

CH5、Deadlock

死結的定義、預防與處理

目錄：

定義：成立的 4 個必要條件(例子)、與 Starvation 比較

死結的處理方式：1.Deadlock Prevention

2.Deadlock Avoidance (Banker's Algorithm)

3.Deadlock Detection & Recovery

4.Ignores it

定理

相關圖形： 1.Resource Allocation Graph(RAG) + 3 點結論

2.Clain Edge + RAG (for Avoidance)

3.Wait for Graph (for Detection)

死結 Deadlock

一、Def：System 中，存在一組 Processes 彼此形成循環等待之情況，造成這些 Processes 皆無法往下執行，並降低 Throughput 之現象

二、死結成立的 4 個必要條件(4 necessary condition)：即缺少一個，死結必不發生

1. 互斥存取(Mutual Exclusion)：

Def：這是對 Resource 而言，具有此性質的 Resource，在任何時間點，最多只允許一個 Process 持有/使用，不可多個 Processes 同時持有/使用

例：大多數的資源皆具有此性質：例：CPU、Memory、Disk、

Printer...。不過 Read only file 不算在內

2. 持有並等待(Hold & Wait)：

Def：Process 持有部分資源，且又在等待其他 Process 所持有的資源

3. 不可搶先(No Preemption)：

Def：Process 不可任意剝奪其他 Process 所持有的資源，必須等對方釋放資源後，才有機會取得資源

Note：若可搶先，必無 Deadlock，頂多只有 Starvation

4. 循環等待(Circular Waiting)：

Def：System 中存在一組 Process 形成循環等待之情況

Note：[恐]：4 implies 2；其他：4 implies 1、2、3

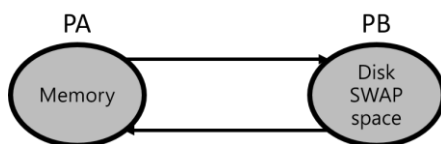
為何 Single Process 不會造成 Deadlock？

=> 因為 Circular waiting 必不存在，死結必不發生

例：(T/F) If these 4 conditions are true, then the deadlock will arise?

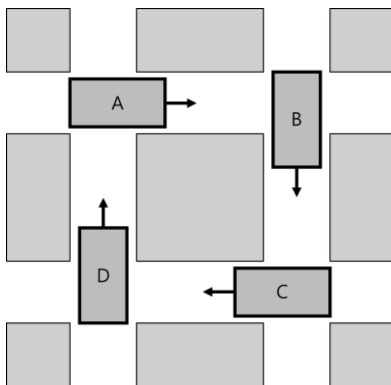
False

三、例 1：硬體的死結



例 2：交通十字路口的死結

路口 => 資源；車子 => Process (4 個條件都成立)



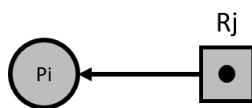
四、與 Starvation 比較：

Deadlock	Starvation
一組 Processes 形成 Circular Waiting，造成這些 Processes 皆無法往下執行，Waiting forever	Process 因為長期無法取得完工所需的各式資源，造成它遲遲無法完工。有完工的機會，但機會渺茫。Indefinite Blocking
會連帶造成 Throughput 低落	與 Throughput 高低無關連
有 4 個必要條件，其中一個一定是 "No Preemption"	容易發生在 "Preemption" 環境
解法有 Prevention、Avoidance、與 Detection & Recovery	採用 "Aging" 技術防止
相同點：皆與『資源分配管理機制設計不良』相關	

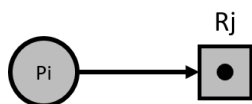
Resource Allocation Graph (RAG：資源分配圖)

一、Def：令 $G=(V, E)$ 有向圖，代表 RAG，其中：

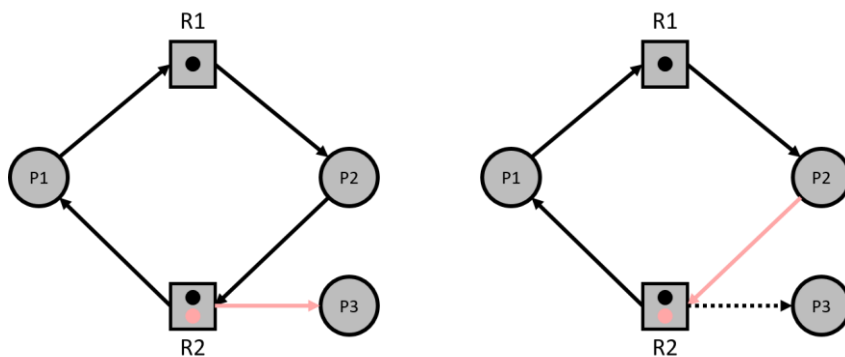
- Vector(頂點)有 2 類型：
 - (1) Process 以 O 表示
 - (2) Resource 以 \square 表示
- Edge(邊)分為 2 種：
 - (1) Allocation Edge：



- (2) Request Edge：



- (3) 例：



此圖雖然有 Cycle，但並無死結，因為 P3 必可完工，會釋放 1 個 R2，可配給 P2，此時的圖片將無 Cycle，因此 No Deadlock

二、RAG 的 3 點結論：

1. No Cycle，則 No Deadlock
2. 有 Cycle 不一定有 Deadlock
3. 除非(若)每一類型資源都是 Single Instance(單一數量)，則有 Cycle 必有 Deadlock

例 P5-29(6)：

例 P5-33(11)：

例 P5-53(39)：

例 P5-53(42)：

Deadlock 處理方式

1. Deadlock Prevention(預防)
2. Deadlock Avoidance(避免)
3. Deadlock Detection & Recovery(偵測&恢復)

[1]及[2]共同的：

優點：保證 System is Deadlock free (or never enters the deadlock state)

缺點： 1.對資源的使用/取得限制多：Resource utilization 偏低，連帶 Throughput 也降低
2.可能造成 Starvation

[3]

優點：資源 Utilization 相對較高，Throughput 也連帶較高

缺點： 1.System 有可能進入 Deadlock state
2.Detection & Recovery 之成本相當高

Deadlock Prevention

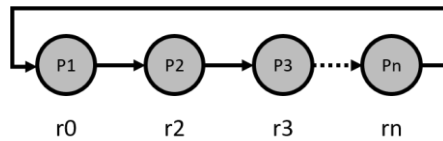
原則：破除 4 個必要條件之其中一個，則 Deadlock 必不發生

1. Mutual Exclusion：無法，因為 Resource”與生俱來 Inherent”之性質無法破除
2. Hold & Wait：
[法一]：規定：除非 Process 可一次取得全部所需資源，才准許持有資源，否則不得持有任何資源。
[法二]：規定：Process 可先持有部分資源，但當 Process 要申請其他資源時，必須先釋放持有的義部資源，才可提出申請
3. No Preemption：改為 Preemption 即可，例：Based on priority-level
4. Circular Waiting：方法叫『Resource ordering』
(1) OS 會賦予每一類型資源一個 unique(唯一的)Resource ID
(2) OS 會規定 Process 必須依照資源 ID Ascending(遞增)的方式對資源提出申請，例：

	持有的	欲申請	
1	R1	R3	申請核准(因為 R1->R3 為 ID 遞增)
2	R5	R3	需先放掉 R5，才可提出申請
3	R1, R5	R3	需先放掉 R5，才可提出申請

Why?

證明：假設在這樣的規定下，System 仍存在一組 Process 形成 Circular Waiting 如下

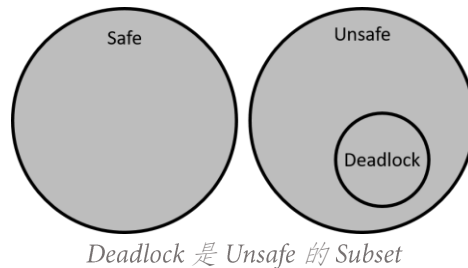


依規定，我們可以推導出 Resource ID 大小關係如下： $r_0 < r_1 < \dots < r_n < r_0$

竟然推出 $r_0 < r_0$ 此一矛盾式子，所以 Circular Waiting 必不存在

Deadlock Avoidance(避免)

Def：當某個 Process 提出某些資源申請時，OS 必須執行”Banker’s Algorithm”以確定倘若分配給 Process 其申請的資源後，System 未來處於 Safe state，若 Safe，則核准其申請；若是 Unsafe，則否決其申請，Process 需等一段時間後，才可以再次提出申請。Safe、Unsafe 與 Deadlock 的文氏圖如下：



Banker’s Algorithm

一、使用的 Data Structures：假設 n 是 Process 個數、 m 是 Resource 種類數

1. int : Request $[1:m] \Rightarrow P_i$ 提出之各式資源申請量
2. int matrix : Allocation $n*m \Rightarrow$ Process 目前的各式資源量
3. int matrix : MAX $n*m \Rightarrow$ 各式資源的最大數量
4. int matrix : Need $n*m \Rightarrow$ 各 Process 尚需要各式資源的數量才能完工
即：MAX – Allocation
5. int matrix : Available $[1:m] \Rightarrow$ System 目前可用的各式資源數量
即：資源總量 – Allocation

二、Procedure

1. Check Request \leq Need?
若成立，則 go to 2.，否則，終止 P_i (因為申請不合格)
2. Check Request \leq Available
若成立，則 go to 3.，否則， P_i waits until Resource available.
3. [試算]
 $Allocation_i = Allocation_i + Request_i;$
 $Need_i = Need_i - Request_i;$
 $Available_i = Available_i - Request_i;$

4. 依上述試算值，執行”Safety Algorithm”

若回傳”Safe” State，則核准 P_i 此申請

若回傳”Unsafe” State，則否決 P_i 之申請， P_i 需等一段時間，才能再重新提出申請

Safety Algorithm

一、Data Structure used：除了上述之外，另外加入：

1. int : Work [1:m] => 表示 System 目前可用 Resource 之累積數量
2. Boolean : Finish [1:m] => True 表 P_i 可以完工；False 表 P_i 無法完工

二、Procedure

1. 設定初值 Work = Available，Finish[i] 皆為 False ($1 \leq i \leq n$)
2. 看可否找到 P_i 滿足(1)Finish[i] 為 False，且(2)Need_i ≤ Work，若可以找到，則 go to 3.；否則 go to 4.
3. 設定 Finish[i]=True，且 Work = Work + Allocation_i，then go to 2.
4. Check Finish array，若皆為 True，則傳回”Safe” State，否則回傳”Unsafe” State

例(5-14)：5 個 Process $P_0 \sim P_4$ ，3 種 Resource A、B、C，資源量(A, B, C) = (10, 5, 7)：

	Allocation				MAX		
	A	B	C		A	B	C
0	0	1	0		7	5	3
1	2	0	0		3	2	2
2	3	0	2		9	0	2
3	2	1	1		2	2	2
4	0	0	2		4	3	3

1. 求 Need 及 Available?
2. P_i 提出(1, 0, 2)是否核准？Why?

1.

Need				Available		
A	B	C		A	B	C
7	4	3		3	3	2
1	2	2				
6	0	0				
0	1	1				
4	3	1				

2. 執行 Banker's Algorithm

(1)檢查 Request_i ≤ Need，成立。Go to (2)

(2)檢查 Request_i ≤ Available，成立。Go to (3)

(3)Allocation 1 = Allocation 1 + Request 1 = (2, 0, 1) + (1, 0, 2) = (3, 0, 2)

Need 1 = (1, 2, 2) - (1, 0, 2) = (0, 2, 0)

Available = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)

(4)執行”Safety Algorithm”

i. 初值：Work = Available = (2, 3, 0)

Finish = [F, F, F, F, F]

ii. 找到 P_1 ，滿足(i)Finish[i]=False、且(ii)Need₁ ≤ Work，go to iii.

iii. 設定 Finish[1]=True，且 Work=Work+Allocation 1 = (2, 3, 0)+(3, 0, 2) = (5, 3, 2)

Then go to ii.

ii. 找到 P3，滿足(i)Finish[]=False、且(ii)Need₃ ≤ Work，go to iii.
 iii. 設定 Finish[3]=True，且 Work=Work+Allocation 3= (5, 3, 2)+(2, 1, 1) = (7, 4, 3)
 Then go to ii.
 以此類推，P0、P2、P4 皆可完成，
 iv. 檢查 Finish=[T, T, T, T, T]，因為皆為 True，故回傳 Safe State
 因為 Safety Algorithm 回傳值為 Safe，所以此申請『核准』

何謂 Safe State?

Def：至少可以找到 ≥1 組 Safe sequence，讓系統依此順序分配 Process 所需資源，使所有 Process 皆可完工

例如上題：P1, P3, P0, P2, P4(可能有多種)，只是其中一組 Safe sequence

例 1：沿此目前資訊，若 P4 提出(3, 3, 0)申請，是否核准？

1. Check Request 4 ≤ Need 4 · (3, 3, 0) ≥ (4, 3, 1)，成立，go to 2.
2. Check Request 4 ≤ Available · (3, 3, 0) ≤ (2, 3, 0)，不成立，故無法核准(因為系統資源不足)

例 2：沿此目前資訊，若 P0 提出(0, 2, 0)申請，是否核准？

否，因為 Unsafe State

例 P5-65(53)：

例 P5-59(47)：

Banker's Algorithm 之 Time 分析

n：Process 數、m：Resource 種類

Banker's：

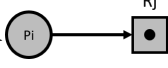
1. O(m)
2. O(n)
3. O(m)
4. Safety Algorithm：O(n²*m)
 - (1) Work：O(n)、Finish：O(n)
 - (2) 最多檢查 n+(n-1)+(n-2)+...+1=1/2 * n(n+1) => O(n²)
 - (3) 每次 check Need≤Work 花 O(m)，最多花 O(n²*m)
 - (4) O(n)

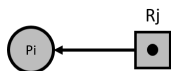
針對每一類型的 Resource 皆為 Single-Instance 情況下，有較簡易的 Avoidance 做法：

一、利用 RAG，搭配”Claim Edge(宣告邊)”使用

二、Claim Edge：代表 P_i 未來會對 R_j 提出申請(即表 MAX/Need 之意義)

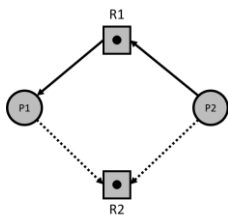
三、當 P_i 提出 R_j 申請後

1. 檢查有沒有若有，則 go to 2.；否則續止 P_i
2. Check R_j 是否 available，若是則 go to 3.；否則 P_i 等待(變申請邊)
3. [試算]：暫時把宣告邊改為配置邊

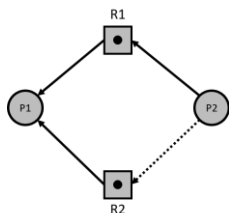


4. Check 圖中，是否有 Cycle 存在，若沒有，則為 Safe => 核准；
若有，則 Unsafe => 不核准

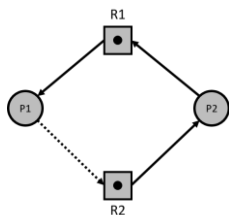
例：



1. 若 P₁ 提出 R₂ 之申請，是否核准？



2. 若 P₂ 提出 R₂ 之申請，是否核准？



[補充]Deadlock 是 Unsafe 之 Subset(或 Unsafe 有可能導致死結，也可能不會有死結)

1. 可能不死結：ex：P₁ 先釋放 R₁，然後才對 R₂ 提申請，此時無 Cycle
=> 無 Deadlock
2. 可能會死結：ex：P₁ 立刻對 R₂ 提出申請，RAG 有 Cycle，且 Resource 都為 Single-Instance => Deadlock

(重要)定理：

System 若有 n 個 Process， m 個 Resource 種類(單一種類)滿足下列 2 個條件：

1. $1 \leq \text{MAX}_i \leq m$
2. $\sum \text{MAX}_i < n+m$

則 System is Deadlock free

例 1：有 6 個 Printer 被 Process 使用，每個 Process 最多需要 2 部 Printers 才可完工，則 System 最多允許幾個 Process 執行，以確保為 Deadlock free 呢？

1. $1 \leq \text{MAX} < 6$ ，成立
2. $2n < n+6 \Rightarrow n < 6 \Rightarrow 5$ 個 Process(完全不會有 Deadlock)

若題目改為： $\text{MAX}=3$ ， $m=10$ ，則： $3n < n+10$ ， $n < 5 \Rightarrow n=4$ 個 Process

例 2：證明？

令資源全部配置給 Processes 了，即 $\sum \text{Allocation}_i = m$ 。依 Banker's Algorithm，可知：

$\sum \text{Need}_i = \sum \text{MAX}_i - \sum \text{Allocation}_i$ ，所以： $\sum \text{MAX}_i = \sum \text{Need}_i + m$

依條件 2： $\sum \text{MAX}_i < n+m$ ，代表至少有 ≥ 1 個 Process_i ，它們的 Need_i 為 0，即此 Process_i 必可順利完成工作。

另外，依條件 1： $1 \leq \text{MAX}_i \leq m$

所以 Process_i 完工後，必可釋放出 ≥ 1 個 Resources

(因為 $\text{MAX}_i \geq 1$ ，且 $\text{Need}_i = 0$ ，所以 $\text{Allocation}_i \geq 1$)，而這些收放出的 Resources 又必定可讓剩下的 Process 中，至少有 ≥ 1 個 Process 之 Need 為 0、可以完工、釋放 Resources。依此類推：所有 Process 皆可順利完工 \Rightarrow 保證無死結發生

Deadlock Detection & Recovery

一、如果放任 Resource 使用較無限制，雖然 Utilization 高，但 System 有可能進入 Deadlock 而不自知，所以需要一個 Deadlock 偵測演算法，及萬一偵測出有 Deadlock，如何破除 Deadlock(也就是 Recovery)的作法

二、Recovery 作法：

1. Kill process in the Deadlock：

[法一]：kill all processes in the Deadlock：成本太高、先前工作成果作廢

[法二]：kill process one-by-one：kill 一個 process 後，再跑偵測演算法，若仍有 Deadlock 存在，再重覆 kill。一樣是成本太高：且 loop 次數*偵測成本

2. Resource Preemption：

(1) 選擇"Victim" Process

(2) 剝奪他們身上的資源

(3) 回復此 Process 當初未取得此剝奪資源的狀態：困難、成本太高、也可能會 Starvation

Deadlock Detection Algorithm

一、Data Structure used : n : Process 數、m : Resource 種類

1. int matrix : Allocation $n \times m$
2. int : Available [1: m]
3. int : Work [1: m]
4. Boolean : Finish [1: n]
5. int matrix : Request $n \times m$: 各 Process 目前提出各式 Resources 的申請量

Note :

1. Avoidance(Banker's Algorithm) : 含有未來的資訊(MAX、Need)
2. Detection 只有現在的資訊

二、Procedures :

1. 初值設定 :
2. 看可否找到 P_i 滿足 :
3. 設定 $Finish[i] = True$
4. Check Finish array :

Time Complexity : $O(n^2 \times m) \Rightarrow$ 死結偵測一次成本很高、再加上乘以偵測頻率，總成本非常可觀

例 P5-18 : 偵測目前有無 Deadlock，若有的話，是哪些？

Allocation			Request			Available		
A	B	C	A	B	C	A	B	C
0	1	0	0	0	0	0	0	0
2	0	0	2	0	2			
3	0	3	0	0	0			
2	1	1	1	0	0			
0	0	2	0	0	2			

(1) $Work = Available = (0, 0, 0)$

$Finish[] = [F F F F F]$ ，所以 $Allocation \neq (0, 0, 0)$

(2) 因為可找到 P_0 ，滿足 $F[0]$ 為 False，且 $Require_0 \leq Work$ ，go to (3)

(3) 設定 $Finish[0] = True$ ，且 $Work = Work + Allocation_0 = (0, 1, 0)$ ，go to (2)

(2) 因為可找到 P_2 ，滿足 $F[2]$ 為 False，且 $Require_2 \leq Work$ ，go to (3)

(3) 設定 $Finish[2] = True$ ，且 $Work = Work + Allocation_2 = (3, 1, 3)$ ，go to (2)

(2)與(3)以此類推...

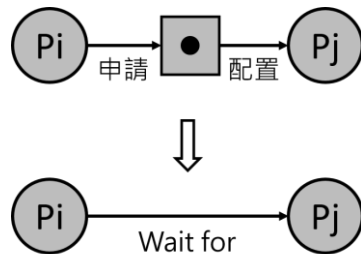
(4)檢查 $Finish$ ，因為皆為 True，故目前無死結存在

Wait-For Graph

若每種 Resources 皆為 Single-Instance，則有較簡化的”Detection”作法：使用”Wait-For” Graph

一、Def：令 $G=(V, E)$ 有向圖，代表 Wait-For Graph，其中：

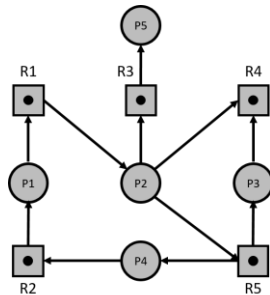
1. Vertex：只有 Process 頂點，沒有 Resources 頂點
2. Edge：稱為 Wait-For Edge



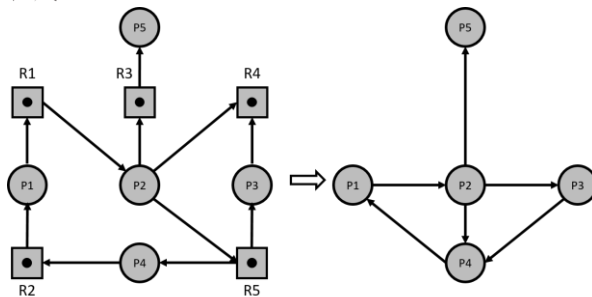
二、是從 RAG 簡化而來，即若 RAG 中，存在： $P_i \rightarrow R \rightarrow P_j$ ，則在 WFG 以： $P_i \rightarrow P_j$ 表示

三、偵測作法：在 WFG 中，若有 Cycle，則目前有 Deadlock，否則目前無 Deadlock

四、例：RAG 圖如下：



1. 化成 WFG



2. 目前有無 Deadlock(因為有 Cycle，故有 Deadlock)