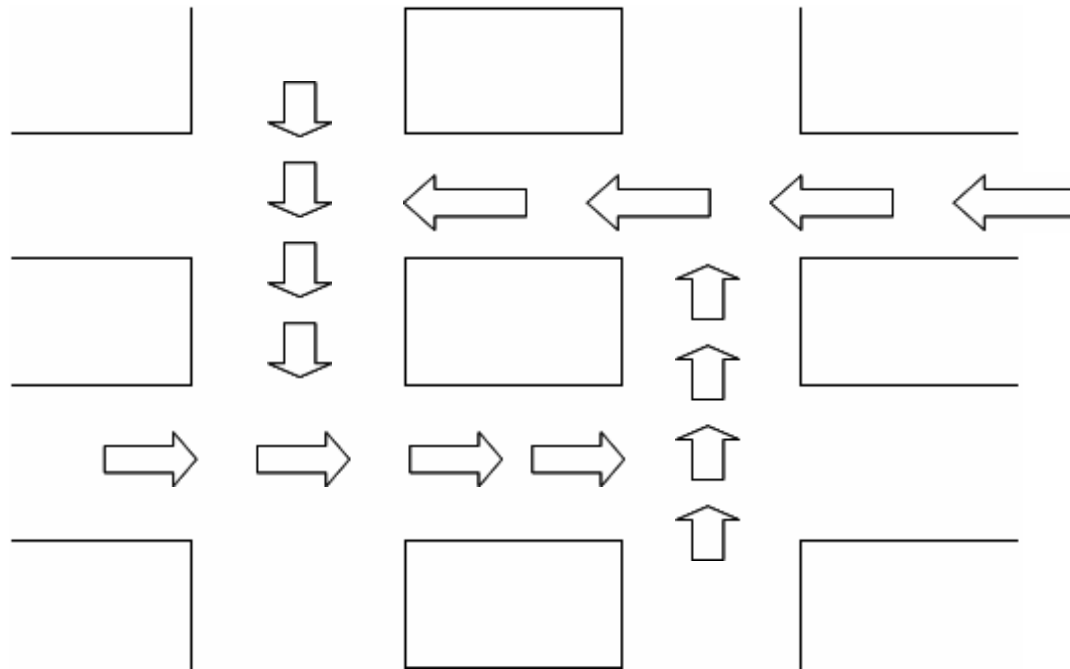


第七章 死結(Deadlock)

- 在日常生活中，我們亦有可能發生死結問題
 1. 十字路口車輛擁塞成死結的範例，因為大家都在等待對方空出通路



死結(Deadlock)(1)

- 死結是指，正在懸置狀態的處理元再也無法改變它的狀態，因為它所要的資源被另一個也在懸置狀態的處理元佔用。
- The waiting processes will never again change state, because the resources they have requested are held by other waiting processes.

死結(Deadlock)(2)

- 處理元發生死結，是指在此組處理元之間，每個處理元均等待該組其他處理元所引發的事件（Event），以便繼續執行，然而卻沒有處理元可以引發事件，以致造成所有該組處理元均無法繼續執行。由於該組處理元均無法繼續執行，**終究會餓死（Starvation）**。
- Every process in the set of processes is **waiting for an event** that can only be caused by another process in the set.

死結(Deadlock)(3)

- 必須至少有二個處理元才會發生死結；然而若我們將一個處理元看作是多個執行線所組成，則一個處理元內多個執行線有可能產生死結。
- 不特別區分處理元及執行線，而一律稱為處理元。

產生死結的四個必要條件 (Necessary Condition) (1)

7.1 死結的特性

- 互斥 (Mutual Exclusion)
 - 某個資源被一個處理元擁有並使用時，它具有獨佔性，其他處理元必須等待此資源被釋放 (Release) 後，才能競爭此資源。
- 擁有和等待 (Hold and Wait)
 - 處理元已經擁有至少一個獨佔性資源，但仍須等待使用其他處理元已經擁有的獨佔性資源。

產生死結的四個必要條件 (Necessary Condition) (2)

- 不可奪取 (Non-Preemption)
 - 若資源已經被某個處理元擁有，則其他處理元不可以強取此資源，而必須等待資源被正常釋放。
- 循環等待 (Circular Wait)
 - 一組處理元 P_0, P_1, \dots, P_n ，其中 P_0 正在等 P_1 所擁有的資源， P_1 正在等 P_2 所擁有的資源， \dots ， P_n 正在等 P_0 所擁有的資源，稱為循環等待。

死結四個必要條件-以哲學家問題為例(1)

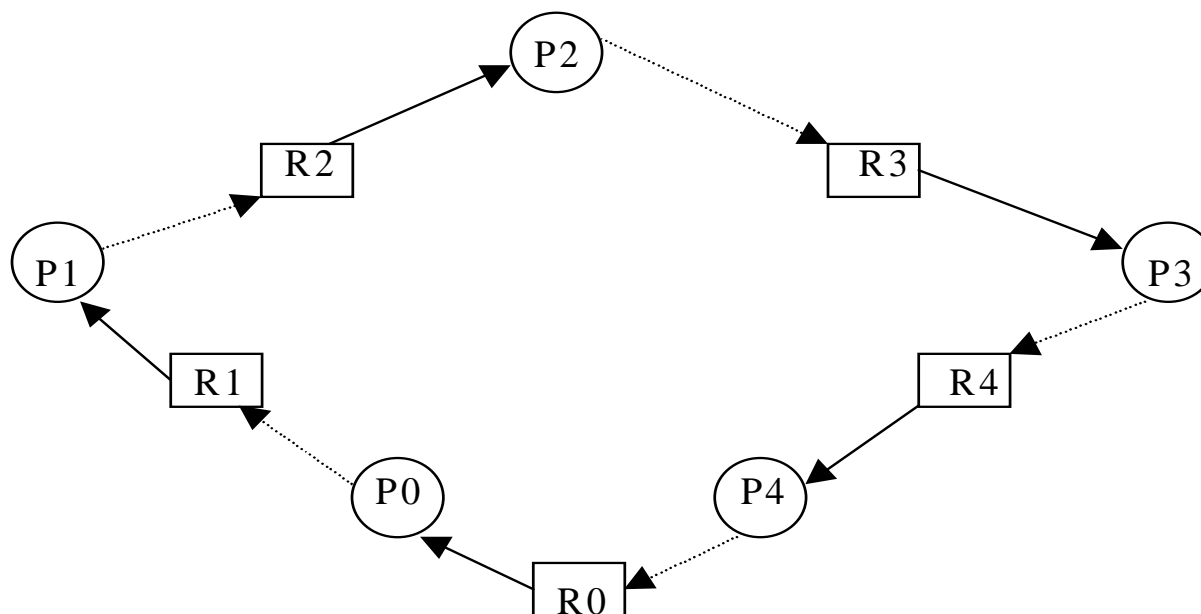
- 互斥
 - 筷子這個資源是互斥的，當一個哲學家擁有筷子，他不允許別的哲學家使用此根筷子。
- 擁有和等待
 - 當每個哲學家均擁有右邊的筷子，卻無法取得左邊的筷子，因為左邊的筷子已被相鄰的哲學家取走了！

```
wait(chopstick[i]);  
wait(chopstick[i+1 mod 5]);  
  
    eat;  
  
signal(chopstick[i]);  
signal(chopstick[i+1 mod 5]);  
  
    think;
```

死結四個必要條件-以哲學家問題為例(2)

- 不可奪取
 - 每個哲學家擁有右邊的筷子，卻堅持已經擁有的筷子不准被別的哲學家奪取。
- 循環等待
 - 哲學家0已經擁有第0號筷子，他等待哲學家1所擁有的第1號筷子，哲學家1等待哲學家2所擁有之第2號筷子，...，哲學家4等待哲學家0所擁有的第0號筷子，因此產生循環等待。

資源配置圖 (Resource Allocation Graph)(1)



資源配置圖(Resource Allocation Graph)(2)

- P1擁有R2並要求R1，P2擁有R1及R2並要求R3，P3擁有R3。並沒有形成循環圈，故沒有產生死結。
- P3又去要求R2，則形成死結，因為它們形成P1、R1、P2、R3、P3、R2的循環圈。

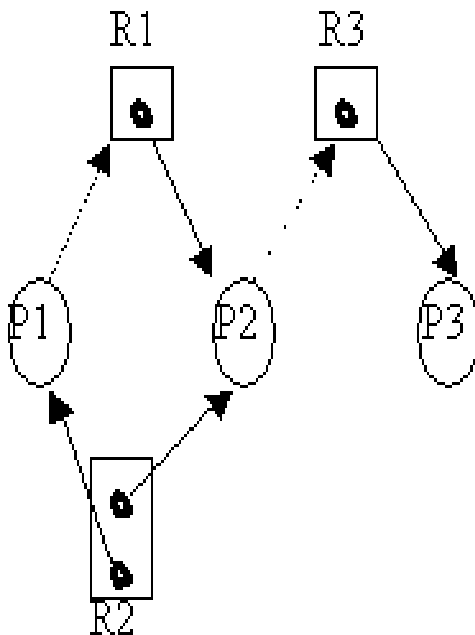


圖 6.4a 沒有形成死結

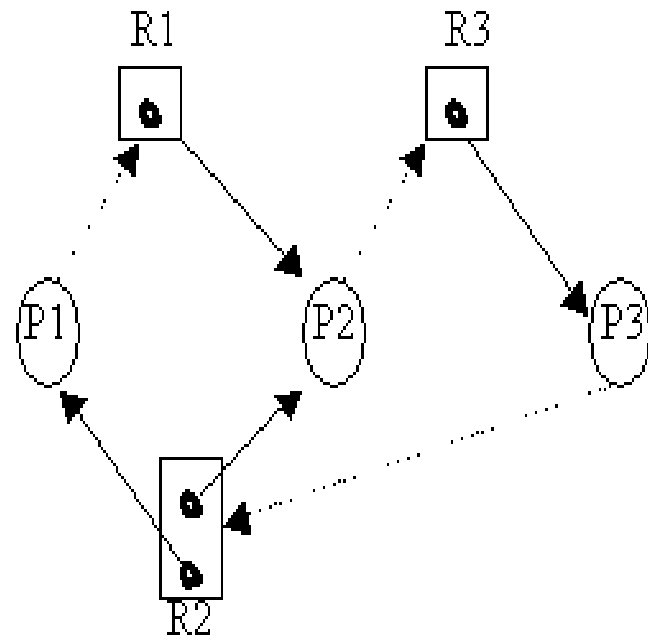
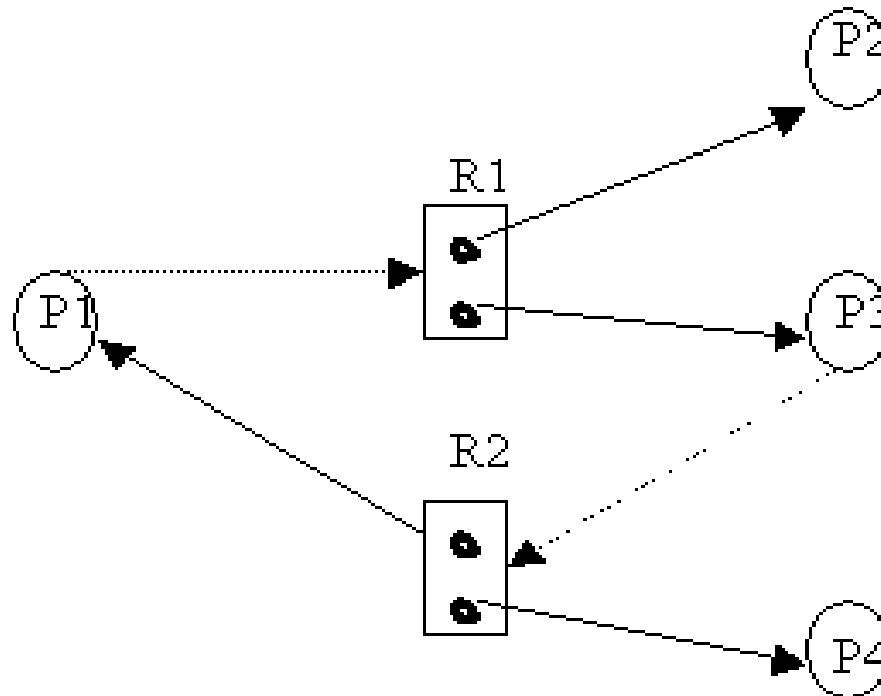


圖 6.4a 形成死結

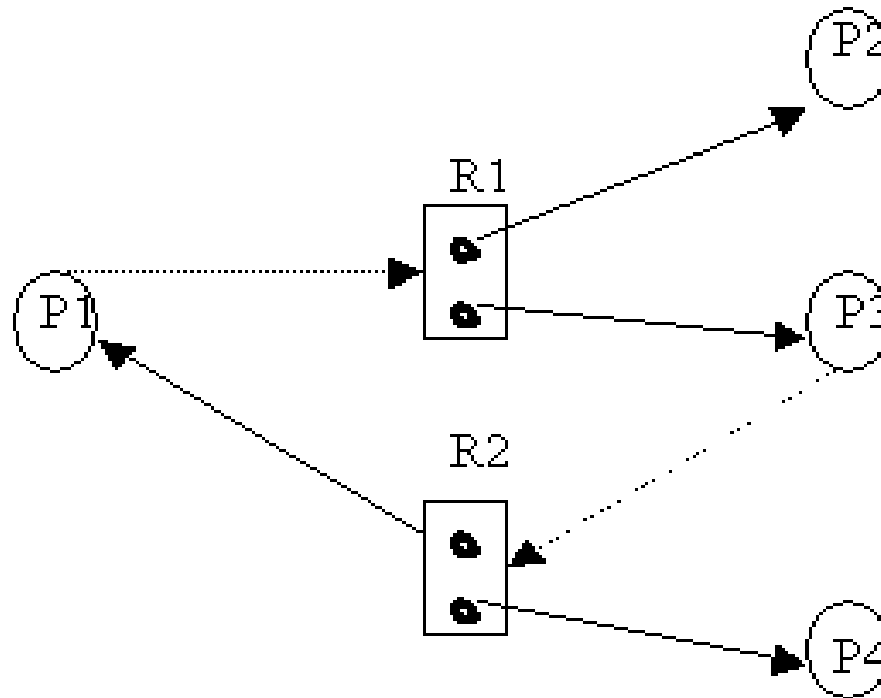
資源配置圖(Resource Allocation Graph)(3)

- 雖然形成P1、R1、P3、R2之循環圈，但是它們沒有形成死結，因為只要P4執行完畢後便會釋放R2供P3使用此時就沒有循環圈了！



資源配置圖(Resource Allocation Graph)(4)

- 形成循環圈並不一定就意味發生死結；但是若每個資源僅有一個的情況之下，**互斥、擁有和等待、不可奪取、循環等待**四個條件成立，則一定發生死結。



處理死結問題的方法

- 預防死結（Deadlock Prevention）及避免死結（Deadlock Avoidance）
 - 在系統內使用某種協定，以確保系統不會進入死結狀態。
- 偵測死結（Deadlock Detection）並作死結復原（Deadlock Recovery）
 - 允許系統進入死結狀態，並有一套方法偵測死結的發生，然後消除死結，並嘗試使系統回復至正常狀態。

預防死結(Deadlock Prevention)(1)

7.2 預防死結

- 預防死結的方法，就是使四個死結的必要條件中一種條件不成立。
- 互斥
 - 如果資源是可以共用的，例如許多處理元均可以同時讀取相同的檔案，則用不著互斥，亦不會發生等待的問題，也就不可能產生死結。
 - 對不可共用的資源（Non-sharable Resource），則必須維持互斥性，例如中央處理器是無法由多個處理元所共用的。
 - 要讓互斥條件不成立，在作業系統環境內是不太可行的，因為系統內並不是每個資源都是共用資源（Sharable Resource）。

預防死結(Deadlock Prevention)(2)

- 擁有和等待
 - 任何處理元必須先獲得所有資源之後才能開始執行，若其未獲得所有資源，則必須釋放資源後，再重新要求所有資源。
 - 處理元必須在沒有擁有資源的情形下，才能提出資源的要求。也就是說，處理元可以先擁有部份資源便執行，但是執行過程中，再要求其他資源時，必須釋放所有已經擁有的資源之後，才能再要求資源。
- Disadvantages
 - Resource utilization may be low, since many of the resources may be allocated but unused for a long time.
 - Starvation is possible.

預防死結(Deadlock Prevention)(3)

- 不可奪取
 - 只要某個處理元要不到所要求的資源時，便把它已經擁有的資源釋放，然後再重新要求所要資源。
 - If a process that is holding some resources requests another resource that **can not be immediately allocated** to it, then all resources currently being held **are preempted (released)**.
 - 當處理元要不到資源時，它便去奪取別的處理元的資源。

預防死結(Deadlock Prevention)(4)

- 不可奪取
 - 要讓不可奪取的條件不成立，在作業系統環境實務上是不可行的，因為
 - 對印表機、磁帶機、讀卡機...等專屬設備 (Dedicated Device) 而言，它們是**不可奪取的資源 (Non-preemptible Resource)**
 - 對中央處理器、記憶體、磁碟機...等共用設備而言，它們是**可奪取的資源 (Preemptible Resource)**，是可以實現的。
 - Spooling: **Non-preemptible Resource ==> Preemptible Resource**
- Starvation is possible.

預防死結(Deadlock Prevention)(5)

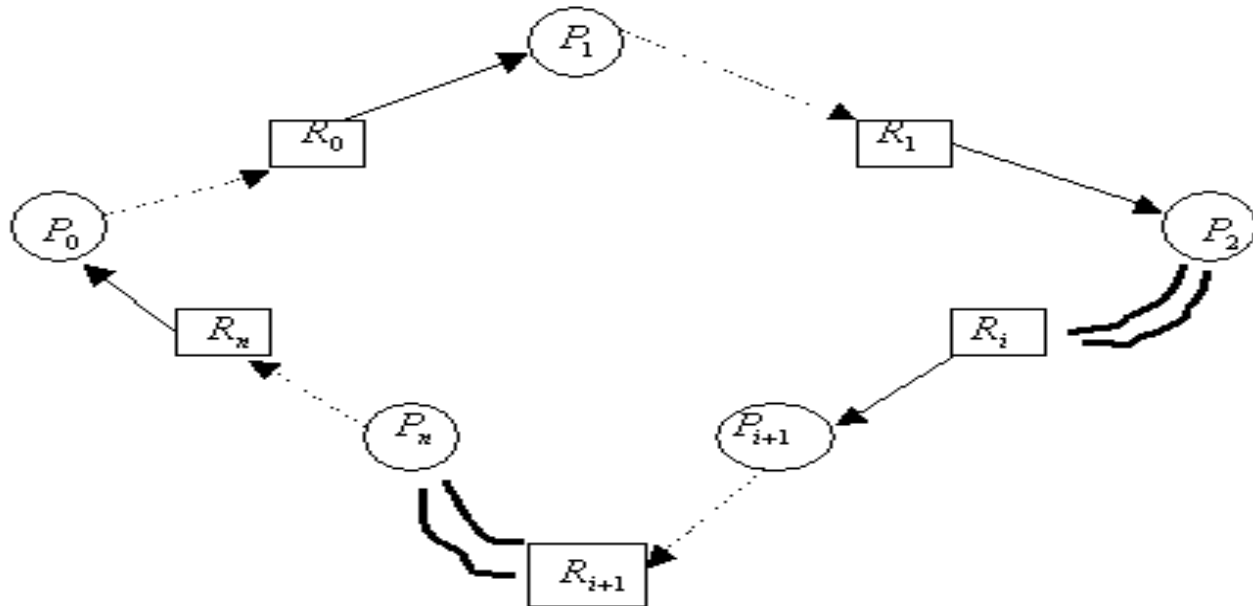
- 循環等待
 - 每個資源均給予一個唯一的編號，處理元可以在任何時間要求資源，但每次所要求資源的編號必須漸增。若系統內所有處理元均依照此方法要求資源，則保證不會發生循環等待，亦即不可能產生死結。
 - Each resource give a unique integer number, when process request resources, it can request only in an increasing order of enumeration.

循環等待證明(1)

- 定義一個一對一函數 $F: R \rightarrow N$ ，其中 $R = \{R_0, R_1, \dots, R_n\}$ ，是所有資源的集合，而 N 是自然數。
- $F(R_i)$ 代表資源的編號。
- 任何二個資源 R_i 及 R_j ， $F(R_i) < > F(R_j)$ 。
- 當處理元已經擁有的資源當中， $F(R_i)$ 為最大，則往後處理元僅能要求 R_j 資源，且符合 $F(R_j) > F(R_i)$ 的條件。

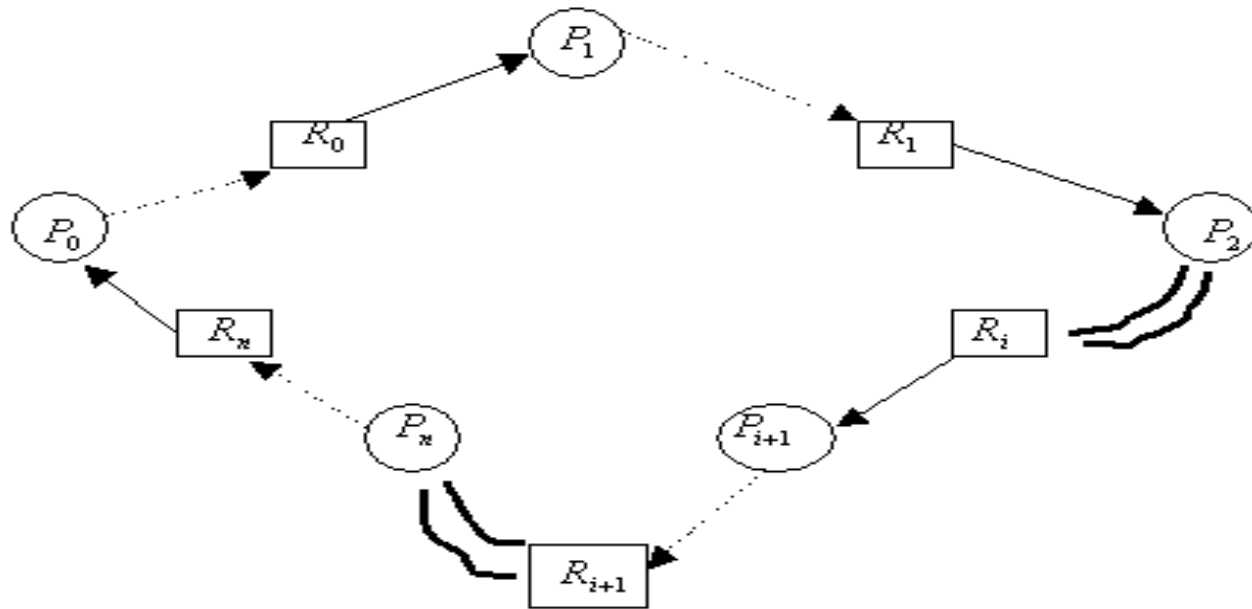
循環等待證明(2)

- 假設利用以上方法仍然發生循環等待，而循環等待的狀況為 R_0 、 P_1 、 R_1 、 P_2 、...、 R_{n-1} 、 P_n 、 R_n 、 P_0 。
- 其中 P_1 已擁有 R_0 正要求 R_1 ， P_2 已擁有 R_1 正要求 $R_2 \rightarrow P_{i+1}$ 已擁有 R_i 正要求 R_{i+1} 。
- $F(R_0) < F(R_1)$ ， $F(R_1) < F(R_2)$ ，...， $F(R_i) < F(R_{i+1})$ ，...， $F(R_{n-1}) < F(R_n)$ 且 $F(R_n) < F(R_0)$ 。



循環等待證明(3)

- 也就是 $F(R_0) < F(R_1) < F(R_2) < \dots < F(R_i) < F(R_{i+1}) < \dots < F(R_n) < F(R_0)$ ，故得到 $F(R_0) < F(R_0)$ ，這是矛盾不可能發生的。



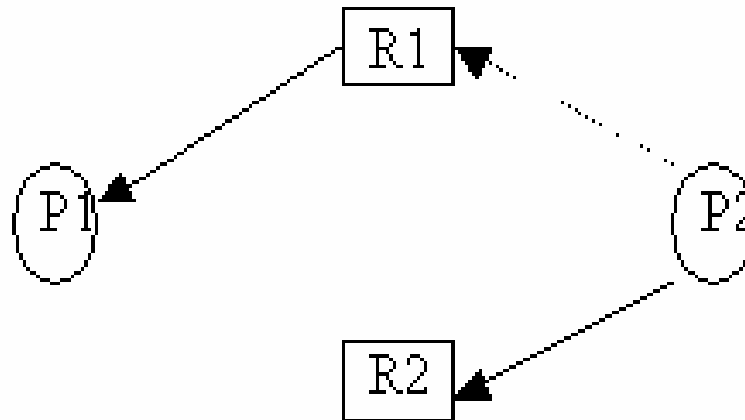
避免死結(Deadlock Avoidance)(1)

7.3 避免死結

- 依據現有可使用的資源、現有已被處理元所擁有資源的狀況、以及未來處理元所要求或釋放資源的因素考量，以決定是否同意目前處理元所提出的資源要求，抑或該處理元必須等待資源，以避免未來發生死結。
- A deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular wait condition.

避免死結(Deadlock Avoidance)(2)

- 利用資源配置圖形除了可以方便了解處理元是否進入死結狀態之外，亦可以了解處理元是處在安全狀態 (Safe State) 或不安全狀態 (Unsafe State)。

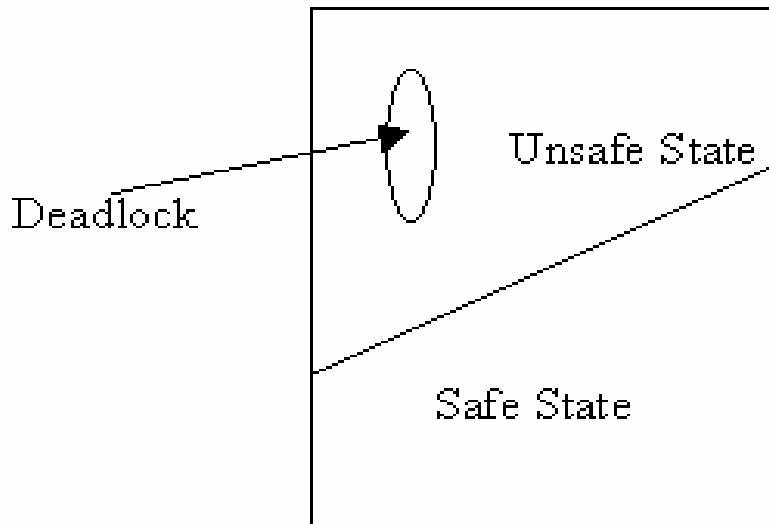


避免死結(Deadlock Avoidance)(3)

- 安全狀態是指，系統依照某些次序分派資源給每個處理元，且能夠避免產生死結。
- Safe state: If the system can allocate resources to each process in some order and still avoid a deadlock.
- 在安全狀態下，雖然 P_i 無法立即取得別的處理元所擁有的資源，但 P_i 能夠等到別的處理元完成工作後，將資源釋放出來，而獲得資源。

安全狀態、不安全狀態及死結的關係

- 在安全狀態絕對不會產生死結。
- 不安全狀態不一定產生死結。
- 但死結一定是從不安全狀態產生。
- 只要讓系統處於安全狀態，便可以避免死結。



避免死結範例(1)

- 系統只剩下3個資源，若將3個資源分配給P0，將使得系統進入不安全狀態；進入不安全狀態不一定產生死結，因為P0可能在執行中又釋放多個資源，但萬一P0不釋放資源，則會造成死結。
- 這3個資源亦不能分配給P2，因為同樣的會進入不安全狀態。

Process	Maximum Needs	Used
P0	10	5
P1	4	2
P2	9	2

Total Resources = 12

避免死結範例(2)

- 若將其中2個資源分配給P1，則仍然在安全狀態，而且當P1執行完畢後，會釋放4個資源，接著將5個資源分配給P0，當P0執行完畢後，便能順利執行P2。
- 若執行的次序是P1、P0、P2，則系統一直處在安全狀態，它不會產生死結。其他的執行次序則會進入不安全狀態，系統必須避免這種狀態發生。

Process	Maximum Needs	Used
P0	10	5
P1	4	2
P2	9	2

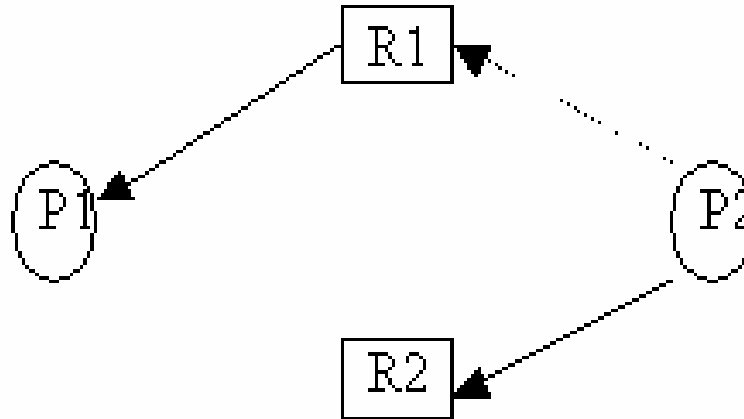
Total Resources = 12

資源配置圖形 (Resource Allocation Graph) 及資源配置狀態 (Resource Allocation State)

- 資源配置圖形 (Resource Allocation Graph)
 - 可以用來處理避免死結的發生，但它較適合每個資源都只有一個的情況。
- 資源配置狀態 (Resource Allocation State)
 - 若每個資源都有許多個時，隨時記錄可供使用之資源及已分派的資源數目，並避免進入不安全狀態。

資源配置圖形

- 若P1已經擁有R1，P2等待R1，這時若P2要求R2，則系統不能將R2分配給P2，因為系統會測試到，若將R2分配給P2，之後若有P1要求R2的需求時，將造成循環等待，而產生死結。
- Detect a cycle in graph.



Banker's演算法範例(1)



- 若將3個資源分配給P0，將使得系統進入不安全狀態。
- P0可能在執行中又釋放多個資源，但萬一P0不釋放資源，則會造成死結。
- 這3個資源亦不能分配給P2，因為同樣的會進入不安全狀態。
- 若將其中2個資源分配給P1，則仍然在安全狀態。
- 若執行的次序是P1、P0、P2，則系統一直處在安全狀態，它不會產生死結。

Process	Maximum Needs	Used
P0	10	5
P1	4	2
P2	9	2

Total Resources = 120

Banker's演算法範例(2)

- 若系統依照P1、P3、P4、P2、P0的次序執行，則保證是在安全狀態，且不會產生死結。

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
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			

Banker's演算法範例(3)

- 給予P1一個A資源及二個C資源之後的資源配置狀態。

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
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			

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	2	3	0
P1	3	0	2	3	2	2	0	2	0			
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			

Banker's演算(1)

- 使用資料結構記錄資源配置狀態。假設系統中有n個處理元及m個不同型態資源。
 - 可用資源 (Available)
 - 為陣列，共有m個元素，用來記錄每個資源的個數。
 - 最大需求 (Max)
 - 為二維陣列，共有n*m個元素，用來記錄每個處理元最多需要那種資源多少個。

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	2	3	0
P1	3	0	2	3	2	2	0	2	0			
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			

Banker's演算(2)

- 配置 (Allocation)
 - 為二維陣列，共有 $n*m$ 個元素，用來記錄每個處理元目前已擁有那種資源多少個。
- 需要量 (Need)
 - 為二維陣列，共有 $n*m$ 個元素，用來記錄每個處理元還需要那種資源多少個，才能完成工作。

Process	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	2	3	0
P1	3	0	2	3	2	2	0	2	0			
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			

Banker's演算(3)

- Banker's演算法分二階段
 - 決定處理元要求資源是否可以被接受。
 - 計算給予資源後是否仍在安全狀態。

Banker's演算(4)

- 步驟1測試是否此需求超過Need的宣告，若是則此Request不合理。步驟2測試是否需求超過Available，若是則暫時無法提供此需求。步驟3則先假設此Request被接受，並計算資源配置狀態。

First Stage: Resource Request Algorithm

Input: P_i Request Resource

Step 1: for $j = 0$ to $m - 1$

```
    if (Request[j] > Need[i][j])
    {   error;
        stop run;
    }
```

Step 2: for $j = 0$ to $m - 1$

```
    if (Request[j] > Available[j]) then
        Reject this request;
```

Step 3: for $j = 0$ to $m - 1$

```
    {   Available[j] = Available[j] - Request[j];
        Allocation[i][j] = Allocation[i][j] + Request[j];
        Need[j] = Need[j] - Request[j];
    }
```

Banker's演算(5)

Second Stage: Safety Algorithm

```
Step 1:  for j = 0 to m -1
          Work[j] = Available[j];
          for j = 0 to n -1
              Finish[j] = false;
Step 2:  Find a Process Pi such that both
          1. Finish[i] = false;
          2. for j = 0 to m -1
              Need[i][j] <= Work[j];
          if no such Pi exists, go to Step4;
Step 3:  for j = 0 to m -1
          {   Work[j] = Work[j] + Allocation[i][j];
              Finish[i] = true;
          }
          go to Step2;
Step 4:  for j = 0 to n -1
          if (Finish[j] < > true) then
              Reject this request;
          Accept this request;
```

偵測死結(Deadlock Detection)(1)

7.4 偵測死結

- 允許處理元進入死結狀態，並於系統內有一套偵測死結發生的方法，然後消除死結，並使系統回復至正常狀態。
- 當偵測死結演算法偵測到死結之後，系統必須將死結解開，以便讓所有處理元能夠繼續執行下去。
- 解開死結的方法是選擇一個犧牲者，並將此犧牲者回復到沒有執行前的狀態。
- 萬一消除（Abort）一個處理元仍無法解開死結，則持續再消除另一個處理元，直到死結完全被解開為止。

偵測死結(Deadlock Detection)(2)


- 當處理元產生死結時，消除（Abort）一個處理元就代表此處理元沒有執行完畢，所以必須撤回（Rollback）。
- Rollback就是，讓此處理元回復到它開始執行前的狀態。

偵測死結(Deadlock Detection)(3)

- 每個資源均只有一個的情況之下，系統可以利用資源配置圖形，來偵測此圖形是否產生循環圈而造成循環等待，以了解死結是否發生。
- 每個資源均有數個的情況下，可以採用類似Banker‘演算法來偵測死結是否發生。

偵測死結範例

- P1、P2、P3、P4產生死結。

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

偵測死結演算

Step 1: for j = 0 to m-1

Work[j] = Available[j];

Next: for j = 0 to n-1

{ Finish[j] = false;

for k= 0 to m-1

if Allocation[j][k] < > 0 then

go to Next;

Finish[j] = true;

}

Step 2: Find the Process P_i , such that both

1. Finish[i] = false;

2. for j = 0 to m-1

Request[i][j] <= Work[j];

if no such P_i exists, go to Step4;

Step 3: for j = 0 to m-1

Work[j] = Work[j] + Allocation[i][j];

Finish[i] = true;

go to Step2;

Step 4: for j = 0 to n-1

if Finish[j] = false then

P_j deadlock with others;

偵測死結時機

- 當某個處理元要求資源而沒有立刻得到時。
- 固定一段時間作一次。
- 當中央處理器使用率降低時。

復原(Recovery)

7.5 復原(Recovery)

- 電腦當機 產生之問題
 - 無法挽回--> 提款機
 - 持續從剛才當的地方繼續 run 下去--> Word
 - 回復到沒有作(Recovery)
- 清除處理元
 - 回復到沒有作(Recovery)
- 要當作剛才的事沒發生過

Type of failure

- Transaction failure
 - 程式不預期的結束
 - if and only if the program terminated in an unplanned fashion.
- System failure
- Media failure

交易記錄 (Transaction Log)

Start Transaction

Read (A)

Write(A)

Read(B)

Read(C)

Write(C)

End Transaction (COMMIT or ABORT)

- COMMIT : Signal successful termination.
- ABORT : Signal unsuccessful termination.

撤回 (ROLLBACK)

- An aborted transaction must be restored to what it was just before the transaction started execution. 當作事情未作過.
- 讓此處理元回復到它開始執行前的狀態。

記錄檔 (Log File) (1)

- 使用記錄檔 (Log File) 來記錄每個處理元的執行過程。Write Ahead Logging
 - 記錄修改的命令 (Command)
 - 修改前的資料值 (Before Image)
 - 修改後的資料值 (After Image)
- The system maintains, on **stable storage**, a data structure called the log. Every time a **change** is made to the data, a record containing the **old** and **new** values of the changed item is written to a special data set called the log.
- 若解開死結的方法是消除一個處理元，則採用以上方法便可以作到復原的工作。

記錄檔 (Log File) (2)

Start TA

TA Write AA (6)

Start TB

TA Before AA(3)

TA After AA(6)

TB Read CC

Commit TA

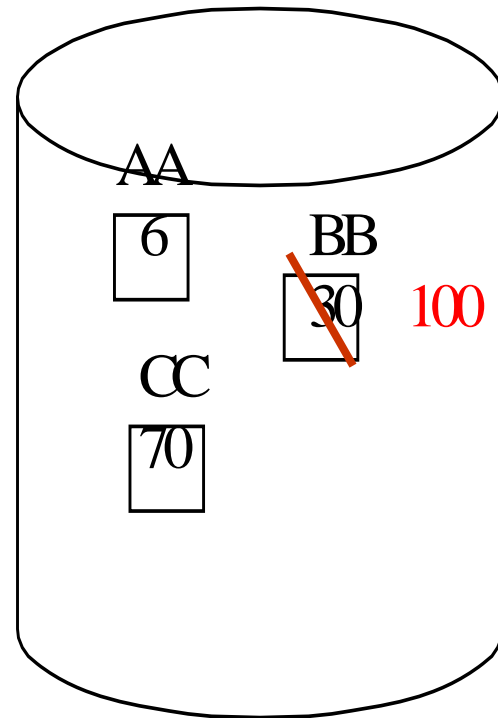
TB Write BB (100)

TB Before BB (30)

TB After BB (100)

TB Write BB (40)

Down



REDO TA

UNDO TB

重作及不作 (Redo and Undo)

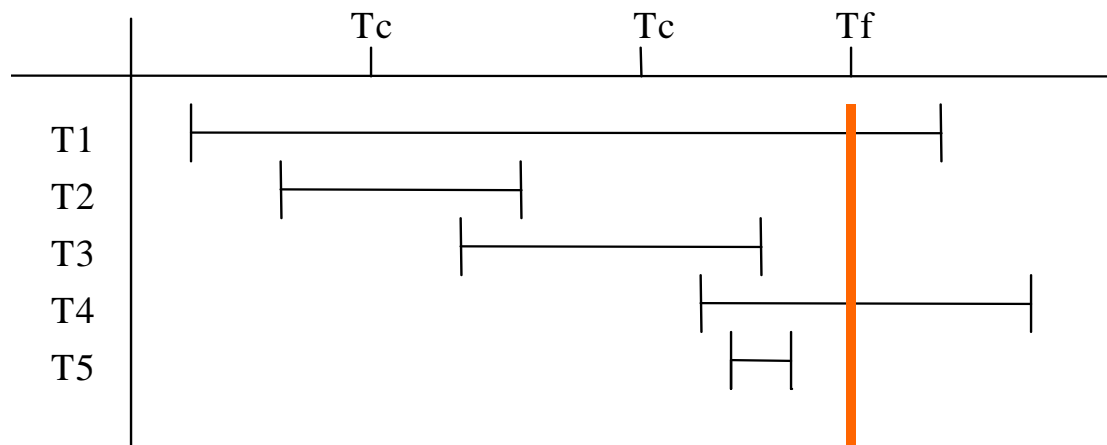
- When failure occurs, redo or undo operations.
- Undo
 - Restores the value of all data updated by transaction to the old values. (看到start, 看不到commit)
- Redo
 - Sets the value of all data update by transaction to the new value. (看到start, 且看到commit)。
 - 有些資料在緩衝區內尚未寫入磁碟。
- 要把交易記錄檔內之訊息均看過一遍，浪費時間。

復原(Recovery)

- 萬一犧牲者是一個資源，也就是強迫某個處理元放棄這個資源，這時若要求某個處理元撤回至執行前狀態，往往會大費週章。例如印表印一半、檔案寫一半。
- 在作業系統中可採用檢查點 (Checkpoint) 的方法，只讓處理元依據記錄檔內容撤回至安全狀態，即是回復到未使用這個資源前的狀態。
- 解開死結讓某些處理元回復至執行前狀態之後，這些處理元仍然可能因為再度執行，而產生另一個死結；因此解開死結並不能保證某些處理元不會餓死 (Starvation)。

檢查點(Checkpoint)

- During execution, the system maintains the write ahead log. In addition, the system **periodically performs checkpoints.**



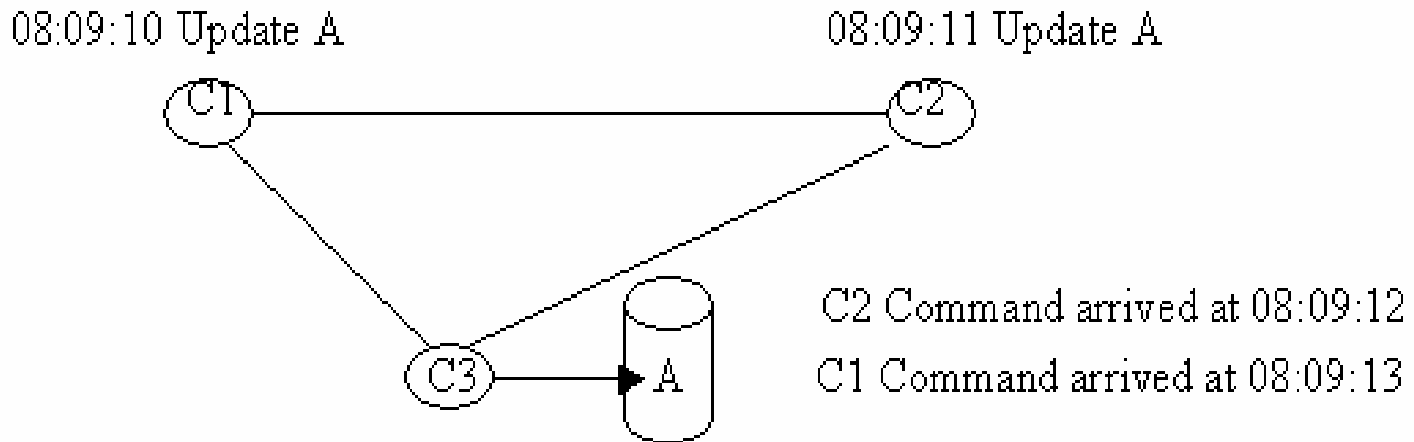
- Success: T2
- Undo: T1, T4
- Redo: T3, T5 After Checkpoint

檢查點演算法(Checkpoint Algorithm)

- Step 1
 - Output all **log records** currently residing in volatile storage onto stable storage.
- Step 2
 - Output all **modified** residing in volatile storage on the stable storage.
- Step 3
 - Output a log record **<checkpoint>** onto stable storage.

時間戳記 (Time Stamp)

- 在一個分散式系統環境中，電腦之間是透過網路來交換訊息，因此有可能產生下圖情況。



- 使用時間戳記 (Time Stamp) 方法
 - 系統承認電腦C2的命令，並撤回電腦C1的命令，並讓於電腦C1內執行此命令的處理元復原。

時間戳記(Time Stamp)演算法(1)

- 任何處理元執行時，均將它開始執行的時間當作一個唯一的識別（時間戳記）。
- Transaction can be assigned a unique identifier (time stamp) which can be through of as the transaction's start time.
- 寫入時間戳記 (W-timestamp)
 - 資源最後是被那個處理元寫入，將該處理元之時間戳記記錄至寫入時間戳記內。
- 讀取時間戳記 (R-timestamp)
 - 資源最後是被那個處理元讀取，將該處理元之時間戳記記錄至讀取時間戳記內。

時間戳記(Time Stamp)演算法(2)

- 對資源Q進行讀取

```
if timestamp(Pi) >= W_timestamp(Q)
{
    R_timestamp(Q) = max(timestamp(Pi), R_timestamp(Q));
    Execute read(Q) Operation;
}
else
    Rollback(Pi) then Restart; /*Assign a New Timestamp */
```


時間戳記(Time Stamp)演算法(3)

- 對資源Q進行寫入

```
if ((timestamp(Pi) >= R_timestamp(Q) )and (timestamp(Pi) >=W_timestamp(Q))
    {
        W_timestamp(Q)=timestamp(Pi);
        Execute write(Q) Operation;
    }
else
    Rollback(Pi) then Restart;  /*Assign a New Timestamp */
```

時間戳記(Time Stamp)演算法(4)

- 時間戳記方法在理論上可以被證明是**可循序化的 (Serializable)**。
- 使用此方法**不會發生死結**，因為一旦發生無法使用資源時，處理元就被撤回，而不會有等待發生。