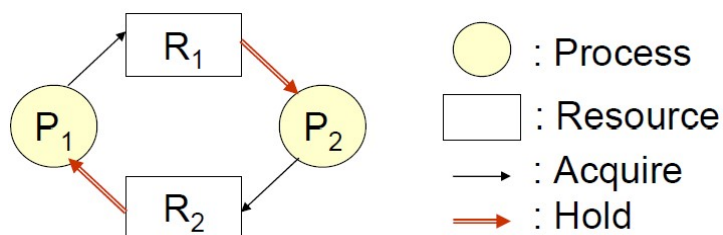


死結

Dead Lock	1
Dead Lock & Starvation	1
Dead Lock 形成的 4 個必要條件	2
Mutual exclusion 互斥	2
Hold and wait 持有並等待	2
No preemptive 不可搶先	2
Circular waiting 循環式等候	2
死結的處理方法	2
Dead Lock Prevention	2
Dead Lock Avoidance	3
Dead Lock Detection & Recovery	9
資源配置圖	11
每個資源都是 Single Instance 的 Dealock Avoidance	11
每個資源都是 Single Instance 的 Dealock Detection	12
Single Instance 的 Deadlock 偵測	12

Dead Lock

- 系統中存在一組 Process 陷入互相等待對方擁有的資源，造成所有的 Process 都無法往下執行，使 CPU 利用率及產能大幅降低
- 案例
 - R_1 配置給 P_2 ， R_2 配置給 P_1
 - R_1 要求 P_1 ， P_2 要求 R_2
 - 這兩個行程在執行過程中互相要求對方使用中的資源 --> 死結



Dead Lock & Starvation

Dead Lock	Starvation
單一組 Process 行程 Circular Waiting	單一或少數 Process 長期無法取得資源，但其他 Process 仍可以正常運作
CPU 利用率和產能大幅下降	CPU 利用率和產能不一定會大幅下降
易發生在 Non-Preemptive	易發生在 Preemptive

Dead Lock	Starvation
	解決方法 Aging 技術

Dead Lock 形成的 4 個必要條件

Mutual exlusion 互斥

- 某些資源在同一時間點，最多只能被一個 Process 使用，其他資源必須等待
- e.g. 印表機、CPU
- 反例：Read-Only File

Hold and wait 持有並等待

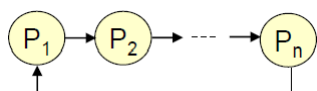
- Process 持有部分資源，且在等待其他 Process 所持有的資源

No preemptive 不可搶先

- Process 不可以任意搶奪其他 Process 所持有的資源

Circular waiting 循環式等候

- 存在一組 Process $\{P_0, P_1, P_2 \dots P_n\}$ ， P_0 等待 P_1 持有的資源， P_1 等待 P_2 持有的資源， P_n 等待 P_0 持有的資源，形成 Circular Waiting



- 死結不會發生在 Single Process 環境中

死結的處理方法

Dead Lock Prevention

- 打破四個必要條件其中之一
 - **Mutual Exclusion**
 - 很困難!!
 - **No Preemptive**
 - 允許 Process 可以搶奪其他 Waiting Process 持有的資源 (Preemptive)
 - 只會有 Starvation，不會有 Dead Lock
 - **Hold and Wait**

- 規定 Process 可以一次取得工作所需的全部資源，才允許 Process 持有資源，否則不准持有任何資源 --> **系統產能低**
- 在執行之初 Process 可以持有部分資源，但若要在申請資源之前，必須先釋放手中所有的資源 --> **系統產能低**

◦ **Circular Waiting**

- OS 須採取下列措施
 - 每個不同類型的資源都有獨一無二的**資源編號** (Unique ID)
 - Process 必須按照**資源標號遞增**的方式提出申請
 - e.g.

Process 持有資源	申請資源	結果
R1, R2	R5	OK
R3	R1	Denied, 需先釋放 R3
R1, R2, R6	R4	Denied, 需先釋放 R6

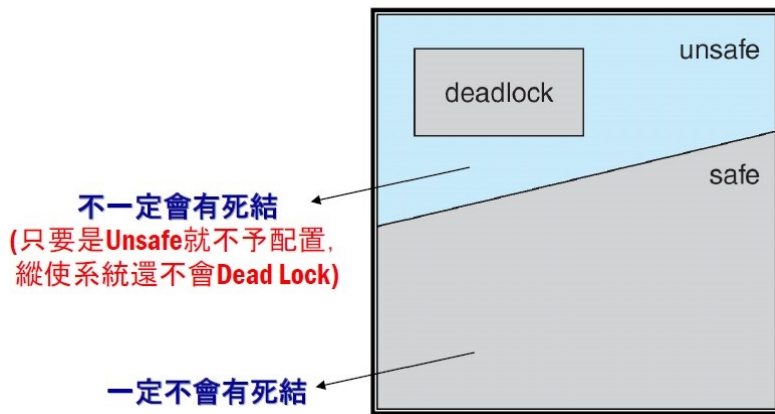
- **證明**
 - 另一組行程 $P_0 \sim P_n$ 分別持有 $r_0 \sim r_n$ 且資源類型皆不同 (編號不同)
 - 根據地曾申請規則, $r_0 < r_1 < \dots < r_n < r_0$
 - $r_0 < r_0$ 矛盾, 假設不成立

Summaries of Dead Lock Prevention

- **優點:** 系統絕對不會有死結
- **缺點:** 資源使用率低、產能低

Dead Lock Avoidance

- 當 Process 提出資源申請, OS 會根據下列資訊執行銀行家演算法, 判斷: 假設核准申請後是否處於 **Safe State**
 - 申請資源數量
 - 各 Process 目前所持有的資源數量
 - 各 Process 上需要之資源需求量
 - 系統目前可用資源數量



銀行家演算法

- 包含 Safety Algorithm

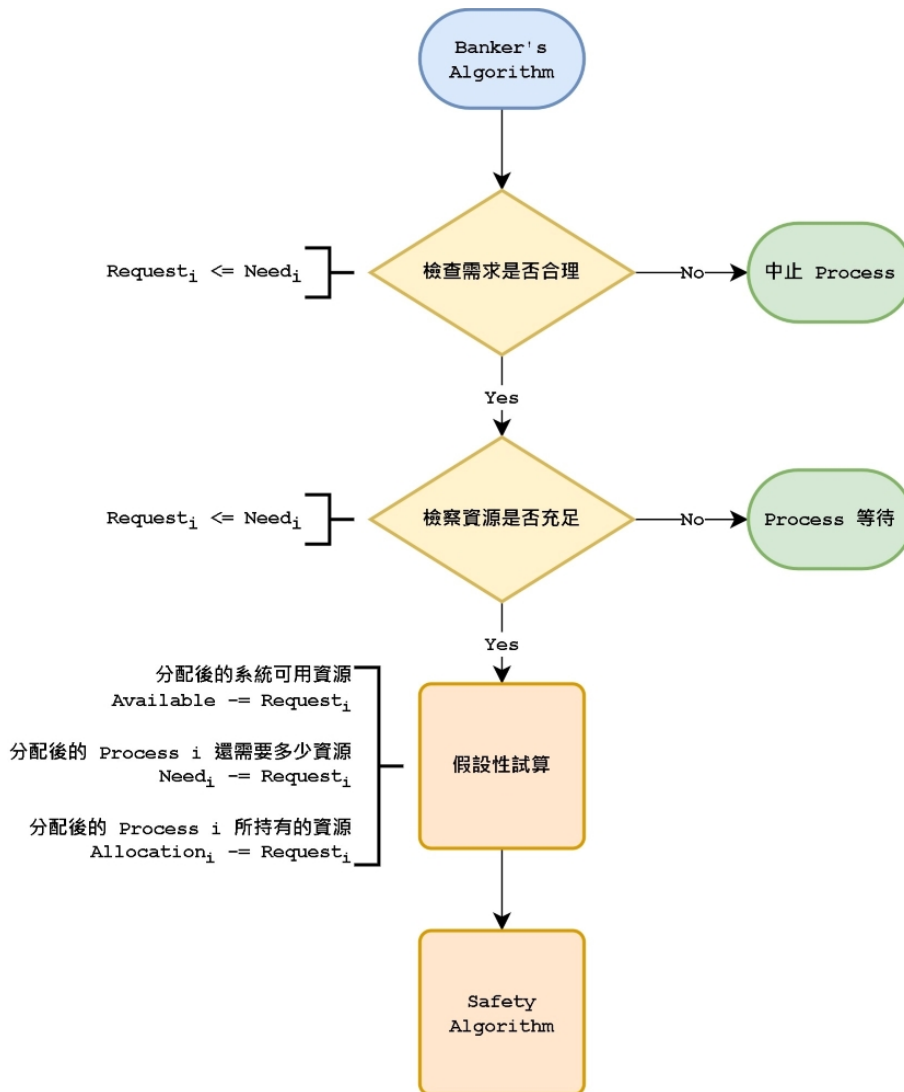
資料結構

- 假設目前系統有 n 個 Process, m 種類型的資源
- $Request_i[1...m]$
 - Process i 所提出的資源申請量
 - $Request_i[j] = k$ 表示 Process i 欲申請 k 個 j 類型資源
- $Allocation[1...n, 1...m]$
 - 表示各 Process 目前持有的各類資源數量
 - $Allocation[i, j] = k$ 表示 Process i 目前持有 k 個 j 類型資源
- $Max[1...n, 1...m]$
 - 表示各 Process 需要哪些資源, 且需要多少數量才能完成工作 (最大需求量)
 - 若 $Max[i, j] = k$ 表示 Process i 最多需要 k 個 j 類型資源才能完成工作
- $Need[1...n, 1...m]$
 - 表示 Process 目前還需要多少數量的資源才能完成工作
 - $Need[i, j] = k$ 表示 Process i 還需要 k 個 j 類型資源才能完成工作
 - $Need_i = Max_i - Allocation_i$
- $Available[1...m]$
 - 系統目前各類資源可用數量
 - $Available = \text{系統資源總量} - Allocation$

步驟

1. 檢查 $Request_i \leq Need_i$
 - 檢查所提出的需求是否合理
 - 若不成立 OS 會視為 illegal, 中止 Process
 - 若成立則 go to 2
2. 檢查 $Request_i \leq Available$
 - 檢查系統是否有足夠資源

- 若不成立 Process 必須等待到資源足夠
 - 若成立則 go to 3
3. **(假設性試算)** 假設系統分配資源給提出申請之 Process，透過以下數值進行**安全演算法**
- $Available = Available - Request_i$
 - $Need_i = Need_i - Request_i$
 - $Allocation_i = Allocation_i + Request_i$
4. Safety Algorithm 若系統判斷會是 Safe State 允許申請，否則拒絕此次申請，稍後重新申請



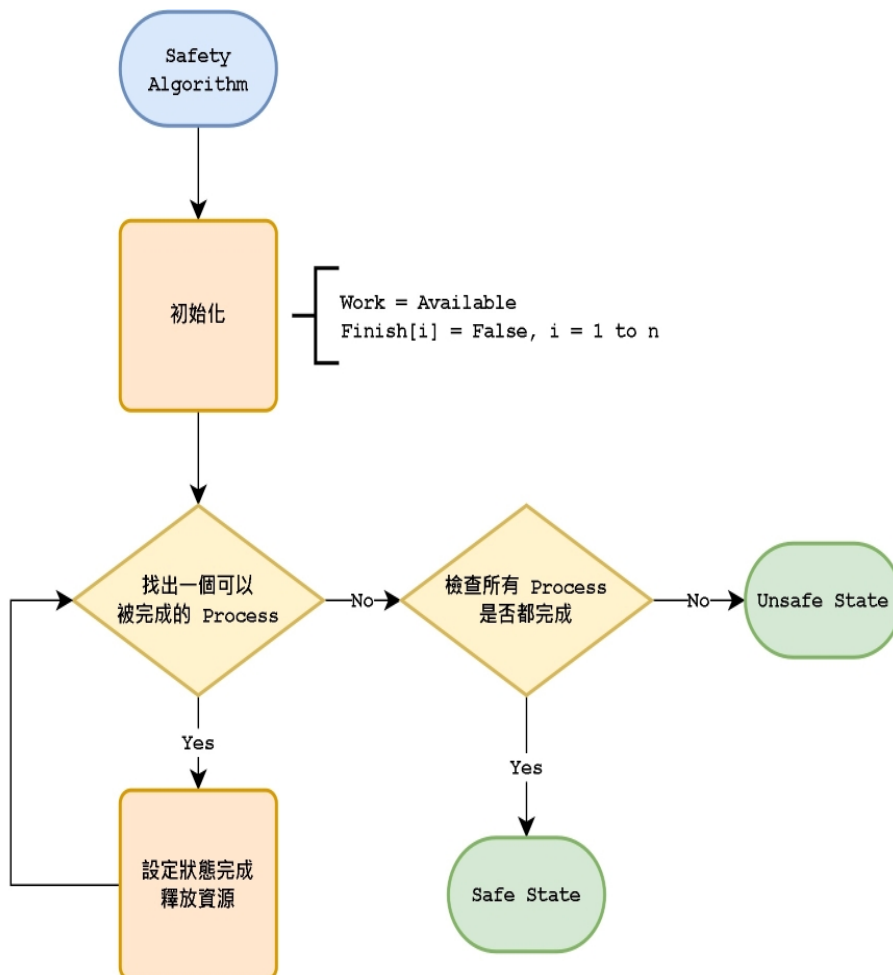
Safety Algorithm

資料結構

- $Work[1 \dots m]$
 - 當假定的配置資源後，目前系統可供作的資源數量統計
 - 初始值 = Available
- $Finish[1 \dots n]$ (Boolean)
 - $Finish[i]$ 表示 Process i 是否完成
 - 初始值 $Finish[i] = False, i = 1 \sim n$

步驟

1. 設定初始值
 - $Work = Available$
 - $Finish[i] = False, i = 1 \text{ to } n$
2. 找出一個 Process i 滿足
 - $Need_i \leq Work$ (可以完成工作)
 - $Finish[i] = False$ (還沒完成工作)
 - 若找到 go to Step 3
 - 沒找到 go to Step 4
3. 設定 $Finish[i] = True$ (狀態設定為完成)
 - $Work = Work + Allocation_i$ (釋放資源)
 - Go to Step 2
4. 檢查 $Finish$ 陣列，若全部為 $True$ 則系統處於 $Safe \text{ State}$ ，否則處於 $Unsafe \text{ State}$



- 若可以照出至少一組 Process 執行順序，讓所有 Process 完成，此順序稱為 **Safe Sequence**

範例

- 通常在實際情況下，下列資訊是已知
 - 資源總量
 - $Max[]$

- Allocation[]
- Request[]
- 上述資訊可以求出 Need 及 Available
- 假設系統內有 5 個 Process (P0 - P4) 及 3 種資源 A, B, C。其中 A 有 10 個, B 有 5 個, C 有 7 個
- 系統目前狀態如下表

	Allocation			Max		
	A	B	C	A	B	C
P0	0	1	0	7	5	3
P1	2	0	0	3	2	2
P2	3	0	2	9	0	2
P3	2	1	1	2	2	2
P4	0	0	2	4	3	3

- Need 及 Available 的內容?
- 若 P1 提出 Request1[1, 0, 2] 則系統是否核准? (Using Banker's Algorithm)

- 找各 Process 還需多少資源來完成工作 (Need[])
 - $Need = Max - Allocation$

	Allocation			Max			Need		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3
P1	2	0	0	3	2	2	1	2	2
P2	3	0	2	9	0	2	6	0	0
P3	2	1	1	2	2	2	0	1	1
P4	0	0	2	4	3	3	4	3	1

- 求系統目前各類資源的可用數量 (Available[])
 - 各類資源總量: $A = 10, B = 5, C = 7$
 - 目前各資源被 Process 持有的數量分別為: $A = 0+2+3+2+3 = 7, B = 1+0+0+1+0 = 2, C = 0+0+2+1+2 = 5$
 - 因此系統目前還可以提供各類資源總量分別為: $A = 10-7 = 3, B = 5-2 = 3, C = 7-5 = 2$
- P1 提出 Request1[1, 0, 2]。利用銀行家演算法
 - 檢查 Requesti 是否小於等於 Needi 若成立 go to 2
 - Request1 = [1, 0, 2], Need = [1, 2, 2] **成立**
 - 檢查 Requesti 是否小於等於 Available 若成立則 go to 3
 - 系統目前的 Available = [3, 3, 2] **成立**
 - (假設性試算)
 - $Available = Available - Request1 = [3, 3, 2] - [1, 0, 2] = [2, 3, 0]$
 - $Need1 = Need1 - Request1 = [1, 2, 2] - [1, 0, 2]$

- = [0, 2, 0]
 - Allocation1 = Allocation1 + Request1 = [2, 0, 0] + [1, 0, 2]
 - = [3, 0, 2]
 - 執行 Safety Algorithm
 - Safety Algorithm
 - 設定初始值
 - Work = Available = [2, 3, 0]
 - Finish[] = [F, F, F, F, F]
 - 找到 P1, 滿足 Finish[1] = False 且 Need1 ≤ Work, then go to 3
 - 設定 Finish[1] = True 且 Work = Work + Allocation1 = [5, 3, 2], then go to 2
 - 步驟 2 和 3 執行數次後, 可依序找到 P3, P4, P0, P2 接滿足 (此序列不唯一), 且當執行完 P2 後再重執行步驟 2 時, 會因不滿足要件而 go to 4
 - 檢查 Finish Array 皆為 True, 系統處於 Safe State → 核准 P1 申請
 - 上述推論找出一組 Safe Sequence: P1, P3, P4, P0, P2 (不唯一)

 - 接上題, 若 P4 再提出 [3, 3, 0] 之請求, 則是否核准?
 - 不核准
 - 當執行銀行家演算法的步驟 2 時, 會發現檢查 Request4 ≤ Available 不成立, P4 需要等待其他 Process 持有資源釋放
 - 若 P0 提出 [0, 2, 0] 之請求, 則是否核准?
 - 不核准
 - 當執行到 Safety Algorithm, 會發現 4 檢查 Finish 陣列並不皆為 True, 系統處於 Unsafe → 否決 P0 申請
-

- Banker's Algorithm
 - 優點避免系統發生死結
 - 缺點演算法需要時間複雜度 $O(m \times n^2)$
 - m: 資源種類數, n: Process 個數

Dead Lock Avoidance 的重要定理

- 假設系統包含 m 個單一種類的資源, 且被 n 個 Process 共用, 如果滿足下列兩個條件, 則系統無死結存在 (Dead Lock Free)
 - $1 \leq Max_i \leq m$
 - $\sum_{i=1}^n Max_i < m + n$
- 有 6 部印表機提供給 n 個 Process 使用, 每個 Process 最大需求量为 2, 再系統不發生 Dead Lock 情況下, 最多允許多少個 Process 在系統內執行? (求 n 最大值)

- **Ans**
 - 已知 $m = 6$, Max_i
 - 滿足條件 1
 - 欲滿足條件 2 則可得 $2n < 6 + n \rightarrow n < 6$
 - n 的最大值為 5

Dead Lock Detection & Recovery

- 若不用 Dead Lock Prevention 和 Avoidance，必須提供下列機制
 - 偵測死結是否存在
 - 若死結存在，則必須打破死結
- 優點
 - 資源利用率較高
 - 產能提升
- 缺點
 - Cost 太高
- Dead Lock Detection 和 Dead Lock Recovery 必須並存

Dead Lock Detection Algorithm

- 偵測所有 Process 是否會進入死結
- Data Structure Used
 - Available[1...m] 系統目前可用資源數量
 - Allocation[1...n, 1...m] 各 Process 目前持有資源數量
 - Request[1...n, 1...m] 各 Process 目前提出資源申請量
 - Work[1...m] 目前系統可用資源之累計
 - Finish[1...n]
 - If Allocation_i = 0, then Finish[i] = True
 - Process i 沒有持有任何資源，不會發生 **Hold and Wait**，假設可以完成工作
 - If Allocation_i ≠ 0, then Finish[i] = False
 - 尚未完成，且 Process 持有資源
- 處理步驟如下
 1. 設定 Work 和 Finish 初始值
 - Work = Available
 - Finish[i] 的初始值視 Process i 是否有資源而定
 - True, if Allocation[i] = 0
 - False, if Allocation[i] ≠ 0
 2. 找到一個滿足下列兩條件的 Process P_i，若找到則 go to 3，否則 go to 4
 - Finish[i] = False (還沒完成工作)
 - Request_i ≤ Work (系統可以滿足 Process 完成工作)

3. 設定 $Finish[i] = \text{True}$, $Work = Work + Allocation[i]$, go to 2
4. 檢查 $Finish$ 陣列
 - 若皆為 True ，則表示系統目前無死結
 - 若不是皆為 True ，則表示有 Dead Lock 且 $Finish[i] = \text{False}$ 者，皆陷入此 Dead Lock 中

範例

- 一個系統目前有五個處理程序 $P_0 - P_4$ 及三種資源 A, B, C
- A 資源 7 個裝置， B 資源 2 個裝置， C 資源 6 個裝置
- 假設在時間 T_0 時，系統分配狀態如下

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

- 求系統目前有無死結？
- **Solution**
 1. 設定初始值
 - $Work = Available = (0, 0, 0)$
 - $Finish = [F, F, F, F, F]$
 2. 找到 P_i 滿足 $Finish[i] = \text{False}$ 且 $Request_i \leq Work$
 - 找到 P_0 滿足 $Finish[0] = \text{False}$ 且 $Request_i \leq Work$
 - 先找 P_0 then go to 3
 3. 設定 $Finish[i] = \text{True}$ 且 $Work = Work + Allocation_i$
 - 設定 $Finish[0] = \text{True}$ 且 $Work = Work + Allocation_0 = (0, 0, 0) + (0, 1, 0) = (0, 1, 0)$ go to 2
 - 步驟 2 與 3 交換執行，可依序得到 P_2, P_1, P_3, P_4 滿足條件且完成工作
 4. 檢查 $Finish$ 陣列發現都為 True ，系統無 Dead Lock

延伸範例

- 假設在時間 T_0 時，系統的資源分配狀態如下

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	1			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			


- **Ans:** P_0 可完成工作，但 P_1, P_2, P_3, P_4 陷入死結

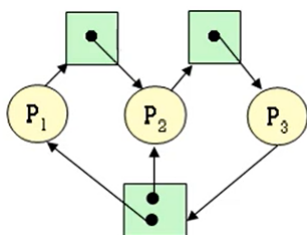
- **優點**可偵測出系統是否有死結，且可知那些 Process 懸入死結
- **缺點**時間複雜度 $O(n^2m)$ (n : Process 個數, m : 資源種類)

Dead Lock Recovery

- 中止 Process
 - Delete All
 - 成本太高
 - 每次只終止一個 Process 直到 Dead Lock 打破為止
 - 每刪一個 Process 後皆須執行 Dead Lock Detection Algorithm 判斷有無死結
 - 成本太高 (偵測、刪除都要成本)
- 資源搶奪
 - 程序
 - 選擇犧牲者 Process (Victim Process)
 - 剝奪其資源
 - 恢復到 Victim Process 原先無該資源的狀態 (困難)
 - OS 需要記錄每一個 Process 的每次資源使用狀況，成本高
 - 需考量 Starvation 問題
 - 把被剝奪的次數列入選擇犧牲者 Process 之考量因素

資源配置圖

- 令 $G = (V, E)$ 為一個有向圖
 - V 頂點集合 可分為兩類
 - Process: 
 - Resource: 
 - E 邊集合 可分為兩類
 - Request Edge (申請邊) 
 - Allocated Edge (配置邊) 
- 範例



- 若圖形沒有 Cycle 存在，則系統無死結
- 若系統中每一類資源為 **Multiple Instances (多個數量)**，則有 Cycle 存在不一定有死結
- 若每類資源皆為 Single Instance，有 Cycle 存在一定有死結

每個資源都是 Single Instance 的 Dealock Avoidence

- **Clain Edge** 宣告邊：表示 P_i 未來會對 R_j 提出申請 (目前尚未申請)



- 類似銀行家演算法的 Need 資料結構，表未來要完成 Process 需要申請的 Resource

每個資源都是 Single Instance 的 Dealock Detection

- 定義：由 Resource Allocation Graph 演變而來， $G = \langle V, E \rangle$

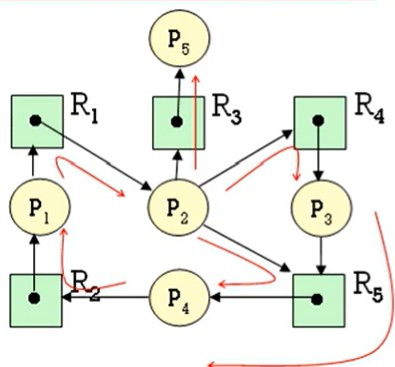
- 表示 Process_i 正在等待 Process_j 持有的資源

- 簡化

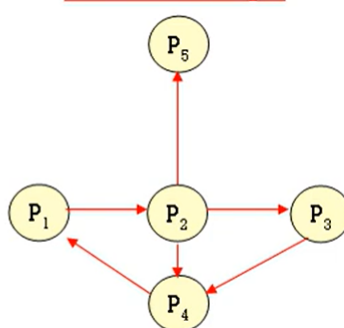
Single Instance 的 Deadlock 偵測

- 使用 Wait-For Graph
 - $G = (V, E)$ 有向圖
 - V 是 Process 組成
 - E 是 Wait-For Edge 表示 Process i 正在等待 Process j 所持有的資源

Resource Allocation Graph



Wait-for Graph



- OS 偵測到 Wait-For Graph 有 Cycle 存在，表示 Deadlock 存在
 - 上圖存在兩個 Cycle，系統有 Dead Lock
- 時間複雜度 $O(n^2)$