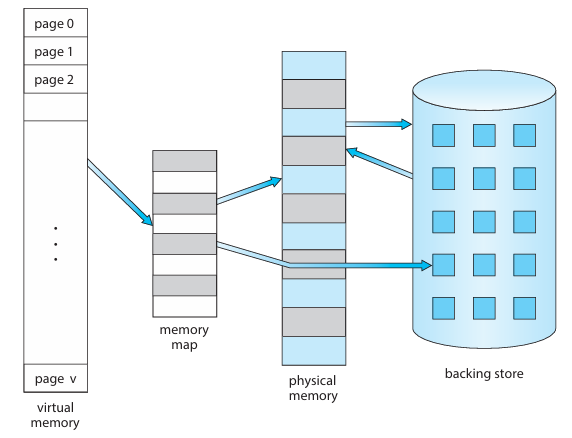
# Basic Concepts

虛擬記憶體（Virtual Memory, VM）是現代作業系統提供的一項重要技術，它讓每個程式擁有「獨立、完整且連續」的記憶體視圖（Virtual Address Space），即使實體記憶體不足，也能執行大程式、多工運行，並提供程式間共享、隔離及保護機制。

## 虛擬記憶體的好處

|  |  |  |
| --- | --- | --- |
| **好處** | **詳細說明** | **生活化比喻** |
| 無需受限於實體記憶體容量 | 可撰寫需求超過實體記憶體容量的大型程式，系統會將部分資料暫時存放於硬碟，稱為「分頁（Paging）」或「交換（Swapping）」。 | 就像你桌面空間有限，但可以把書暫時放進書櫃，需要時再拿出來使用。 |
| 提高 CPU 使用率、同時執行程式數量 | 每個程式實際只載入部分頁面即可執行，系統可同時配置資源給多個程式，達成多工運作。 | 就像餐廳一次只把部分食材放在桌上，其他放在後廚，讓更多桌客人同時用餐。 |
| 減少 I/O 操作、加速程式執行 | 只需載入需要的頁面，避免整個程式一次讀入，並可利用快取機制減少磁碟讀寫次數。 | 就像看書時只翻讀需要的頁數，不必把整本書影印出來才能閱讀。 |
| 提供程式間隔離與安全保護 | 每個程式擁有獨立的虛擬地址空間，避免互相干擾，也能防止惡意程式存取其他程式資料。 | 就像飯店每個房間都有不同房卡，確保客人隱私與安全。 |
| 支援共享記憶體，提高資源利用率 | 系統程式庫、常用函式可被多個程式同時映射使用，避免重複載入。 | 就像圖書館的書籍可被多人共用，而不是每人都要買一份。 |

Figure 1：Diagram showing virtual memory that is larger than physical memory.



程式看到的是完整的虛擬地址空間，作業系統與硬體（MMU）會將這些虛擬頁面對應到實體頁框。

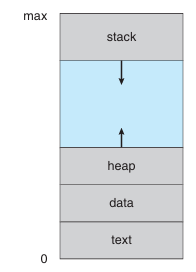
## 虛擬地址空間（Virtual Address Space, VAS）

每個程式「看到」的記憶體是虛擬化後的連續空間，作業系統與硬體（MMU：Memory Management Unit）負責將其映射到實際的物理記憶體。

Table 1：虛擬地址空間區段分布範例

|  |  |
| --- | --- |
| **區段名稱** | **功能** |
| Text（程式碼區） | 儲存可執行指令，通常為唯讀區域，避免程式被隨意改動。 |
| Data（全域變數區） | 儲存初始化過的全域變數、靜態變數。 |
| Heap（堆積區） | 動態配置記憶體空間，例如 malloc() 申請的區域。 |
| Stack（堆疊區） | 儲存函式呼叫、區域變數、回傳位址等，隨函式呼叫增減。 |
| Memory-Mapped區域 | 包含共享函式庫、檔案映射、共享記憶體區段等。 |

Figure 2： Virtual address space of a process in memory.

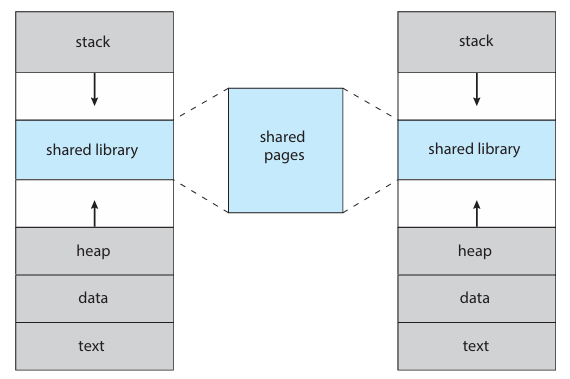


## 虛擬記憶體的共享機制

虛擬記憶體不僅提供隔離，也能讓多個程式共享特定資源，提高記憶體利用率。

* 共享程式庫：例如 libc（C 標準函式庫）只需載入一次，可被多個程式同時使用。減少重複載入相同函式的記憶體消耗。
* 共享記憶體區段：透過 mmap() 系統呼叫或 POSIX 共享記憶體（shm\_open）讓程式之間共享資料。例如，網路瀏覽器的多個分頁可透過共享記憶體存取同一組快取資料。
* fork() 呼叫後的 Copy-on-Write（COW）：子行程最初共享父行程的記憶體頁面，僅在寫入時才複製，節省建立行程的成本。例如fork() 之後 exec() 之前，父子行程大多數頁面都是共享的。

Figure 3：Shared library using virtual memory



簡單運作原理示意：

1. 程式請求一個虛擬位址（Virtual Address）。

2. MMU 將此虛擬位址對應（Mapping）至實體位址（Physical Address）。

3. 若該頁面不在記憶體中，產生 Page Fault，OS 將所需頁面從硬碟載入記憶體。

程式可像擁有完整記憶體一樣運作，而不需要知道背後的分頁機制。

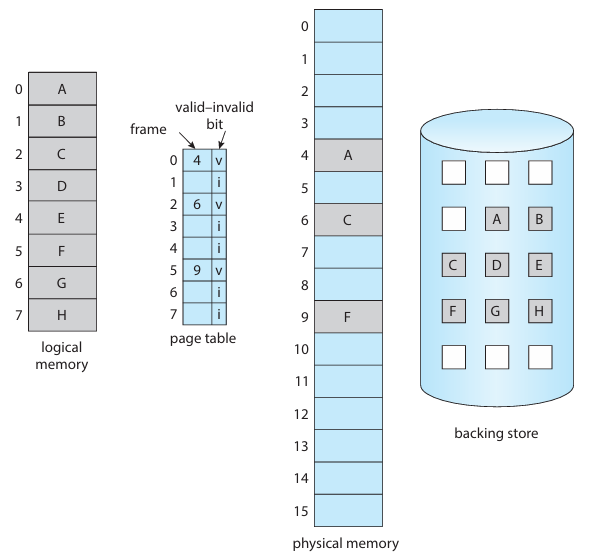
# Demand Paging（需求分頁）

需求分頁是現代虛擬記憶體管理中的重要技術，它允許系統只在「程式實際需要用到某頁面（Page）」時才將該頁載入主記憶體，而不是一次載入整個程式。這大幅降低了記憶體浪費，讓多工環境能同時執行更多程式。

## 需求分頁的主要特性

* 延遲加載（Lazy Load）：程式啟動時只載入必要頁面，其餘頁面僅在首次被存取時才載入。
* 減少記憶體浪費：沒有被使用的頁面永遠不會載入，騰出記憶體讓其他程式使用。
* 借助硬體支援：使用Valid-Invalid bit（有效位元）判斷頁面是否在記憶體，無效位元會觸發 Page Fault。

Figure 4： Page table when some pages are not in main memory.



## 分頁錯誤處理流程（Page Fault Handling）

此機制稱為 pure demand paging：初始時不載入任何頁，完全依需求加載。

1. 當程式試圖存取尚未載入的頁時，會觸發 Page Fault（分頁錯誤），由作業系統處理：

2. 陷入（trap）作業系統。

3. 作業系統檢查該記憶體存取是否合法。

- 若非法 → 終止程序。

- 若合法但尚未載入 → 繼續以下步驟。

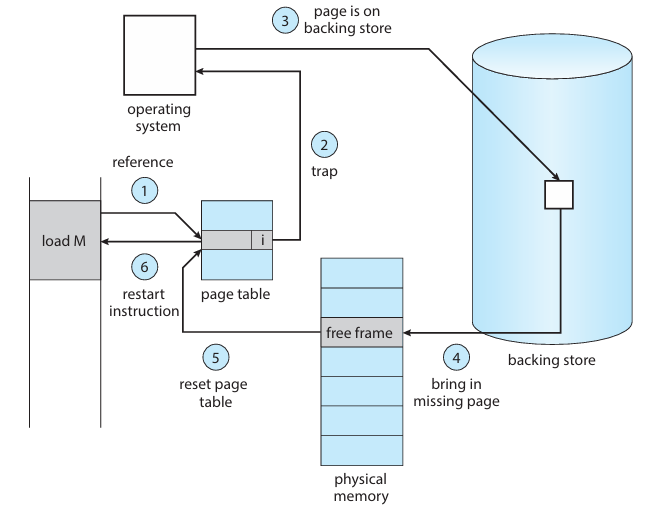
4. 從「自由框架清單（Free Frame List）」中找出一個空閒 frame。

5. 從磁碟（通常是 swap 空間或檔案系統）載入該頁。

6. 更新頁表與內部表格（將 invalid 改成 valid）。

7. 重新執行原本中斷的指令（可從同一位置繼續執行）。

Figure 5： Steps in handling a page fault



## 指令重啟問題（Instruction Restart Challenge）

部分指令可能在執行過程中途遇到 Page Fault，導致指令執行「一半」時被中斷，可能造成資料不一致。

例如：

指令：ADD A, B → C

1. 取出與解碼指令

2. 取出 A

3. 取出 B

4. 相加

5. 寫入 C

若在 C 不在記憶體 → Page Fault

更複雜情況：MVC（Move Character）指令，會搬移大量資料塊。若中途缺頁，可能只搬移一半資料。

解決方法：

1. 預先檢查頁面存取權限：在執行前先檢查所有要用到的頁面是否有效。

2. 暫存區（Buffering）回復機制：在搬移資料前先做快照，若缺頁中斷，恢復狀態重新執行。

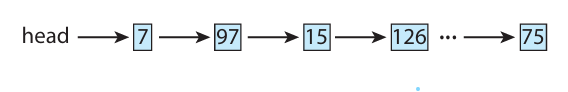
## Free-Frame List（空框架清單）

當 Page Fault 需要載入新頁面時，必須找到一個空框架（frame）來放置該頁。

Free-Frame List 就像一個「空抽屜清單」，OS 從中挑一格出來使用。系統通常會在分配前做 Zero-Fill-On-Demand：

* 先將框架內容清零，避免前一程式的機密資料被新程式讀取。
* 如果沒有可用空框架：進入 Page Replacement（頁面替換） 機制，將某些頁面換出（swap out）再載入新的頁

Figure 6：List of free frames.



# Page Replacement（頁面替換）

當程式運行中發生 Page Fault，但主記憶體中已經沒有空的 Frame（框架） 可以放新頁面時，系統必須決定「犧牲哪一頁」來騰出空間。這個選擇過程稱為 Page Replacement（頁面替換）。

這個機制影響系統效能很大，因為不當的替換策略會造成 Page Fault 率升高（Thrashing），導致 CPU 大部分時間都在等待 I/O。

## 頁面替換的處理流程

1. Page Fault 發生：程式請求一個不在記憶體中的頁面。

2. 查詢二級儲存（Secondary Storage）：找到該頁面在硬碟（Swap Space 或檔案）的位址。

3. 檢查 Free Frame List

\* 如果有空的 Frame → 直接載入，不需替換。

\* 如果沒有空 Frame → 進入頁面替換程序。

4. 選擇 Victim Page（受害者頁面）：使用 頁面替換演算法 決定哪一頁要被踢出。

5. 檢查 Dirty Bit（修改位元）

\* 如果該頁 Dirty Bit=1，代表內容已被修改 → 必須先寫回硬碟，再釋放。

\* 如果 Dirty Bit=0，該頁與硬碟一致 → 可直接捨棄。

6. 載入新頁面至 Victim Frame：更新頁表內容，將新頁面標記為 Valid。

7. 恢復被中斷的指令，程式繼續執行。

Figure 7：Need for page replacement.

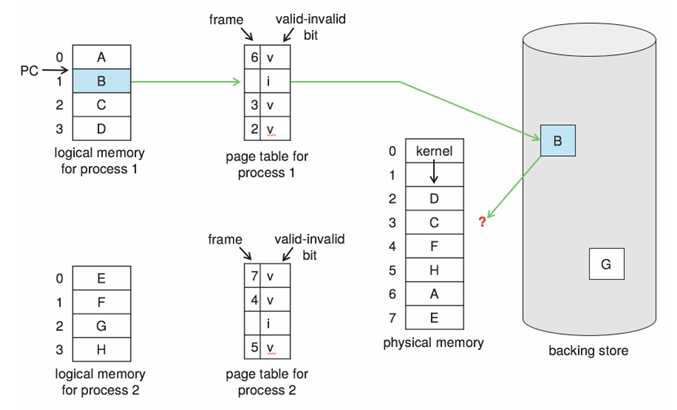
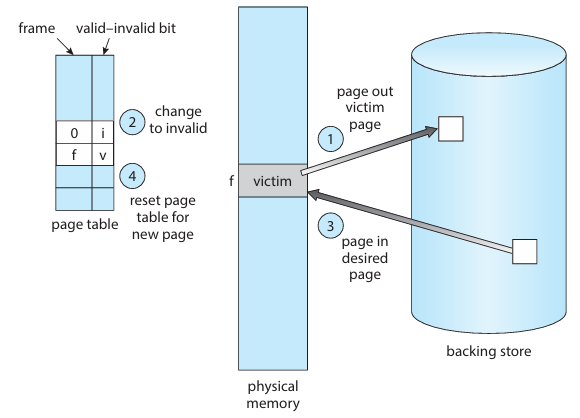


Figure 8：Page replacement



## Dirty Bit（修改位元）的作用

為了避免不必要的磁碟寫入（I/O 成本很高），系統利用 Dirty Bit（Modify Bit） 來標記該頁面是否曾被修改。

\* 0：該頁未被改動，內容與硬碟相同 → 可直接捨棄，不寫回硬碟。這樣可以減少不必要的 I/O，提升整體效率。

\* 1：該頁已被修改 → 需先將內容寫回硬碟，再釋放給其他頁面使用。

# 常見的頁面替換演算法

當記憶體不足，需要將某一頁面換出時，作業系統必須選擇一個「Victim Page（受害者頁面）」。不同演算法對 Victim Page 的選擇策略不同，影響 Page Fault 率 與 整體效能。

## FIFO（First-In-First-Out，先進先出）

最早被載入記憶體的頁面，最先被替換。使用「佇列結構」實現，頭部頁面最早進來，先被淘汰。

優點是實作簡單，僅需記錄頁面進入時間順序

缺點是不考慮頁面實際使用頻率，可能會淘汰「經常使用」的頁面。此外，可能出現 Belady's Anomaly：增加 Frame 反而 Page Fault 增加。

Figure 9： FIFO page-replacement algorithm

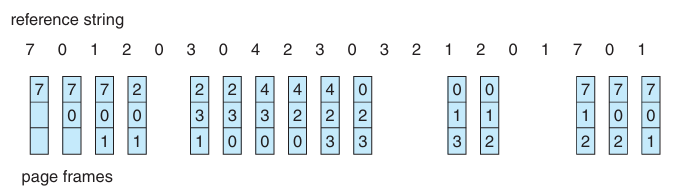
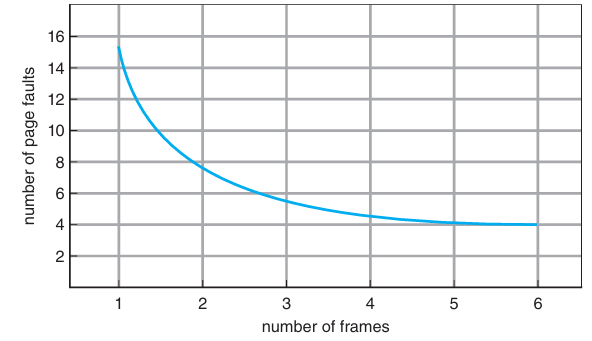


Figure 10：Graph of page faults versus number of frames.



範例：

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

Frame: 3

Page Fault 次數可能比 Frame=4 還低（違反直覺）。

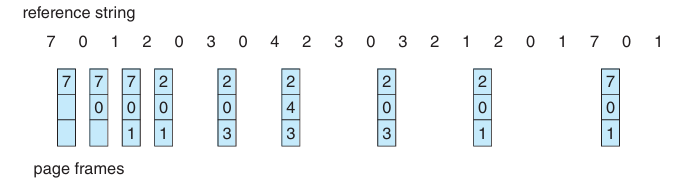
## OPT（Optimal Page Replacement，最佳演算法）

預測未來存取行為，選擇「未來最久不會被使用」的頁面換出。可達到理論上最少的 Page Fault。

優點是在理論分析中作為最優基準，評估其他演算法表現。

缺點是需要預知未來的記憶體存取序列，無法在實務系統中實作。

Figure 11：Optimal page-replacement algorithm.



## LRU（Least Recently Used，最久未使用）

是 OPT 的一種近似實現。假設「近期使用過的頁面將來也更可能被使用」，因此替換「最久沒被使用」的頁面。

優點是表現良好，Page Fault 率接近理想值。較符合程式的局部性原則（Locality Principle）。

缺點是需要額外硬體或軟體記錄「最近使用時間」：

* 計數器法（Counter）：每次存取頁面時更新時間戳記。
* 堆疊法（Stack）：將最近使用的頁面移到堆疊頂端，底部是最久未使用。

## LRU Approximation

由於完整 LRU 需大量硬體支援，實務上常使用近似演算法。

1. Reference Bit（參考位元）

* 每個 page 有一個參考位元（由硬體設定），有被訪問過設為 1。
* 無法得知時間順序，但可判斷是否被使用。

2. Additional-Reference-Bits Algorithm

* 每隔一段時間（如 100ms），將 reference bit shift 到 8-bit register。
* 越小的值代表越久沒使用。
* 可看成 page 使用的時間序列紀錄。

3. Second-Chance Algorithm（Clock Algorithm）

* 基於 FIFO，但加入參考位元判斷。
* 若 reference bit = 1 → 不替換，給它第二次機會並清為 0。
* 像時鐘一樣繞一圈，直到找到 reference bit = 0 為止。

# Frame Allocation（框架分配）

虛擬記憶體系統中，實體記憶體被切分為許多固定大小的單位，稱為 Frame（頁框）。程式被分割為相同大小的 Page（頁面），執行時必須將頁面載入 Frame 中。

但問題是：若系統共有 m 個 Frame，同時有 n 個程式在執行，作業系統應如何將 Frame 分配給這些程式，才能兼顧公平性與效能？

## 系統 Frame 資源分配概念

1. 保留部分 Frame 給 OS：作業系統本身需要 Frame 來執行核心（kernel code）、資料結構、I/O 緩衝區等。

2. 其餘 Frame 分配給多個程式。分配策略需考量：

* 每個程式的最小需求（Minimum Number of Frames）。
* 程式的大小或優先權。
* 系統的公平性與Page Fault 率。

舉例：系統有 128 個 Frame，OS 使用 35 個 → 剩下 93 個 Frame 給使用者程式。

## Minimum Number of Frames（最小 Frame 數量）

每個程式必須至少擁有一定數量的 Frame，才能保證 CPU 指令執行不中斷。若分配低於最小需求，會導致頻繁 Page Fault 或根本無法執行。

例子：

某 CPU 架構執行一條指令可能需要同時存取：

* 1 個 Frame → 存放指令本身。
* 1 個 Frame → 存放操作數（operand）。
* 1 個 Frame → 存放間接參考（indirect addressing）。

→ 每個程式至少需要 3 個 Frame，否則可能在指令執行中途就發生 Page Fault。

# Thrashing（抖動）

當系統為某個程序分配的 page frame 數量太少，不足以容納它當前正在使用的所有頁面（locality），就會發生：

1. 頁面不斷被替換出去

2. 下一條指令馬上又需要剛剛被替換出去的頁面 → 再次 page fault。

3. 不斷 page-in 與 page-out，程式花大量時間在 I/O，而不是執行。

當程序花費更多時間在分頁（paging）而不是執行真正的指令，稱為 thrashing（抖動）。

## Thrashing 的成因：系統誤以為 CPU 利用率低，增加多工處理數量

1. 系統偵測到 CPU 利用率下降 → 嘗試「新增程序」以提升利用率。

2. 新程序加入 → 分掉原本就不足的 frame。

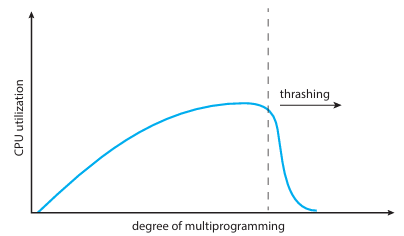
3. 所有程序都因記憶體不足而產生大量 page fault。

4. Page fault → 需要 I/O → 程序等待 → CPU idle 更久。

5. 系統錯誤地再新增更多程序（想解決 idle 問題）。

6. 最終結果是系統進入「thrashing」狀態，效率崩潰。

Figure 12：：Thrashing



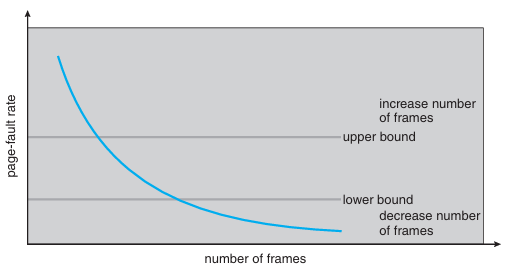
## Page-Fault Frequency (PFF) 策略

直接以「page-fault rate」作為控制參數。設定一個上限與下限，若超出上限 → 加 frame，低於下限 → 減 frame。

優點是直接、更簡潔。

缺點是需要複雜的 working set 計算與紀錄。

Figure 13：Page-fault frequency



# Memory Compression（記憶體壓縮）

記憶體壓縮（Memory Compression） 是現代作業系統用來延緩或避免 Swap Out（將頁面寫回磁碟） 的技術。

當系統記憶體壓力過大、即將不足時，與其將不常用的頁面直接換出到磁碟，系統可以選擇：

* 在 RAM 中壓縮頁面
* 將多個壓縮後的頁面裝入同一個 Frame
* 以此釋放更多實體記憶體空間

這樣可避免昂貴的磁碟 I/O 操作，提高系統反應速度。

## 傳統虛擬記憶體管理中：

* 當記憶體不足時，OS 將部分頁面移至 Swap 空間（磁碟）。
* 硬碟存取速度比 RAM 慢數萬倍 → 造成系統延遲。

記憶體壓縮的想法：

* 壓縮算法運算成本遠低於磁碟 I/O 成本。
* 利用 CPU 計算能力換取 I/O 優化，整體系統效率提升。

## 運作原理：

1. 系統偵測到可用 Frame 數量低於閾值。

2. 選擇部分低頻使用的頁面，將其內容壓縮（如 LZO、LZ4、WKdm）。

3. 將多個壓縮後的頁面打包存入單一 Frame，釋放其他 Frame。

4. 當頁面被再次需要：

\* 從壓縮區解壓縮 → 放回獨立 Frame。

\* 無需從磁碟讀回，速度更快。

Table 2：與傳統分頁機制比較

|  |  |  |
| --- | --- | --- |
| **項目** | **傳統分頁（Swap Out）** | **記憶體壓縮（Memory Compression）** |
| 儲存位置 | 硬碟（Swap 空間） | RAM（壓縮區域） |
| 成本 | I/O 延遲高（毫秒級） | CPU 計算壓縮成本較低（微秒級） |
| 效能 | 頻繁換頁造成整體效能下降 | 減少 I/O，提升系統反應速度 |
| 適用場景 | 桌面系統、傳統伺服器 | 手機、筆電、現代作業系統（macOS、Linux、Windows） |
| 空間效率 | 需完整 Frame 儲存 | 壓縮後可在一個 Frame 存放多個頁面 |

# Prepaging（預先載入）

當一個程式初次啟動或從暫停狀態恢復時，記憶體是空的，沒有任何頁面被載入。

若完全依照需求分頁（Demand Paging）方式執行，會在程式前期產生大量 Page Fault，降低程式啟動速度。

Prepaging（預先載入）是一種最佳化策略，在程式啟動或恢復時，提前載入預期會被使用的頁面，降低初期 Page Fault 次數。

優點：減少程式啟動時的 Page Fault 洪峰。也加快系統回應時間，特別是大型程式或頻繁暫停/恢復的程式。

缺點：若預測錯誤，會浪費 I/O 與記憶體空間。且難以百分之百準確預測，實務多採取保守策略（僅預載鄰近頁面）

運作機制：

1. 程式啟動或換入時，OS 預測哪些頁面會被使用。

2. 將這些頁面一次載入記憶體。

3. 後續執行時，可避免頻繁 Page Fault。

成本 vs 效益分析：

假設系統預先載入 s 頁：有 α 比例頁面最終被真正使用。

則：

\* 成本 (Cost) = 載入沒使用的頁面數量 = s × (1 – α)

\* 效益 (Benefit) = 避免的 Page Fault 次數 = s × α

\* 當 α → 1（預測準確） → 預載效益高。

\* 當 α → 0（預測失誤） → 浪費記憶體與 I/O。

# Page Size（頁面大小）

虛擬記憶體與實體記憶體皆以「Page/Frame」為單位進行管理。

頁面大小（Page Size）通常固定，例如：

* 典型值：4 KB、8 KB、16 KB
* 現代 x86-64 CPU 支援 4 KB、2 MB、1 GB 巨頁（Huge Pages）

Page Size 的選擇會影響：

* Page Table 大小
* Fragmentation（內部浪費）
* I/O 效率
* Locality 精準度
* Page Fault 次數

|  |  |  |
| --- | --- | --- |
| **考量因素** | **小頁面較佳** | **大頁面較佳** |
| **Page Table 大小** | ❌ 頁數多 → Page Table 變大 | ✅ 頁數少 → Page Table 縮小 |
| **Internal Fragmentation（內部碎裂）** | ✅ 記憶體浪費少，每頁不易留空白 | ❌ 浪費空間多，部分頁面可能沒填滿 |
| **I/O 傳輸效率** | ❌ 多次 I/O，傳輸開銷高 | ✅ 一次讀寫更多資料，效率高 |
| **Locality 精準度** | ✅ 更精準，避免載入不必要資料 | ❌ 精準度低，可能載入多餘資料 |
| **Page Fault 次數** | ❌ 頁面數多，換頁頻繁 | ✅ 頁面數少，換頁次數減少 |