CH07-Deadlock

Deadlocks(死結)

* 當一個佇列裡的行程皆需等待來自另一個行程的一個事件發生才能夠執行時，就會造成死結
* System Model
  + 系統包含有限資源
  + 資源可分成許多種型式(type)
    - Ex：CPU、記憶體空間、輸入/輸出裝置
    - 每個型式(type)都有例證(instance)

→一個系統有兩個CPU，則CPU這種形式的資源有兩個instances

* + 每個行程要依據下列順序使用資源
    1. Request
    2. Use
    3. Release

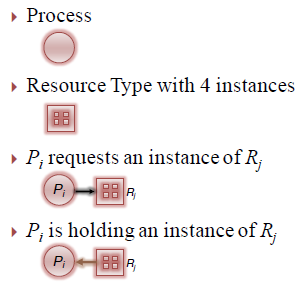
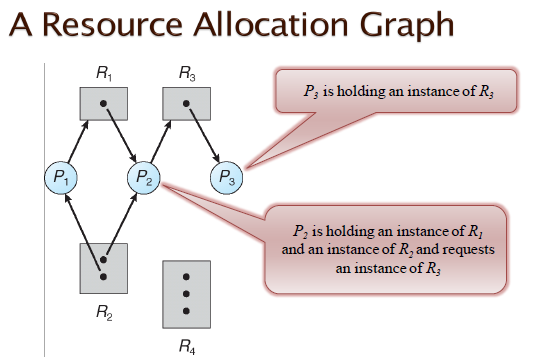
Deadlock Characterization(死結的特性)

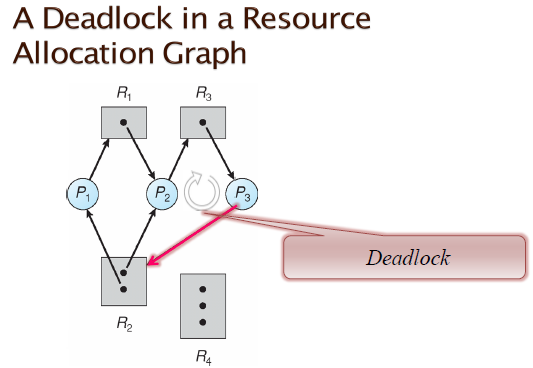
死結可能會發生在以下四種狀況同時成立：

* Mutual exclusion(互斥)：一次只能有一個行程可用此資源
* Hold and wait(占用與等候)：必須存在一個至少已佔用一個資源，且正等候其他行程已佔用另外資源之行程
* No preemption(不可搶先)：資源只能在被佔用它的行程結束後，才能釋放
* Circular wait(循環式等候)：存在一等候行程{P0,P1,…,Pn-1}，其中Pi在等待P((i+1)%n)所占用的資源

Resource-Allocation Graph(系統資源配置圖)

* 由一組頂點V及一組邊E所組成
* V分為兩個不同的集合
  + P = {P1, P2, …, Pn}，系統中所有正在執行之行程的集合
  + R = {R1, R2, …, Rm}，系統中所有資源型式的集合
* E分為兩個不同的種類
  + Request edge：Directed edge(有向邊) Pi → Rj
  + Assignment edge：Directed edge(有向邊) Rj → Pi

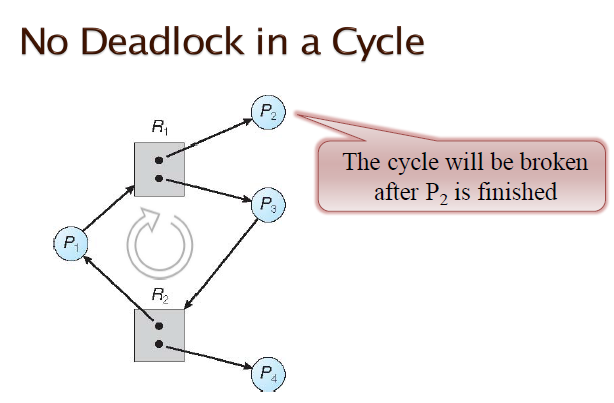


兩循環：

P1->R1->P2->R3->P3->R2->P1

P2->R3->P3->R2->P2

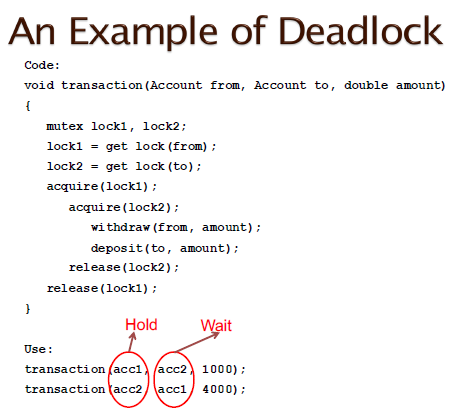
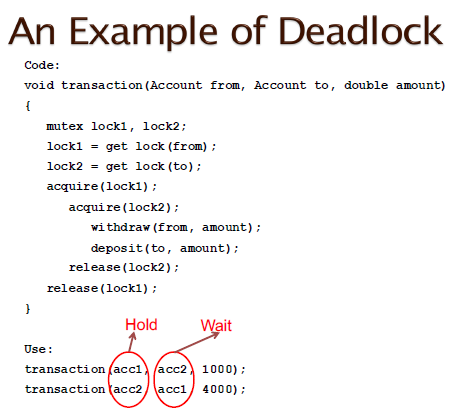
行程P1、P2、P3產生死結，P2在等P3釋放資源R3，P3在等P1或P2釋放資源R2，P1在等P2釋放資源R1。



循環：

P1->R1->P3->R2->P1

P2結束後會釋放一個source的instance，可供P1要求使用，P4也行釋放資源供P3使用



Method for Handling Deadlocks(處理死結的方法)

* 保證系統不會進入死結狀態
  + Deadlock Prevention：必免必要條件(4種)都發生
  + Deadlock Avoidance：要求作業系統預先取得行程要要求的那些資源
* 允許進入死結狀態，偵測出來再想辦法回復
  + 偵測死結
  + 修復
* 忽視此問題假裝沒發生過死結
  + 若系統看似進入死結狀態或停止運作則重新啟動
  + 但系統可能只是暫時凍結

Deadlock Prevention

* 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

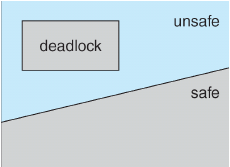
* 目標：
* 動態偵測資源的分配，確保沒有circuit-wait情況發生
* Keep the system at a **safe state**
* 要求時系統要有些附加資訊

對於每個資源

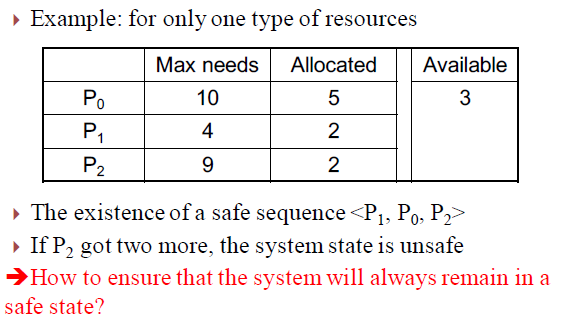
* 計算分配的資源
* 紀錄可用的資源

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

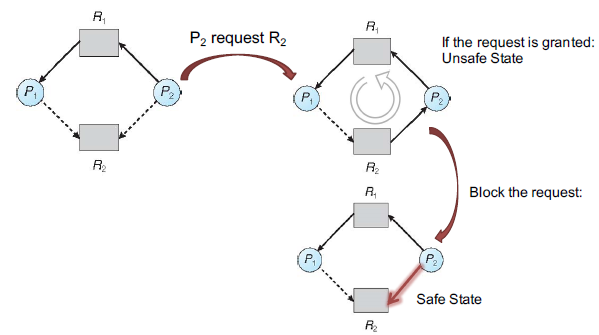
* 系統處於safe state → no deadlocks
* 系統處於unsafe state → possibility of deadlock
* Avoidance→確保系統永遠不會進入unsafe state
* 如果所有行程有safe sequence則系統處於safe state
* 一個序列 <P1, P2, …, Pn>，對每一個Pj（j<i），若Pi所要求的資源能被所有目前可用的資源與Pj所占有的資源所滿足，則此序列為safe sequence
* 也就是說：當Pj結束後Pi可以獲得它需要的資源，執行完並釋放資源，當Pi結束時，Pi+1即可得到所需的資源。



* + - 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

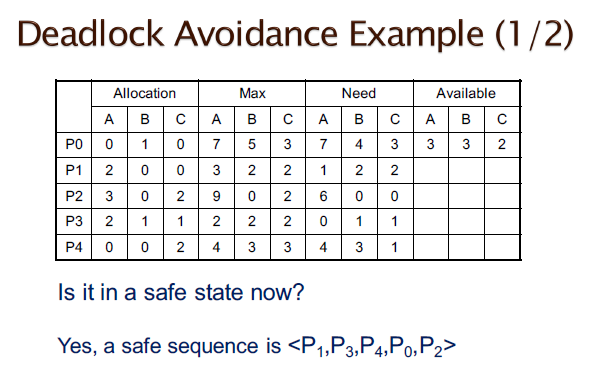
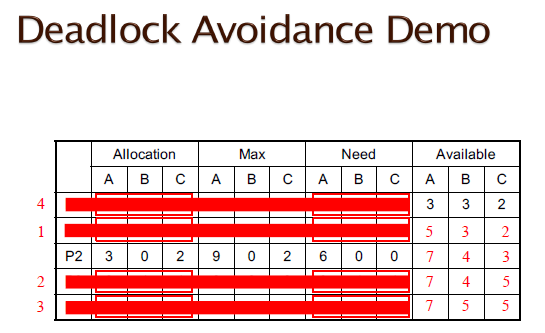
1. 若Finish[i]=true for all i，則系統處於safe state

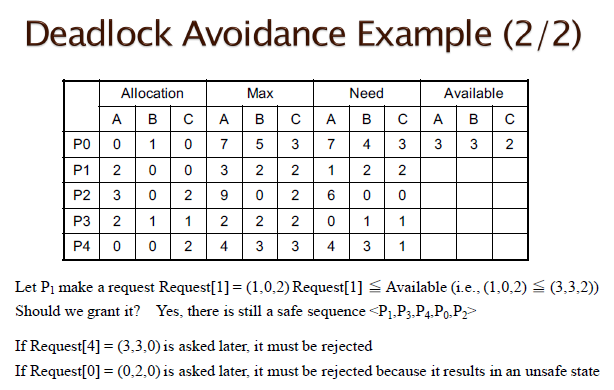
Banker’s Algorithm – Resource-Request Algorithm

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



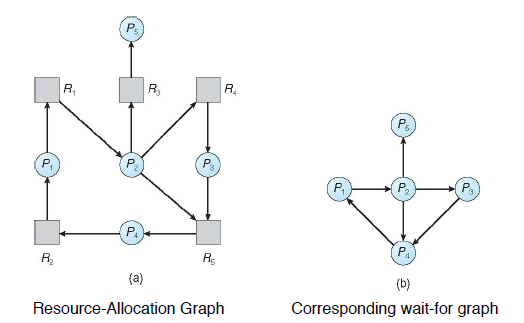


Deadlock Detection

* 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

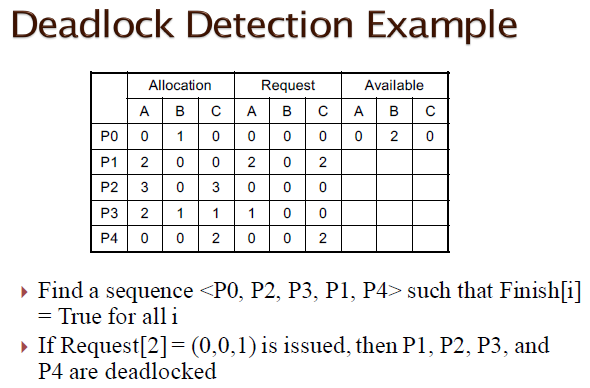
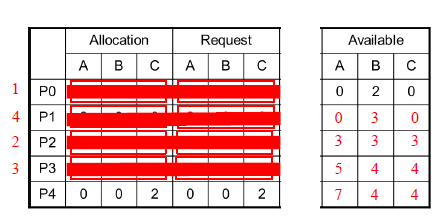
若沒有符合條件的行程，跳到step 4

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