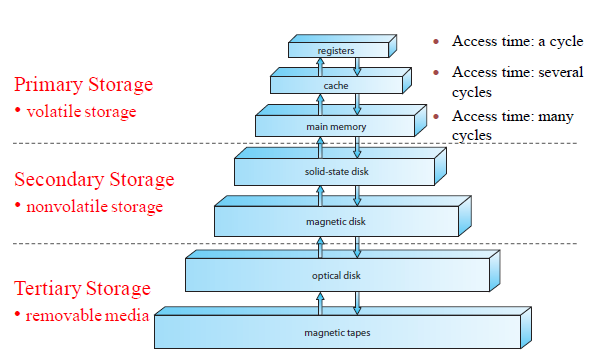
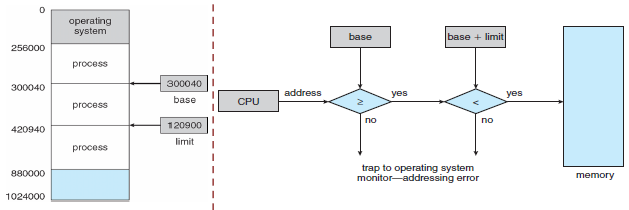
CH08-Memory

Storage-Device Hierarchy



Base and Limit Registers

* 定義logical adress space：
  + Base：放置的基礎位置
  + Limit registers：基礎位置+放置所需的空間
* CPU必須確認在user mode的可用記憶體位置，存在於base和limit間



Binding of Instructions and Data to Memory

Binding：決定程式的執行位置

發生在三種狀態：

* Compile time：如果儲存位址已經先知道了，absolute code可以產生→若啟動位置更改，則需重新編譯
* Load time：如果在compile time時不知道memory location，必須產生relocatable code
* Execution time：如果程式執行期間可以改變他的起始位置，則在binding會延後到程式run time

Logival and Physical Address Space

* Logical address：由CPU產生，也稱為virtual address
* Physical address：seen by the memory unit
* Logical 和Physical address在comile-time和load-time address-binding schmem中是相同的，在execution-time中不同
* Logical address space是所有logical adresses的集合，由program產生
* Physical address space是所有physical adresses的集合，由program產生

Dynamic Linking

* 目標：試著不讓任何一個必要條件成立
* 必要條件：
* Mutual exclusion
* Hold and wait
* No preemption
* Circular wait

Mutual exclusion

資源是不可共用的，可共用資源除外，因為不會發生死結。

Hold and wait

Rule：

* 在執行之前必須要求並取得所需的資源

Or

* 在要求一項資源時，不可佔用其他資源

Disadvantage：

* 資源使用率很低
* 飢餓現象

No preemption

資源只配置給可以執行(saved and restored)行程，也就是說，如果有一個行程已經占用某些資源，而它還要求一個無法立即取得的資源（必須等待），則它目前所持有的資源應可由他人搶先使用

Circular wait

對所有資源型式強迫安排一個線性的順序，而行程提出要求時必須依照數字大小，遞增地提出要求

Deadlock Avoidance

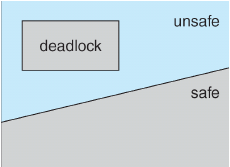
* 目標：
* 動態偵測資源的分配，確保沒有circuit-wait情況發生
* Keep the system at a **safe state**
* 要求時系統要有些附加資訊

對於每個資源

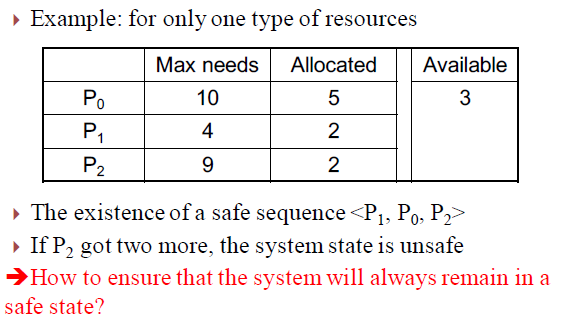
* 計算分配的資源
* 紀錄可用的資源

對於每個行程

* 聲明對於每個型式資源的對大用量
* 計算每個資源分配到的數量

Safe State

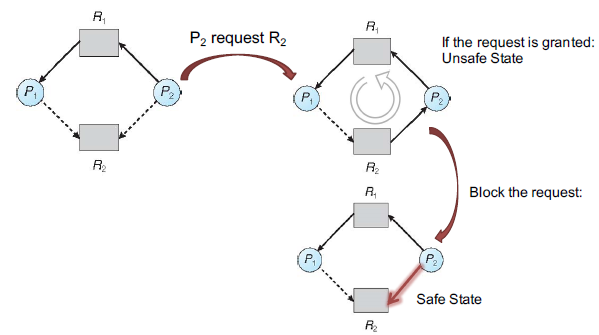
* 系統處於safe state → no deadlocks
* 系統處於unsafe state → possibility of deadlock
* Avoidance→確保系統永遠不會進入unsafe state
* 如果所有行程有safe sequence則系統處於safe state
* 一個序列 <P1, P2, …, Pn>，對每一個Pj（j<i），若Pi所要求的資源能被所有目前可用的資源與Pj所占有的資源所滿足，則此序列為safe sequence
* 也就是說：當Pj結束後Pi可以獲得它需要的資源，執行完並釋放資源，當Pi結束時，Pi+1即可得到所需的資源。



* + - 起初P0被分到5個P1被分到2個P2被分到2個剩下3個，此時可用的資源只夠P1再拿兩個去達到Max needs，而P1執行完後釋放資源，此時可用的資源有5個，足夠P0達到Max needs，所以換P0去執行，結束後釋放10個資源，可滿足P2需求。
    - 若P2一開始多拿兩個，則剩餘的資源只有一個，不夠分給任何行程，故為unsafe的狀態

Resource-Allocation Graph Scheme

* Claim(申請) edge Pi → Rj ，表示行程Pi要求資源Rj，以虛線表示
* 當一個行程要求一個資源，claim edge轉換成request edge
* 當資源被分配到一個行程，request edge轉換成assignment edge
* 當行程釋放資源（結束），assignment edge轉換成claim edge



Banker’s Algorithm

n：Process的數量

m：Resource Types的數量

* Available[m]：表示每個Resource type的instances有幾個

Avalible[j]=k → Resource type Rj有k個instances

* Max[n,m]：定義各行程所需的最大需求量

Max[i,j]=k → 行程Pi最多可要求k個Rj Resource type的instances

* Allocation[n,m]：定義每個行程所占用之各Resource type的數量

Allocation[i,j] = k → 行程Pi現已占用Resource type Rj中k個instances

* Need[n,m]：表示每一個行程的剩餘需求量

Need[i,j]=k → 行程Pi還需佔用Resource type Rj中的k個instances

Need [i,j] = Max [i,j] – Allocation [i,j]

Banker’s Algorithm – Safe State Checking

1. 令Work和Finish分別為長度m和n的向量，初始為：

* Work[i]=Available[i]，表示每個資源目前可用的實例
* Finish[i]=false，表示行程Pi已完成

1. 尋找符合下面兩個條件的行程Pi
   * Finish[i]=false
   * Need[i]<=Work

若沒有符合條件的行程，跳到step 4

1. Work = Work + Allocation[i]

Finish[i]=true

Go to step 2

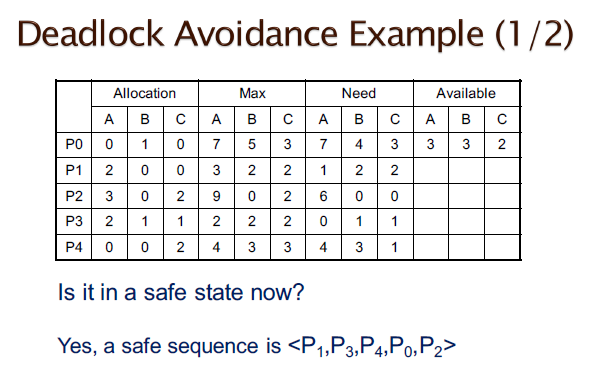
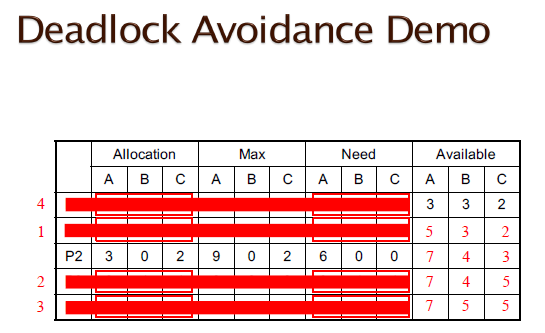
1. 若Finish[i]=true for all i，則系統處於safe state

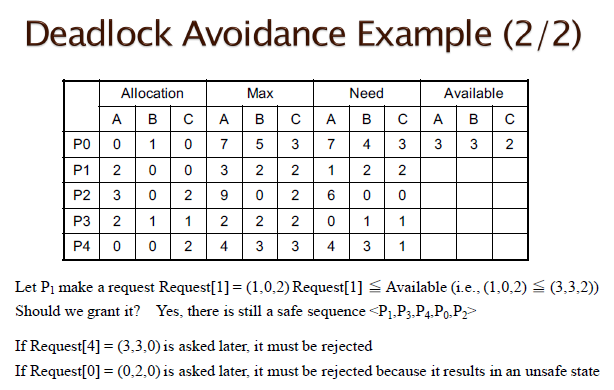
Banker’s Algorithm – Resource-Request Algorithm

Request[i]是行程Pi的需求向量

Request[I,j]=k → 行程Pi需要k個resource type Rj的instances

1. 如果Request[i]<=Need[i]，go to step 2，否則發生錯誤，因為行程已超過最大需求
2. 如果Request[i]<= Available，go to step 3 ，否則Pi必須等待
3. 若行程Pi以配置到資源則
   * Available = Availabe – Request[i]
   * Allocation[i] = Allocation[i] + Request[i]
   * Need[i] = Need[i] – Request[i]
4. 執行”Safe State Checking”，如果系統狀態是safe，則配置資源給Pi，若不是，則Pi必須等待，且恢復舊有的資源配置狀態



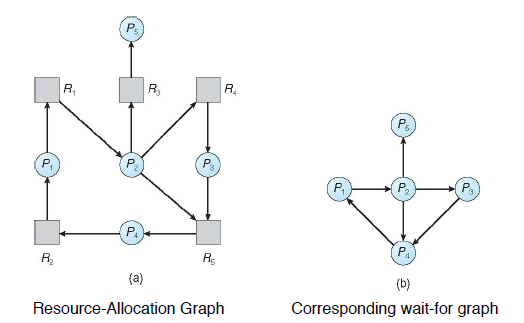


Deadlock Detection

* Approach：允許系統進入死結狀態
* Thus, we need：
  + 檢測的演算法
  + 回復的方案

Single Instance of Each Resource Type

* Wait-for graph
  + Nodes are processes
  + Pi → Pj ：Pi 正在等待 Pj
* 週期性的使用演算法來尋找圖中的cycle
* 如果圖中含有循環，則系統必有死結



Single Instance of Each Resource Type

n：Process的數量

m：Resource Types的數量

* Available[m]：表示每個Resource type的instances有幾個

Avalible[j]=k → Resource type Rj有k個instances

* Allocation[n,m]：定義每個行程所占用之各Resource type的數量

Allocation[i,j] = k → 行程Pi現已占用Resource type Rj中k個instances

* Request[i]是行程Pi的需求向量

Request[I,j]=k → 行程Pi需要k個resource type Rj的instances

1. 令Work和Finish分別為長度m和n的向量，初始為：

* Work[i]=Available[i]，表示每個資源目前可用的實例
* Finish[i]=false，表示行程Pi已完成

1. 尋找符合下面兩個條件的行程Pi
   1. Finish[i]=false
   2. Need[i]<=Work

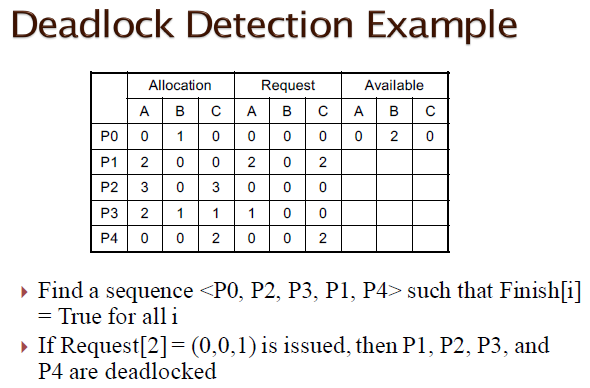
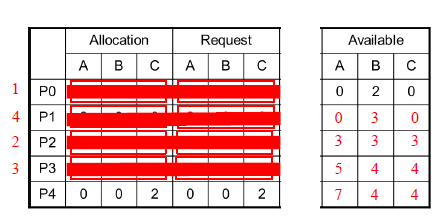
若沒有符合條件的行程，跳到step 4

1. Work = Work + Allocation[i]

Finish[i]=true

Go to step 2

1. 若Finish[i]=False for some Pi，則系統處於死結狀態，Pi陷入死結狀態



Detection-Algorithm Usage

應該在何時使用偵測演算法，取決於兩個因素？

* 多久會產生一次死結
* 有多少個行程受死結的影響

Recovery from Deadlock：

Process Termination

* 取消所有死結的行程
* 一次取消一個行程直到死結循環被消除為止
* 選擇取消哪個行程的因素：
  + 行程優先權
  + 此行程以工作多久，還需要運算多久才能完成
  + 行程已用了什麼資源
  + 行程還需要多少資源才能完成
  + 有多少行程需要被終止
  + 行程為交談式還是批次

Resource Preemption

* 選擇犧牲者（Selecting a victim）–最小成本
* 回撤（Rollback）–回到某一安全狀態，從這狀態重新開始
* 飢餓（Starvation）–有些行程可能永遠被選為犧牲者，為防止飢餓發 生，將撤回的次數包括在成本因素中