
ASIA EDITION
作業系統
Operating System Concepts
TENTH EDITION
趙涵捷 審閱
吳庭育 駱詩軒 譯
ABRAHAM SILBERSCHATZ
PETER BAER GALVIN
GREG GAGNE
東華書局
WILEY

Chapter 8

死　結

章節目標

- 說明使用互斥鎖時如何發生死結
- 定義發生死結的四個必要條件
- 在資源分配圖中辨識別死結情況
- 提出防止死結的四種不同方法
- 應用銀行家演算法避免死結
- 應用死結偵測演算法
- 提出從死結中恢復的方法

8.1 系統模型

- 在正常運作的模式之下，一個執行緒只能依據下列的順序來使用資源：
  1. 要求：執行緒要求資源。若此要求不能立即被認可，則執行緒必須等候以獲得此資源
     - 例如，此互斥鎖正被其它的執行緒所取得
  2. 使用：執行緒能夠在此資源上運作
     - 例如，若此資源為互斥鎖，則執行緒可存取其臨界區間
  3. 釋放：執行緒釋放資源

8.2 多執行緒應用程式中的死結

- 死結的例子

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

8.2.1 活　結

- **活結** (livelock) 是另一種形式的執行失敗，這類似於死結
  - 兩者都阻止兩個或多個執行緒繼續進行，但是由於不同的原因，執行緒無法繼續進行

8.2.1 活　結

- 活結的範例

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&first_mutex);
        if (pthread_mutex_trylock(&second_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&second_mutex);
            pthread_mutex_unlock(&first_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&first_mutex);
    }

    pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
    int done = 0;

    while (!done) {
        pthread_mutex_lock(&second_mutex);
        if (pthread_mutex_trylock(&first_mutex)) {
            /**
             * Do some work
             */
            pthread_mutex_unlock(&first_mutex);
            pthread_mutex_unlock(&second_mutex);
            done = 1;
        }
        else
            pthread_mutex_unlock(&second_mutex);
    }

    pthread_exit(0);
}
```

8.3 死結的特性

- 如果下列四種狀況在系統中同時成立時，就可能發生死結的狀況：
  - 互斥 (mutual exclusion)：至少有一資源必須是不可共用的形式
    - 一次只有一個執行緒可使用此資源
    - 若有另一執行緒想使用此資源，則必須延遲至此資源被釋放後才可以
  - 占用與等候 (hold and wait)：必須存在一個至少已占用一個資源，且正等候其它執行緒已占用另外資源之執行緒

8.3 死結的特性

- 不可搶先 (no preemption)：資源不能被搶先
  - 因此，一個資源只能被占用它的執行緒在完成工作目標之後才被釋放
- 循環式等候 (circular wait)：必須存在一等候執行緒的集合 $\{T_0, T_1, ..., T_n\}$
  - 其中 T_0 等候的資源已被 T_0 占用、T_0 等候的資源已被 T_2 占用, ……, T_{n-1} 等候的資源已被 T_n 占用，而 T_n 等候的資源已被 T_0 占用

8.3.2 資源配置圖

- 系統資源配置圖的有向圖由一組頂點 V 及一組邊 E 所組成
  - 頂點所成的集合 V 又可區分為兩個不同的集合：
    - 系統中所有正在執行之執行緒的集合 $T = \{T_1, T_2, ..., T_n\}$
    - 與系統中所有資源形式的集合 $R = \{R_1, R_2, R_3, ..., R_m\}$
- 有向邊 $T_i \rightarrow R_j$ 稱為**要求邊** (request edge)；而有向邊 $R_j \rightarrow T_i$ 則稱為**分配邊** (assignment edge)

圖 8.3　在圖 8.1 中程式的資源分配圖

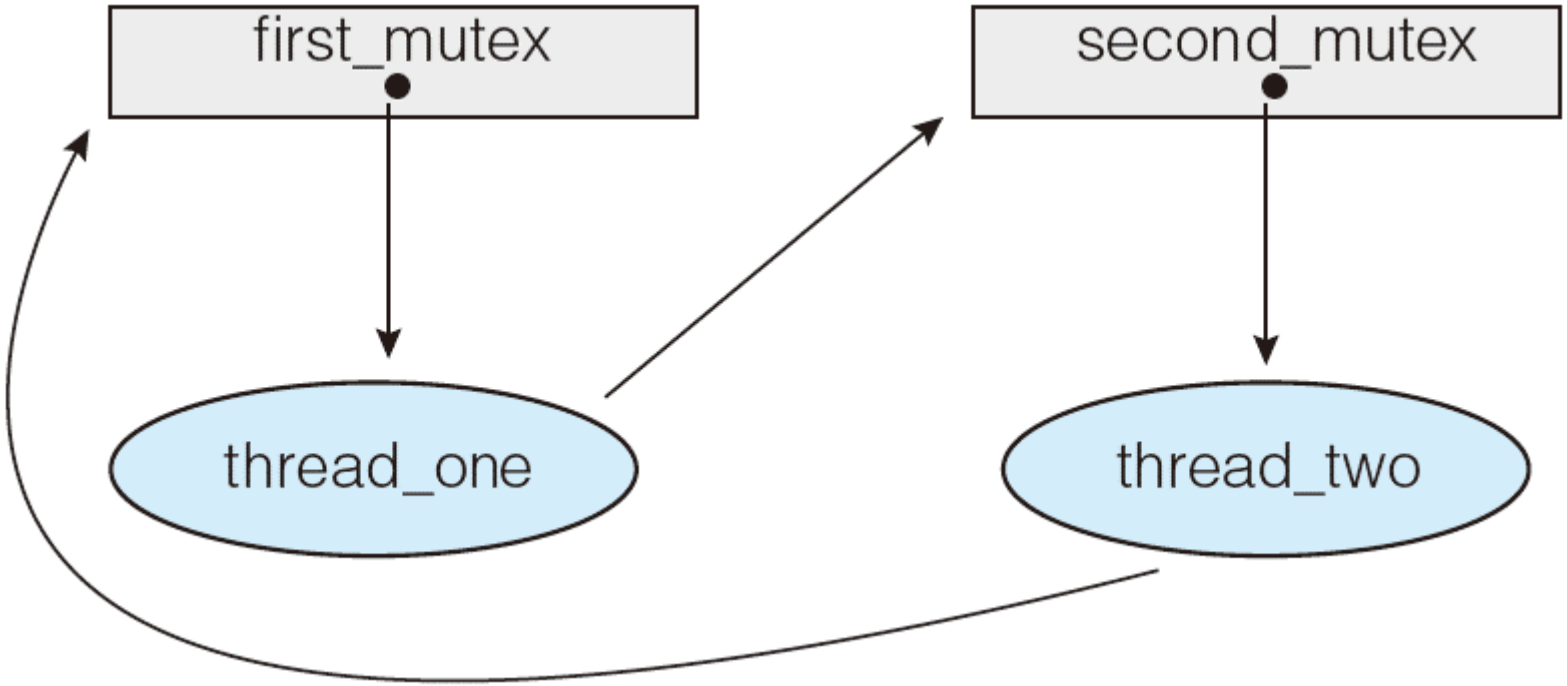


圖 8.4　資源配置圖

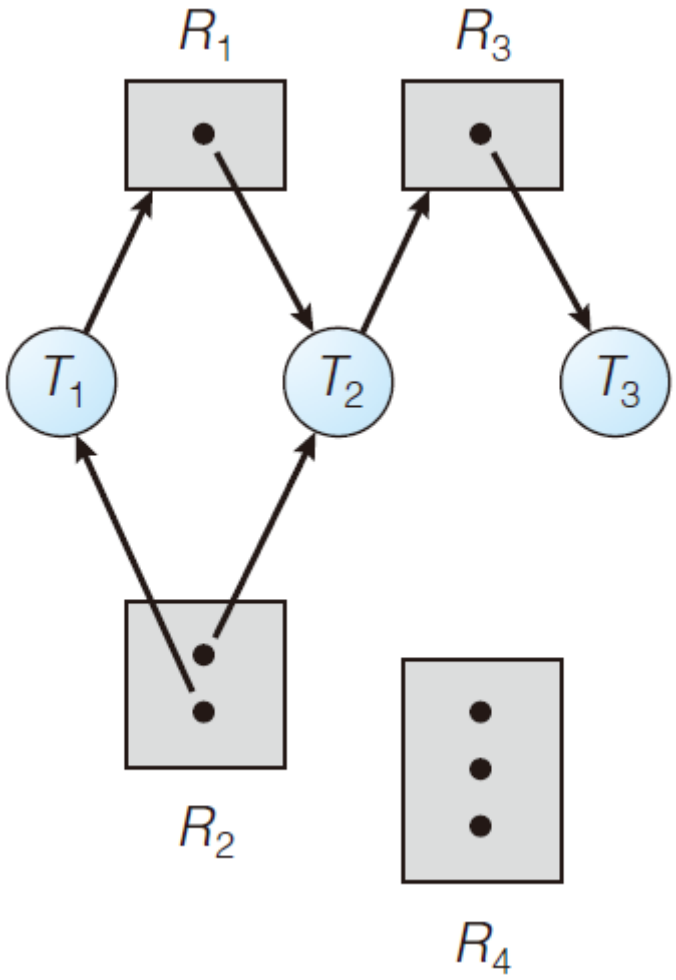


圖 8.5　含死結的資源配置圖

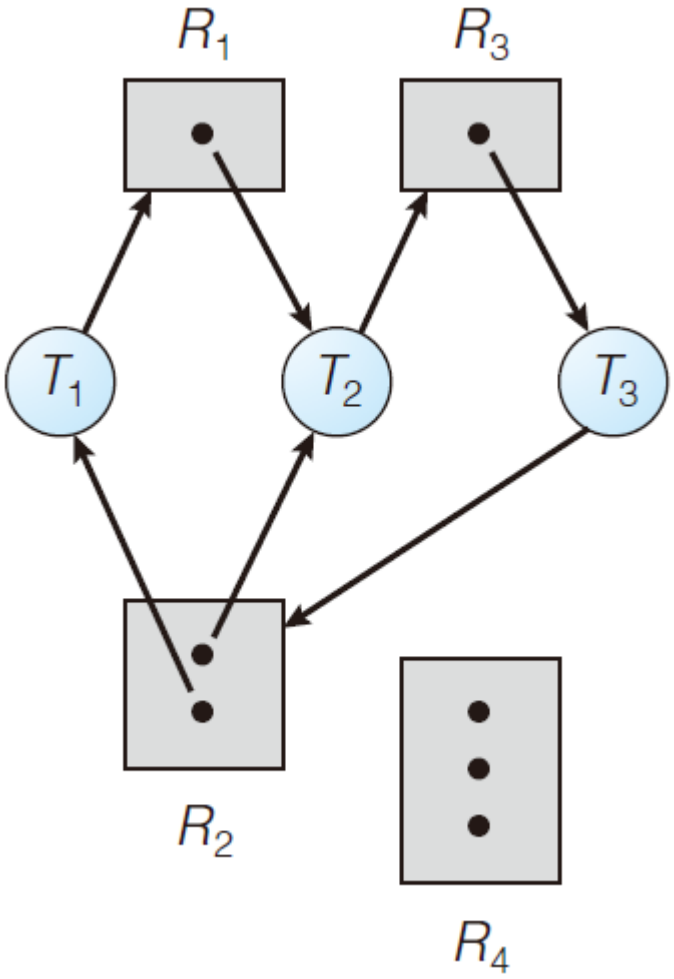
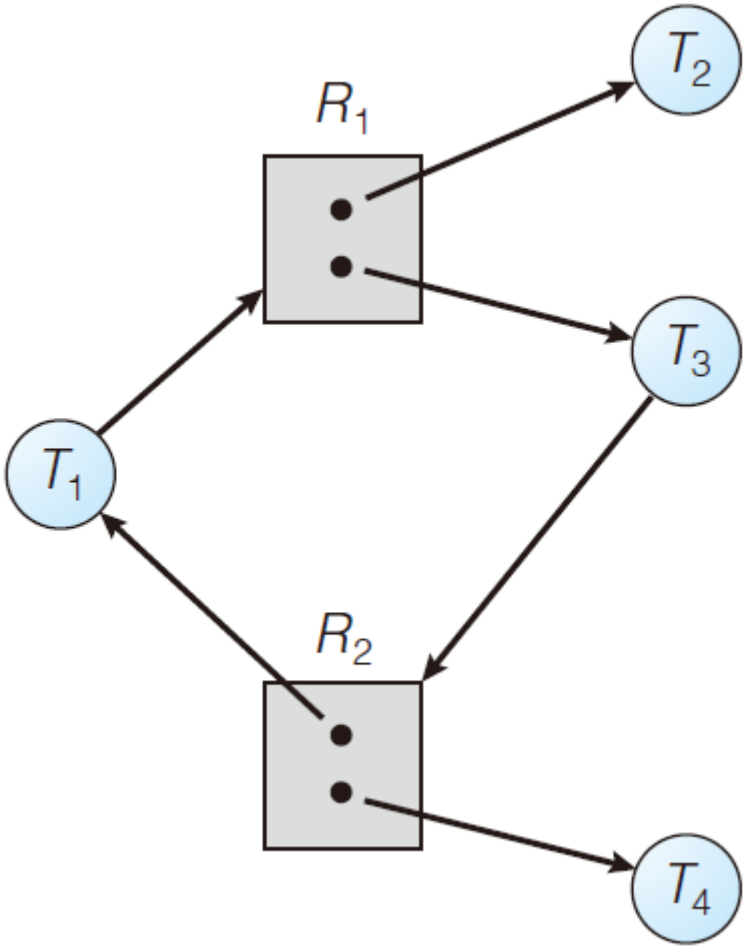


圖 8.6　含循環但無死結現象之資源配置圖



8.3.2 資源配置圖

- 總括來說，如果資源配置圖中無循環出現，則系統就不會進入死結狀態
- 反之，若有循環形成，則此系統並不一定會進入死結狀態
- 在處理死結問題上這是一個非常重要的觀點

8.4 處理死結的方法

- 為了確保死結永不發生，我們可以使用預防死結或是避免死結方法
  - **預防死結** (deadlock prevention) 提供一組方法，可以確保死結必要條件至少有一項不會發生的方法
  - 這些方法藉由限制對資源的要求來避免死結發生

8.4 處理死結的方法

- **避免死結** (deadlock avoidance) 則要求作業系統預先取得一個執行緒在它的生命週期將會要求及使用哪些資源
  - 有了這些資訊後，作業系統可以決定此執行緒的每一要求是否該等待
  - 為了決定目前的要求可以被滿足或是必須被延遲，系統考慮現有的可用資源，已分配給其它執行緒的資源
    - 以及將來的要求和其它執行緒會釋放的資源

8.5 預防死結

- 當產生死結時，四個必要條件必須成立
  - 互　斥
  - 占用與等候
  - **不可搶先**
  - **循環式等候**
    - 因此只要這些條件有一條件不成立，即可預防死結產生
    - 藉由分別檢驗這四個必要條件來闡述此方法

8.5.1 互　斥

- **互斥** (mutual exclusion) 的條件必須成立
  - 至少有一項資源必須是不可共用的
  - 可共用資源並不需以互斥的方式存取，因而不會產生死結
  - 唯讀檔案 (readony file) 就是一個共用資源的良好例證

8.5.2 占用與等候

- 為了確保在系統中**占用與等候**條件不成立，必須保證一個執行緒在要求一項資源時，不可占用其它的資源
  - 可使用一種協定，就是每個執行緒在執行之前必須先要求並取得所需的資源
  - 要求資源的動態特性，這對大多數應用程式是不切實際的
  - 兩種協定有兩個主要的缺點
    - 第一，資源使用率 (resource utilization) 可能很低，因為在長時間內，許多資源可能只被分配而未使用
    - 第二，可能產生飢餓現象

8.5.3 不可搶先

- 配置的資源不可被別的執行緒搶先占用
  - 要確定這種情況不會發生，就必須使用以下的協定方式
  - 如果有一個執行緒已經占用某些資源，而它還要求一個無法立即取得的資源 (也就是該執行緒必須等待)，則它目前持有的資源應該可以由他人搶先使用
  - 也就是說，這些資源事實上是已被釋放的
  - 這些搶先的資源就被加到執行緒正在等候的資源表上
  - 這個執行緒只有當它重新獲得原有的那些資源，以及正要求的那個新資源之後，才能重新開始

8.5.4 循環式等候

- 第四個也是最後一個死結的條件是循環式等候的條件
  - 讓必要條件之一變成無效為提供實際解決方法的機會
  - 確保循環式等候的條件不成立的一個方法是，我們對所有的資源形式強迫安排一個線性的順序
  - 而執行緒要求資源時必須依數字大小，遞增地提出要求

圖 8.7　上鎖有順序的死結範例

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);

   acquire(lock1);
      acquire(lock2);

         withdraw(from, amount);
         deposit(to, amount);

      release(lock2);
   release(lock1);
}
```


8.6 避免死結

- 最簡單且最有用的模式就是，要每個執行緒聲明其所需每種形式的資源之最大數量
- 給予這種前置資訊 (priori information) 時，就可能建立一演算法，以保證系統永遠不會進入死結狀態
- 避免死結演算法又可動態地檢查資源配置的狀態，以確保系統不會產生循環等候的狀況
- 資源 配置狀態則由可用的與已被占用的資源數量，以及各執行緒之最大要求數量所定義

8.6.1 安全狀態

- 如果系統能以某種順序將其資源分配給各個執行緒 (到達它的最大值)，而且仍能避免 死結者，則稱其狀態為 安全 (safe) 狀態
  - 更正式的說法，一個系統只有在**安全序列** (safe sequence) 存在時，才算處於安全狀態
  - 對每個 T_j 及 $j < i$ 的狀況下，若 T_i 所要求的資源能滿足所有目前可用的資源與 T_i 所占用的資源之總和
    - 稱此執行緒之序列 $\langle T_1, T_2, ..., T_n \rangle$ 為目前分配狀態下的一個安全序列
  - 在此狀況下，若 T_i 所需資源仍未可用，則 T_i 將一直等候到所有 T_j 完成為止

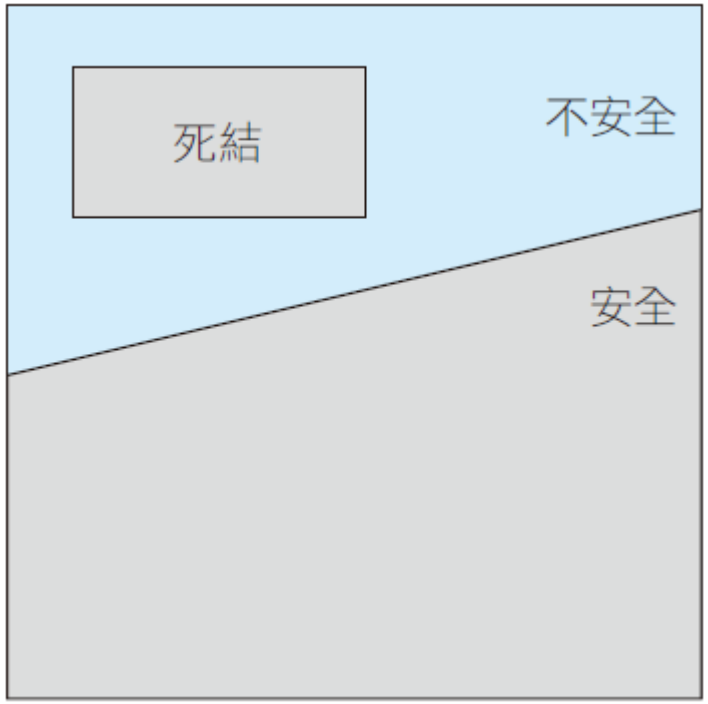
8.6.1 安全狀態

- 當它們已經完成後，T_i 便能得到其所需之所有資源，完成其預定工作，然後釋放其占用的所有資源且結束其工作
- 當 T_i 結束時，T_{i+1} 即可得到其需要的所有資源，依此類推

8.6.1 安全狀態

- 安全狀態並不是死結狀態；相反地，死結狀態就是不安全狀態
  - 然而，並非所有不安全狀態都是死結 (見圖 8.8)
  - 一個不安全狀態可能導致一個死結
  - 只要狀態是安全的，作業系統便能避免不安全 (和死結) 的狀態
  - 在一個不安全狀態下，作業系統不能夠防範因執行緒需求資源而發生死結的情況，執行緒的行為控制不安全狀態

圖 8.8　安全、不安全和死結的狀態空間



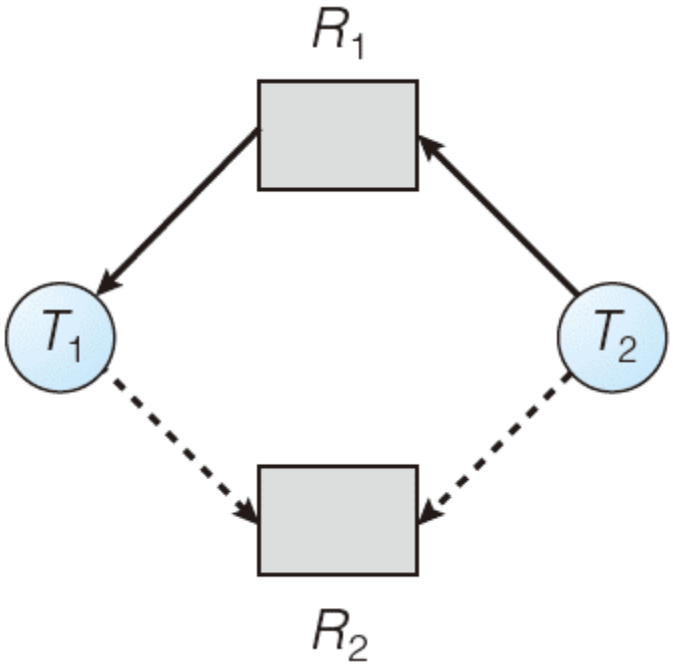
8.6.2 資源配置圖演算法

- 如果我們有一個資源配置系統，其中每種資源型態均只含有一個例證，則 8.3.2 節所定義的資源配置圖變形可以用來避免死結
  - **申請邊** $T_i \rightarrow R_j$ 表示出執行緒 T_i 可能在未來需要求資源 R_j
- 這種邊與要求邊類似，都具方向性，但以虛線表示
  - 當執行緒 T_i 要求資源時，則需把申請邊 $T_i \rightarrow R_j$ 轉換成要求邊
  - 同理，在 T_i 釋放資源 R_j 時，必須將此分配邊 $R_j \rightarrow T_i$ 再轉換成為要求邊 $T_i \rightarrow R_j$

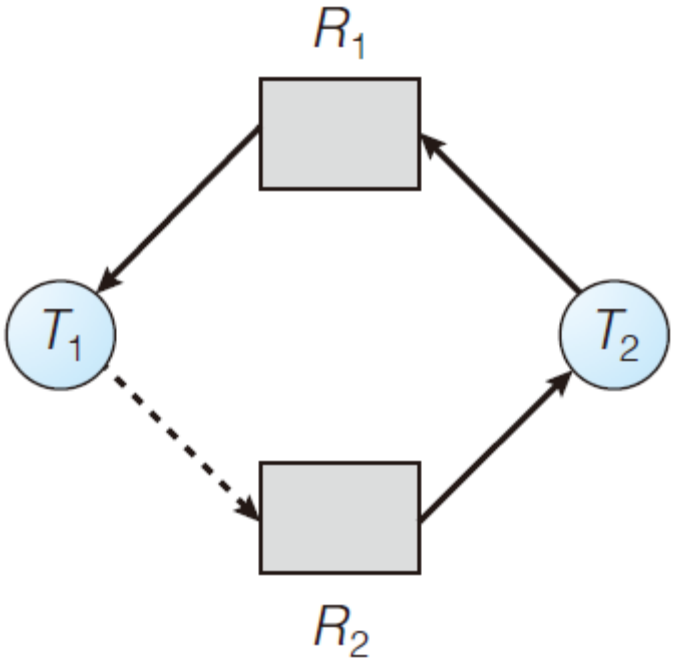
8.6.2 資源配置圖演算法

- 注意，資源必須在系統中預先設置
  - 因此，在執行緒 T_i 開始執行前，所有的申請邊均應已出現在資源配置圖上
  - 我們可以放寬此條件，讓只有在伴隨著執行緒 T_i 的所有邊要求是申請邊的狀況下，才可把申請邊 $T_i \rightarrow R_j$ 加入圖中

圖 8.9　避免死結的資源分配圖



8.10 在資源配置圖中的不安全狀態



8.6.3 銀行家演算法

- 資源配置圖演算法並不適用於有多個例證的資源所形成的資源配置系統
  - 接下來我們所敘述的避免死結演算方法可以用在這種系統上，但是這種方法比資源配置圖演算法的效率低
  - 這種演算法就是眾所周知的**銀行家演算法** (banker's algorithm)
  - 此名稱之選用是因為這種演算法可用在銀行系統中，以確保銀行分配其可用的現金方式，不會使它不能夠滿足所有顧客的要求。

8.6.3 銀行家演算法

- 當一個新的執行緒進入一系統時，它必須聲明可能需要的每一種資源形式例證的最大數量
  - 此數量不能夠超過該系統中資源的總數
  - 當一個使用者要求一組資源時，就必須決定系統是否會因為這些資源的被占用而脫離安全狀態
  - 若是仍在安全狀態下，這些資源就可被占用
  - 這個執行緒必須等候，直到其它的執行緒釋放足夠的資源

8.6.3 銀行家演算法

- 可用的 (Available)：
  - 為一長度為 m 的向量，可表示出每種形式的可用資源量
  - 若 ***Available***$[j] = k$，即表示資源形式 R_j 中有 k 個例證為可用
- 最大值 (Max)：
  - 為一 $n \times m$ 的矩陣，可定義出各執行緒的最大需求量
  - 若 ***Max***$[i][j] = k$，則表示出執行緒 T_i 至多可要求 k 個例證之資源形式 R_j

8.6.3 銀行家演算法

- 占用；分配 (Allocation)：
  - 為一 $n \times m$ 的矩陣，可定義現在每一執行緒所占用之各形式資源的數量
  - 若 ***Allocation***$[i][j] = k$，則執行緒 T_i 現已占用資源形式 R_j 中之 k 個例證
- 需求 (Need)：
  - 為一 $n \times m$ 的矩陣，可表示出每一執行緒剩餘的需求量
  - 若 ***Need***$[i][j] = k$，則表示執行緒 T_i 還需要再占用資源形式 R_j 中之 k 個例證，以完成其工作
  - 注意，***Need***$[i][j]$ = ***Max***$[i][j]$ - ***Allocation***$[i][j]$

安全演算法

1. 令 ***Work*** 與 ***Finish*** 分別為長度 m 及 n 的向量
   - 開始時，令 ***Work*** = ***Available*** 且***Finish***[i] = ***false*** 對 $i = 0, 1, ..., n-1$
2. 尋找滿足下述兩個條件之 i
   - a. ***Finish***[i] == ***false***
   - b. ***Needi*** $\leq$ ***Work***
   - 如果 i 不存在，就跳到步驟 4
3. ***Work*** = ***Work*** + ***Allocationi***
   - ***Finish***[i] = ***true***
   - 回到步驟 2
4. 若對所有 i，***Finish***[i] == ***true***，則此系統處於安全狀態

資源要求演算法

- 接下來我們描述如果要求可以安全地被允許的演算法
  - 令 $\boldsymbol{Request}_i$ 為執行緒 T_i 的要求向量
  - 如果 $\boldsymbol{Request}_i[j] == k$，即表示出行程 T_i 欲要求資源形式 R_j 中的 k 個例證
  - 當行程 T_i 要求資源時，可能會發生下列數種狀況：
    1. 如果 $\boldsymbol{Request}_i \leq \boldsymbol{Need}_i$ 那麼跳到步驟 2
       - 否則即發生一項錯誤，因為執行緒已經超過它的最大需求
    2. 如果 $\boldsymbol{Request}_i \leq \boldsymbol{Available}$，那麼跳到步驟 3
       - 否則因無可用的資源 T_i 必須等候

資源要求演算法

3. 假設系統已配置執行緒 T_i 占用其所要求的資源，並將各陣列資料改為：

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- 如果所產生的資源配置狀態為安全，則完成此項配置資源給執行緒 T_i 的處理
- 假若此新狀態為不安全，則 T_i 必須等候 $Request_i$，且恢復舊有的資源配置狀態

一個例子

- 考慮某一系統具有 5 個執行緒 T_0 到 T_4 以及 3 個資源形式 A、B 和 C

	Allocation			*Max*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	5	3	3	3	2
T_1	2	0	0	3	2	2			
T_2	3	0	2	9	0	2			
T_3	2	1	1	2	2	2			
T_4	0	0	2	4	3	3			

一個例子

- 矩陣 ***Need*** 的內容被定義為 ***Max - Allocation*** 如下：

	Need		
	A	*B*	*C*
T_0	7	4	3
T_1	1	2	2
T_2	6	0	0
T_3	0	1	1
T_4	4	3	1

一個例子

- 事實上，序列〈T_1, T_3, T_4, T_2, T_0〉能滿足安全準則

	Allocation			*Need*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	7	4	3	2	3	0
T_1	3	0	2	0	2	0			
T_2	3	0	2	6	0	0			
T_3	2	1	1	0	1	1			
T_4	0	0	2	4	3	1			

- 必須決定這個新的系統狀態是否安全，因此執行我們的安全演算法，並且發現序列〈T_1, T_3, T_4, T_2, T_0〉滿足安全條件
  - 所以可以立刻答應執行緒 T_1 的要求

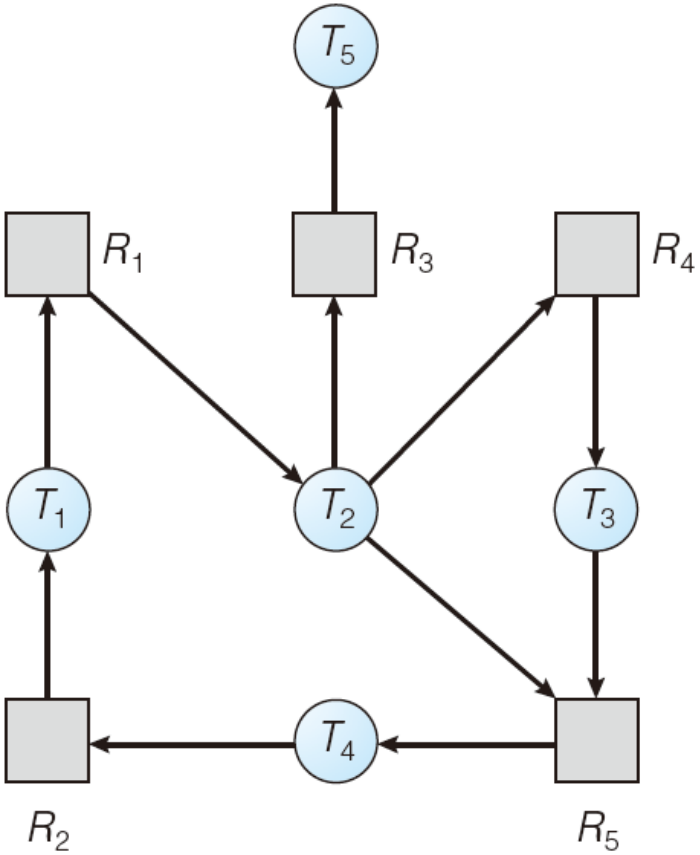
8.7 死結的偵測

- 檢查系統狀態，以決定死結是否產生的演算法
- 由死結中回復的演算法

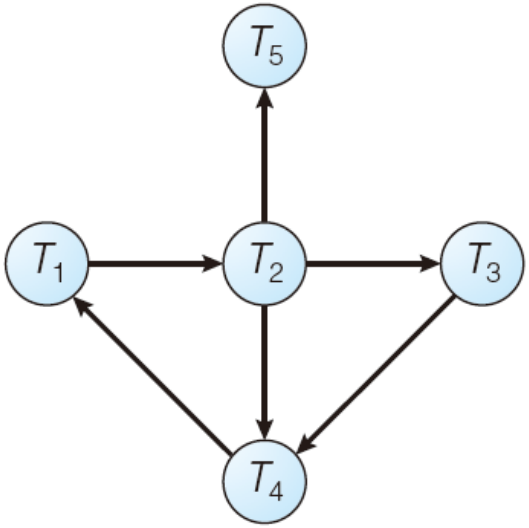
8.7.1 具單一例證的資源形式

- 若所有資源均只含有一個例證，可以使用資源配置圖的變形，叫作**等候** (waitfor) 圖定義一種偵測死結的演算法
  - 可以藉由移除資源端點並除去相對應的邊，以便從資源配置圖獲得等候圖
- 更嚴格地說，在等候圖中自 T_i 至 T_j 的邊，意味著 T_i 必須等候 T_j 釋放 T_i 所需之資源
  - 對某項資源 R_q 而言，若且唯若在相應的資源配置圖中含有兩邊 $T_i \rightarrow R_q$ 與 $R_q \rightarrow T_j$，則邊 $T_i \rightarrow T_j$ 必存在等候圖中
  - 例如在圖 8.11 的例子中，我們將可發現到一資源配置及其相應之等候圖

圖 8.11 (a) 資源配置圖；(b) 對應等候圖



(a)



(b)

8.7.1 具單一例證的資源形式

- 如前所述，若且唯若等候圖中含有循環，則系統必有死結
  - 為了偵測死結，此系統必須維護整個等候圖，與週期性地使用一演算法以尋找圖中的循環
  - 尋找圖中循環的演算法共需 $O(n^2)$ 次的運算，其中 n 為圖中的端點數量

8.7.2 具有多個例證的資源形式

- 可用的 (Available)：是一長度為 m 的向量，可表示每種資源形式的可用數量
- 占用 (Allocation)：為一 $n \times m$ 的矩陣，可定義出每一行程所占用之各種資源形式的數量
- 需求 (Request)：為一 $n \times m$ 的矩陣，可表示出每一個執行緒的現在需求
  - 若 ***Request***$[i][j] = k$，則表示出執行緒 T_i 還要資源形式 R_j 中之 k 個例證

8.7.2 具有多個例證的資源形式

1. 令 ***Work*** 與 ***Finish*** 分別是長度為 m 與 n 的向量
   - 開始時，令 ***Work*** = ***Available***
   - 對 $i = 0, 1, ..., n-1$，若 $\boldsymbol{Allocation}_i \neq 0$，則 ***Finish***[i] = ***false***；否則，***Finish***[i] = ***true***
2. 搜尋滿足下列條件的 i：
   - a. ***Finish***[i] == ***false***
   - b. $\boldsymbol{Request}_i \leq \boldsymbol{Work}$
   - 若無滿足這兩個的 i 存在，則跳到步驟 4

8.7.2 具有多個例證的資源形式

3. ***Work*** = ***Work*** + ***Allocation***$_i$
   - ***Finish***[i] = ***true***
   - 回到步驟 **2**
4. 對某些 $0 \leq i < n$ 的 i，若 ***Finish***[i] = = ***false***，則系統必處於死結態
   - 此外，若 ***Finish***[i] = = ***false***，則執行緒 T_i 便陷入死結狀態

- 這個演算法需要 $m \times n^2$ 次運算來檢查系統是否處於死結的狀態

8.7.2 具有多個例證的資源形式

- 為了說明這個演算法，我們考慮一個具有 5 個執行緒 T_0 到 T_4 的系統和 3 個資源形式A、B 和 C 的系統
  - 資源形式 A 有 7 個例證，資源形式 B 有 2 個例證，而資源形式 C 有 6 個例證
  - 假定在時間 T_0 時，我們有下列資源分配狀態：

	Allocation			*Request*			*Available*		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

8.7.2 具有多個例證的資源形式

- 宣稱此系統並非在死結狀態中
  - 執行的演算法，發現序列〈T_0, T_2, T_3, T_1, T_4〉將使得對所有的 i 而言 ***Finish***$[i]$ == *true*

8.7.2 具有多個例證的資源形式

	Request		
	A	*B*	*C*
T_0	0	0	0
T_1	2	0	2
T_2	0	0	1
T_3	1	0	0
T_4	0	0	2

- 我們宣稱系統現在是死結狀態
  - 雖然我們能夠重新要求執行緒 T_0 所占用的資源，但是可用資源的數目不足以實現其它執行緒要求
  - 因此，一個死結便存在了，包括執行緒T_1、T_2、T_3和 T_4

8.7.3 偵測演算法之使用

- 我們應在何時使用偵測演算法？這個答案取決於兩個因素：
  1. 多久會產生一次死結？
  2. 有多少個行程受死結發生的影響？
- 若經常產生死結，則偵測演算法的使用就要較為頻繁。又在死結現象未消除前，系統不會把資源分配給執行緒，因此牽涉到死結循環中執行緒數目就可能愈來愈多

8.8 自死結恢復

- 取消所有死結中的行程
- 一次取消一個行程直到死結循環被消除為

8.8.1 行程和執行緒的終止

- 不幸地，最小費用並不是很明確的。許多因素都可以決定那個選擇行程的因素，包括：
  1. 行程的優先權為何？
  2. 此行程已運作了多久？在完成它指定的工作之前還需要運算多少時間才完成？
  3. 行程已經使用了多少資源，各是什麼形式？(譬如，資源是否很容易被搶先)
  4. 行程還需要多少資源，才能完成此行程？
  5. 有多少行程需被終止？

8.8.2 資源的搶先

1. 選擇犧牲者 (select a victim)：
   - 哪一個資源與哪一個行程將被搶先？
2. 回撤 (rollback)：
   - 如果我們從一個行程搶先一項資源時，我們對這行程有哪些事要做呢？
3. 飢餓 (starvation)：
   - 我們要如何才能確定飢餓的情形不會發生呢？