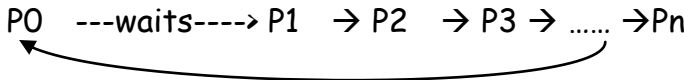


Chapter 5 Deadlock 及處理策略

● Deadlock vs. Starvation

	Deadlock	Starvation
Def.	系統中存在一組 Process，彼此形成"Circular Waiting"情況，導致 process 無法向下執行，CPU 及 System throughput 低落的結果	Process 因為長期無法取得完工所需資源，導致 Process 遲遲無法完工，形成"indefinite blocking"(無止盡等待)
Preemption 否?	必須是 NO Preemption	在 Preemption 環境下，容易不公平
CPU 使用率 & Throughput	CPU utilization 及 System throughput 低落	與 CPU, throughput 無關
解決方法	解決方法有四 Deadlock Prevention, Avoidance, Detection, Recovery.	採用 Aging 技術
相同處	(1) 皆為 Resource 分配機制設計不良 or 影響有關 (2) Process 皆有無窮止境的停滯現象	

● Deadlock 成立之四個必要條件(缺一個不可)

	Definition
Mutual Exclusion	資源在任何之時間點，最多只允許一個 process 使用，其他 process 必須等待，不可有多個 process 同時使用。 EX: CPU, Memory, Disk, Printer, etc. 皆具有互斥性質 EX: read-only File 不具互斥性質
Hold & Wait	又稱"Partial Allocation"，即 Process 目前持有部分資源且又在等待其他 process 所擁有的資源
No Preemption	Process 不可任意搶奪其他 process 所持有的資源，必須等到對方釋放後才可使用
Circular Waiting	一組 Process(如下)形成循環等待的情形  <pre> P0 ---waits----> P1 -> P2 -> P3 -> -> Pn _____ </pre>

● Deadlock 之處理

■ Deadlock Prevention

原則：打破四個必要條件之其中一個。

打破 what?	作法
Mutual Exclusion	做不到


	因為互斥是大多數資源與生俱來的性質，無法破除
Hold & Wait	<p>[協定一] 規定“Process 若無法取得全部資源，則不可持有任何資源，否則才可持有”</p> <p>缺: CPU utilization 低, 會經常 idle 不用</p> <p>[協定二] 允許 Process 建立之初先持有部分資源，若要再申請其他資源，必須先 release 所有的 resource，才可提出申請</p> <p>缺: 若 resource 是被 processes 交替使用則會多出許多釋放再提申請之程序, waiting time 拉長, throughput 亦不高</p>
No Preemption	<p>改成“Preemption”即可</p> <p>⇒ Process 可以搶奪其他 Waiting Process 所持有的資源(ex, 依優先權高低)</p> <p>⇒ No Deadlock, but maybe Starvation</p>
Circular Waiting	<p>Step1: 替每一類型的資源賦予一個 unique Resource ID</p> <p>Step2: 規定 Process 須依 Resource ID 遞增(Ascending)方式提出申請</p> <p>證明: 若在此規定下，依舊存在 Circular waiting</p> <p>$P_0 \longrightarrow P_1 \longrightarrow P_2 \longrightarrow \dots \longrightarrow P_n$</p> <p>且假定每個 process 所擁有的 resource 為 R_0, R_1, \dots, R_n</p> <p>則 $R_0 < R_1 < \dots < R_n$, 矛盾</p>
優點	保證系統不會發生死結
缺點	Resource utilization 不高, Throughput 自然無法提升

● Resource Allocation Graph

(一) Def. 令 $G = (V, E)$ 是一個有向圖，代表 R.A.G.，其中

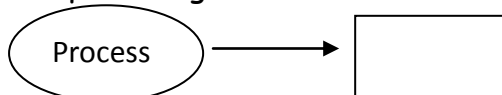
1. Vertex 集合由兩種頂點組合而成

i. Process: 以 ○ 表示

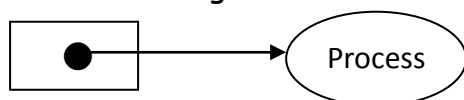
ii. Resource: 以  表示，其中數目代表 resource instance 的量

2. Edge 集合由兩類行組合

i. Request Edge



ii. Allocation Edge



(二) 重要論述

1. No cycle → No Deadlock

2. 若圖形中有 cycle 存在，則不一定有 Deadlock 發生

3. 若每類資源皆為“Single instance”，則若有 cycle → 必有 Deadlock

- Deadlock Avoidance

Def	<p>若某個 Process 提出資源申請，則 OS 必須依照下列資訊：</p> <p>(1) 此 Process 提出的申請量 (Request)</p> <p>(2) 各 process 目前持有各類型的資源數量 (Allocation)</p> <p>(3) 系統目前可用的資源數量 (Available)</p> <p>(4) 各 process 尚需多少資源量才能完工 Banker's Algo. (含 Safety Algo.)</p> <p>判斷若給予資源後，系統是否處於 Safe State，</p> <p>若 Safe，則批准資源</p> <p>否則(Unsafe)，就否決此次申請，Process 需等一段時間再重提申請</p> <p>✓ 優點：保證系統不會死結</p> <p>✓ 缺點：(1) Resource utilization 不高, Throughput 不高</p> <p>(2) 執行 Banker's Algo. 成本高</p>		
	Data Structures used	Procedures	NOTE
Banker' Algorithm	<p>1. Allocation : $[n \times m]$ 矩陣</p> <p>➤ 代表各 process 目前持有得各類型資源數量</p> <p>2. MAX : $[n \times m]$ 矩陣</p> <p>➤ 代表各 process 之最大資源量</p> <p>3. Request : $[1 \sim m]$ 一維陣列</p> <p>➤ 代表各 process 目前提出之資源申請量</p> <p>4. Need : $[n \times m]$ 矩陣</p> <p>➤ 代表各 process 尚需多少資源量才完工, $Need = MAX - Allocation$</p>	<p>Steps:</p> <p>[1] 檢查 $Request_i \leq Need_i$</p> <p>若是，則 goto [2]</p> <p>否則，終止 Process(因為不合理申請)</p> <p>[2] 檢查 $Request_i \leq Available$</p> <p>若成立，goto [3]</p> <p>否則，Process wait until 資源足夠</p> <p>[3] 若[1], [2]成立，則計算</p> <p>$Available = Available - Request_i$</p> <p>$Allocation_i = Allocation_i + Request_i$</p> <p>$Need_i = Need_i - Request_i$</p>	<ul style="list-style-type: none"> ● Safe State 即存在 ≥ 1 組 Safe Sequence，使得系統依照此 Process 順序分配其所需 Resource，讓所有 Process 皆完工 ● Banker's Algo. 之 Time Complexity <ul style="list-style-type: none"> ● $O(n^2 \times m)$ ● 定理： <p>若系統中有 n 個 Process 共享 m 個</p>

	<p>5. Available : [1~m]一維陣列</p> <ul style="list-style-type: none"> ● 代表系統目前可用資源量, Available = 資源總量 - Allocation 	<p>[4] 執行“Safety Algorithm”, 若傳回 Safe State, 則核准 P_i 此次申請, 否則, 否決 P_i 需等一段時間再提申請</p>	<p>resource(單一類型), 則要確保 System is Deadlock Free, 必須滿足以下兩個條件:</p> <p>(1) $1 \leq MAX_i \leq m$</p> <p>(2) $\sum_{i=1}^n MAX_i < n + m$</p>
Safety Algorithm	<p>1 ~ 5 與 Banker's Algorithm</p> <p>6. Work : [1~m]一維陣列 代表系統目前可用資源的累積數量</p> <p>7. Finish[i] : [1~n] of Boolean</p> <ul style="list-style-type: none"> ➤ Finish[i] = False : 表 P_i 尚未完工 True : 表 P_i 已經完工 	<p>Steps:</p> <p>[1] 設初值 Work = Available Finish[i] 皆為 False; $1 \leq i \leq n$</p> <p>[2] 找出 P_i 滿足 (i) Finish[i] = False 且 (ii) Need_i ≤ Work 若可以找到, 則 goto[3], 否則 goto[2]</p> <p>[3] 設定 Finish[i] = True 且 Work = Work + Allocation; (資源釋放出來) then goto [2]</p> <p>[4] 檢查 Finish, 若皆為 True, 則傳回 Safe, 否則, 傳回 Unsafe</p>	

※Compare the circular-waiting scheme with deadlock-avoidance scheme(like the Banker's Algo) with respect to the following issues

- Runtime overhead
- System throughput

Ans: a. 會增加 Runtime overheads, 因為在執行 deadlock avoidance 之過程中, 需對 Resource Allocation 監控, 而增加 overhead
b. deadlock avoidance allow more current resources than statically prevent the formation of deadlock. In this sense, the

deadlock avoidance scheme can increase the system throughput

- Deadlock Detection & Recovery

- 優點：resource utilization 相對較高

Throughput 也提升

- 缺點：1. 系統可能進入 deadlock state (會允許 resource cycle 的存在)
2. Cost 高

	Data Structure	Procedure
Detection Algo.	<p>n : process 個數 m : resource 種類數</p> <ol style="list-style-type: none"> 1. Allocation : [n*m] 2. Available : [1~m] 3. Request : [n*m]矩陣 <ul style="list-style-type: none"> ➢ 代表各 process 目前提出各類資源申請量 4. Work[1~m] 5. Finish[1~n] of Boolean 	<p>Steps:</p> <ol style="list-style-type: none"> 1. 設定初值 $Work = Available$ $Finish[i] = False$: if Allocation \neq False $True$: if Allocation = True 2. 找到 P_i 滿足 (i) $Finish[i] = False$ 且 (ii) $Request_i \leq Work$ 若可找到，則 goto 3. 否則 goto 4. 3. 設定 $Finish[i] = True$ 且 $Work = Work + Allocation_i$ 則 goto 2. 4. 檢查 Finish, 若皆為 True, 則表示系統無死結存在，否則，即有(且 $Finish[i]$ 為 False 者，陷入 deadlock) <p>NOTE: Time complexity : $O(n^2 * m)$</p>
Recovery	<pre> graph LR A[I. Kill Process] --> B[1. Kill All Process in the Deadlock] A --> C[2. Kill one process, then apply "Detection" Algo.] </pre> <p>I. Kill Process</p> <ol style="list-style-type: none"> 1. Kill All Process in the Deadlock → Cost 太高(先前工作皆白費) 2. Kill one process, then apply "Detection" Algo., 若死節還存在，則繼續做2. → 執行 Detection Algo. 的 Cost 仍高 	

II. Resource Preemption

Steps:

1. 選擇victim process
2. 剝奪其resources給其他process用
3. 將此process恢復成原先無此resource的 state (此步驟困難，Cost也高)

另外，也可能會有Starvation.

※what is the optimistic assumption made in the deadlock-detection algo? How could this assumption be violated?

Ans: the optimistic assumption is that there will not form any circular-waiting in term of resource allocated and processes making request for them. But actually, this assumption could be violated if circular-waiting indeed in practice