# 虛擬記憶體 (virtual memory)

# 09


9.1 虛擬記憶體 (Virtual memory) 簡介

9.1.1 實體記憶體的抽象化

9.1.2 虛擬記憶體技術

9.2 分段式的虛擬記憶體機制 (segmentation)

9.2.1 分段法的原理

9.2.2 分段法的實作

9.3 分頁的虛擬記憶體機制 (paging)

9.3.1 靜態分頁配置演算法 (static paging algorithm)

9.3.2 動態分頁配置演算法 (dynamic paging algorithm)

9.4 分段分頁的記憶體配置
(segmented/demand paged memory allocation)

生意好的餐廳有所謂的翻桌率，也就是在同一段進餐時間，同一桌可能可以陸續讓好幾組客人依序用餐。所以即使餐廳只有 10 張桌面，可能一個中午出了 30 桌的菜量。電腦的記憶體空間也有限，為了服務更多的處理元，作業系統也採用類似的概念，而且有過之而無不及，處理元執行時需要的資料放置到記憶體中，即使有的放不下，也暫時放到磁碟上，等需要用到時再想辦法移到記憶體中。

隨選分頁 (demand paging) 的方法可以讓一個程式在沒有完全載入到主記憶體中的情況下，依然能夠繼續執行。如此一來，程式占有空間的大小就比較沒有限制了。這種技術稱為虛擬記憶體 (virtual memory) ，分頁本身在系統的安排下，能夠在磁碟與主記憶體中移動，讓使用者覺得執行的程式一直都位於主記憶體中。

早期的程式設計者要想辦法讓程式寫得不占有那麼多空間，後來有人想到可以把程式分段，這些段落也稱為 overlay，需要用到的段落就搬到主記憶體，雖然系統可以做到 overlay 在磁碟與主記憶體之間的搬移，但是對於程式設計者來說，程式的分段實在煞費苦心。倒是 overlay 的觀念引發了後來虛擬記憶體的觀念，解決了之前的問題。

**虛擬記憶體的技術可以分成分頁法 (paging) 與分段法 (segmentation) 兩大類**，虛擬記憶體要靠記憶體管理程式 (memory manager) 與處理器的合作，記憶體管理程式追蹤分頁 (page) 與分段 (segment) 的使用，處理器則在需要在適當的時機產生中斷 (interrupt)，處理虛擬位址。表 9-1 整理出分頁與分段虛擬記憶體技術的比較。

表9-1 分頁(paging)與分段(segmentation)虛擬記憶體技術的比較

| 分頁(paging) | 分段(segmentation) |
|---|---|
| 容許分頁框內部的碎片 (fragmentation) 存在 | 不容許內部的碎片 (internal fragmentation) 存在 |
| 不容許外部的碎片 (external fragmentation) 存在 | 容許外部的碎片 (external fragmentation) 存在 |
| 程式分割成大小一樣的分頁 (pages) | 程式分割成大小不同的分段 (segments) |
| 使用 page number 與位移 (displacement) 來計算絕對位址 (absolute address) | 使用 segment number 與位移 (displacement) 來計算絕對位址 (absolute address) |
| 需要 PMT(page map table) | 需要 SMT(segment map table) |

圖 9-1 顯示虛擬記憶體運作的方式，在多工的環境下，由於很多程式都經常處於等待的狀態，假如有多一點程式一起進行，可以讓處理器的時間不會因為某些程式進入等待的狀態而閒置，虛擬記憶體的技術有幾項優點：

1. 執行程式的大小可以不受限於主記憶體空間的大小。
2. 記憶體的使用會更有效率，因為程式會用到的部分才會占用記憶體的空間。
3. 可以進行更廣泛的多工 (multiprogramming)。
4. 避免 external fragmentation 的問題，降低 internal fragmentation 的程度。
5. 容許程式碼與資料的共用。
6. 讓程式片段的動態連結 (dynamic linking) 更容易。

當然，虛擬記憶體也有一些缺點，例如處理器硬體成本增加、處理分頁中斷 (page interrupts) 的額外成本，以及為了避免頻繁置換 (thrashing) 而增加的軟體複雜性。但是一般說來，虛擬記憶體技術所帶來的好處還是遠超過這些缺點，這是虛擬記憶體技術一直在記憶體管理中占有重要地位的原因之一。

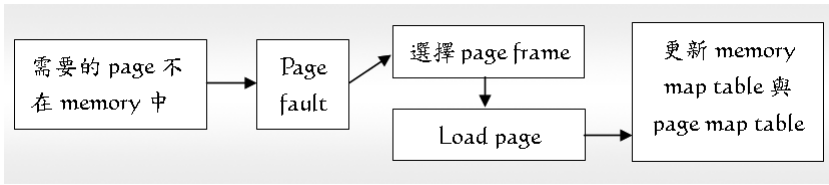



圖9-1 虛擬記憶體運作的方式

## 9.1 虛擬記憶體(Virtual memory)簡介

虛擬記憶體的機制可以讓沒有完全存在於主記憶體中的處理元能夠執行，換句話說，處理元的位址空間 (address space) 並沒有完全載入到主記憶體中。做法是把處理元的位址空間分割，如此一來，需要用到的位址空間分割 (address space partition) 必須載入，還用不上的就可以先存在 secondary memory 裡頭。圖 9-2 顯示虛擬記憶體擴充主記憶體空間的原理，虛擬記憶體是位於磁碟上的空間，利用軟體的技術扮演類似於主記憶體的角色，所以 process address space 才能延伸到磁碟中。

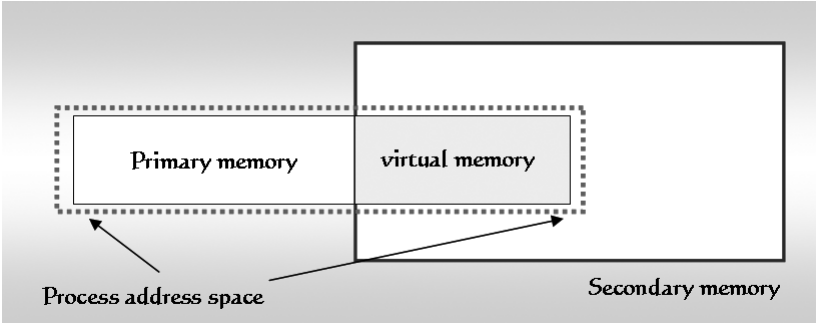



圖9-2 虛擬記憶體擴充主記憶體空間的原理

**以程式碼區段 (code segment) 來說，不同的執行階段往往用到的分割不同，這裡的分割是指主記憶體上的一部分儲存空間**，例如多數程式在剛開始的時候多半會對一些資料結構做啟始化的處理，並且輸入資料，然後會進入密集的運算階段，最後則會輸出結果。

資料區段 (data segment) 也有類似的特徵，我們把這種特性稱為空間引用的區域化特性 (spatial reference locality)。當處理元在進行某一階段的執行時，它的 spatial locality 可定義為所引用位址的集合，因為這一階段的引用會集中在這個集合中的位址。當處理元進入另一個階段的執行時，locality 也會跟著改變。

虛擬記憶體管理員 (virtual memory manager) 的主要工作就是判斷處理元各執行階段集中使用的位址，然後在進入執行階段時儘量把對應的位址空間分割載入到主記憶體中。假如虛擬記憶體管理員能很精確地維持處理元會用到的位址空間，基本上，系統整體的記憶體空間的需求就會降低，只是實際上並沒有那麼容易。有下列幾項困難：

1. 記憶體管理員必須能接受位址空間的分割 。
2. 位址空間的分割必須能載入並且動態地連結到程式所用的位址 。

**學習活動**

虛擬記憶體的技術大約在 1970 年代的末期開始出現，當時主記憶體的價格很貴，虛擬記憶體的技術可以降低程式執行時對於主記憶體空間的需求。隨著主記憶體的價格下降，目前虛擬記憶體的技術主要的目的在於提昇系統的效能。

### 9.1.1 實體記憶體的抽象化

虛擬記憶體管理員在 secondary memory 建立的虛擬位址空間 (virtual address space) 可以看成是實體記憶體的抽象化，當系統運作時，虛擬記憶體管理員會自動控制主記憶體與虛擬記憶體之間的對應，促成資料方塊在主記憶體與次記憶體間自動的移轉。**我們可以這麼想，在虛擬位址空間存在的情況下，對於使用者來說，其實虛擬位址空間跟實體位址空間並沒有差別，只是在程式用到虛擬位址空間的時候，作業系統要把資料搬到實體記憶體中**。要完全了解實際的機制與細節，必須進一步地認識 memory map table 與 page map table 的內涵與使用方式。

當程式編譯時，其實已經開始有虛擬位址空間與程式位址空間之間的對應關係，不過得等到執行時期，虛擬位址才會連結 (bind) 到實體的記憶體位址。簡單地說，虛擬記憶體技術可以讓處理元引用 (reference) 很大的位址空間，但實際執行時卻只需要較少量的實體記憶體空間，當引用的資料不在實體記憶體中，則需要運用軟體的技術將資料找到並且載入。雖然主記憶體的價格已經降低，但是處理元對於記憶體的需求卻增加得更快，所以虛擬記憶體技術仍然使用中。

### 9.1.2 虛擬記憶體技術

對於交換系統 (swapping system) 來說，並不需要區分絕對模組 (absolute module) 與主記憶體中的位址空間，因為兩者大小相同，差異只在於重定位的值 (relocation value) 。但對於虛擬記憶體來說，符號名稱 (symbolic name)、虛擬位址 (virtual address) 和實體位址空間 (physical address space) 是有差異的，3 者之間有對應的關係 (mappings) 。虛擬位址與實體位址之間的對應有兩種方式：

1. 分段法 (segmentation)。
2. 分頁法 (paging)。

原始程式中有符號名稱 (symbolic identifier)、標籤 (label) 與變數 (variable) 等形成所謂的名稱空間 (name space)。當程式轉譯成絕對映像 (absolute image) 時，這些名稱會對應到虛擬位址，當程式的絕對映像轉換成可執行的映像 (executable image) 時，虛擬位址會再對應到主記憶體中的實體位址 (physical address)。圖 9-3 畫出這個位址對應的過程。

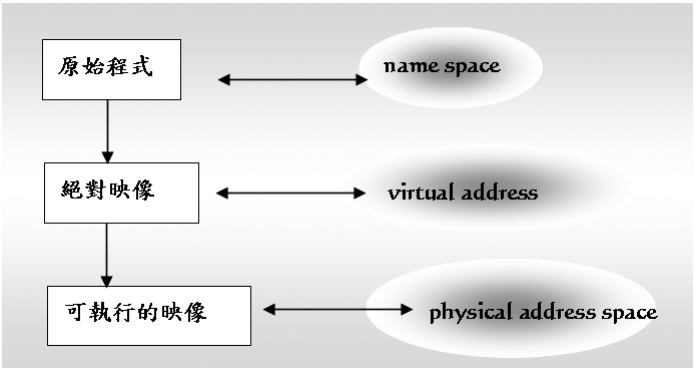



圖9-3 位址對應的過程

虛擬記憶體管理員要支援所謂的動態載入 (dynamic loading) 的功能，在執行時期把虛擬位址與實體位址連結 (bind) 起來。這種對應會隨時間而改變，所以是一種動態的對應關係。在真正運作的過程中，程式引用的物件不在記憶體內時，處理元的執行會暫停，虛擬記憶體管理員要載入引用的物件，同時重建所謂的位址轉換對應關係 (address translation map) ，了解了虛擬記憶體管理員的功能之後，我們下面就可以一起來探討虛擬記憶體實作的方法：segmentation 與 paging。

**學習活動**

當程式執行時引用到的物件位於 virtual address space 時，系統馬上會偵測到該物件不在實體記憶體中，所以一連串的處理程序會因此而啟動，把物件載入到主記憶體，同時重新定義對應，即 address translation map。在虛擬記憶體系統中，處理器要有辦法先暫停指令的執行，等到物件載入而且對應也重新定義以後才繼續執行原來的指令。

## 9.2 分段式的虛擬記憶體機制(segmentation)

分段式的虛擬記憶體機制 (segmentation) 和前面提到的 relocation register 與 limit register 的方法近似，由程式設計者本身決定程式的分割方式，產生大小不一的分段 (segment) ，例如 UNIX C compiler 訂的 text, data 與 stack segment。分割之後，記憶體空間的位置就可以用：**< 分段號碼 , 平移量** (offset)> 來決定。分段號碼指定記憶體的某個區塊，平移量指所在之處與分段起點之間的距離。**分段本身就成為虛擬記憶體管理員在主記憶體與次記憶體之間移動資料的單位**。

分段式的虛擬記憶體機制可由程式設計者做設定與調整，由於程式設計者比較清楚程式的行為，這種方式可能可以在效率上占一點優勢，但是由分段造成的外部空間散佈 (external fragmentation) 問題依然存在，是比較不利的因素。分段式的虛擬記憶體機制在使用與實作上的難度比較高。

## 9.2.1 分段法的原理

分段的由來是程式的結構化，也就是把程式分成模組 (modules)，所以在分段式的記憶體配置法中，我們把 job 分成大小不一的幾個分段 (segment)，每個分段剛好對應到一個模組，通常模組內執行的是相關的功能，副程式 (subroutine) 就可以看成是一種模組。在分段法中主記憶體不必再分割成 page frames，當程式編譯與處理的過程中，系統會開始建立分段的資訊，每個 job 會有一個分段對應表格 (SMT, segment map table)，裡面記載分段的號碼、分段的長度、存取的權限、狀態與在記憶體中的位置。

記憶體管理員必須記錄記憶體中的分段，結合動態分割與 demand paging 的記憶體管理技術。Job table 列出處理中的 jobs，整個系統只有一個 job table，SMT(segment map table) 列出每個分段的細節，每個 job 有一個 SMT，MMT(memory map table) 記錄主記憶體的分配，整個系統只有一個 MMT。圖 9-4 顯示的程式含有兩個副程式，分成 3 個分段。每個分段中的指令依序排列，但是程式的所有分段不必儲存在記憶體的連續位置上。

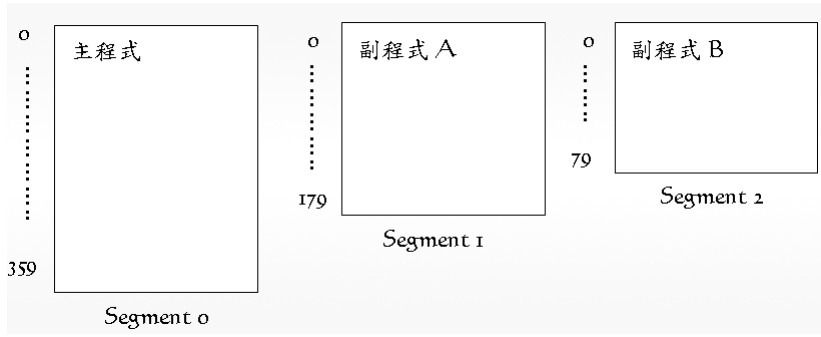



圖9-4 分段式的記憶體配置方式

memory manager 必須記錄記憶體中有關於分段的資訊，圖 9-5 顯示 segment 1 載入以後從 8000 的位置開始，job 1 的 SMT 有這樣的記錄，所以系統可以找到 segment 1。假如某個指令要求從副程式 A 的第 100 行開始執行，則系統要把第 100 行距離副程式 A 開端的距離再加上 8000 才能得到在主記憶體中的實際位址。

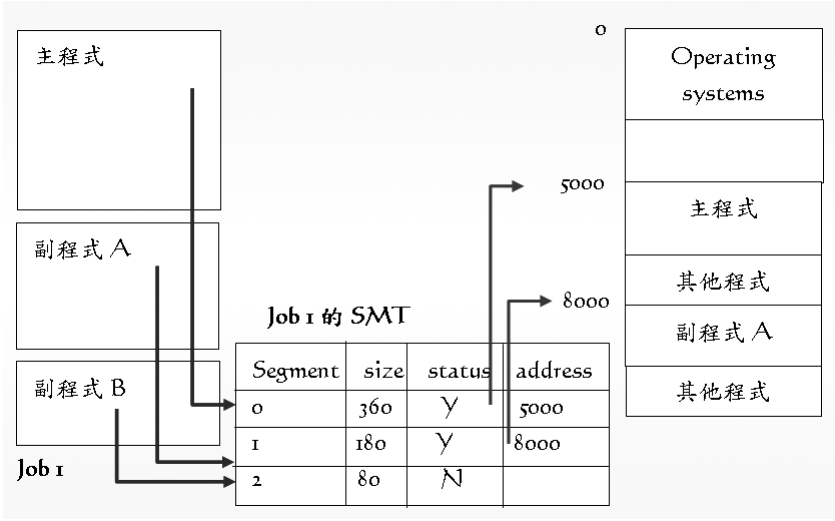



圖9-5 SMT的內容與用途

## 9.2.2 分段法的實作

分段法的實作方式很多，分段式的記憶體系統要如何實作決定於作業系統的設計者，圖 9-6 顯示一個可能的實作方式，4 個分段暫存器 (segment register) 的功能如下：

1. STR(segment table register)：儲存指向 segment table 的指標 。
2. CBR(code base register)：儲存 code segment 的 base value，有時候也稱為 PBR(procedure base register)。
3. DBR(data base register)：用來為靜態資料引用進行動態的重定位 。
4. SBR(stack base register)：指向含有 process stack 的分段 。

上面的描述是假設硬體上有這樣的配備，實際的情況當然會有一些差異，Multics 作業系統在設計上支援分段法，軟體與硬體的功能配合得很好，是相當有名的分段虛擬記憶體系統。假如想對圖 9-6 的架構進行更深入的了解，可以參考 Multics 作業系統。

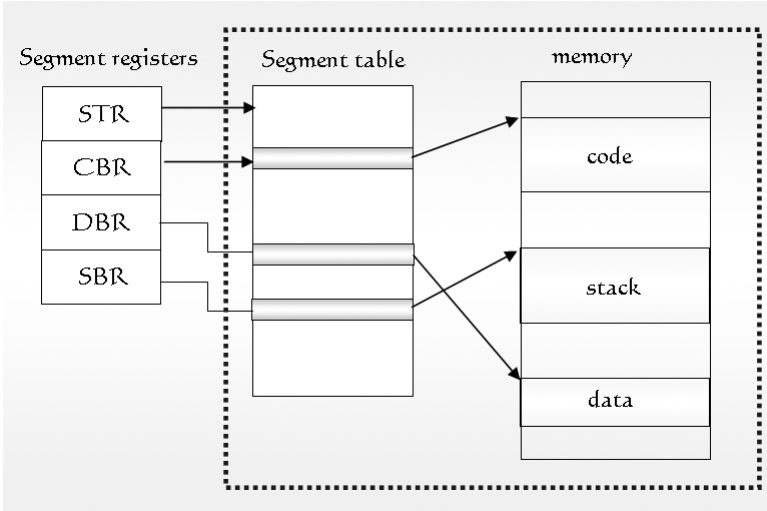



圖9-6 分段法的實作

## 9.3 分頁的虛擬記憶體機制(paging)

分頁的記憶體配置 (paged memory allocation) 方式將 CPU 所要處理的 job 分成大小一樣的分頁 (page)，有的作業系統以記憶體區塊的大小為分頁的大小，而且也剛好跟磁碟上區塊的大小一樣，假如要以名詞來稱呼這些不同的儲存單位，可以用下面的名稱：

1. 磁碟上的區塊：稱為 sector 或 block。
2. 主記憶體的區塊：稱為 page frame。
3. job 的區塊：稱為 page。

在程式執行以前，memory manager 要查知程式所需要的 page 數目，然後在主記憶體中尋找足夠的 page frames，接著將含有程式的 pages 載入到所找到的 page frames 中。當程式準備要載入執行的時候，其 pages 有邏輯上的順序，也就是第 1 頁含有第 1 部分的指令，最後一頁含有最後部分的指令。**在分頁的配置方式中，分頁不必一定要載入到主記憶體相鄰連續的 page frames 上**。把 job 放到不連續的 page frames 上的好處是：空白的 page frame 可以分配給任何 jobs，這樣主記憶體空間的運用會比較有效率。External fragmentation 的現象不會出現在分頁法中，internal fragmentation 只在少數的 page frame 中存在，例如圖 9-7 左邊的 page 3 載入後會造成右邊 main memory 中 job-3/page 3 的 internal fragmentation。不過分頁的方式會形成作業系統額外的負擔，因為必須記載 job 是載入到那些 page frames 上。

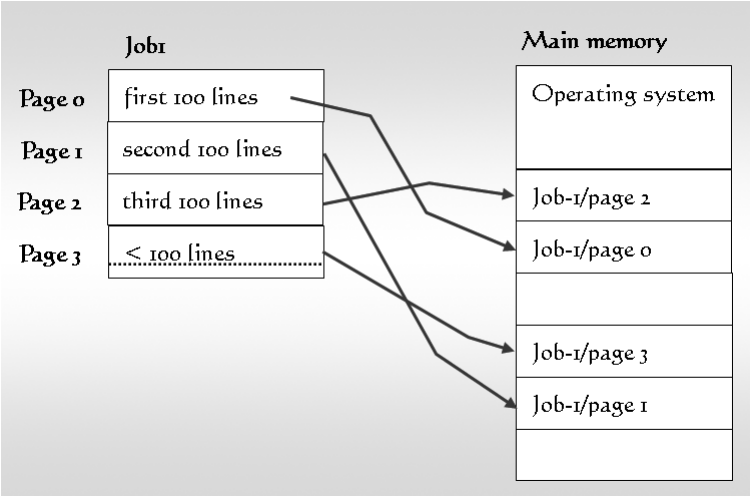



圖9-7 分頁的記憶體配置(paged memory allocation)方式

分頁的虛擬記憶體機制 (paging) 使用單一成份的位址，虛擬記憶體空間分成線性的 (linear) 虛擬位址，程式設計者不需要知道虛擬記憶體空間如何運作，完全由虛擬記憶體管理程式負責把固定大小的分頁 (page) 依照需求在主記憶體與次記憶體之間移動。在實作上，分頁機制比較簡單，使用者對於技術上的細節不必了解。每個處理元的虛擬位址空間分成邏輯上的分頁，外部空間散佈的問題 (external fragmentation) 比較小。

主記憶體配置給處理元的單位以分頁框 (page frame) 稱之，大小和分頁是一樣的，**分配給同一個處理元的分頁框不必在連續的位址上，因為分頁對應 (page map) 會記載虛擬空間與實體空間之間分頁的對應關係**。這裡還是要強調一個基本觀念，就是程式執行時通常在資料的引用上會有區域性的現象 (reference locality)，所以不必把可能用到的所有分頁都載入到主記憶體中。常見的分頁配置演算法有兩種：

1. 靜態配置演算法 (static allocation)：當處理元建立的時候可以分配到固定數目的分頁框 (page frame)。然後由分頁配置的政策來處理後來分頁的載入與卸載 。
2. 動態配置演算法 (dynamic allocation)：在程式執行的時候 ，依照程式的需求改變記憶體空間的分配 。

## 9.3.1 靜態分頁配置演算法(static paging algorithm)

靜態配置演算法在處理元產生時會配置固定數目的頁框給處理元，然後定義虛擬記憶體系統如何進行後續的載入 (load) 與卸載 (unload) ，分頁配置演算法中通常有以下的政策：

1. 取用政策 (fetch policy)：決定分頁何時載入主記憶體 。
2. 替換政策 (replacement policy)：決定系統資源不足時那一個分頁應卸載 。
3. 置放政策 (placement policy)：決定取用的分頁應放在何處 。

靜態記憶體配置的演算法中，分配的 page frames 的數目是固定的，新的 page 分配到的 page frame 一定是被替換的 page 原先載入的地方，所以置放政策是固定的。各種靜態記憶體配置的演算法的主要差異是在取用政策與替換政策。假設 N 是 virtual address space 中的分頁的集合，則分頁引用的串列 (page reference stream) 可以表示如下：

```
w = r1, r2, r3, ……, ri, ……
```

w 代表程式執行過程中引用到的分頁，這些分頁被引用的順序對於系統效能的影響很大，不過以實際的情況來看，很難在程式執行以前精確地預測出分頁引用的順序。

#### 9.3.1.1 取用政策(fetch policy)

取用政策決定分頁那時候會被載入到主記憶體中，分頁演算法通常不會預先知道分頁引用的順序，所以要做到預先擷取 (prefetch) 是不太可能的，prefetch 是指在分頁被引用之前就先載入到主記憶體中。大多數的演算法都採用所謂的依需求取用分頁 (demand paging) 的方法，也就是當程式引用到分頁時才將分頁載入到主記憶體中。所以在程式執行過程中的任意時間 t，出現在引用順序中的分頁為 Pt 時，下面所列的是可能出現的幾種狀況：

1. 假如 $P_t$ 在 (t-1) 時已經載入 ，則不需要進行任何處理 。
2. 假如 $P_t$ 在 (t-1) 時還未載入 ，而且分配給程式的 page frame 還有空的 ，則將 [ 引用的分頁載入到空的 page frame 中 。
3. 假如 $P_t$ 在 (t-1) 時還未載入 ，而且分配給程式的 page frame 沒有空的 ，則必須進行置換 。

#### 9.3.1.2 需求分頁法(demand paging)

依需求分頁 (demand paging) 的觀念是指只將程式的一部分載入到記憶體中，原本在 job 開始執行一直到結束，整個 job 都要放在記憶體中，假如程式引用到沒有載入的分頁，再將分頁載入到 page frame。**前面已經介紹過 fetch policy 與 placement policy，在靜態記憶體配置的演算法中，這兩者都比較固定，所以下面的介紹以 replacement policy 為主**。

在隨機的替換 (random replacement) 政策中，被替換的分頁是隨機挑選的，所以完全沒有考慮到資料使用的特性。一般說來，這種方法的效能不佳，會造成很多 page faults。Belady 的演算法是一種理想中的演算法，也稱為 Belady's optimal algorithm，在挑選替換的分頁時，會選最久會再被用到的分頁來替換。假設主記憶體有 3 個 page frame 分配給目前的 job，而分頁使用的順序為：

```
w=0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7
```

則依照 Belady 的演算法，分頁替換的方式會像表 9-2 所表示的。星號 (*) 表示產生了 page fault，所以一共產生了 10 個 page faults。

表9-2 Belady的靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1* | 1 | 1 | 4* | 4 | 4 | 7* |
| 1 | | 1* | 1 | 1 | 1 | 1 | 2* | 2 | 2 | 2 | 2 | 2 | 2 | 5* | 5 | 5 |
| 2 | | | 2* | 3* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 6* | 6 |

最久沒用的 (LRU, least recently used) 先替換的演算法選擇過去一段時間最久沒用到的分頁來替換。一般說來，程式在執行的時候常會有區域性引用的現象，例如執行到程式迴路的時候，可能引用到的資料都集中在某個分頁中，在這種情形下，越是最近用到的分頁，再被引用到的機會應該比較高。表 9-3 顯示 LRU 的靜態配置演算法，一共造成 16 次 page faults。

表9-3 LRU的靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 1* | 1 | 1 | 4* | 4 | 4 | 7* |
| 1 | | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 5* | 5 | 5 |
| 2 | | | 2* | 2 | 2 | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 6* | 6 |

最不常用的 (LFU, least frequently used) 先替換的演算法選擇過去最不常用到的分頁來替換，可能有多個分頁都滿足條件，系統必須逕行選擇其一。LFU 對於引用區域性的形成，在反應上會比較慢一點，因為剛進入一個引用區域時，還無法從使用的頻率來做正確的判斷。LFU 的另外一個問題是使用頻率是從一開始算起。有時候程式開始執行時會有一些啟始的程式碼，對於 LFU 的判斷沒有幫助。所以 LFU 的改善方式之一是在計算使用頻率時從分頁上次載入的時間開始算起。表 9-4 顯示 LFU 的靜態配置演算法，一共造成 12 次 page faults。

表9-4 LFU的靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3* | 3 | 3 | 3 | 3 |
| 1 | | 1* | 1 | 1 | 1 | 1 | 1 | 3* | 3 | 1* | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | | | 2* | 3* | 3 | 3 | 2* | 2 | 2 | 2 | 2 | 2 | 4* | 5* | 6* | 7* |

先進先出 (FIFO, first-in, first-out) 的置換演算法把存在於記憶體中最久的分頁先替換掉，FIFO 判斷的標準是分頁在主記憶體存在的時間長短，所以很容易實作，但是跟多數程式執行時的實際行為有落差。表 9-5 顯示 FIFO 的靜態配置演算法，一共造成 16 次 page faults。

表9-5 FIFO的靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 1* | 1 | 1 | 4* | 4 | 4 | 7* |
| 1 | | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 5* | 5 | 5 |
| 2 | | | 2* | 2 | 2 | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 6* | 6 |

### 9.3.1.3 堆疊演算法(stack algorithms)

依照需求來配置分頁的演算法有時候表現出來的效果會有一些變化，可能會超出我們直覺的想像，下面用一個特殊的例子來說明，假設分頁引用的順序變成：

```
w = 0 1 2 3 0 1 4 0 1 2 3 4
```

假如分頁的配置採用 FIFO，分配的 page frames 的數目為 3，則依照表 9-6 的分析會產生 9 次的 page faults，這時候我們可能會想到，若是增加分配的 page frames 的數目，可能 page faults 的數目會降低。

表9-6 只使用了3個page frame的FIFO靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 3* | 3 | 3 | 4* | 4 | 4 | 4 | 4 | 4 |
| 1 | | 1* | 1 | 1 | 0* | 0 | 0 | 0 | 0 | 2* | 2 | 2 |
| 2 | | | 2* | 2 | 2 | 1* | 1 | 1 | 1 | 1 | 3* | 3 |

假如現在分配的 page frames 的數目增加為 4 個，仍然採用 FIFO 的演算法，則依照表 9-7 的分析，一共會造成 10 個 page faults，這是很奇怪的現象，也稱為 Belady's anomaly，為什麼分配的 page frames 增加了，page faults 反而沒有減少？

表9-7 使用4個page frames的FIFO的靜態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 4 | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 4* | 4 | 4 | 4 | 3* | 3 |
| 1 | | 1* | 1 | 1 | 1 | 1 | 1 | 0* | 0 | 0 | 0 | 4* |
| 2 | | | 2* | 2 | 2 | 2 | 2 | 2 | 1* | 1 | 1 | 1 |
| 3 | | | | 3* | 3 | 3 | 3 | 3 | 3 | 2* | 2 | 2 |

基本上，載入到主記憶體的分頁在配置的 page frames 數目不同時也會不一樣，例如表 9-6 中的 page 4 第 1 次引用時，page 1 位於主記憶體中，但是在表 9-7 中，page 4 第 1 次引用時，page 1 被替換，接下來的置換情況差異更大。對於某些分頁演算法在分配的 page frames 的數目增加時，原來可以載入的分頁的集合會是 page frames 數目增加以後載入的分頁的集合之子集合，也就是說，分配的 page frames 的數目增加，載入的分頁的數目除了原來載入的之外還會增加額外的分頁。這種特性稱為包含特性 (inclusion property)，這一類的演算法稱為堆疊演算法 (stack algorithms)，堆疊演算法不會有 Belady's anomaly 的現象。FIFO 演算法不滿足包含特性，以 page 4 第 1 次被引用時的情況來看，載入分頁的集合在表 9-6 是 {4,0,1}，在表 9-7 則是 {4,1,2,3}，並沒有子集合的關係，LRU 與 LFU 則都是堆疊演算法。

## 9.3.2 動態分頁配置演算法(dynamic paging algorithm)

動態配置演算法會考量處理元執行過程中需求的改變，修正記憶體的配置情況。處理元引用記憶體位址的區域性關聯 (locality) 在此就可以派上用場，工作集演算法 (working set algorithm) 就是著名的動態配置演算法。

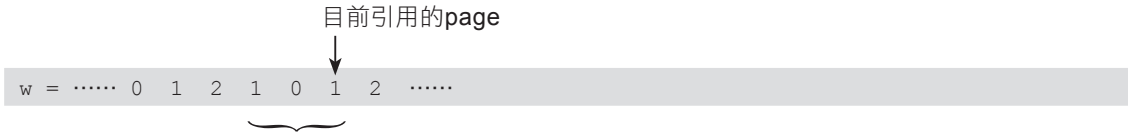
**一個 job 的工作集 (working set) 是記憶體中 job 使用的 pages 的集合**，當使用者開始執行程式以後，開始會有 pages 載入到記憶體中，經過一陣子以後，大多數的程式對於 pages 的存取都會進入穩定的狀態，很少有 page fault 發生，代表 job 會用到的 pages 都已經在主記憶體中，這些 pages 就形成了 job 的 working set。不過程式的執行有可能在進入另一個階段以後又開始產生 page faults，因為用到的 working set 改變了。

比較沒有結構的程式可能需要把所有用到的 pages 都載入到記憶體中，才會開始執行。不過多數的程式都會有某種結構，使得 pages 的引用到達一種引用區域化 (locality of reference) 的狀態。例如程式進入迴路 (loop) 的執行時，很明顯地會反覆使用某些 pages，一直持續到迴路執行結束。下面先來思考一下工作集中兩個重要的觀念：

1. 為什麼不乾脆把 job 的 working set 中所有的 pages 一次全部都載入到記憶體中呢？首先，working set 會隨 locality 而改變，所以除非是把所有用道 pages 都載入，否則無法全部一次都載入。

2. 分時系統中會有 job swapping 的現象，重新載入記憶體的 job 一開始都會產生很多 page faults，影響系統的效能。

有很多 paging system 試著找出 job 的 working set，然後在 job 開始執行前先把 working set 載入到記憶體中。不過在 job 開始執行之前要找出 working set 並不容易。Working set 導入所謂的 window size 的觀念，代表進行判斷時需要考量的最近的分頁的數目，假設 window size, ws=3，我們可以觀察以下的分頁引用順序：




既然 ws=3，表示 working set 是 {0,1}，因為從目前引用的 page 往前推（包括 page 本身）一共考量 3 個 pages，所引用到的只有 0 與 1 兩個 pages，所以 job 只需要 2 個 page frames 即可。除了 window size 以外，也可以用 page fault 的數目為指標，當這個數目高過某個門檻時，就把分配的 page frames 的數目提高，若是 page fault 低於某個數目，則可降低分配的 page frames 的數目。工作集演算法的效能和 locality 與 window size 都有關，通常若是 window size 太小的話，工作集演算法容易造成反覆置換 (thrashing)。假如知道 page faults 的發生率，可以調整 window size，例如 page faults 發生太快時，window size 應該要大一點。表 9-8 顯示 Window size 為 3 的工作集動態配置演算法，產生 16 次 page faults。

表9-8 Window size為3的工作集動態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 1* | 1 | 1 | 4* | 4 | 4 | 7* |
| 1 | | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 2* | 2 | 2 | 5* | 5 | 5 |
| 2 | | | 2* | 2 | 2 | 1* | 1 | 1 | 0* | 0 | 0 | 3* | 3 | 3 | 6* | 6 |
| 分配 | 1 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

圖 9-8 中分配的 page frames 的數目從 0 開始增加到 3，假如 window size 增加為 4，剛好可以容納引用序列的 locality，則效果大幅改善，page faults 降到 8 次。事實上引用序列中曾經引用到的不同的分頁數目為 8，所以只發生 8 次 page faults 是最佳的狀況了！表 9-9 顯示 Window size 為 4 的工作集動態配置演算法，跟表 9-8 的差異在於 page 3 第 1 次引用時，假如 window size 為 3，則 page 0 會被替換，若是 window size 為 4，由於還放得下，所以沒有分頁需要置換，注意工作集演算法置換的政策與 LRU 類似。

表9-9 Window size為4的工作集動態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4* | 4 | 4 | 4 |
| 1 | | 1* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 5* | 5 | 5 |
| 2 | | | 2* | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 6* | 6 |
| 3 | | | | 3* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 7* |
| 分配 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |

當 window size 超過引用區域 (locality) 的大小時，其實分配的 page frame 是會少於 window size 的，像圖 9-10 的例子中，window size 是 9，但是 job 最多引用到的分頁數目才不過 8 個，雖然分配了比較多的記憶體空間，但是這時候 page faults 的數目並沒有再降低，仍然是 8。

表9-10 Window size為9的工作集動態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | | 1* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | | | 2* | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | | | | 3* | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | | | | | | | | | | | | | 4* | 4 | 4 | 4 |
| 5 | | | | | | | | | | | | | | 5* | 5 | 5 |
| 6 | | | | | | | | | | | | | | | 6* | 6 |
| 7 | | | | | | | | | | | | | | | | 7* |
| 配置 | 1 | 2 | 3 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 | 5 | 6 | 7 | 8 |

下面表 9-11 的例子中，隨著 job 的執行，所分配的 page frames 反而有減少的趨勢，注意這跟分頁引用序列的改變有關，表 9-11 的分頁引用序列為：

```
w = 0 1 2 3 0 1 0 1 2 3 2 3 4 5 6 7
```

其中 0101 與 2323 的兩段都只需要分配 2 個 page frames，這代表 locality 縮小為 2 個分頁，從這裡可以看到動態分頁配置在主記憶體運用上的彈性。不過以實作來說，動態分頁配置的演算法比 LRU 演算法還要困難。

表9-11 使用4個page frames但是引用次序不同的工作集動態配置演算法

| frame | 0 | 1 | 2 | 3 | 0 | 1 | 0 | 1 | 2 | 3 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | 4* | 4 | 4 | 4 |
| 1 | | 1* | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | | 5* | 5 | 5 |
| 2 | | | 2* | 2 | 2 | 2 | | | 2* | 2 | 2 | 2 | 2 | 2 | 6* | 6 |
| 3 | | | | 3* | 3 | 3 | 3 | | | 3* | 3 | 3 | 3 | 3 | 3 | 7* |
| 配置 | 1 | 2 | 3 | 4 | 4 | 4 | 3 | 2 | 3 | 4 | 3 | 2 | 3 | 4 | 4 | 4 |

## 9.4 分段分頁的記憶體配置(segmented/demand paged memory allocation)

分段分頁的記憶體配置方法綜合了分段與分頁的記憶體配置方法，主要的目的是希望能獲得兩種方法的好處。在這個配置方法中，分段 (segment) 不再是單一的連續空間，而是細分成大小一致的 pages。通常單一的分頁是要比分段來得小的。如此一來，可以避免分段法的一些缺失，例如 external fragmentation。分段分頁的記憶體配置方法需要下列 4 種表格：

1. 系統要保存一個記載處理中的 jobs 的 Job Table。
2. SMT(segment map table) 要列出每個分段的細節 ，每個 job 都有一個 SMT。
3. 每個 segment 有一個 PMT(page map table)，用來記載每個 page 的細節 。
4. 系統保存一個 MMT(memory map table)，用來監控 page frames 在記憶體中配置的狀況 。

圖 9-8 顯示分段分頁法的原理，雖然表個格之間的關係有表示出來，但是跟實際的表格比較起來，已經簡化了，例如 SMT 中有關於使用的權限保護設定就省略了，還有那些使用者可以使用分段的資訊也沒有呈現出來。PMT 原本也有一些位元用來代表 status、modified 與 referenced。在存取主記憶體的某個位置時，系統必須先確認位址 (address)，由 segment number、segment 中的 page number 與 page 內的位移 (displacement) 共 3 個部分所組成。

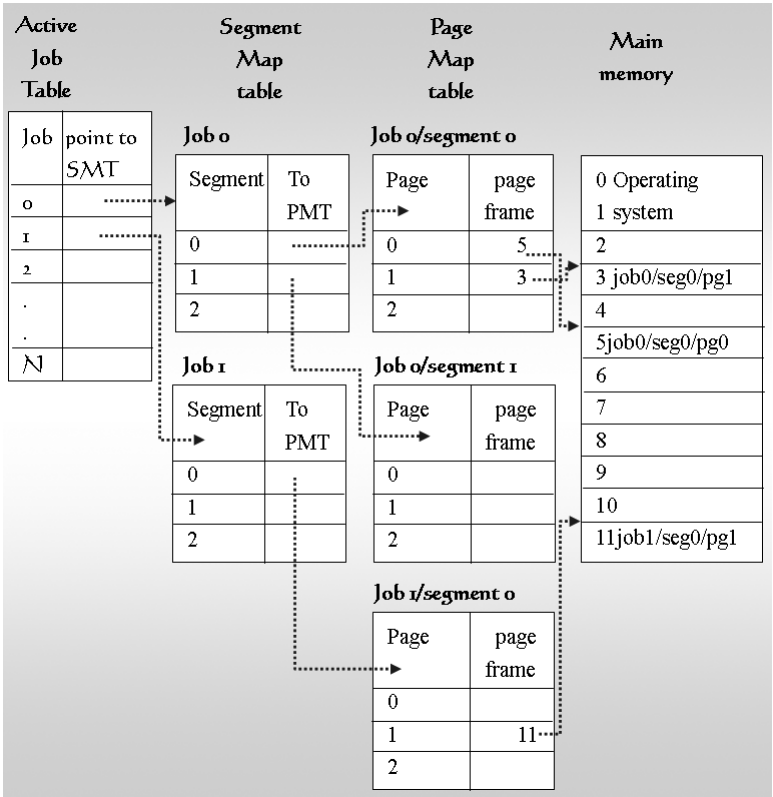



圖9-8 分段分頁法的原理

分段分頁法的主要缺點是維護各種表格所花費的時間，為了降低引用各種表格的次數，很多作業系統採用關聯記憶體 (associative memory) 的技術，associative memory 是指與分配給 job 的幾個暫存器 (registers)，

當 job 載入以後，其 SMT 也會載入到記憶體中，但是 PMT 只有用到的時候才會載入，當 pages 在主記憶體與磁碟之間移動的時候，相關的表格也會跟著更新。

1. 當某個 page 被引用時，系統會先搜尋 job 的 SMT，找到對應的 PMT。

2. PMT 接著載入到記憶體中，決定 page 在記憶體中的位置。

3. 假如 page 不在記憶體中，發出 page interrupt，將 page 載入到記憶體中，然後更新相關的 tables。

Associative memory 把最近使用過的 pages 的資訊存在記憶體中，當 page request 產生的時候，系統除了循上述的程序找尋 page 之外，也會從 associative registers 中進行搜尋。假如在 associative registers 中已經找到 page 的相關資料，則其他的搜尋程序都會停止。

1. 請說明分段 (segmentation) 與分頁 (page) 的差異。
2. 系統發生反覆替換 (thrashing) 的現象時，分頁會一直在主記憶體與次記憶體之間快速地反覆移動，對於記憶體的使用來說是很沒有效率的，請說明反覆替換發生的原因？作業系統要如何偵測與避免反覆替換的現象？
3. 請說明虛擬記憶體技術的優點以及可能產生的缺失。
4. 請說明分段分頁記憶體配置法的原理。
5. 假設分頁引用的順序為 w=0 1 3 0 4 6 7 5 0 1 0 1，請仿照表 9-7 使用 4 個 page frames 的 FIFO 的靜態配置演算法，將表格中的資料寫出來。