作業系統：Deadlock

**目錄**

[作業系統：Deadlock 1](#_Toc204368441)

[1. 系統模型（System Model） 1](#_Toc204368442)

[1.1 資源使用的流程 2](#_Toc204368443)

[2. 多執行緒程式中的死結 (Deadlock in Multithreaded Applications) 2](#_Toc204368444)

[2.1 範例：Deadlock Example 2](#_Toc204368445)

[3. 活結（Livelock） 3](#_Toc204368446)

[4. 死結的四個必要條件(Necessary Conditions) 4](#_Toc204368447)

[4.1 資源配置圖（Resource-Allocation Graph，RAG） 4](#_Toc204368448)

[4.2 範例說明：Figure 8.4 4](#_Toc204368449)

[5. 死結預防（Deadlock Prevention） 5](#_Toc204368450)

[5.1 互斥（Mutual Exclusion） 6](#_Toc204368451)

[5.2 Hold and Wait（保有並等待） 6](#_Toc204368452)

[5.3 No Preemption（不可搶佔） 6](#_Toc204368453)

[5.4 Circular Wait（循環等待） 6](#_Toc204368454)

[6. Deadlock Avoidance 6](#_Toc204368455)

[6.1 Safe State（安全狀態） 6](#_Toc204368456)

[6.2 相關演算法(這格算重要，但太多了) 7](#_Toc204368457)

[7. 死結復原(Recovery from Deadlock 7](#_Toc204368458)

[7.1 方法01：終止程序（Process Termination） 7](#_Toc204368459)

[7.2 方法 02：Resource Preemption（資源搶回） 8](#_Toc204368460)

# 系統模型（System Model）

一個作業系統包含「有限數量」的資源，要分配給「多個執行緒（threads）」使用。資源可能會分成幾種類型（resource types），每種類型內有數個相同的實例（instances）。如果同一類資源內部的實例不完全相同，那就代表資源分類不正確。

|  |  |  |
| --- | --- | --- |
| **資源類型** | **實例數量** | **說明** |
| CPU | 4 顆 | 有 4 個可用的處理器 |
| 網路卡 | 2 個 | 有兩張網路介面卡 |
| 檔案、I/O 裝置 | 視系統而定 | DVD、硬碟、音效卡等 |

每個鎖通常對應一個特定的資料結構（如佇列、串列等），所以系統會將「每個鎖視為獨立的資源類別」。

## 資源使用的流程

每個執行緒使用資源必須遵循：Request → Use → Release

* Request（請求）：執行緒向系統請求資源。若該資源已被其他執行緒占用，則該執行緒會進入等待狀態。
* Use（使用）：成功獲得資源後，執行緒可以操作它（例如進入 critical section）。
* Release（釋放）：使用完畢後，執行緒必須歸還該資源。

# 多執行緒程式中的死結 (Deadlock in Multithreaded Applications)

以 POSIX 的 mutex 鎖（pthread）為例：

* pthread\_mutex\_t 是 POSIX 標準的互斥鎖。
* 初始化：pthread\_mutex\_init(&mutex, NULL)
* 鎖住：pthread\_mutex\_lock(&mutex)
* 解鎖：pthread\_mutex\_unlock(&mutex)

## 範例：Deadlock Example

// 建立了兩把鎖 first\_mutex 和 second\_mutex

pthread\_mutex\_t first\_mutex**;**

pthread\_mutex\_t second\_mutex**;**

// 建立兩個執行緒 thread\_one 和 thread\_two

pthread\_mutex\_init**(&**first\_mutex**,** NULL**);**

pthread\_mutex\_init**(&**second\_mutex**,** NULL**);**

// 執行緒函數：

// 執行緒一

**void** **\***do\_work\_one**(void** **\***param**)** **{**

pthread\_mutex\_lock**(&**first\_mutex**);** // 鎖住第一把鎖

pthread\_mutex\_lock**(&**second\_mutex**);** // 嘗試鎖住第二把鎖

// 做一些工作...

pthread\_mutex\_unlock**(&**second\_mutex**);** // 釋放第二把鎖

pthread\_mutex\_unlock**(&**first\_mutex**);** // 釋放第一把鎖

pthread\_exit**(**0**);**

**}**

// 執行緒二

**void** **\***do\_work\_two**(void** **\***param**)** **{**

pthread\_mutex\_lock**(&**second\_mutex**);** // 鎖住第二把鎖

pthread\_mutex\_lock**(&**first\_mutex**);** // 嘗試鎖住第一把鎖

// 做一些工作...

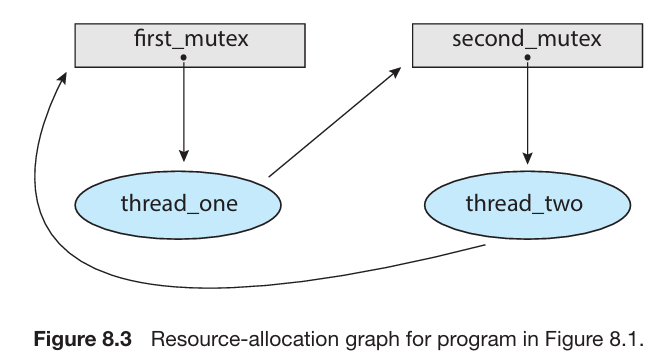
pthread\_mutex\_unlock**(&**first\_mutex**);** // 釋放第一把鎖

pthread\_mutex\_unlock**(&**second\_mutex**);** // 釋放第二把鎖

pthread\_exit**(**0**);**

**}**

Figure 1： Resource-allocation graph for program



如果發生以下情況，會造成死結（Deadlock）：

* thread\_one 成功鎖住 first\_mutex
* thread\_two 成功鎖住 second\_mutex
* 兩者都在等待對方的鎖 → 永遠卡住不動。

為何這種死結難以偵測？因為它不是每次都會發生，而是要在某個特定的排程時機（scheduling timing）下才會產生。

# 活結（Livelock）

活結與死結類似，都會讓執行緒無法繼續執行。差別是：

* 死結：執行緒被「阻塞」住，完全卡死。
* 活結：執行緒還在「跑」，但一直重試同一個失敗的動作，沒有實質進展。

**if** **(**pthread\_mutex\_trylock**(&**second\_mutex**)** **!=** 0**)** **{**

pthread\_mutex\_unlock**(&**first\_mutex**);** // 嘗試失敗就放棄再重試

**continue;**

**}**

// pthread\_mutex\_trylock不會阻塞，而是馬上回傳：

// 成功就鎖住

// 失敗就回傳錯誤值

活結的發生流程：

1. thread\_one 鎖住 first\_mutex
2. thread\_two 鎖住 second\_mutex
3. 然後thread\_one 嘗試 trylock(second\_mutex) → 失敗 → 解鎖 first\_mutex → 重試
4. 然後thread\_two 嘗試 trylock(first\_mutex) → 失敗 → 解鎖 second\_mutex → 重試
5. 他們彼此一直重試又碰撞，就像走廊中兩人互相閃避卻擋住對方。

活結的解法：在重試之間加入隨機延遲（random backoff）來避免無限碰撞。

# 死結的四個必要條件(Necessary Conditions)

|  |  |
| --- | --- |
| 條件名稱 | 說明 |
| 互斥（Mutual Exclusion） | 資源一次只能被一個執行緒佔用，不可共享。例如：鎖。 |
| 保持並等待（Hold and Wait） | 執行緒持有一部分資源的同時，還在等待其他資源。 |
| 不可搶佔（No Preemption） | 資源不能被強制奪回，只能由持有者主動釋放。 |
| 循環等待（Circular Wait） | 存在一個執行緒集合 {T0, T1, ..., Tn}，彼此環狀等待對方所持有的資源。 |

## 資源配置圖（Resource-Allocation Graph，RAG）

資源配置圖是一種用來可視化死結情況的有向圖，圖中的循環可以幫助我們判斷是否有死結。

|  |  |
| --- | --- |
| 執行緒 Ti | 圓圈表示 |
| 資源類型 Rj | 用矩形表示 |
| Ti → Rj | **請求邊（Request Edge）**，表示執行緒 Ti 正在等待 Rj |
| Rj → Ti | **分配邊（Assignment Edge）**，表示資源 Rj 的某個實例分配給 Ti |

## 範例說明：Figure 8.4

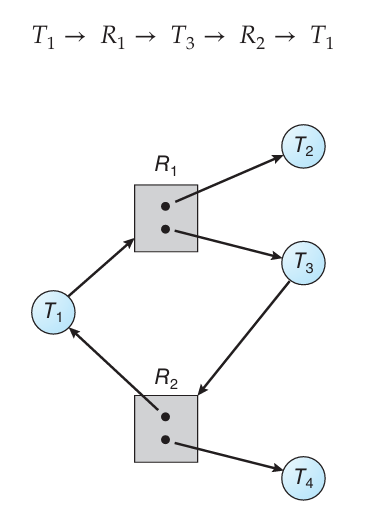
* 執行緒集合：T = {T1, T2, T3}
* 資源集合：R = {R1, R2, R3, R4}
* 資源實例數量：
  + R1、R3 各有 1 個
  + R2 有 2 個
  + R4 有 3 個
* 資源配置情形：
  + T1：擁有 R2，等待 R1
  + T2：擁有 R1 和 R2，等待 R3
  + T3：擁有 R3

|  |  |
| --- | --- |
| Figure 2：Resource-allocation graph. | Figure 3：若此時 T3 又請求 R2，但沒資源，就新增 T3 → R2 請求邊 → 死結就形成了 |

但循環也不一定會有deadlock，無死結的循環案例：

* 圖中有循環 T1 → R1 → T3 → R2 → T1
* 但：T4 正在使用 R2，有可能釋放 R2，解開循環

Figure 4：Resource-allocation graph with a cycle but no deadlock



# 死結預防（Deadlock Prevention）

死結的四個必要條件：

1. Mutual Exclusion（互斥）
2. Hold and Wait（保有並等待）
3. No Preemption（不可搶佔）
4. Circular Wait（循環等待）

## 互斥（Mutual Exclusion）

若資源是可共用的，就不需要鎖定，也不會產生死結。

可共用：如唯讀檔案（read-only file），多個執行緒可以同時讀取。

但大多數資源（如 mutex、I/O 裝置）是天生不可共用的，因此這條通常無法被破壞。

## Hold and Wait（保有並等待）

* 方法一（全部一次請完）：
  + 規定執行緒必須在開始執行前，一次請求全部資源。
  + 如果有任何資源還沒拿到，就不能開始。
* 方法二（只能在沒持有時請求）：
  + 如果一個程式已經持有某些資源，它不能直接再要求新資源
  + 如果它真的想請求新的，就必須先把目前拿到的資源釋放掉。

問題：資源利用率低（可能先佔著不用）且容易飢餓（一直請不到一整組資源）

## No Preemption（不可搶佔）

預防方法：若某個執行緒請求的資源無法立即取得，則把它持有的資源通通搶回來（preempted），讓它重新排隊。

類似機制：被搶回的資源會加入等待佇列，執行緒等到它原本的資源與新請求的資源全部都能取得時才繼續執行。

限制：

* ✅ 適用於可保存狀態的資源（如 CPU 資源、資料庫交易）
* ❌ 不適用於 mutex、semaphore（狀態難保存、難回復）

## Circular Wait（循環等待）

解法：資源編號 + 遞增順序請求

* 先給所有資源設定一個「唯一整數編號」F(Ri)。
* 規定執行緒請求資源時只能照編號遞增順序請求：
* 若要請 Rj，必須釋放所有 Ri 且 F(Ri) ≥ F(Rj)。可以先請 R1 再請 R5，但不能先請 R5 再請 R1。

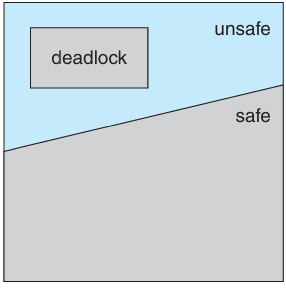
# Deadlock Avoidance

與其像 Deadlock Prevention 一開始就限制資源請求方式，Deadlock Avoidance 的策略是：在執行緒每次請求資源時，動態判斷分配這筆資源會不會導致未來進入死結。

## Safe State（安全狀態）

安全狀態（safe state）代表：存在一個執行緒的順序 <T1, T2, ..., Tn>，讓每個執行緒在資源可用時最終都能成功完成並釋放資源。

Figure 5：Safe, unsafe, and deadlocked state spaces



安全狀態 ≠ 死結狀態。

* 安全 → 絕不會死結
* 不安全 → 不一定死結，但有可能進入死結
* 死結狀態 → 已經死結，是不安全狀態的一種

## 相關演算法(這格算重要，但太多了)

* 資源配置圖演算法（只有一個資源實例時可用）
* 銀行家演算法（Banker’s Algorithm）
* 安全狀態演算法（Safety Algorithm）

# 死結復原(Recovery from Deadlock

## 方法01：終止程序（Process Termination）

|  |  |
| --- | --- |
| 終止方式 01：全部終止 | 終止方式 02：逐一終止（partial termination） |
| 把所有死結相關的執行緒都終止。 | 一次只殺掉一個執行緒，並每次都重新檢查是否還有死結。 |
| ✅ 簡單有效，一定會解除死結 | ✅ 節省資源損失 |
| ❌ 缺點：計算結果全丟失，損失很大。 | ❌ 缺點：檢查次數多，效率差 |

執行緒選擇策略（誰該被終止？）：這屬於 policy decision（策略判斷），可依下列因素選擇成本最低的受害者：

|  |  |
| --- | --- |
| **考慮因素** | **說明** |
| 優先權 | 優先權低者優先終止 |
| 已經執行多久？還剩多久？ | 越早終止代價越小 |
| 佔用了多少資源？ | 資源越多 → 成本越高 |
| 還需要多少資源？ | 需求大者可能更難完成任務 |
| 需要終止多少人？ | 若終止一人即可解決，最好別動太多人 |

## 方法 02：Resource Preemption（資源搶回）

這是另一種選擇：不殺執行緒，而是搶資源。會有三個關鍵問題：

* 選擇受害者（Select a victim）：依照成本考量，選擇「損失最小」的犧牲者開始搶佔資源
* 回滾（Rollback）：搶走資源後，該執行緒不能繼續跑，必須回到某個安全狀態（Safe State）重來。
* 避免飢餓（Avoid Starvation）：若一直搶某個執行緒的資源，它可能永遠無法完成工作 ⇒ 飢餓（Starvation）
  + 解法：限制該執行緒被選為犧牲者的次數（例如：超過 3 次就優先保護它）