虛擬記憶體

資科系
林偉川

虛擬記憶體

- 實體記憶體管理的目的：
  - 同時執行多個行程，並對CPU作最有效的利用
- 行程的大小與數目受限於實體記憶體的容量
- 虛擬記憶體：允許程式不必完全載入到記憶體中就可以執行的機制(96tpu 6(5), 98tku 1(b))
  - 能夠執行記憶體需求大於實體記憶體的程式
  - 程式規劃上變得容易

虛擬記憶體

- 頁框配置的原則與方法
- 使用需求分頁來探討虛擬記憶體
- 系統設計上需要考量的因素，如預先分頁、程式結構、分頁大小等

3

基本概念

- 如果只需要部份程式在記憶體中就可以執行，會有下列優點：(dim a(10000,10000) as integer)
  - 程式不會被實體記憶體的容量所限制
    - 程式設計師可以設計超過實體記憶體容量的程式；簡化程式設計的工作。
  - 剩餘的記憶體可讓更多行程同時在記憶體中執行
    - 可增加CPU使用率與產量
    - 反應和回覆時間並不因此增加。
  - 置換次數會減少；每一個使用者程式平均可以更快地被執行

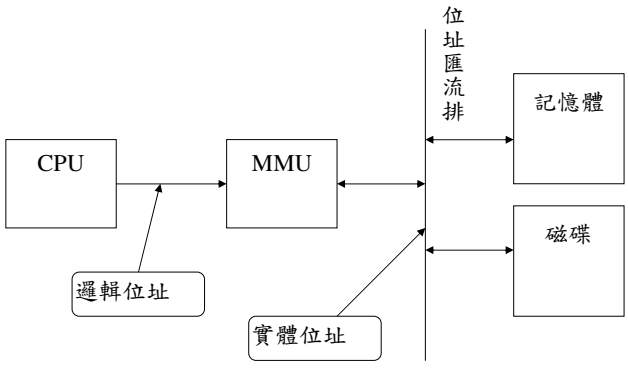
4

基本概念

- 虛擬記憶體的基本想法：
  - 僅把目前需要的部份程式載入到主記憶體
  - 其餘的則儲存在磁碟中，等到有需要時再載入
  - 程式設計師所能運用的記憶體容量，從原來的實體記憶體空間增加到整個磁碟的空間
  - 透過記憶體管理單元(MMU)的硬體支援，將邏輯位址轉換成實體位址；如果所要的資料不在實體記憶體中，會從磁碟中載入

5

虛擬記憶體下的位址轉換



6

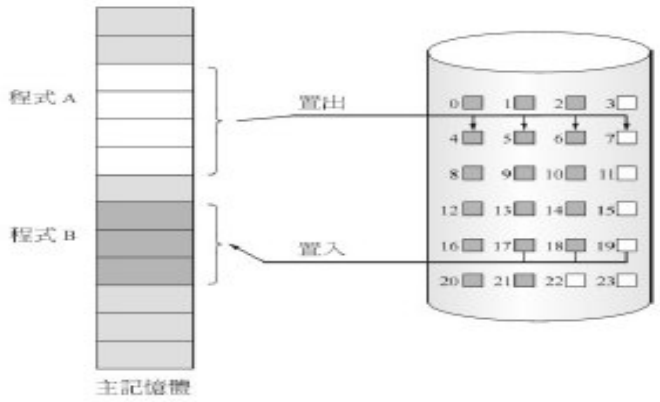
基本概念

- 虛擬記憶體的機制
  - 使用需求分頁、需求分段、分頁式分段的方式來實作
- 區域性(locality)
  - 時間區域性：最近執行過的指令不久會再地被行程執行，例如：迴圈、副程式、堆疊、計數
  - 空間區域性：執行過的指令，其附近的指令很快會被執行的機率相當大，例如：陣列、循序指令
- 區域性的觀點：執行時所參考到的同分頁中的指令會頻繁地被重複執行

7

需求分頁

- 需求分頁法就是使用置換法的分頁系統，行程存放在硬碟中。當要執行某個行程的時候，就把那個行程置換至記憶體。並不是把整個行程都置換進來，而是使用一個懶惰置換程式(lazy swapper)。懶惰置換程式只有當需要某一頁的時候才把該頁置換進來。



需求分頁

- 只將部份程式的分頁載入到記憶體中；只在行程需要執行某分頁時，才將此分頁載入到記憶體中
- 分頁表中的效用位元
  - 被設定為v：表示此分頁是有效(valid)的，且放在記憶體中
  - 被設定為i；表示此分頁可能不是有效(invalid)的
    - 此分頁沒有用
    - 此分頁是有效的但目前卻放在硬碟上

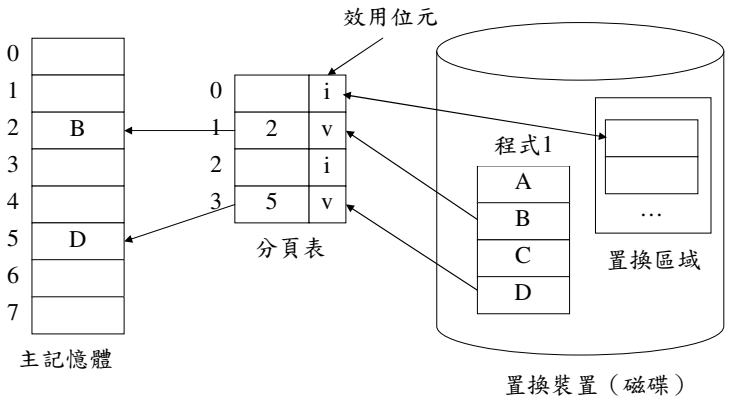
9

需求分頁

- 硬碟：儲存不在記憶體的分頁所用的置換裝置；為此目的而使用的磁碟區段稱為置換區域(較大的獨立連續區塊)
  - 所有行程被換出的分頁會被儲存在置換區域中
  - 在置換區域中進行需求分頁
  - 一次將行程所需要的分頁由置換區域中置入記憶體，不是一頁一頁地置入，以增進分頁效率

10

效用位元、置換區域與置換裝置



11

效用位元、置換區域與置換裝置

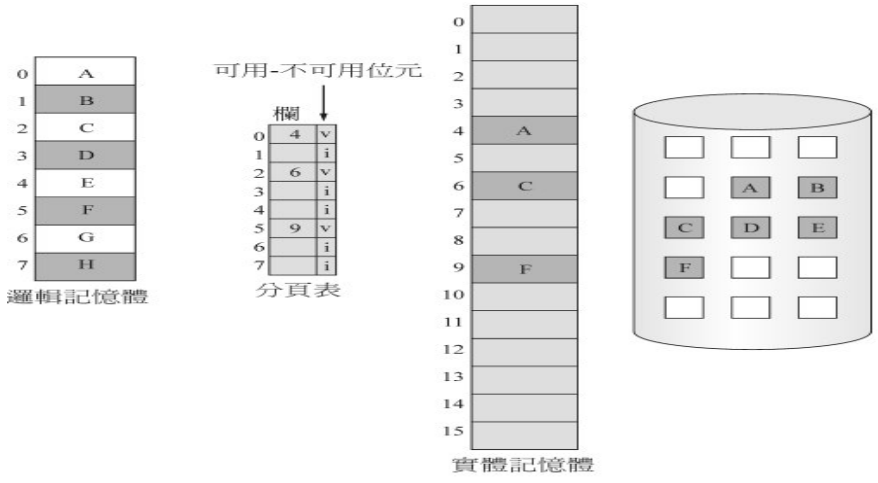


圖 9.5　當有些頁不在主記憶體中時的分頁表

需求分頁

- 分頁錯誤(page fault)：若一個行程想要使用一個不在記憶體中的分頁(存取一個標記為無效的分頁)
  - 作業系統產生例外中斷，此中斷之因是作業系統沒有把行程需要用的分頁載入到記憶體中，而不是企圖存取一個不合法的記憶體位址

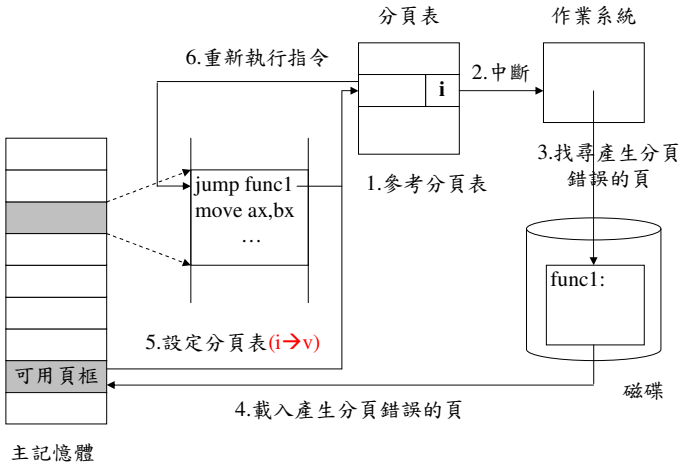
13

需求分頁

- 分頁錯誤處理的過程
  - 參考分頁表，決定是有效或是無效的記憶體參考
  - 若非法，則終止該行程執行；若合法，但此分頁尚未載入到記憶體中，則對作業系統發出分頁錯誤的例外中斷
  - 作業系統在硬碟中找尋引發分頁錯誤的分頁。
  - 在記憶體中找出一個頁框，並排定一個磁碟I/O，把引發分頁錯誤的分頁載入到該頁框中
  - 修改分頁表把此頁框設成有效(該分頁已經載入到記憶體中) i ➔ v
  - 重新執行引發分頁錯誤的指令

14

分頁錯誤處理步驟



15

分頁錯誤處理步驟

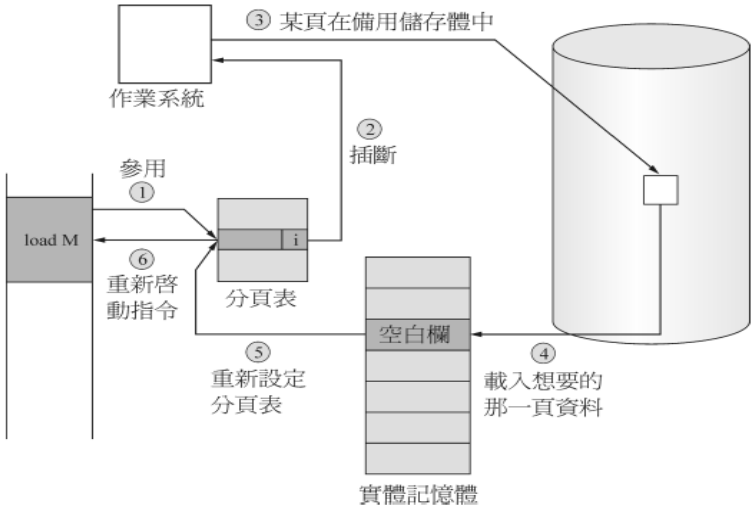


圖 9.6 處理分頁錯誤的步驟

16

需求分頁

- 分頁錯誤發生時，要儲存：
  - 當時被中斷的行程狀態，包含暫存器、狀態碼、程式計數器等
- 分頁錯誤處理完畢後，再把行程的狀態全都還原，並繼續執行此行程
- 執行行程的第一個指令時就會發生分頁錯誤，行程會不斷地發生分頁錯誤，直到所有目前需要的分頁都載入到記憶體為止，之後行程就可以順利執行，稱為純粹需求分頁
- 純粹需求分頁，就是在需要某分頁時才將那分頁載入執行

17

需求分頁

- 嚴格要求在分頁錯誤發生後，要能夠重新執行任何的指令；容易達到！
- 分頁錯誤發生在
  - 指令擷取的階段：藉由重新擷取指令來重新開始執行
  - 運算元擷取時：必須重新再擷取指令，並將指令解碼，然後擷取運算元
  - 儲存結果的階段時：要重新開始執行包含了指令擷取、運算元擷取、執行運算等步驟
- 主要的困難：一個指令執行時有可能會修改到很多不同位址的資料，導致無法重新執行指令

18

需求分頁效能

- 評估需求分頁的效能：計算有效存取時間
- 大多數的電腦系統中，記憶體存取時間都介於 10 奈秒與 200 奈秒間(t)
- 若無分頁錯誤，有效存取時間將會和記憶體存取時間相同；若發生分頁錯誤，將從磁碟中讀入相關的分頁→存取想要的位元組
- 假設分頁錯誤發生的機率為 p， p 介於 0 與 1 之間
  - 有效存取時間為(1–p)× t + p × 分頁錯誤處理時間
  - 如果 p 能夠相當地接近 0，有效存取時間可以非常接近記憶體存取時間(t)

19

需求分頁效能

- 分頁錯誤時系統所進行的處理：(94nttu 9, 97nttu 2(a))
  - 硬體對作業系統發出例外中斷
  - 儲存行程的狀態及一般暫存器內容
  - 作業系統判斷是否發生分頁錯誤
  - 作業系統檢查邏輯位址是否有效
    - 若沒問題，則找出所需要被載入記憶體的分頁；要求一個空的頁框
    - 假如沒有空的頁框，則利用分頁替換演算法決定要替換那一個分頁

20

需求分頁效能

- 若所選擇的頁框資料已被更改過，則發生內文切換，暫停發生分頁錯誤的行程，直到要置換出的分頁儲存回硬碟
- 若所選擇的分頁資料未經更改，或是資料已被寫回硬碟，作業系統會安排一個磁碟 I/O 將所需要的分頁載入
  - 在等待 I/O 完成時， CPU 先內文切換到別的行程
- 當分頁已被載入，分頁表更新，所對應的頁框也顯示有效狀態(i➔v)
- 發生分頁錯誤的行程等待重新拿回 CPU 使用權
- 還原暫存器、新的分頁表和其它變動的資訊，繼續執行此行程

21

需求分頁效能

- 處理分頁錯誤所花費的時間分為 3 部份
  1. 處理分頁錯誤所產生的中斷
  2. 讀取分頁
  3. 重新執行行程
- 第1和第3部份可藉由撰寫程式碼的技巧而降低
- 分頁的替換時間大致為 25 毫秒，一個典型的硬碟平均有8 毫秒的旋轉延遲時間、15毫秒的搜尋時間、與1毫秒的資料傳遞時間，因此總共的分頁時間接近 25 毫秒
  - 這包含硬體與軟體所需的時間

22

需求分頁效能

- 假設平均 25 毫秒的分頁錯誤處理時間及 100 奈秒記憶體存取時間(分頁錯誤發生率 p)
  - 有效存取時間=(1-p)×100+p × 25 毫秒
    =(1-p)×100+p × 25000000 = 100 + 24999900 × p
- 有效存取時間與分頁錯誤率成正比(96tpu 二3)
- 在需求分頁的系統中，降低分頁錯誤發生次數很重要
  - 不然有效存取時間的增加，會大幅延遲行程的處理時間

需求分頁效能

令一次分頁錯誤出現的機率爲 p $(0 \le p \le 1)$。我們當然希望 p 愈接近 0 愈好；也就是說只有一點點分頁錯誤出現。於是**有效存取時間** (effective access time) 就是

有效存取時間=$(1-p)\times ma+p\times$分頁錯誤的時間

如果我們花費了 8 毫秒的平均分頁錯誤處理時間以及 200 奈秒的記憶體存取時間，於是

有效存取時間=$(1-p)\times(200)+p$(8 毫秒)

需求分頁效能

$$= (1-p) \times 200 + p \times 8{,}000{,}000$$
$$= 200 + 7{,}999{,}800 \times p.$$

我們可以發現有效存取時間和**分頁錯誤的比率**（page-fault rate）乃是成正比的。如果一千次存取裏面有一次引起了分頁錯誤，那麼有效存取時間就等於 8.2 微秒。因爲需求分頁而造成電腦速度減緩的因數是 40。如果我們希望有小於百分之十的減緩。那就需要

$$220 > 200 + 7{,}999{,}800 \times p,$$
$$20 > 7{,}999{,}800 \times p,$$
$$p < 0.0000025.$$

--

考題

- Consider a two-level memory hierarchy M1 and M2. Denote the hit rate of M1 as h. Let C1 and C2 be the cost per kilobytes, S1 and S2 be the memory capacities, T1 and T2 be the access time
  (1) What is the effective memory access time of this hierarchy?
  (2) Let r=T2/T1 be the speed ratio of the two memories. Let E=T1/T2 be the access efficiency of the memory system. Explain E in term of r and h

26

分頁替換

- 行程在執行時若只需要載入行程所要使用的分頁
  - 程式多元度提高
    - 指系統同時有多個程式執行的程度
    - 程式多元度越高越好
  - CPU 使用率與產量提高
  - 記憶體頁框的使用率提高

27

分頁替換

- 發生分頁錯誤時，若記憶體中已經沒有空的頁框，解決方法如下：
  - 可終止發生分頁錯誤的行程
    - 不是一個好方法，因為此行程可能很重要不允許被終止
  - 可選擇將在記憶體中的某行程置換掉，將它所使用的頁框全部都空出來
    - 降低程式多元度
  - 利用分頁替換，換出其它的分頁以載入行程目前所需要的分頁

28

分頁替換

- 使用分頁替換，若無空頁框可使用，需在記憶體中選擇一個替換分頁將它寫入磁碟中→修改被替換的行程分頁表(由v→i)→把引起分頁錯誤的分頁載入到空出的頁框中→修改引起分頁錯誤的行程分頁表(由i →v)

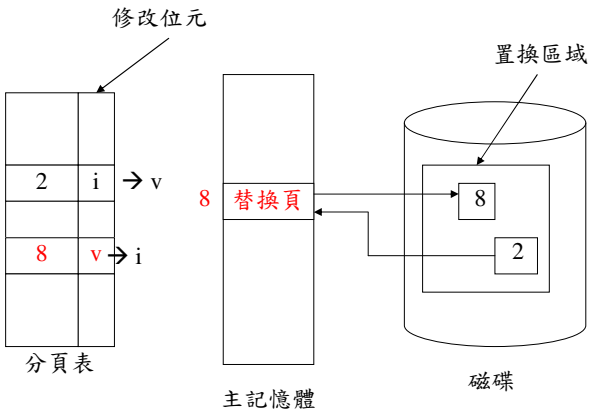
29

分頁替換

- 需要替換兩頁
  - 一從記憶體換出到磁碟，另一從磁碟置入記憶體中
- 磁碟I/O非常花時間，故利用修改位元來減少分頁的替換次數
  - 當有某個位元組被寫入到某分頁時，此分頁所屬的修改位元被硬體設定為i表示此分頁已被修改過；否則對應的修改位元為v，表示此分頁只進行讀取沒有任何寫入
  - 進行分頁替換時，會先選擇沒有被修改過的分頁進行替換，置入的分頁直接覆蓋該頁框即可，可以減少替換一個分頁的時間

30

分頁的替換



31

分頁替換

- 過度配置以下列方式呈現。當執行一個使用者行程的時候，出現了一個分頁錯誤。作業系統確定所要的那一頁存在磁碟什麼地方，卻發現在可用空白欄的表中已無空白欄了;所有記憶空間都已被使用。

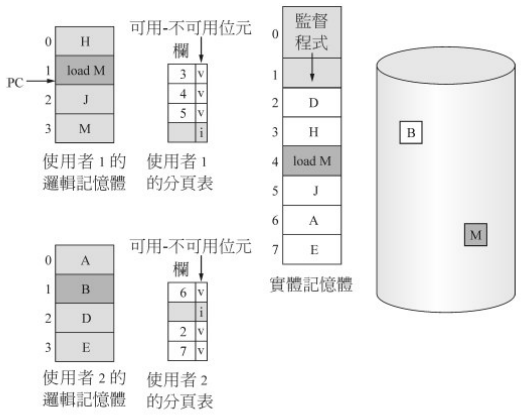


圖 9.9 需要做頁替換的時候

分頁替換

- 分頁替換的方法如下。如果沒有空白欄可用，就去找一個目前並未使用的欄把它空出來。要把某個欄空出來，可以把該欄的內涵寫入置換空間中，並且更改分頁表(以及所有相關的表格)以標示該頁已不在記憶體中。空出來的欄現在就可以用來存放引起分頁錯誤的那一頁。

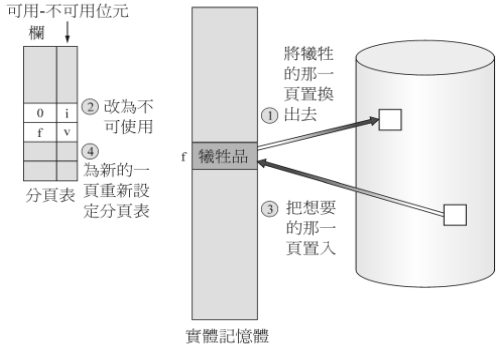


圖 9.10 頁替換

分頁替換

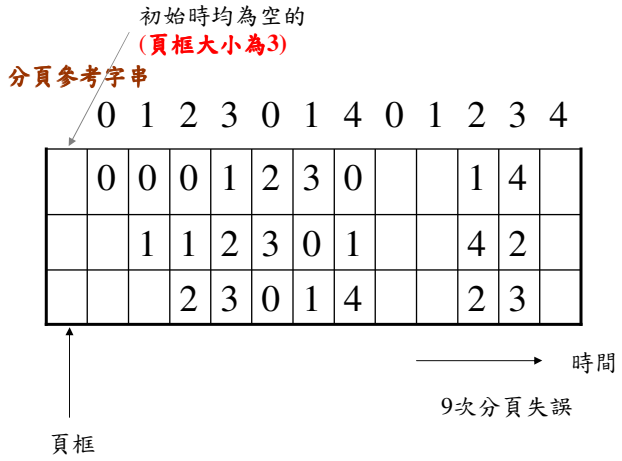
- 評估一個分頁替換法的好壞
  - 以分頁錯誤比率來當標準
  - 如果一個分頁替換演算法不好
    - 會增加分頁錯誤的次數➔ Disk I/O
    - 會降低程式執行的速度
    - 不會造成程式的執行結果不正確
  - 所有使用分頁的編號稱為參考字串

先進先出演算法(FIFO)

- 每次有新的分頁需置入時，會選擇置入記憶體時間最久的分頁換出(90nttu,95tku, 92tpu 4, 94tpu 2(b), 96tpu, 95ncu)
- 兩種實作的方法：
  - 記錄每個分頁被置入到頁框的時間，當每次需要換出分頁時，會找置入時間最早的一頁
  - 利用 FIFO 佇列來實作，當要進行分頁替換時，就把佇列最前端的分頁換出，再把要置入的分頁放到佇列的末端
- Belady 反常：配置的頁框數目增加，分頁錯誤次數反而增加(頁框為3增加至4)

35

先進先出分頁替換法



36

先進先出演算法(Belady 反常)

初始時均為空的

(頁框大小為4)

分頁參考字串

0	1	2	3	0	1	4	0	1	2	3	4
0	0	0	0			1	2	3	4	0	1
	1	1	1			2	3	4	0	1	2
		2	2			3	4	0	1	2	3
			3			4	0	1	2	3	4

時間

10次分頁失誤

頁框

37

先進先出分頁替換法

初始時均為空的

(頁框大小為3)

分頁參考字串

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	0		1	2	3	0	4	2			3	0			1	2	7
	0	0	1		2	3	0	4	2	3			0	1			2	7	0
		1	2		3	0	4	2	3	0			1	2			7	0	1

時間

15 次分頁失誤

頁框

38

考題

- Explain the following items as details as possible (94tku)
  Thrashing
  DMA
  RISC
  Belady's Anomaly
  RAID
- 分頁參考字串為：ABCDABEABCDE，當頁框大小為3,4,5時，採用FIFO會有何結果？
  9,10,5(FIFO), 7,6,5(Optimal), 10,8,5(LRU)

39

考題

- 分頁參考字串為：
  12345341678789789545 42，當頁框大小為3,4,5時，採用FIFO、Optimal、LRU、二次機會3(?),4(13),5(?)、LFU有何結果？

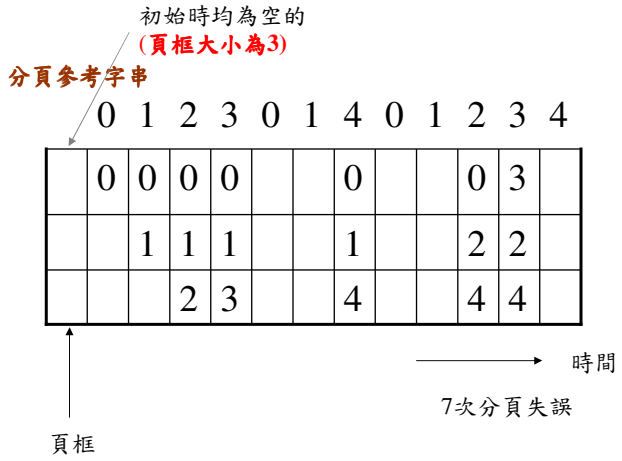
40

最佳演算法(Optimal)

- 分頁錯誤比率最低
- 替換未來最不可能被使用到的分頁
- 保證在頁框的數目固定之下，會得到最少的分頁錯誤次數
- 最佳頁替換演算法是所有演算法中分頁錯誤比率最低的一種。它永遠不會遭遇到Belady反常的問題。目前存在的最佳頁替換演算法稱為OPT或MIN。
- 實際上無法實作，因為對於未來將要使用的分頁不可能完全預測成功
- 可提供分頁替換演算法的比較基準

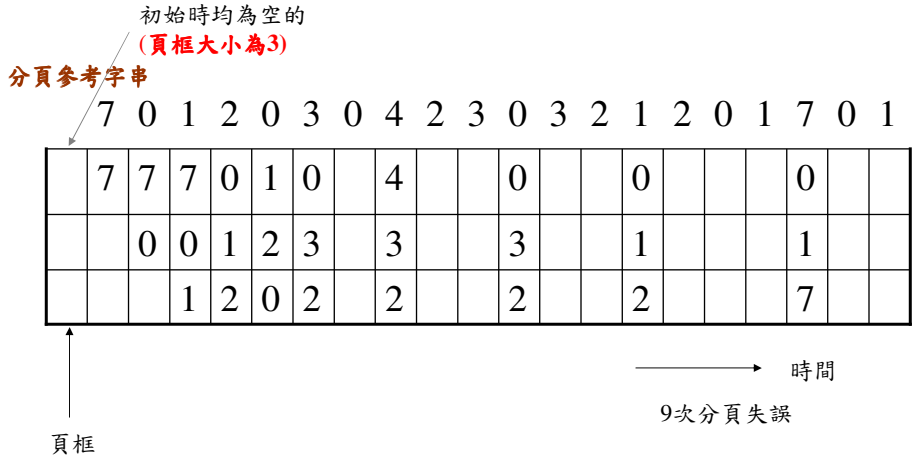
41

最佳演算法



42

最佳演算法



43

最久未用演算法(LRU)

- 近似最佳演算法：把頁框中最久未被使用到的分頁替換出去(96tpu 二2、97ncu 6、95ncu 1)
- 當記憶體中的某分頁被存取時，重新給予該分頁一個時間標記(不同於先進先出演算法)
- 對於行程的區域性而言，此方法最佳，但系統所花費的代價卻十分昂貴
  - 在記憶體中維護一個以最近的存取時間標記排序的鏈結串列
  - 每次行程使用過一分頁後，就必須對鏈結串列作一次更新，並可能要刪除其中一個分頁作替換，這是最花時間的動作

44

最久未用演算法

初始時均為空的

(頁框大小為3)

分頁參考字串

0 1 2 3 0 1 4 0 1 2 3 4

	0	0	0	1	2	3	0			0	1	2
		1	1	2	3	0	1			1	2	3
			2	3	0	1	4			2	3	4

時間

10次分頁失誤

頁框

最久未用演算法

初始時均為空的

(頁框大小為3)

分頁參考字串

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

	7	7	7	0		2		3	0	4	2			3		1		0		
		0	0	1		0		0	4	2	3			2		2		1		
			1	2		3		4	2	3	0			1		0		7		

時間

12次分頁失誤

頁框

最久未用演算法

- 其它方式實作，需要硬體支援
  - 在 CPU 中加入一個邏輯時鐘或計數器；在分頁表中的每個項目都增加時間欄位
  - 行程每使用一分頁後，就把 CPU 的邏輯時鐘加一，並將邏輯時鐘的數值寫入時間欄位
  - 要進行分頁替換時，系統檢查分頁表中所有項目的時間欄位值，找到最小的，並把此分頁替換掉
- 有了硬體的支援，可降低額外負擔

47

最久未用近似演算法

- 使用參考位元來達到近似最久未用演算法的效果
- 分頁表中的每個項目都加上一個參考位元，預設為 0；當某個分頁被行程使用，參考位元會被設定為 1
- 利用參考位元可知哪些分頁曾被行程使用過
  - 雖然無法得知這些分頁被行程使用的先後順序，卻是最簡單的最久未用近似演算法

48

最久未用近似演算法

- 其他的最久未用近似演算法：額外參考位元演算法
  - 記憶體中，每個頁框設置一組參考位元與移位暫存器，均初始化為 0
  - 每經過一次計時器中斷，行程會把控制權交給作業系統。若頁框中的分頁正被該行程使用，則將頁框的參考位元設定為 1；反之則設定為 0
  - 每隔一段時間發出中斷，作業系統將頁框的參考位元右移入移位暫存器，來比較其位元值0表示皆沒被使用

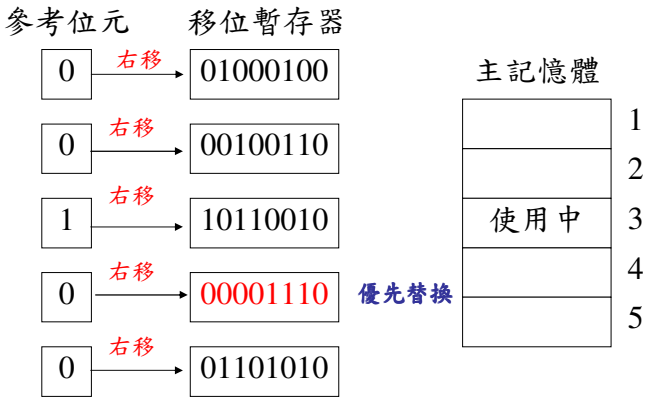
49

最久未用近似演算法

- 作業系統比較所有頁框移位暫存器值的大小，數值最小的分頁，就是最久未被行程所使用的分頁，便優先替換此頁框內的分頁
- 移位暫存器的數值可作為判斷分頁多常被行程使用的基準
  - 如果數值很大，表示此分頁常被行程使用
  - 可能有好幾個頁框移位暫存器的數值是相同的，它們是否要被替換，端看系統選用的方式 (可用FIFO方法)

50

額外參考位元演算法



51

額外的參考位元法則

- 藉著定期地記錄參考位元可以得到一些額外的次序資訊。在一個表中為每一頁保存一個8位元的內容。每經過一段時間(100msec)，一個計時器中斷就把控制權轉交給作業系統。作業系統把每一頁的參考位元移到其 8位元中較高次的位元上，把其它位元右移一位元，捨棄掉低次的位元。
- 用來記錄的位元數量是可以改變的，在選擇的時候當然是能促使更換作業愈快愈好。在最極端的情形下，可以減少至0個，只留下參考位元本身。這種演算法稱之為第二次機會頁替換演算法 (second-chance page-replacement algorithm)。

52

二次機會演算法

- 二次機會演算法的基本演算法是一種FIFO替換法。當某頁被選出來之後，檢視它的參考位元。如果參考位元為0，就進行替換這一頁的工作。如果參考位元是1，則將此分頁的參考位元設為0，給那一頁第二次機會，不馬上替換掉，且把此分頁的時間標記設為目前的時間，並且繼續用先進先出的方式，去選出下一頁。

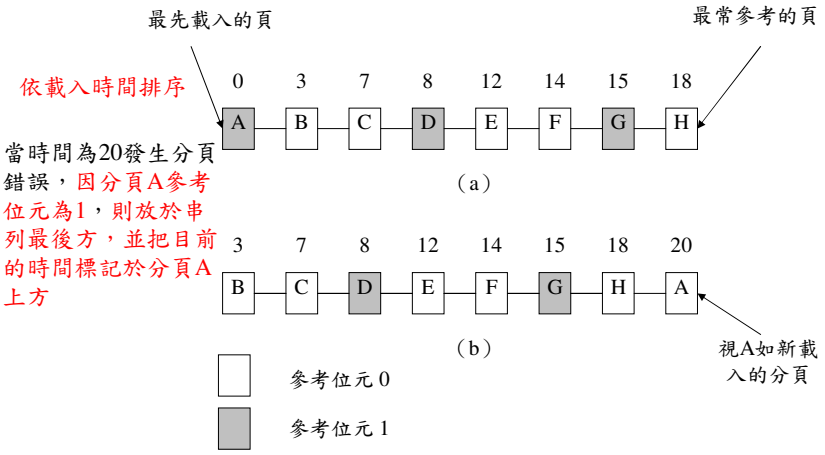
53

二次機會演算法

- 其他的最久未用近似演算法：移位暫存器的大小設定為 0 個位元(95tku 2)
  - 記憶體中每個頁框均對應到一個參考位元，其初始值為0；當某分頁被行程使用時，該參考位元會被設定為 1
  - 進行分頁替換時，以先進先出的方式找尋被替換分頁；若找到的頁框參考位元為 0，則進行替換；若參考位元為1，則將此分頁的參考位元設定為 0，給予此分頁第二次機會，不馬上將它替換；並把此分頁的時間標記設為目前的時間
  - 繼續用先進先出的方式找尋被替換分頁，直到所有其它的分頁都被替換掉，或是都被給予第二次機會之後，該分頁才會被替換掉
  - 利用鏈結串列較無效率，可換用環狀佇列方式

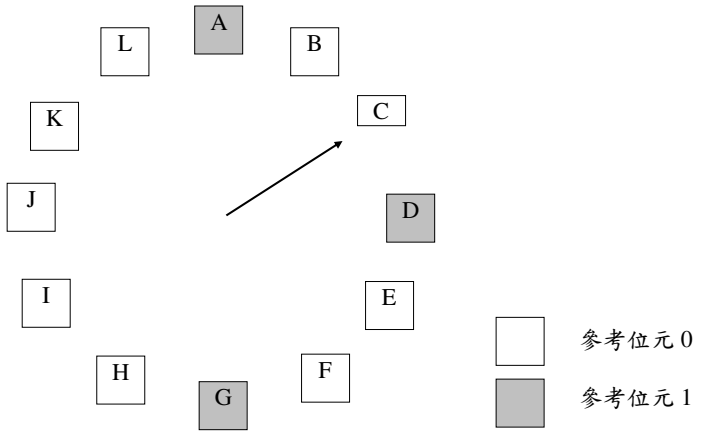
54

二次機會演算法－鏈結串列



55

二次機會演算法－環狀佇列



56

二次機會演算法

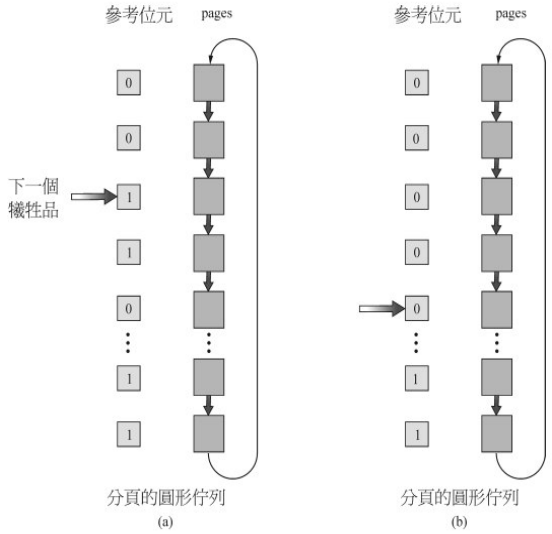


圖 9.17 第二次機會頁替換法

加強二次機會演算法

- 其他的最久未用近似演算法：
  - 和二次機會演算法相同，但增加了修改位元
  - 作業系統會每隔一段時間將分頁的參考位元設定為 0(表示因太久未再參考而視為新載入的分頁)
  - 此 2 位元有下列 4 種組合，依據順序進行替換(**參考,修改**)
    - (0, 0)：表示最近沒有被行程使用，也沒有被行程修改，是最佳的替換分頁(√)
    - (0, 1)：表示最近沒有被行程使用，但是已經被修改過，此分頁須先寫回磁碟後，才可進行替換(?)
    - (1, 0)：表示最近曾被行程使用，但是沒被修改過，由於可能再次被使用，故盡量不要替換此分頁(╳)
    - (1, 1)：表示最近曾被行程使用，也被修改過，所以需寫回磁碟中，是最差的替換分頁選擇(╳)

58

最不常用演算法(LFU)

- 考慮參考存取頻率
- 每分頁均使用一個計數器來計算被行程使用過的次數，初始值為 0 ；當某分頁被行程使用或修改時，對應的計數器便加 1
- 進行分頁替換時，找尋計數器值最小的分頁進行替換
- 若每分頁的計數器值相同，以先進先出的方式找尋被替換分頁

59

最不常用演算法(LFU)

- 問題1：某一分頁剛開始常常被行程使用，經過一段時間這分頁不再被使用，但因計數器的值已累加到很大，所以會被一直留在記憶體中
- 問題2：剛進來的分頁其計數器的值最小，所以會馬上被替換
- 解決方法：系統每經過一段時間，就將分頁的計數器數值向右移動一個位元(除以 2)，以降低所記錄的使用次數，讓某些時間很久為用的分頁有機會被替換

60

最不常用演算法(LFU)

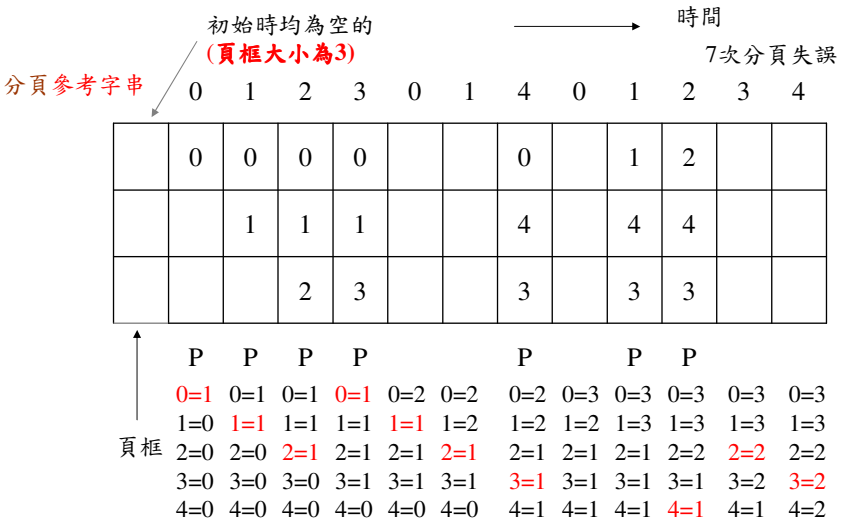
61

最常用演算法(MFU)

- 與最不常使用演算法相反：當要進行分頁替換時，找計數器值最大的分頁進行替換
  - 原因：在記憶體中使用很頻繁的分頁，行程可能已經執行完畢了，之後不會再被使用
  - 若每分頁的計數器值相同，以最近使用過之頁的方式找尋被替換分頁
  - 不一定是好方法，因為在記憶體中頻繁使用的分頁，通常都很重要，可能在不久的將來就會再次被使用

62

最常用演算法(MFU)



63

問題

- 分頁參考字串為：FABCADAECDADCBCABFAB，當頁框大小為3時，採用下列替換方法，分別會造成多少次頁失誤？1. FIFO 2. Optimal(9) 3. LRU 4. LFU(11) 5. MFU 6. 二次機會(12)
- 分頁參考字串為：DCBADCEDCBAE，當頁框大小為4時，又會有何結果？
- 分頁參考字串為：ABCDAABDEABCCDA，當頁框大小為4時，又會有何結果？

64

問題

- 給予的位址，為一長度為600 bytes的程式中會被參考到：
  12,37,128,180,76,209,135,246,248,520,436,448
  ，假設每一頁框為100 bytes，其頁參考字串為何？若記憶體僅有2個頁框(200bytes)，採用下列替換方法，分別會造成多少次頁中斷？
  A. FIFO　B. Optimal　C. LRU　D. LFU
  E. MFU　F. Second-chance

65

近來未用的頁替換(NRU)

- 近似LRU，但額外負擔小
- 最近未被使用到的頁，優先被替換掉
- 每頁附加兩個取用位元，且隨時更新
  - 參考位元：0表示未被使用，1表示已使用過
  - 修改位元：0表示未被修改，1表示已使修改

66

近來未用的頁替換(NRU)

- 策略：要替換頁時，以一個未被使用過的頁為優先考慮，若全部都使用過，則以未被修改過的頁優先考慮，若全被修改過，則選被修改過時間最久的頁為優先
- 當被選中的頁已被修改過，則必須重新寫回硬碟，否則即可直接覆蓋此頁。
- 採用此法時，必須週期性的把取用位元重新設定為0，否則會喪失應用的功能(94tpu 3)

67

分頁緩衝演算法

- 分頁替換的步驟：
  - 把被替換的分頁移出記憶體並寫回磁碟裡
  - 將要載入的分頁，由磁碟置入到記憶體中
  - 執行該分頁的程式或是存取資料
- 必須先等待替換分頁寫回磁碟後，才能進行後續的動作(Linux之sync、DB之refresh指令)
- 將分頁寫回磁碟的動作並不緊急，所以可以等到系統有空的時候再做

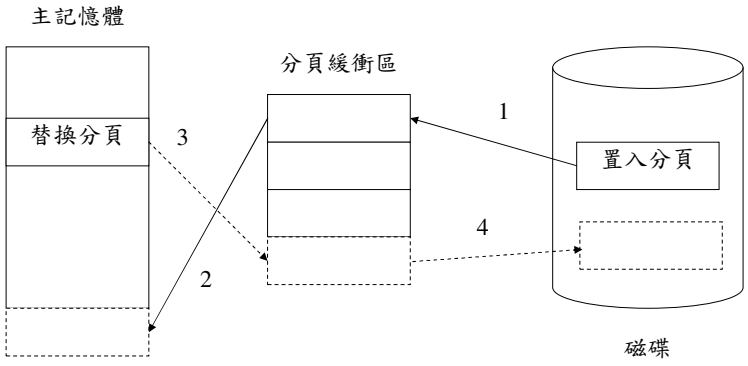
68

分頁緩衝演算法

- 系統內除了主記憶體的頁框外，還要有幾個可用的頁框當作分頁緩衝區
- 分頁錯誤發生時
  - 先將要載入到記憶體的分頁，由磁碟載入到分頁緩衝區中(1)
  - 將緩衝區中的該分頁併入到主記憶體中，然後執行此分頁的程式或是存取資料(2)
  - 再從主記憶體中挑選被替換分頁，將它寫回到磁碟中(3)
  - 再把空出來的頁框併入到分頁緩衝區中(4)
- 行程能很快地繼續執行

69

分頁緩衝區演算法



70

頁框配置

- 多行程的記憶體環境中，重要的事是要決定一個行程能夠配置多少個頁框
  - 配置過多造成空間上的浪費，當行程更多，將會找不到可用的頁框來置入行程的分頁
  - 配置過少會造成行程頻繁地發生分頁錯誤
- 好的頁框配置演算法能增進系統效能

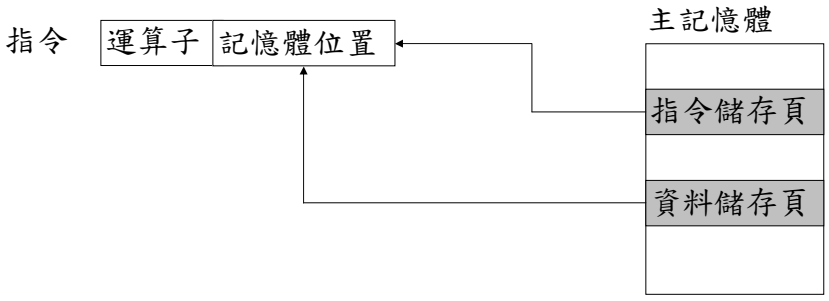
71

最少頁框數目

- 行程能正常執行所需的最少頁框數目與 CPU 結構和指令架構有關
- 指令架構：
  - 直接定址：需要一個放置頁框儲存指令的分頁，另一個存放頁框儲存資料所在的分頁→每個行程至少要配置 2 個頁框
  - 間接定址：需要一個存放頁框儲存指令所在的分頁、一個存放頁框儲存資料位址的分頁、還有一個放置頁框儲存資料的分頁→每個行程至少需要 3 個頁框
    - 若只分配兩個頁框，每當 CPU 處理間接定址指令時，就會發生分頁錯誤，因而降低系統效率

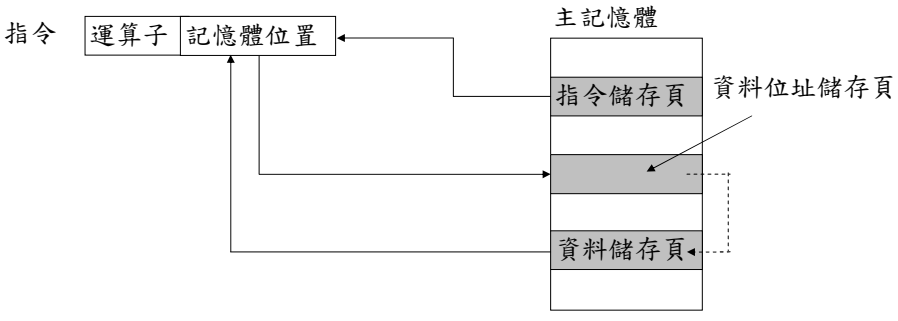
72

直接定址



73

間接定址



74

最少頁框數目

- CPU 結構：有多個一般暫存器(4 AX,BX,CX,DX) 的架構下，可以把指令分成
  - 含有 2 個運算元的系統：a=a+b
    - 假如每一個運算元都使用間接定址的方式
    - 一個運算元會需要兩個頁框，而指令需要一個頁框，因此每個行程最少需要 1+2*2=5 個頁框
  - 含有 3 個運算元的系統：若使用間接定址，每個行程最少需要1+2*3=7個頁框 (c=a+b)
- 最大的頁框數目，可由實體記憶體的容量來決定
- 最少頁框數目可由CPU 結構和指令架構有關

75

二運算元指令與三運算元指令

運算子	運算元 1	運算元 2

運算子	運算元 1	運算元 2	運算元 3

76

配置演算法

- 若系統中有 m 個頁框要分配給 n 個行程：
  - 平均配置的方法：每個行程會平均分配記憶體中的所有頁框，所以每個行程可以分配到 m / n 個頁框(485個頁框分給10個行程➔每個48頁框)
    - 不公平，因行程大或小分配皆相同，行程大分頁錯誤發生的次數會增加
  - 比例配置的方法：假設行程 P_i 的大小為 S_i，系統中可用的頁框數目為 m，則 P_i 可分配到 $(S_i/S) \times m$ 個頁框，其中 S 為所有行程大小的總和(485個頁框分給2個行程300k、30k➔ 300/330*485=441頁框)

77

配置演算法

- 兩種配置方法下，分配給每個行程的頁框數目都會因為程式多元度而改變
  - 程式多元度提高，每個行程需釋放一些頁框給新的行程使用；程式多元度降低，將離開行程的頁框會分配給仍在執行的行程

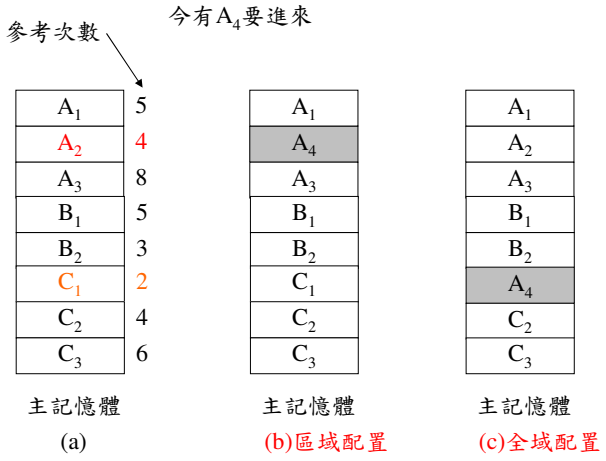
78

全域與區域配置

- 全域配置：當一個行程要進行分頁替換，可從記憶體中所有的頁框中挑選被替換分頁(一個行程可以從其它行程獲得一個頁框)
- 區域配置：每個行程所使用的頁框數不會改變；進行分頁替換時，由該行程所擁有的頁框中挑選出被替換分頁
- 全域配置的方式較好
  - 若使用區域配置，當一個行程需要額外的頁框載入分頁，但由於配置固定頁框數目，此行程仍會頻繁地發生分頁錯誤
  - 若使用全域配置，可以增加許多被替換分頁的選擇

79

全域與區域配置



80

輾轉現象(Thrashing)的成因

- 行程所使用到的分頁比分配到的還多，必定不停地把之後會使用到的分頁換出，並隨後再立刻置入
  - 造成分頁在記憶體與磁碟中來回搬動做虛工(94tku,96tpu三)
- 當 CPU 使用率低時，CPU 排程器為了增加 CPU 的使用率，會提高程式多元度(在輸入佇列中選擇一個行程載入)(93ncu)
  - 因為此新的行程需要使用到許多頁框，系統若採用全域配置頁框，此行程可能會搶其他行程所使用的頁框，
  - 但若其它行程在執行時也需要這些被換出的分頁，又發生分頁錯誤，造成產生分頁錯誤的行程都在等待分頁裝置將它們所需要的分頁置入→更造成 CPU 使用率降低

81

輾轉現象的成因

- 當 CPU 排程器又發現 CPU 使用率降低，再載入一個等待執行的行程，此新的行程又會從其他行程中搶頁框，造成分頁錯誤的現象更加頻繁，使得 CPU 使用率降更低
- 此現象不斷發生，會使系統效率極低
  - 行程所有的時間都花在分頁置換上面

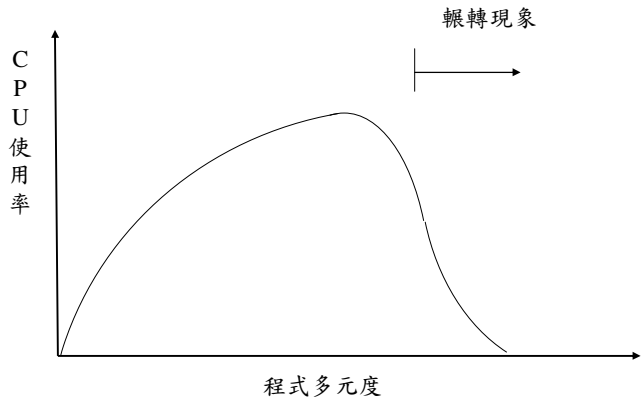
82

輾轉現象的成因

- 當程式多元度增多，CPU 使用率將成長；之後，成長的幅度會趨緩；如果程式多元度繼續增加，系統會發生輾轉現象，使得 CPU 使用率快速降低
- 為了解除輾轉現象，系統必須減少程式多元度，使剩餘的行程有足夠頁框可用
  - 減輕分頁錯誤現象；再度提高 CPU使用率

83

輾轉現象



84

輾轉現象的成因

- 系統要分配給每個行程多少頁框，才不會發生輾轉現象呢？行程執行有其區域性
  - 至少要有足夠的頁框可供目前區域性所涵蓋的分頁使用
  - 減少行程發生分頁錯誤的機會
- 只要發現，增加行程的數目仍無助於提高CPU使用率，就可能發生輾轉現象

85

工作集合模型

- 隨著行程的執行，區域性的改變，又會使分頁錯誤增加(90tku,93tpu 7,92tpu 5,96tpu三,94tpu 2(j))
- 工作集合：某段時間內行程使用過分頁的集合
  - 藉由選擇適當工作集合，可代表程式執行的區域性
  - 當工作集合中所有的分頁都置入記憶體後，就不會發生分頁錯誤，直到行程執行的區域性改變為止
- 區域性(locality)
  - 時間區域性：最近執行過的指令不久會再地被行程執行，例如：迴圈、副程式、堆疊、計數
  - 空間區域性：執行過的指令，其附近的指令很快會被執行的機率相當大，例如：陣列、循序指令

86

工作集合模型

- 選擇適當的工作集合不容易
  - 若選擇的工作集合太小：無法代表行程執行的區域性
  - 若選擇太大的工作集合，可能會橫跨數個局部區域
- 工作集合是為了使CPU 執行時更有效率，而必須放在主記憶體中的分頁所形成的集合
- 在多元程式的環境下，許多分頁系統會記錄每個行程的工作集合，在行程重新執行前將所有工作集合中的分頁都置入記憶體中
  - 可減輕分頁錯誤的發生
  - 預先分頁：在行程尚未執行前先將分頁置入記憶體中

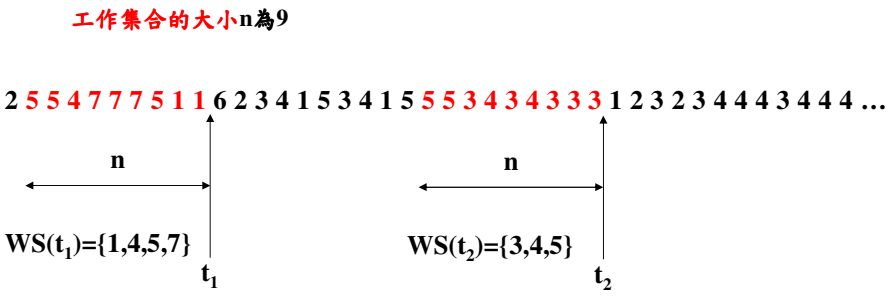
87

工作集合模型

- 實作：作業系統記錄工作集合中存在哪些分頁
  - 定義工作集合的大小為 n
  - 利用老化演算法決定工作集合中的分頁，將太久沒用的分頁移出工作集合
- 做法：行程中每一分頁會對應到一個計數器，計數器中的 n 個位元由高而低分別代表最近 n 次行程對本分頁的使用與否
  - 設定為 1 代表被行程使用，0 表示沒有被使用
- 須適當選用參數 n
  - n 太小：工作集合無法代表行程的區域性
  - n 太大：會橫跨許多行程使用的區域，無法確定行程目前在哪一個局部區域中執行

88

工作集合模式



89

工作集合模型

- 作業系統監督每個行程的工作集合(98tku3,4)
  - 若能給予每個行程足夠的頁框，則不會發生分頁置換的問題
  - 若系統內所有行程的所有工作集合所需頁框的總和超過記憶體所能提供就發生輾轉現象
  - 解決方式：給予行程更多的頁框數目；或降低程式多元度
- 工作集合的做法可以有效預防輾轉現象；也盡可能維持較高的程式多元度，使 CPU 有較高的使用率

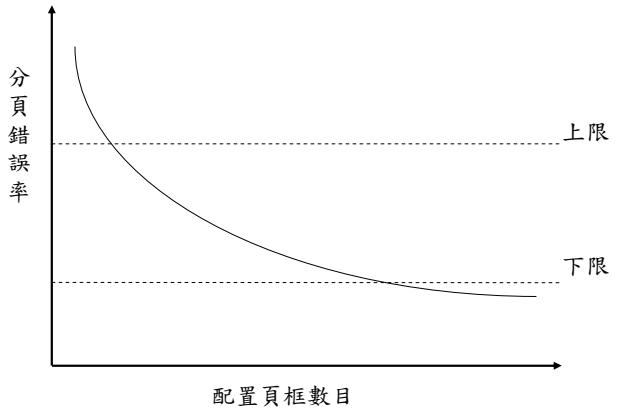
90

分頁錯誤頻率

- 藉由控制系統中的分頁錯誤頻率以避免發生輾轉現象
- 作業系統中先定義分頁錯誤頻率的上限與下限
  - 當某行程的分頁錯誤頻率大於系統定義上限，表示此行程所需頁框數目不足，必須再配置
  - 若分頁錯誤頻率比下限還低，表示此行程擁有過多的頁框，系統可以收回未使用頁框
- 若行程的分頁錯誤頻率增加但系統中已無空頁框可使用
  - 可暫停部份行程，將其頁框收回並配置給其餘行程使用

91

分頁錯誤頻率



92

預先分頁

- 在純需求分頁系統中的一項明顯特徵就是當一個行程開始執行的時候就會出現一大堆分頁錯誤。這個情況乃是因為嘗試要將起始局部區域，載入記憶體所造成的。相同的事情可能在其它時間發生。當一個被置換出去的行程想要重新啟動的時候，它所有的頁都存在備用儲存體中並且必須藉著各個分頁錯誤才能把所有的頁都載入記憶體。
- 預先分頁(prepaging)就是想要防止這種高度的起始分頁錯誤。

93

預先分頁

- 將行程所需的分頁，在實際使用前先置入記憶體，以降低分頁錯誤的發生
- 可能預先分頁處理的代價高於處理分頁錯誤，例：許多被預先載入記憶體的分頁最後並沒有被使用
- 謹慎地設計程式與資料結構，可增加行程局部區域性，並降低分頁錯誤的次數
- 分頁大小對分頁錯誤的影響也相當大

94

一個 3 x 2 陣列(93nttu 9)

M：array[0 … 2，0 … 1] of integer

B[0][0]	B[0][1]
B[1][0]	B[1][1]
B[2][0]	B[2][1]

M[0][0]	M[0][1]	M[1][0]	M[1][1]	M[2][0]	M[2][1]

(a) 列優先

M[0][0]	M[1][0]	M[2][0]	M[0][1]	M[1][1]	M[2][1]

(b) 行優先

95

練習

- Suppose that an array X(base address is 1000) in C language as followed: int x[7][9];
  Assume that each integer occupies 4 bytes. What is the address of the array element x[5][7]
  a. if the row-major representation is used?
  b. if the column-major representation is used?
  c. derive a deneral formula for computing the address of an integer array element A[i1][i2]..[in] of array A[r1][r2]…[rn]

96

練習

- Suppose A is a two dimension array, if A(2,3) is in location 1756 and A(3,3) is in the location 1760. Assume that each integer occupies 1 byte. What is the location of A(4,4)？(92,96tku)
- 一陣列(Array)每個陣列元素佔用4個位元組的記憶體，若A(3,4)的起始位置是1640，A(4,4)的起始位置是1680，A(5,5)存放位置為何？
- 陣列a宣告為 a[10][20][30]，若 a[1][1][1]為其中第一個元素，a[1][1][2]為第二個元素，則a[5][5][5]為第幾個元素？(A) 2524 (B) 2525 (C) 3154 (D) 3155

97

程式結構

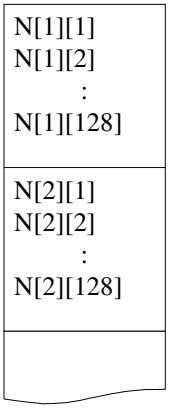
- 分頁大小為256位元組，一個列優先陣列N大小128*128，每一整數大小為2位元組，陣列N[1][1]…N[1][128]在同一分頁，N[2][1]…N[2][128]在同一分頁…，程式結構如後
- 當程式碼的內迴圈每執行一次N[i][j]=0，就會發生一次分頁錯誤，所以總共發生128*128次分頁錯誤
- 若內迴圈改為N[j][i]=0，內迴圈就只會發生一次分頁錯誤，所以總共發生128次分頁錯誤

98

程式結構

```
N：[1 … 128,1 … 128] of integer
   for j= 1 to 128
            for i = 1 128
                     N[i][j] = 0;
```

列優先　每頁 256 位元



99

程式結構

- 分頁大小為128位元組，當程式碼的內迴圈每執行一次N[i][j]=0，就會發生一次分頁錯誤，所以總共發生128*128次分頁錯誤
- 分頁大小為128位元組，若內迴圈改為N[j][i]=0，內迴圈就只會發生兩次分頁錯誤，所以總共發生256次分頁錯誤，比之前的分頁大小為256位元組發生128次分頁錯誤增加了一倍。

100

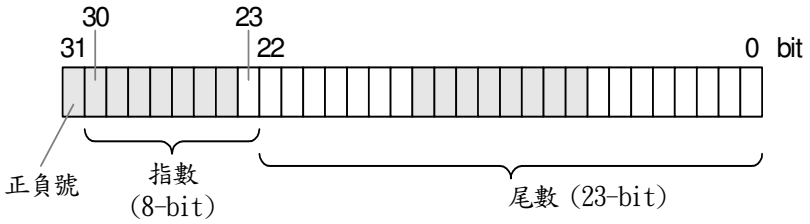
浮點數IEEE 754 (91,92,94tku)

- **Float 資料型別(4 bytes)**
- **Values:** You can enter the words "Infinity", "-Infinity" or "NaN" to get the corresponding special values for IEEE-754. Please note there are two kinds of zero: +0 and -0.

101

浮點數IEEE 754

- **Float 資料型別(4 bytes)**
- The sign is stored in bit 31. The exponent can be computed from bits 23-30 by subtracting 127.



102

浮點數IEEE 754

- The mantissa (also known as significand or fraction) is stored in bits 0-22. An invisible leading bit (i.e. it is not actually stored) with value 1.0 is placed in front, then bit 22 has a value of 1/2, bit 21 has value 1/4 etc. As a result, the mantissa has a value between 1.0 and 2. If the exponent reaches -127 (binary 00000000), the leading 1 is no longer used to enable gradual underflow.

103

浮點數IEEE 754

- 0xC0A00000 ➔ -5.0

http://www.h-schmidt.net/FloatApplet/IEEE754.html

- 18.5 ➔ ??(10010.1_2= $1.00101_2*2^{指數}$= 1.00101_2*2^4) 127+4=131=10000011_2=83_{16}，mantissa為00101，所以為01000001100101 $0\ldots0_2$=0X41940000
- -100.0 ➔ ??(1100100_2= $1.100100_2*2^{指數}$ = 1.100100_2*2^6) 127+6=133=10000101_2=85_{16}，mantissa為100100，所以為11000010110010 $0\ldots0_2$=0XC2C80000

104

浮點數IEEE 754

- float數值表示方式的四個特列：
  - 0：無法使用2的次方表示，使用所有bits皆為0表示。
  - POSITIVE_INFINITY：以所有exponential-bits為1，其餘bits為0表示。
  - NEGATIVE_INFINITY：以sign-bit為1，所有exponential-bits為1，所有mantissa-bits為0表示。
  - NaN：以sign-bit為0，所有exponential-bits為1，mantissa-bits中左側第一個為1其餘為0表示。

105

浮點數IEEE 754

- Infinity為0x7F800000(exponent bits are all 1)
- -Infinity為0xFF800000(exponent bits are all 1)
- NaN為0x7FC00000(exponent bits are all 1，mantissa為1 0…0)

- Infinity為0x7FF0000000000000
- -Infinity為0xFFF0000000000000
- NaN為0x7FF8000000000000
- 21.125??

106

3. IEEE 754 浮點數單倍精確度內部儲存格式如下(佔 32 位元) --

符號	指 數	小 數	位元
31	30 ... 23	22 ... 0	

base = 2

bit 31 -- 小數的符號

bits 30 -- 23 -- 指數(excess-127)，最小只能存 1，最大只能存 254，存入 255 表示無窮大，NaN...

bits 22 -- 0 -- 小數 mantissa(with hidden 1.)

請回答下列問題：(共 20%)

(a) 將 21.125 轉換為 IEEE 754 單倍精確度內部儲存格式。(7%) (答案用 16 進位表示)

(b) 除無窮大以外，單倍精確度能夠儲存的最大數值內部儲存格式為何？(3%)(答案先用 2 進位寫出，再寫成 16 進位)

(c) 第 2 大的數值內部儲存格式為何？ (3%)(答案先用 2 進位寫出，再寫成 16 進位)

(d) 計算

最大的數值 - 第 2 大的數值 =？ (4%)(答案用 10 進位表示)

(e) 除 0 以外，單倍精確度能夠儲存的最小數值內部儲存格式為何？ (3%)(答案先用 2 進位寫出，再寫成 16 進位)

10. Given a real number -1756375E-3 (in C language format), convert it to the floating-point notation of the IEEE Standard 754 using 32-bit single-precision format. Show each step in detail and write the final result in hexadecimal. (10 pts)

7 The IEEE standard double-precision floating-point operand format consists of 64 bits. The sign occupies 1 bit, the exponent has 11 bits, and the fraction occupies 52 bits. The exponent bias is 1,023 and the base is 2. There is an implied bit to the left of the binary point in the fraction. Infinity is represented with a biased exponent equal to 2,047 and a fraction of 0.

(a) Give the formula for finding the decimal value of a normalized number.

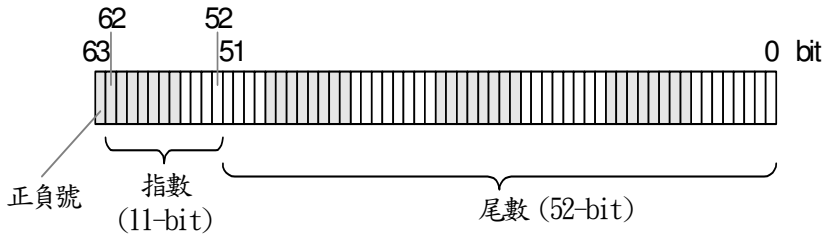
(b) Calculate the largest and smallest positive normalized numbers that can be accommodated.

7. (a) What is the IEEE 754 representation of 32-bit integer binary format of a signed decimal number, -300? (7%)

(b) What is the IEEE 754 representation of 32 bits floating point binary format of a signed decimal number, -29.75? (8%)

浮點數IEEE 754

- **Double 資料型別(8 bytes)**
- The sign is stored in bit 63. The exponent can be computed from bits 62-52 by subtracting 1023.



109

浮點數IEEE 754

- 10.5 ➔ ??($1010.1_2 = 1.0101_2 * 2^{指數} = 1.0101_2 * 2^3$) 1023+3=1026=$10000000010_2$=$402_{16}$，mantissa 為0101，所以為0100000000100101 0…0_2=0X4025000000000000
- -5.0 ➔ 0XC014000000000000
- -100.0 ➔ ??
- 18.5 ➔ 0X4032800000000000

110

浮點數IEEE 754

- **Rounding errors:** Not every decimal number can be expressed exactly as a floating point number. This can be seen when entering "0.1" and examining its binary representation which is either slightly smaller or larger, depending on the last bit.

111

變型題

- 某電腦的浮點數值表示法是由8個位元所組成，$b_7b_6b_5b_4b_3b_2b_1b_0$，其中b_7為正負號S，$b_6b_5b_4$為指數E，其餘為mantissa (M)，其數值公式為$(-1)^s*1.M*2^{(E-4)}$，一實數為4.75，其表示法十六進位為何？
  A. 43 B. 76 C. 56 D. 63
- 6.5 ➔ 0X6A

112

分頁伺服精靈(Page Daemon)

- 在頁框數目足夠的狀況下，分頁法會有較好效果
  - 當分頁錯誤發生時，只要直接將所要求的分頁置入所分配頁框即可
  - 可省下將換出分頁寫回磁碟的時間
- 分頁伺服精靈：每隔一段時間檢查記憶體的狀態
  - 若可用的頁框太少，伺服精靈會利用分頁替換演算法選擇一個分頁，並在系統較空閒時置換掉；被選中的分頁如果被修改過，才會被存回磁碟
  - 若某分頁行程需要再度使用該分頁，且原本記憶體中資料沒被修改，分頁伺服精靈會直接取得此分頁，不必重新從磁碟中載入
  - 可確保系統中有足夠空的頁框可用；保持記憶體中頁框供應無虞

113

分頁上鎖

- 有些狀況必須讓分頁被鎖在記憶體中，不進行置換
  - 例：當某行程由硬碟讀取檔案，或等待資料讀入而作系統呼叫
    - 該行程會被放到 I/O 等待佇列中
    - 系統切換到另一個行程執行
  - 如果新執行的行程發生分頁錯誤，且使用全域配置頁框法，就有可能將等待 I/O 的行程分頁置換掉，會發生問題➔等待I/O 的行程已被置換出去，DMA會將資料寫到錯誤的分頁

114

分頁上鎖

- 解決方法：
  - 允許分頁能夠被鎖在記憶體中，不會被置換
  - 所有 I/O 的動作都必須先寫入系統緩衝區，再由系統將資料搬移到使用者分頁

115

分頁大小

- 決定最佳分頁的大小，需考慮：
  - 內部斷裂的情形：
    - 每一個行程平均浪費半頁的空間；使用較小的分頁，會減少記憶體的浪費
  - 區域性：使用較小分頁可以精確表現行程的區域性
- 使用較大的分頁：
  - 分頁的數目減少使得分頁表較小；節省空間也節省搜尋分頁表的時間
  - 可涵蓋較多的指令與資料，降低分頁錯誤的發生

116

分頁大小

- 從 I/O 的角度，要載入相同的資料，只需載入較少數量的分頁，所花費的時間比較小分頁所花費的時間少
- 進行內文切換時，只需切換較小的分頁表

- 使用較大分頁較好：因為現在 CPU 執行速度很快；記憶體容量大又便宜，存取的速度也很快，可提高系統的效能

117

其他考量

- 即時處理方面：
  - 需要即時處理的行程，取得 CPU 執行權後，得在行程的時間限制內執行完畢
  - 由於虛擬記憶體在行程執行的過程中，必須等待某些分頁置換入記憶體，造成不可測的時間延遲
  - 因此即時系統中幾乎不使用虛擬記憶體

118

其他考量

- 共用分頁方面：
  - 大型多元程式的系統中，許多使用者在同時間內執行同一個程式，共用同一個分頁比在系統中載入許多相同的分頁更有效率
  - 並不是所有的分頁都可以共用：包含程式碼的分頁可共用，存放資料的分頁不能共用
- 置換共用分頁時(MDI/SDI)
  - 系統必須確定已經沒有任何行程在使用此共用分頁，否則行程將發生分頁錯誤
- 需要特殊的資料結構來追蹤共用分頁

119

摘要

- 執行一個記憶體需求大於實體位址空間的行程有幾種方式：
  - 重疊法，程式寫作比較困難
  - 利用虛擬記憶體的機制，使得行程的邏輯位址空間可以大於實體位址空間；增加程式多元度，提高CPU使用率
- 需求分頁
  - 以分頁的方式實作虛擬記憶體
  - 在行程需要某分頁的時候，才將該分頁從磁碟中載入
  - 該分頁的第一次使用會發生分頁錯誤
  - 虛擬記憶體也可以使用需求分頁或分頁式分段來實作。

120

摘要

- 分頁的替換：當記憶體中所有的頁框都在使用時，方法如下
  - 先進先出演算法，設計簡單，會有Belady的反常現象
  - 最佳演算法，實際上不可能實作
  - 最久未用演算法，近似於最佳演算法，需要硬體支援才有效率，實作上有困難
  - 二次機會演算法、額外參考位元演算法、加強二次機會演算法，做法上近似最久未用演算法

121

摘要

- 頁框配置的策略
  - 使用靜態配置
    - 平均配置法是每個行程都分配到一樣多的頁框
    - 比例配置法是依照行程的大小分配每個行程的可用頁框
  - 區域配置法是在同一個行程所分配到的頁框中選擇替換的對象
  - 全域配置法是一種動態配置法，以系統中所有的頁框作為選擇替換的對象

122

摘要

- 輾轉現象
  - 造成系統一直進行分頁的置換動作
  - CPU 的使用率降低，行程無法進展
  - 避免輾轉現象：實作工作集合模型或是根據分頁錯誤頻率的統計
- 實作需求分頁的系統
  - 最重要是選擇一個合適的分頁替換演算法與配置頁框的策略
  - 其它諸多因素需考慮，如預先分頁、程式結構、分頁伺服精靈、分頁上鎖、與分頁大小等。

123