記憶體管理

資科系
林偉川

目標與趨勢

- 目標
  - 范紐曼電腦架構➔將程式載入記憶體才能執行
  - 追蹤記憶體空間使用與否
  - 配置記憶體給需要的行程，可能超出實際記憶體
  - 動態配置記憶體，回收行程釋放出的記憶體
  - 記憶體不夠時，有效率的置換(swapping)方法
- 趨勢
  - 程式成長的速度快於記憶體成長的速度
  - 多媒體應用環境，使用更多的記憶體

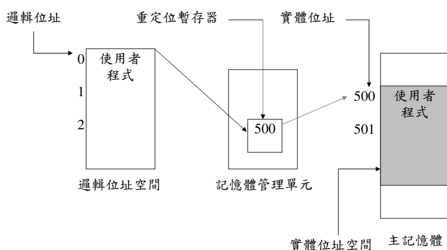
2

位址空間

- 記憶體位址
  - 邏輯位址，邏輯位址空間(執行檔中記憶體位址)
  - 實體位址，實體位址空間(記憶體實際位址)
- 執行程式時，邏輯與實體空間的位址轉換
  - 載入器(loader)：在主記憶體中找一塊可供使用的記憶體來載入程式
  - 基底暫存器(base register) 又名重定址暫存器，存放邏輯位址轉換成實體位址的基底值
  - 記憶體管理單元(memory management unit,MMU)：負責將邏輯位址加上基底值，以轉換成實體位址

3

邏輯位址空間轉換到實體位址空間



4

邏輯位址空間和實體位址空間

- CPU所產生的位址通常稱為邏輯位址(logical address)，而記憶體單元所看到的位址(也就是載入到記憶體的記憶體位址暫存器(memory-address register)之數值)通常叫做實體位址(Physical address)。

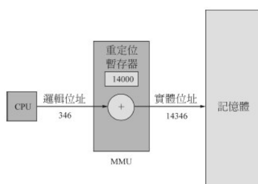


圖 8.4 使用一個重新定位暫存器來動態地重新定位

5

邏輯位址空間和實體位址空間

- 動態載入
  - 行程大小受限於實體記憶體大小。要得到較佳之記憶體空間使用效率，可採行動態載入(dynamic loading)。
  - 主程式儲存在主記憶體並執行，當需要呼叫其它程式時，首先看看此程式是不是已經存在記憶體內，如果不是，便呼叫重定位 鏈結載入程式(re-locatable linking loading)，將所需要的程式載入主記憶體內，並更新行程位址表的內容。然後控制就轉移給新的載入程式。

6

邏輯位址空間和實體位址空間

- 動態鏈結和共用程式庫(.dll 舉例說明)
  - 採用動態鏈結，在程式參用程式庫副程式處便做一記號 (stub)，該記號為一小段程式來指示如何去找尋適當的記憶體常駐程式庫副程式，或是如何載入程式庫副程式(如果不在記憶體中時)。當執行到記號處時，首先檢查所需要的副程式是否已經存在記憶體中。如果該副程式不在記憶體中，程式會將它載入到記憶體中。

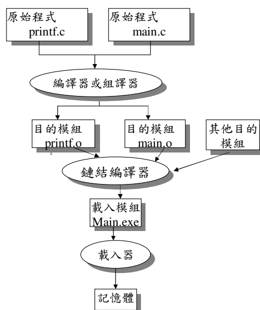
7

位址連結

- 當許多行程都要求將程式載入記憶體時：
  - 行程均進入輸入佇列
  - 依據排程器的排程結果選擇一個行程載入
  - 行程執行時，從記憶體取得指令與資料；執行結束後，會釋放所佔有的記憶體空間
- 位址轉換的步驟：
  - 原始程式中的位址：以符號(標籤)表示
  - 編譯器或組譯器：將符號所指之位址連結(binding)到一可重新定址的相對位址
  - 鏈結編譯器或載入器：將可重新定址(re-locatable)的位址連結到記憶體中的絕對實體位址

8

程式執行前的處理過程



9

位址連結

- 對於同一份資料或指令而言，位址是隨時間而變的
- 資料或指令連結到實體位址的動作可在下列任一階段完成
  - 編譯階段
    - 已確定程式要在記憶體的某個位址執行，絕對位址的程式碼就可以產生
    - 當起始位址改變，程式必須重新編譯，以產生新的絕對位址的程式碼

10

位址連結

– 載入階段
  - 編譯過程，不知道程式將在記憶體何處執行
  - 程式需編譯成可重新定址(re-locatable)的程式碼，當程式載入執行時，連結動作才進行。若起始位址改變，程式碼只需重新載入就可與記憶子重新連結
  - 例：動態鏈結程式庫需要在載入階段連結

– 執行階段
  - 若在執行時，行程會從一記憶體區塊移動到另一區塊；或是內含執行時才能確定的資料型態，連結動作要被延遲到執行時才發生
  - 例：大部分現代的作業系統中都提供執行階段連結的功能，在動態產生行程或執行緒(Object-Oriented Method)

靜態和動態配置

Int n;
Int sample[500];
For (int i=0; i<500; i++)
    scanf("%d",sample[i]);
	Int n;
	Int sample[n];
	For (int i=0; i<n; i++)
	    scanf("%d",sample[i]);

記憶體的配置模式

- **程式碼**主要包括了主程式、函式的程式碼。
- **靜態的資料**是編譯器創造出來，並成為該程式的一個單獨的元件。**動態的資料**則是由像Pascal、C、ADA 或C++的程式語言的執行時支援所配置的。
- 這種類型的資料就是我們前面所提的**程式控制的動態資料**(Program Controlled Dynamic data, PCD data)，而關於這種資料的記憶體配置則稱為**程式控制的動態配置**(Program Controlled Dynamic data allocation, PCD allocation)。

13

位址連結

- 編譯時間:在編譯時間時，若已確知程式在記憶體中的位置， 那麼絕對碼便可產生。
- 載入時間:程式在編譯時間若不能確定在記憶體中的位置，那麼編 譯程式就必須產生重定碼(re-locatable code)。
- 執行時間:如果行程在執行時能夠被由原來的記憶段落移動至另一段落位置，那麼連結的動作就會被延遲至執行時間才發生。

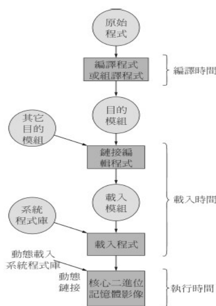
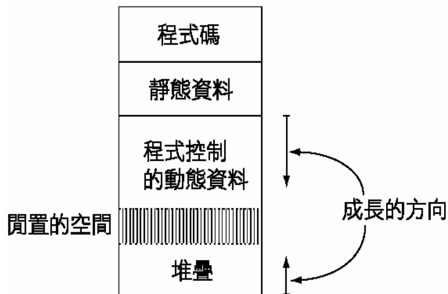


圖 8.3 一個使用者程式的多重步驟處理過程

14

執行中的程式



15

重疊

- 目的：解決記憶體容量的限制，程式大小可能比作業系統配置給行程的記憶體容量來的大，為讓程式順利執行，可利用重疊技術來解決記憶體容量的限制
- 做法：在編譯時，將程式與資料分割成多個獨立區域。在執行時，記憶體中只保留有需要的區段
- 重疊驅動器：載入目前要用的區段
- 若有區段可共用記憶體，新載入的區段會覆蓋舊區段

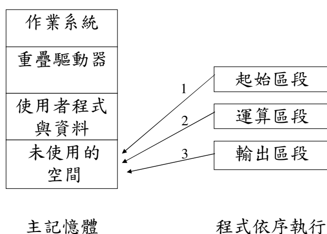
16

重疊

- 程式除從開始到結束都需要的指令與資料為一區段外，其餘程式可分成三個區段：起始、運算與輸出區段，且可共用記憶體空間。起始區段程式會先被載入，執行完後，接著載入運算區段並覆蓋起始區段，最後再載入輸出區段並覆蓋運算區段

17

重疊



18

重疊

- 多重重疊：重疊區段中還可分區段來重疊，此工作是由程式設計師來做，造成程式設計師的負擔，一般會避免使用，因為：
  - 區段分割太多→置換次數過多→降低程式執行效能
  - 區段分割太少→可重疊的程式部分過少→記憶體可能不夠，系統效能降低
- 除非像嵌入式系統(記憶體有限，沒有虛擬記憶體)，才用重疊，否則一般皆避免使用重疊

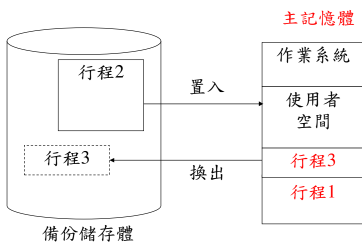
19

置換

- 時機：系統無足夠空間容納所有行程
  - 非執行中的行程暫時移到備份儲存體，要執行時再搬回記憶體中
  - 備份儲存體：一般而言指磁碟
- 換出、置入過程
  - CPU排程器決定下一個執行的行程
  - 分派程式到記憶體中尋找該行程
    - 若不存在且無足夠記憶體空間→先換出某些行程
    - 在置入該行程時，需重新載入暫存器內容，將控制權交給該行程

20

置換兩個行程



21

置換

- 產生內文切換的額外負擔
  - 時間浪費在資料傳遞上
  - 如果OS知道一個行程真正會用到多少記憶體空間，便可以只置換實際所需要的記憶體，而節省置換所消耗的時間
- 為了提高效率減少置換記憶體大小
  - 行程必須隨時告知作業系統：行程對記憶體需求的變化
  - 以便作業系統只置換實際所需的記憶體空間，節省置換所消耗的時間

22

置換

- 當I/O 操作完成而返回時，因行程被置換出而發生記憶體存取的錯誤
  - 原因：置換出不處於閒置狀態的行程(例：置換出的行程正在等待非同步的 I/O 操作→DMA)
- 解決方式：
  1. 任何企圖作I/O操作的行程不會被置換
  2. 只有進入作業系統緩衝區的行程才可作I/O操作，而作業系統與行程記憶體間的資料傳遞，只有在行程被置入時才可以進行

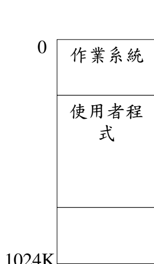
23

置換

- 置換在內文切換需要大量時間。可以實際情況來說明：若有一行程佔10MB，而備份儲存體為硬碟，其傳輸術速度每秒40MB。傳遞10MB到記憶體的時間為10/40=0.25秒=250毫秒
- 若不需磁頭尋找且平均潛伏時間為8毫秒，所以總置換時間為258毫秒，但因置換動作是雙向的，所以總共需516毫秒

24

記憶體分割



- 一部分供作業系統常駐使用
  - 放置位址常考量中斷向量的位址，因此作業系統常位於低位址
- 另一部分供使用者行程使用
- 配置方式：
  - 單一分割配置(單一使用者)
  - 多重分割配置(多元程式概念)
    - 將記憶體分成許多固定大小的區塊，再配置給各行程使用

25

單一分割配置

- 單一使用者
- 記憶體分隔成兩部分，一用來常駐作業系統，剩餘僅供一個使用者行程執行
- 缺點：
  - 僅讓一個行程執行，造成記憶體空間的浪費
  - 若行程執行 I/O 操作， CPU 閒置，使整體系統效能降低
  - 若行程大小超過可用的記憶體空間，將導致程式無法執行(可能解決方法：重疊)

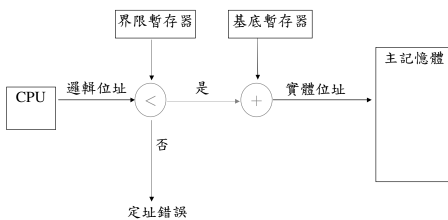
26

單一分割配置

- 記憶體相連，如何保護作業系統與使用者程式不會遭到對方不當修改？
  - 利用基底暫存器與界限暫存器的輔助
    - 基底暫存器：存放最小的記憶體實體位址
    - 界限暫存器：存放邏輯位址範圍
    - 每一個邏輯位址都須小於界限暫存器的邏輯位址範圍
    - 邏輯位址＋基底暫存器內的數值→記憶體的實體位址
  - CPU排程器選定一行程→分派程式將正確的值載入到基底和界限兩暫存器中→CPU每存取一次邏輯位址都經由兩暫存器的核對轉換，確保作業系統與使用者程式不會互相影響

27

位址保護機制



28

基本硬體

- 主記憶體和建立在處理器內的暫存器是CPU唯一可以直接存取的儲存體。機器指令使用記憶體位址做為參數，為使用磁碟位址做為參數。因此，任何執行的指令及被這些指令使用的資料必須放在這些直接存取儲存裝置之內。如果資料不在記憶體內，則必須在CPU操作它們之前移到記憶體之中。

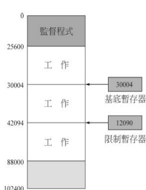


圖 8.1 一個基底及一個限制暫存器定義了一個邏輯位址空間

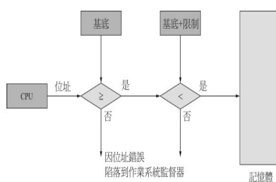


圖 8.2 使用基底及限制暫存器的硬體位址保護

多重分割配置

- 多元程式概念：多使用者程式同時要求載入到記憶體而先會進入輸入佇列，等待CPU排程器選擇下一個執行程式載入
- 主記憶體的容量有限，作業系統如何將可用的記憶體配置給正在輸入佇列中等待的多個使用者程式？
  - 將記憶體劃分成許多固定大小的區域或分割
  - 每個分割只能容納一個行程
  - 分割數目的多寡會影響到同時放置的使用者行程數量
- 系統依據輸入佇列的設計方式，找一塊適合的記憶體分割載入該程式

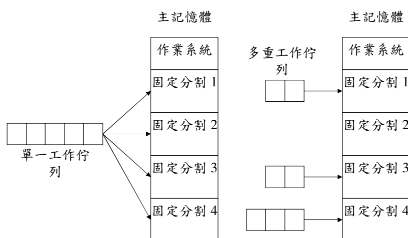
30

多重分割配置

- 設計輸入佇列兩方法：
  - 單一工作佇列：所有的分割都對應到同一個輸入佇列
    - 記憶體使用率較高；程式的等待時間較一致，不會讓特定程式有較快的反應，也不會有等待太久的程式
  - 多重工作佇列：每個分割均有一個對應的輸入佇列
    - 缺點：作業系統須針對每個程式找到適合的輸入佇列，有些分割的輸入佇列幾乎是空的，而有些卻佔滿要載入記憶體的程式，造成記憶體中雖有可用空間卻無法使用的情況

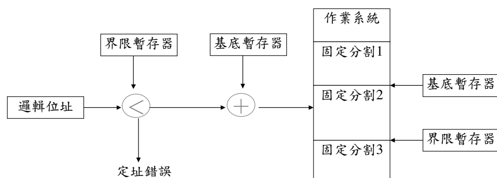
31

單一與多重佇列



32

多重分割下的系統保護



33

多重分割配置

- 記憶體劃分成固定大小的分割之問題：
  - 若程式小於分割大小的行程，會造成記憶體的浪費
  - 而程式大於分割大小的行程，則因為需要跨越分割，增加系統額外的負擔
- 解決程式大於分割大小方法：動態分割法
  - 依照程式執行時的大小，在記憶體中找到夠大的可用區塊給此行程使用
  - 需在作業系統維護一個表格，隨時記錄記憶體中哪些區塊使用中、哪些區塊空閒

34

連續與不連續配置記憶體

- 連續配置中，記憶體配置是採用最先配置或最適配置大小的方法
- 不連續配置中，在程式中每個部份都得進行，記憶體配置方法要利用串列來管理配置

35

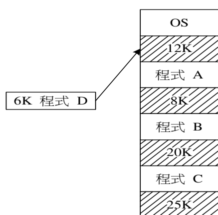
多重分割配置

- 作業系統以動態分割法載入程式時，依據 3 種策略：(94ncu 7)
  - 最先符合法(First Fit)：從第一個可用的區塊開始循序找起，只要找到夠大的空間，就把程式載入。
  - 最佳符合法(Best Fit)：找到一個與載入程式大小最為接近的區塊，再把程式載入。
  - 最差符合法(Worst Fit)：找到最大的區塊，再把程式載入。

36

最先適用法(First - Fit Strategy)

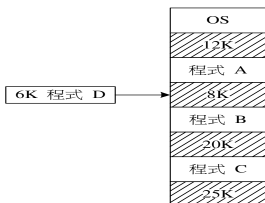
尋找第一個能容納程式或資料的記憶區塊。此法花費於尋找適合的記憶體區塊的時間會最小。 (6k, 8k, 12k,15k)



37

最佳適用法(Best - Fit Strategy)

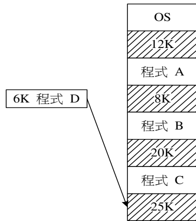
- 尋找一個記憶體區塊其大小與程式的大小最接近者。此法需花費較多的時間去尋找適合的記憶體區塊，但所產的間隙 (Hole) 會最小。



38

最差適用法(Worst - Fit Strategy)

尋找最大的記憶體區塊供程式或資料存放。此法會造成最大的間隙。



39

多重分割配置

- 根據電腦模擬分析
  - 最先符合法與最佳符合法在搜尋時間和記憶體的使用率都優於最差符合法
  - 最先符合法的執行速度通常比最佳符合法與最差符合法要快
  - 若不考慮搜尋時間，行程大小變化較大的適合最佳符合法；反之，則適合最差符合法。

40

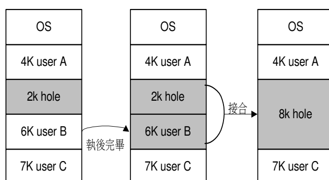
多重分割配置

- 行程執行結束後，作業系統將會
  - 釋放此行程所佔用的記憶體區塊
  - 檢查此被釋放區塊是否可與可用的相鄰區塊合併
  - 同時作業系統檢查輸入佇列中是否有程式正等待配置記憶體→如果有，則檢查此新合併的區塊大小是否夠該行程所用。(間隙接合)

41

間隙接合

若有兩個間隙 (Hole) 彼此相鄰，則將之合併成為一個較大的區塊，以便能供其它程式使用。



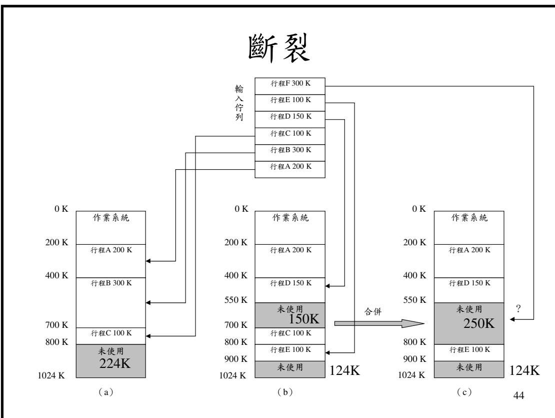
42

配置範例

- 整個主記憶體1024K，OS位於低位址記憶體(200K)，行程A、B、C分別使用記憶體200K、300K、100K，系統於行程B結束時接收其釋放的記憶體，並載入行程D(150K)、E(100K)，爾後行程C結束，系統將行程C所釋放的記憶體與其相鄰的記憶體合併成一較大的記憶體。
- 行程F(300K)仍不能執行。
- 兩塊記憶體合起來的空間足夠容納行程F，但分開卻不行，就產生斷裂現象(fragmentation)

43

斷裂



44

連續配置記憶體

- 在早先的電腦系統，是用**靜態**的方式來進行記憶體配置，也就是在程式執行之前進行
- 連續性記憶體配置在實作上的議題有：
  - 程式與程式之間的**保護**，也就是確保程式在執行時不會互相干擾
  - 程式的靜態與動態的**重新定址**，讓程式可以從它被配置的記憶區域中執行
  - **記憶體的斷裂**(它會產生一些無法使用的記憶區域)，以及克服它的方法

45

斷裂

- 外部斷裂(External fragment)(98tku 2)
  - 因為行程持續地被載入與置換，使得可用的記憶體空間被分割成許多不連續的區塊
  - 雖然記憶體所剩空間總和足夠讓此行程執行，卻因為空間不連續，導致程式無法載入執行
- 內部斷裂(Internal fragment)
  - 發生在以固定分割方式配置的記憶體
  - 當一個程式載入到固定大小的分割，假如程式小於此分割，則此分割剩餘空間無法被使用
  - 不能使用的空間散佈在各個分割內，造成輸入佇列中的程式無法順利載入執行，造成浪費

46

斷裂

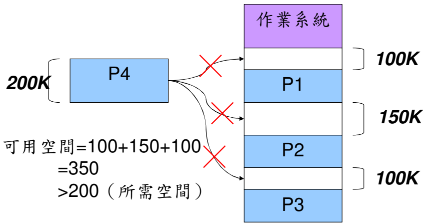
- 記憶體配置的最先符合法和最佳符合法這二種策略都會遭遇到記憶體外部斷裂問題。由於行程的載入移出，可用的記憶空間將被劃分為許多小區間。當有足夠的總記憶體得以滿足要求，而這些區間卻是非連續的時候，外部斷裂的現象便發生，儲存體被分成了許多的小區間。
- 外部斷裂是否會造成嚴重的問題，完全視記憶空間的總容量大小和行程平均大小而定。例如：最先符合法的統計分析顯示縱使具有最佳條件，給予N配置區間由於斷裂現象也將遺失其它的0.5N，那麼三分之一的記憶體可能沒有利用到，這就是著名的百分之五十規則(50-percent rule)。

47

內部斷裂(每個區塊皆200K)

48

外部斷裂之範例

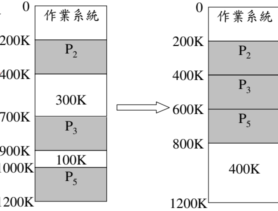


斷裂

- 有記憶體空間卻無法用(記憶體各個區塊太小)
  - 剩餘記憶體空間分散於各處，可能加起來的總和是足夠讓某些程式執行！➔聚集(compaction)

聚集

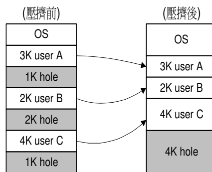
- 解決外部斷裂的問題
- 將記憶體中可用的不連續空間聚集成一大塊連續空間
  - 程式必須都可重新定位
  - 代價高，因為要搬動許多行程的實體記憶體空間
  - 都在記憶體不足時，才使用此法(磁碟重組工具)



51

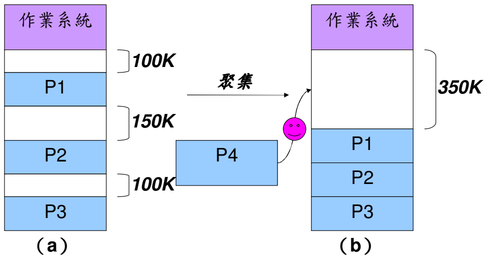
聚集

- 間隙接合僅適用於將彼此相連的間隙 (Hole)合併成一個較大的區塊聚集法(Compaction)。聚集法能將散佈於記憶體各處之小間隙合併成一個大區塊，以便能供其它程式使用。



52

聚集的範例



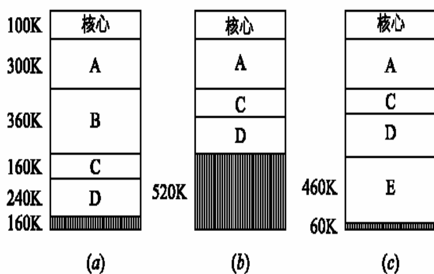
53

執行聚集法的時機

(1) 記憶體空間已經不足時。
(2) 定期執行壓擠的工作。

54

記憶體壓縮



A：300K，B：360K(先做完)，C：160K， D：240K ， E：460K

55

練習

- 某OS之記憶體管理是採用可重定位址之方式配置主記憶體僅128K，今OS佔8K，行程A佔32K、行程B佔28K，行程C佔28K，編譯器佔32K、連結器佔32K，行程E佔42K，行程F佔35K，執行次序及狀態應如何配置？
  - 行程A、B、C已經在記憶體中，行程D 32K進入系統且需要編譯
  - 行程B執行完畢
  - 行程C執行完畢
  - 行程D編譯完需連結器作成執行檔
  - 行程D執行檔32K作完要執行

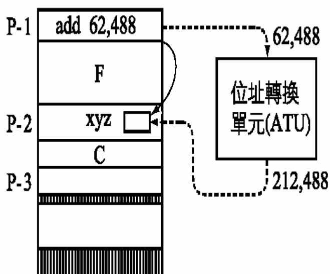
56

非連續性記憶體配置

- 位址的修正稱為**位址轉換**(address translation)。記憶體中一個運算元實際存放位置的記憶體位址，稱為這個運算元的**有效記憶位址**(effective memory address)
- P程式大小為140k被分成50k、30k、60k，分別放於起始記憶體為0、200k、280k的位址，實心箭頭表示P的指令存取xyz，而虛線箭題則表示P 執行時實際的存取情形。稱作**位址轉換單元**(address translation unit, ATU)的硬體單元將xyz 的位址，訂正成有效位址212,488

57

位址轉換支援不連續的配置



58

連續與不連續記憶體配置的比較

特性	連續配置	不連續配置
配置	配置記憶體的單一區域	配置數個記憶體區域——一個記憶體區域配置給程式的每個部份
執行的額外消耗	程式執行期間沒有額外的消耗	程式執行期間要執行位址轉換
置換	程式中的置換必須被置於其原來配置的區域。因此程式的進展依賴使用同個記憶體區域的其他程式	程式中的置換能被置於任何地方。這特性提供比連續配置較佳的執行進展
重複使用	內部斷裂存在於分割區配置。外部斷裂存在於最先合適／最合適配置	在節段中：外部斷裂存在，但是無內部斷裂。在頁中，無外部斷裂，但是有內部斷裂存在

59

分頁基本方法

- 程式可被不連續放置，沒有外部斷裂的問題
  - 將載入的程式分割成固定大小的分頁
  - 主記憶體也分割成固定大小的頁框，大小與分頁相同
  - 執行程式時，把程式所有的分頁載入記憶體任何可用的頁框中，且這些分頁不需要相鄰
- 每個程式有一個分頁表，存有每分頁在記憶體中的起始位址。當程式的分頁被載入到主記憶體時，程式分頁表中記錄該分頁被載入至主記憶體的哪一個頁框中
- 頁框表：作業系統須知道主記憶體中頁框使用與否、系統中總頁框數目有多少等資訊

60

分頁法

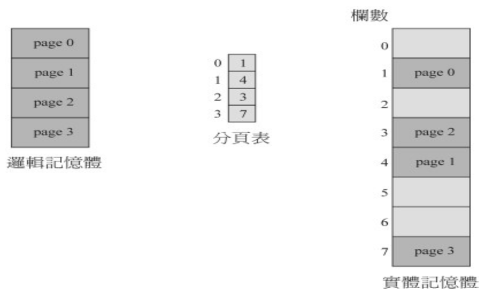
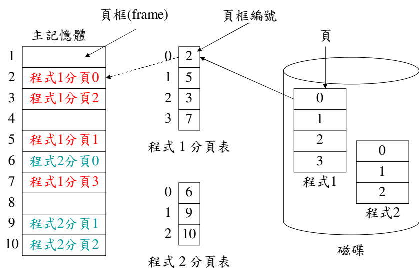


圖 8.8 邏輯與實體記憶體中的分頁模式

分頁法



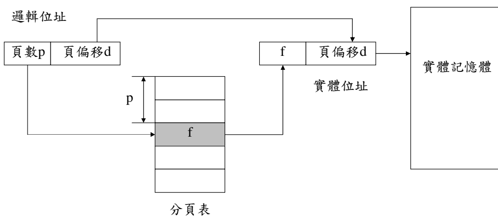
62

基本方法

- CPU 產生的邏輯位址分成兩部分：(96tku 4) (p,d)
  – 分頁碼為指向分頁表的索引(p)→(f)
  – 頁偏移表示與該分頁起始位址的距離(d)
- 在分頁表中找到此分頁在實體記憶體中對應的基底位址後，再和頁偏移組合定義出實體記憶體位址
- 分頁法不會有外部斷裂仍有內部斷裂的問題
  – 如果一個程式所需要的記憶體大小不能被頁框大小整除，最後一個頁框就不會全部被使用而形成內部斷裂
  – 每個程式平均會有 1/2 分頁的內部斷裂
  – 若使分頁大小縮小，就可節省因內部斷裂而產生記憶體空間浪費；但須更大空間儲存分頁表(分頁的數量增加)

63

分頁邏輯位址空間與實體位址空間的轉換



64

分頁表之邏輯位址轉換至實際位址

- 1. 處理單元A的邏輯位址(p,d)=(1,20)其實際位址？
  2. 處理單元B的邏輯位址(p,d)=(3,60)其實際位址？

頁框編號	實際位址
0	1000
1	40
2	3200
3	500
4	100
5	600
6	400
7	2000
8	1500

65

分頁表的結構

- 儲存分頁表：若儲存在主記憶體中，系統需作兩次記憶體存取(先分頁表位址加上頁偏移)，效率低
- 解決方法：將分頁表放在關聯式記憶體的小型記憶體(也稱位址查閱緩衝TLB)中
  - 每筆資料有分頁碼、頁框編號兩欄位。關聯式記憶體以分頁碼為索引，平行地在相對的頁框中找尋
  - 很短的時間就可找尋到，時間複雜度為 O(1)

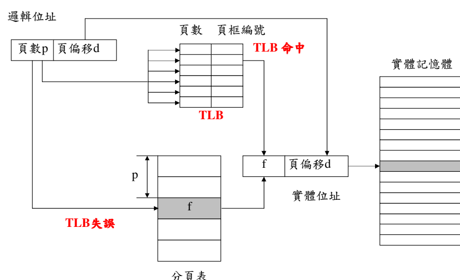
66

分頁表的結構

- 實際上的做法(因關聯式記憶體昂貴)：使用主記憶體建立分頁表，將關聯式記憶體當成快取記憶體，只保存分頁表的部分內容
- 若頁框編號在關聯式記憶體中找得到，就直接與頁偏移相加得到實體記憶體位址(Hit)，否則再到分頁表中找尋 ，最後再決定是否將找到的分頁碼與頁框編號放入關聯式記憶體
- 在更新關聯式記憶體時，如果關聯式記憶體已經存滿，則系統必須置換掉其中一筆記錄(參考分頁置換規則)(96tpu 5)

67

使用 TLB 硬體支援的分頁法(94tku)



68

考題

- Draw a diagram to show the transition from logical address to physical address under single-level paging system with TLB?(94tku 4)

69

分頁表的結構

- 使用關聯式記憶體(TLB)來儲存分頁表的效率：(95ncu 4,6、96ncu 2,4)
  - 如果要尋找的分頁碼已經在關聯式記憶體中，則稱為命中(Hit)，否則稱為失誤(Loss)
  - 命中率(Hit Rate)的定義為[命中次數/(命中次數＋失誤次數)]*100%
  - 例：假設存取關聯式記憶體的時間為 20 奈秒，直接存取主記憶體的時間為 100 奈秒。
  - 若全部都以TLB來儲存分頁表：存取實體位址內資料所需的時間為120奈秒，20奈秒先到TLB找頁框編號，100奈秒是存取主記憶體內的資料(命中時間)

70

分頁表的結構

- 若全部都以主記憶體來儲存分頁表：存取實體位址內資料所需的時間為200奈秒，100奈秒先到主記憶體找頁框編號，100奈秒是存取主記憶體內的資料
- 若混合TLB與主記憶體來儲存分頁表(但得先到TLB檢查)：假設在關聯式記憶體內命中率為 95%，則所需的時間為：

  $0.95 \times 120\ ns + 0.05 \times 220\ ns = 125\ ns$

  - 其中 120 奈秒的 20 奈秒花費在關聯式記憶體中找尋(命中)，100 奈秒花費在存取主記憶體的資料；而 220 奈秒中的 20 奈秒花費在關聯式記憶體中找尋(失誤)，100 奈秒花費在主記憶體中找尋，另外 100 奈秒花費在存取主記憶體的資料

- 適當地使用關聯式記憶體，可以降低主記憶體的存取時間，也可以節省成本(90tku 2、94tku 5、93ncu 8、94ncu 1)

71

效能的評估--兩種指標

- TLB的命中率(hit ratio)
  - 要存取的分頁資料可以在TLB中找到的機率
- 有效記憶體存取時間(effective memory access time)

[有效記憶體存取時間] =

[命中率] × [在TLB可以找到分頁的記憶體存取總時間]

\+

(1-[命中率]) × [在TLB找不到分頁的記憶體存取總時間]

72

有效記憶體存取時間

- 採用分頁方式來管理記憶體，有下列資料：
  hit rate: 80%
  TLB access time: 50ms
  Memory access time: 700ms
  what is the effective memory access time?

73

分頁表的結構

- 分頁的環境中，記憶體的保護可以使用分頁上面的保護位元來完成(通常保存在分頁表)
- 保護位元可以定義某一分頁是可讀、可寫或者兩者皆可
- 每次的位址轉換均會去參考相對應的保護位元；企圖以非保護位元所提供的動作來存取此分頁，將會引發例外中斷

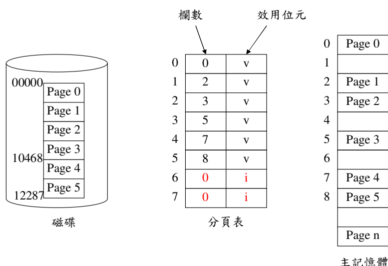
74

分頁表的結構

- 分頁表每個紀錄都會有一個效用位元：
  - 當此位元被設定為有效(v)時，表示所對應的分頁目前在主記憶體中；如果此位元被設定成無效(i)，則表示所對應的分頁在輔助記憶體中
  - 作業系統會設定每分頁的有效位元，以核對該分頁的存取動作；當系統進行位址轉換時，也會去檢查此位元，若企圖存取無效的分頁也會引發例外中斷

75

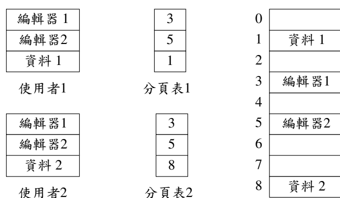
分頁表中的效用位元



76

兩個使用者共用頁框

- 使用分頁法另一項好處
  - 簡單達到程式碼共用，OS可在各行程的分頁表內做適度的管理，使得共用分頁的程式碼僅在頁框中出現一次



77

分頁

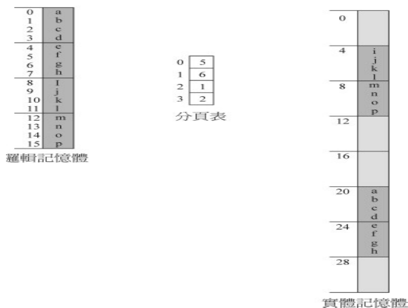


圖 8.9 以每頁 4 字組分頁之 32 字組記憶體的例子

78

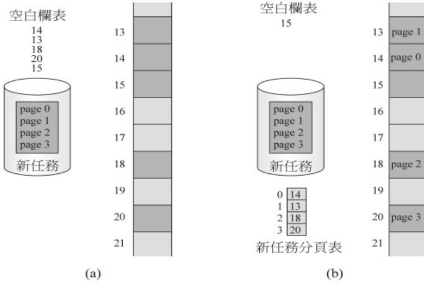


圖 8.10 空白欄的配置 (a) 之前和 (b) 之後

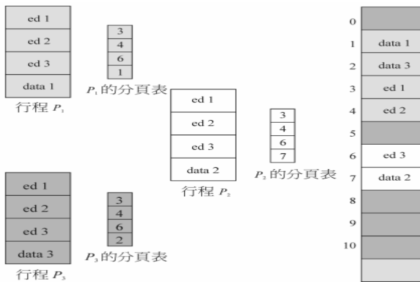


圖 8.13 在分頁環境中的共用程式碼

多層分頁法(Multi-level paging)

- 在一32位元電腦中，每個分頁大小為4K位元組(2^{12})，每頁的偏移位址有12個位元，其餘20位元用於選擇頁框編號，所以共有2^{20}個分頁，若每個分頁表的記錄佔用4個位元組，則得花4MB儲存分頁表(4* 2^{20}位元組)
- 不希望分頁表佔用連續的記憶體空間→想把分頁表分成較小的單位儲存(95tku 1, 93tku 3, 95tpu 1)

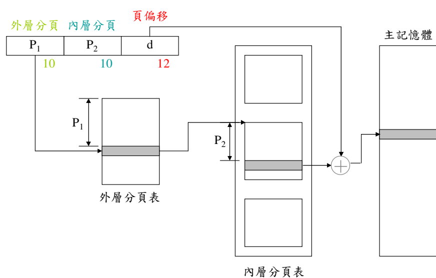
81

多層分頁法

- 多層式分頁將分頁表也進行分頁
  - 以兩層式的分頁來說，在由邏輯位址轉換成實體位址時；
  - 先以 P_1為索引到外層分頁表中找尋，所找到的記錄會指向一個相對應的內層分頁表
  - 然後再以 P_2為索引到該內層分頁表中找尋，所找到的記錄會指向一個頁框
  - 最後和頁偏移 d 相加成為實體記憶體位址

82

兩層式分頁法(32位元的邏輯位址)



83

階層式的分頁

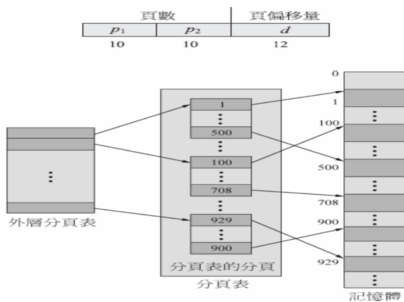


圖 8.14 兩層分頁表的技巧

84

階層式的分頁

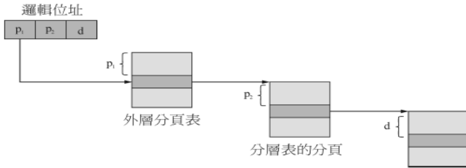


圖 8.15 32 位元兩層分頁架構的位址轉換

在 VAX 架構下的位址如下所示：

段落	頁數	頁偏移量
s	p	d
2	21	9

外層分頁	內層分頁	頁偏移量
p_1	p_2	d
42	10	12

64位元邏輯位址空間的系統

85

階層式的分頁

- 可用不同方式分割外層分頁表，例如使用三層分頁的方法，再對外層的分頁表做分頁。假設外層分頁表是由標準大小的頁數(2^{10}頁及2^{12}頁)所組成，一個64位元的位址空間為

第二外層分頁	外層分頁	內層分頁	頁偏移量
p1	p2	p3	d
32bits	10bits	10bits	12bits

86

多層分頁法

- 64 位元的架構不適合階層式的分頁方式(外層分頁表將佔用太大的連續記憶體空間)，每個分頁大小為4K位元組(2^{12})，每頁的偏移位址有12個位元，其餘52位元用於分層頁框，所以共有2^{52}個分頁，故除分為內、外層各10個位元外，還加上一個第2外層分頁32位元
- 對於邏輯位址大於 32 位元的硬體架構，可使用雜湊分頁表
  - 利用雜湊函數(http://knight.fcu.edu.tw/~d9046876/ds/d_71.htm)將邏輯位址對應到實體位址
  - 可能發生雜湊碰撞而降低效率，但是可有效減少分頁表空間

87

雜湊函數

- **雜湊 (hashing)** 的搜尋與一般的搜尋法不同。在Hashing 中，鍵值或識別字在記憶體的位址是經由函數 (function) 轉換而得。此種函數，一般稱之為雜湊函數或鍵值對應位址轉換
- 雜湊法具有以下四項優點：(1) 使用雜湊法搜尋，檔案不須事先排序。(2) 在沒有碰撞及溢位的情形下，只需一次讀取即可，且搜尋的速度與資料量的多寡無關。(3) 保密性高，若不知雜湊函數，無法擷取到資料。(4) 可做資料的壓縮，利用適當的散置函數，可將資料壓縮到一個較小的範圍，以節省空間。

88

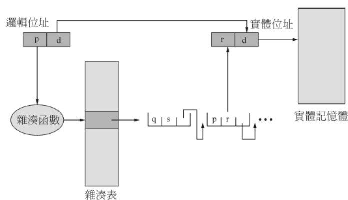
雜湊分頁表

- 處理位址空間大於32位元的一種常見方法是使用雜湊分頁表(hash page table)。其中雜湊值即是虛擬分頁值。雜湊表中每一項包括了雜湊到相同位置之單元的鏈結串列。
  - 每一個單元由三個欄位所組成:
    1. 虛擬分頁的數值
    2. 對映分頁的數值
    3. 指向鏈結串列下一單元的指標。

89

雜湊分頁表

•演算法依照下列的方式進行:虛擬位址的虛擬分頁數值被雜湊(hashed)到雜湊表。虛擬分頁數值和鏈結串列中第一個單元的欄位(1)做比較。如果兩者相同，相關分頁欄位 (欄位(2))就被用來形成所需要的實體位址。如果不吻合，鏈結串列中接下來的單元會被搜尋以找出符合的虛擬分頁數值。



90

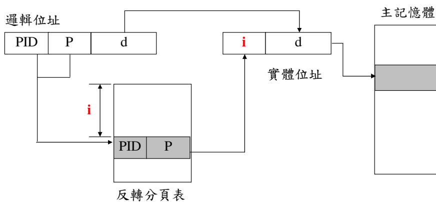
圖 8.16 雜湊分頁表

反轉分頁表

- 每個行程都有各自的分頁表，若邏輯位址空間很大的行程，需要很大的分頁空間。
- 解決分頁表空間太大的問題：整個系統僅用一個分頁表
  - 行程辨識碼(PID)加上邏輯分頁與實體頁框是一對一的對應
  - 反轉分頁表的記錄數目要與頁框一致
- 反轉分頁表的架構下(93tpu 8,96tku 4)
  - 一個邏輯位址包含三個欄位：行程辨識碼(PID) 、分頁碼(P)、與頁偏移(d)
  - 反轉分頁表中每個記錄包含：行程辨識碼與分頁碼
  - 當一個行程要進行位址轉換，必須以 PID 與分頁碼當索引，到反轉分頁表中找尋所屬的頁框編號，再與頁偏移相加就可以得到實體位址

91

反轉分頁表



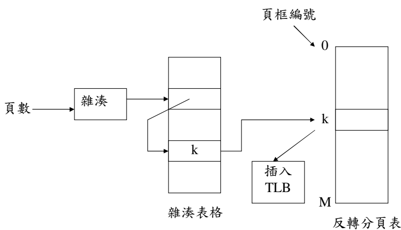
92

反轉分頁表

- 反轉分頁表可以降低儲存每個分頁表所需要的空間，但是搜尋分頁表所用的時間卻大量增加
  - 減輕此問題：使用雜湊表格 (一個分頁對應到一個頁框)
- 從取得邏輯位址開始到在記憶體中存得資料，至少需要兩次的記憶體讀取：一次是雜湊表格，另一次是反轉分頁表
  - 解決的辦法是利用類似關聯式記憶體(TLB)的方式來加快搜尋
- 在一個使用反轉分頁表的系統中，無法分頁共用(共用需把2個邏輯位址對應到一個實際位址)

93

反轉分頁表－雜湊表格



94

分頁表之邏輯位址轉換至實際位址

- 若每一頁大小為q，則存放資料的頁數p與頁偏移d，可經由邏輯位址a計算得之，其公式：
  p=a div q
  d=a mod q ➔ (p,d) ➔ (f,d) ➔實際位址

95

分頁表之邏輯位址轉換至實際位址

- 有3個處理單元A,B,C，在執行時的分頁表基底暫存器的值分別為2,6,3，試就下列之邏輯位址將之轉換為實際位址？(每頁大小為100)
  1. 處理單元B的邏輯位址320？頁插斷(page interrupt)
  2. 處理單元C的邏輯位址250？650

頁框編號	實際位址
0	1000
1	40
2	3200
3	500
4	100
5	600
6	400
7	2000
8	1500

96

分頁表之邏輯位址轉換至實際位址

- 有3個處理單元A,B,C，在執行時的分頁表基底暫存器的值分別為2,4,6，試就下列之邏輯位址將之轉換為實際位址？
  1. 處理單元A的邏輯位址(p,d)=(3,20) ？
  2. 處理單元B的邏輯位址(p,d)=(3,60) ？
  3. 處理單元C的邏輯位址(p,d)=(2,60) ？

頁框編號	實際位址
0	1000
1	40
2	3200
3	500
4	100
5	600
6	400
7	2000
8	1500

97

分頁表之邏輯位址轉換至實際位址

- 若每一頁大小為4K，則存放資料的頁數p與頁偏移d，可經由邏輯位址0x2576將之轉換為實際位址？

頁框編號	實際位址
0	0x1000
1	0x2000
2	0x3200

98

分段

- 依照程式的邏輯功能將邏輯位址空間切割成許多分段
- 分頁法將邏輯位址空間切割成許多固定大小的分頁
- 每個分段的長度都不盡相同，長度是由此分段中程式的大小來決定
- 若要參考分段中某一位元組，可以藉由此分段的起始位址配合段偏移來決定

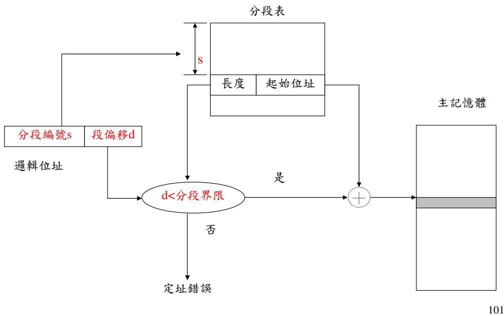
99

基本方法

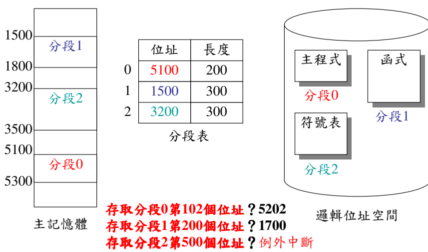
- 一個行程的邏輯位址被分成許多分段，每個分段都有一個名稱和長度，也有一個分段編號
  - 一個行程的邏輯位址被分成兩個欄位：分段編號(當作分段表的索引)與段偏移(s,d)
- 每個程式均有一個分段表，其中每一記錄都有：
  - 分段基底值：記錄分段在主記憶體中實際開始的位址
  - 分段界限值：記錄該分段的大小，以避免程式執行時超過該分段界限，若超過則產生例外中斷

100

分段法的使用

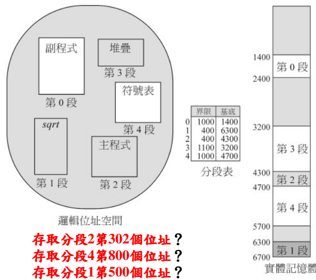


分段法



存取分段0第102個位址？5202
存取分段1第200個位址？1700
存取分段2第500個位址？例外中斷

分段法



存取分段2第302個位址？
存取分段4第800個位址？
存取分段1第500個位址？

圖 8.20 分段法的例子

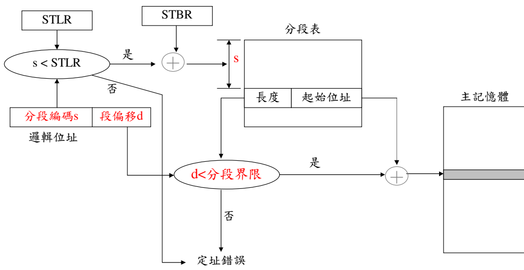
103

基本方法

- 分段表可以存放在
  - 快速的暫存器：節省對界限暫存器相加與基底暫存器比較的時間
  - 主記憶體：對於有大量分段的程式而言，不宜放在快速暫存器中
    - 利用一個分段表基底暫存器(STBR)來指向分段表；使用一個分段表長度基底暫存器(STLR)記錄分段大小
    - 將分段基底值與段偏移相加，取得要存取位元組的實體位址
    - 缺點：每個邏輯位址需要對實際記憶體存取兩次，所以速度比較慢
    - 解決方法：也可利用關聯式暫存器儲存經常會被使用的分段表記錄

104

配合 STLR 與 STBR 的分段法(s,d)



105

分段

- STBR=0試就下列之邏輯位址將之轉換為實際位址？
  1. (0,480)？792
  2. (1,36)？錯誤例外
- 3. (2,652)？4176
- STBR=1試就下列之邏輯位址將之轉換為實際位址？
  1. (0,80)？錯誤例外
  2. (1,200)？3724

段編號	段對應位址	大小
0	312	500
1	2566	25
2	3524	752

106

分段

- STBR=0試就下列之邏輯位址將之轉換為實際位址？
  1. (0,120)？
  2. (1,440)？
  3. (2,36)？
  4. (3,100)？
  5. (4,420)？
  6. (5,60) ？

段編號	段對應位址	大小
0	340	450
1	2560	25
2	190	125
3	1336	536
4	1950	87
5	2054	40

Intel Pentium

- Pentium架構允許區段可以大到4GB，每個行程的分段數目最多是16KB。一個行程的邏輯位址被分成兩部份。第一部份是由該行程私有的分段(至多有8KB)所組成。第二部份是由所有行程共同使用的區段(至多有8KB)所組成。
- 第一部份的資料存放在區域描述表LDT(local descriptor table)中，第二部份的資料存放在全域描述表GDT(global descriptor table)中。LDT和GDT中的每一項皆佔用8個位元組，其中包含關於某一特殊區段的詳細資料，包括基底位址以及該區段的長度。

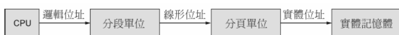
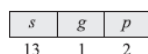


圖 8.21 在 Pentium 中邏輯的實體位址轉譯

邏輯位址是一組（選擇器，偏移量），其中選擇器（selector）是一個 16 位元的數目：



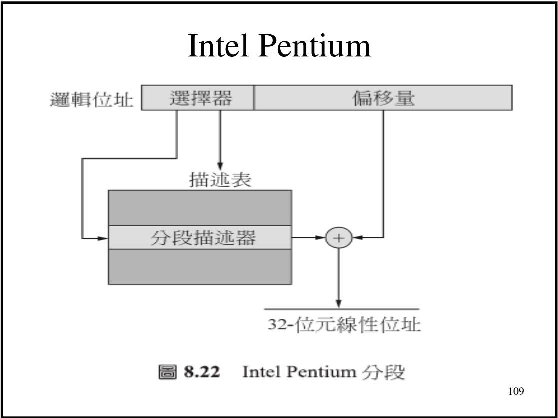


圖 8.22 Intel Pentium 分段

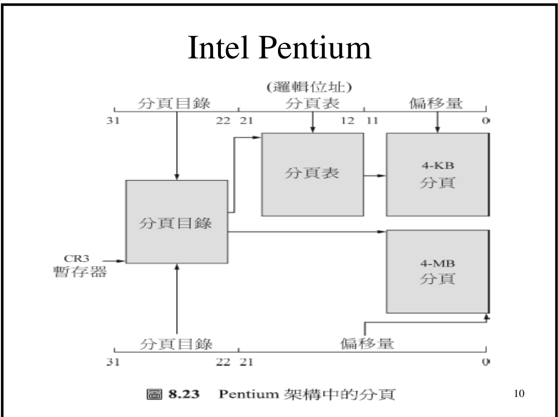


圖 8.23 Pentium 架構中的分頁

Linux在Pentium系統

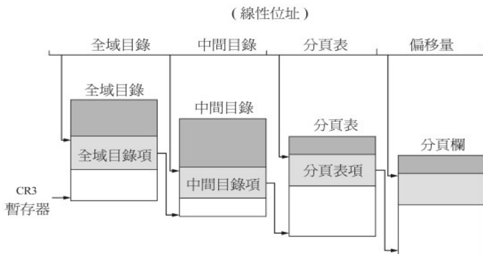


圖 8.24 Linux 中三層分頁

111

基本方法

- 分段法可達到分段保護的功能
  - 可以指定指令分段是唯讀(r)或是唯執行(e)的方式、資料分段是可讀寫(rw)的方式
  - 分段表中每個項目有一組保護位元，以避免不正當的存取

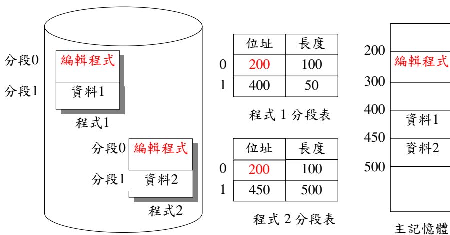
112

基本方法

- 分段法可達到資料或程式碼共用的功能
  - 兩個分段表中有共同的項目指向同一個實體位址
- 問題：行程可能跳到本身以外的記憶體位址執行程式碼
  - 如果此記憶體位址的程式碼共用，勢必所有共用行程所屬共用分段的分段編號都要相同
  - 該編輯程式如何參考本身的程式？且共用的行程增多後，想要找到一個可共用的編號將會更加地困難(一個編輯程式不同分段號碼？)
    pure (reentrance) code V.S. impure code(96tpu 14)

113

分段共用



114

基本方法

- 分段法中分段長度是變動的，若有外部斷裂，便可能導致記憶體中根本找不到足夠空間供一個分段載入(96tpu 11)
  - 該行程可以等待別的行程釋放記憶體，或是利用聚集法取得夠大記憶體以供載入
  - 若選擇等待別的行程釋放記憶體，CPU 排程程式可以考慮讓優先權較高、需要記憶體較小的行程先行，以增進效率

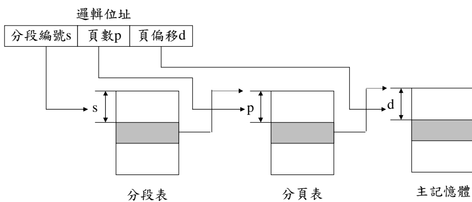
115

分頁式分段

- 在分頁式分段的方式下(98tku 1(d))
  - 每個行程會依據邏輯功能切成分段，而每個分段會再被分割成分頁，所以在這種架構下外層為分段表，內層為分頁表(s,p,d)
  - 一個邏輯位址包含了分段編號 s、分頁碼 p 與頁偏移 d
  - 系統會先透過 s 在分段表中找到分頁表的起始位址，再透過 p 在分頁表中找到分頁的頁框位置，最後再與 d 相加成為實體記憶體位址
- 平均每個分段也會有 1/2 分頁的內部斷裂
  - 所以分頁式分段中每個行程會有較多的內部斷裂
  - 著名的 OS/2 作業系統便是使用分頁式分段法，使用的分頁大小為 4 KB

116

分頁式分段



分頁式分段

- 有3個處理單元A,B,C，在執行時的分段表基底暫存器的值皆為1,試就下列之邏輯位址將之轉換為實際位址？
  1. 處理單元A的邏輯位址(s,p,d)=(2,3,20)？5120
  2. 處理單元B的邏輯位址(s,p,d)=(1,1,40)？4140
  3. 處理單元C的邏輯位址(s,p,d)=(3,1,30)？4730
  4. 處理單元C的邏輯位址(s,p,d)=(0,4,60)？錯誤例外

段編號	段對應位址
0	1209
1	1205
2	1214
3	1200
4	1216

分頁式分段

	頁編號	實際位址
1200	0	3500
	1	4400
	2	4300
	3	5100
	4	3800
1205	0	4800
	1	4500
	2	5400
	3	3700
1209	0	4900
	1	4200
	2	4600
	3	5000
	4	3600
1214	0	5300
	1	4100
1216	0	5200
	1	4700

119

摘要

- 記憶體管理的主要目的：
  - 有效地運用與維護記憶體空間
- 程式以二進位可執行檔的型態儲存於磁碟中
- 程式在執行時透過記憶體管理單元將邏輯位址轉換成實體位址
- 程式位址連結的工作可在編譯、載入或執行階段完成
- 當行程所需的執行空間大於配置給行程的記憶體時：
  - 利用重疊技術
  - 置換

120

摘要

- 記憶體管理的方式可分為
  - 連續配置：採用的方式有
    - 單一分割配置：單一使用者的系統
    - 多重分割配置：多元程式的環境
      - 最先、最佳、或最差符合法來分配記憶體
      - 不當的配置方式將會加重記憶體外部斷裂與內部斷裂的問題
  - 非連續配置
    - 分頁法，用來解決記憶體外部斷裂的問題，需要硬體的支援
      - 將載入的程式分割成固定大小的分頁，主記憶體也分割成固定大小的頁框，頁框與分頁大小相同
      - 執行程式時，把程式所有的分頁載入記憶體任何可用的頁框中，不需相鄰

121

摘要

- 多層次分頁
- 反轉分頁表
- 分段法
  - 依照程式的邏輯功能將邏輯位址切割成許多分段
  - 達到資料或程式碼共用的目的
- 分頁式分段：每個行程會依據邏輯功能切成分段，而每個分段再分割成分頁

122