Operating Systems
作業系統

SYNCHRONIZATION AND DEADLOCK
同步與死結

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
- 死結避免
- 摘要

行程同步 (Process Synchronization)

- 多個行程同時去存取相同的資料時，會因爲不同的指令執行順序而得到不同結果的現象
  - 稱爲**競爭情況**(Race Condition)
- 爲了避免競爭情況的發生
  - 同一時間只能讓一個行程去存取一個變數。
  - 行程之間需要互相**同步**，讓一個行程更改資料的動作不會影響到其他行程的執行結果。

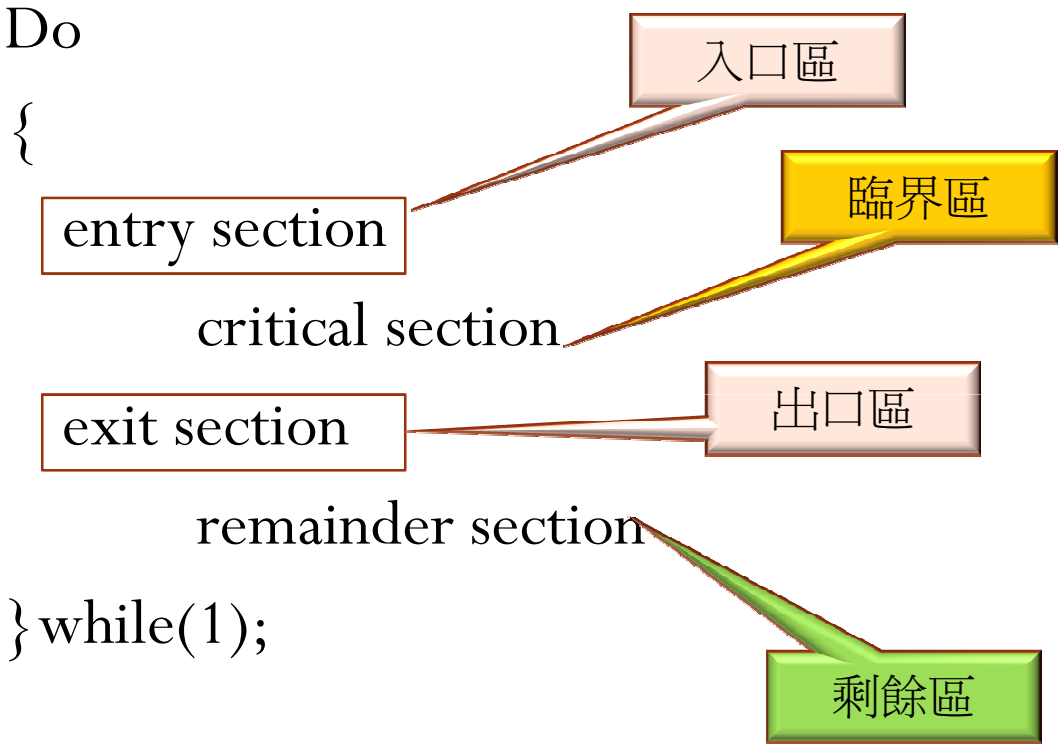
同步與死結

- 行程同步
- 臨界區
  - 交替演算法
  - 旗標演算法
  - 綜合演算法
  - 麵包店演算法
  - 硬體支援
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
- 死結避免
- 摘要

臨界區 (Critical Section)

- **臨界區**是一段**不能**讓多個行程同時執行的程式碼。
  - 系統中的某個行程在執行臨界區的這段程式時，其他的行程不能在這段時間內進入同一個臨界區執行。
  - 可以解決因行程**共享資料**而造成**資料可能不一致**的問題。
  - 臨界區的設定必須同時符合**互斥**(Mutual Exclusion)、**進行**(Progress)與**有限等待**(Bounded Waiting)三項條件。
- 位在臨界區之前負責協調行程的程式碼稱之爲**入口區**
- 在臨界區之後會接著一個**出口區**，負責處理出臨界區後的動作。
- 而剩餘的程式部分則稱之爲**剩餘區**

臨界區 (Critical Section) (Cont.)

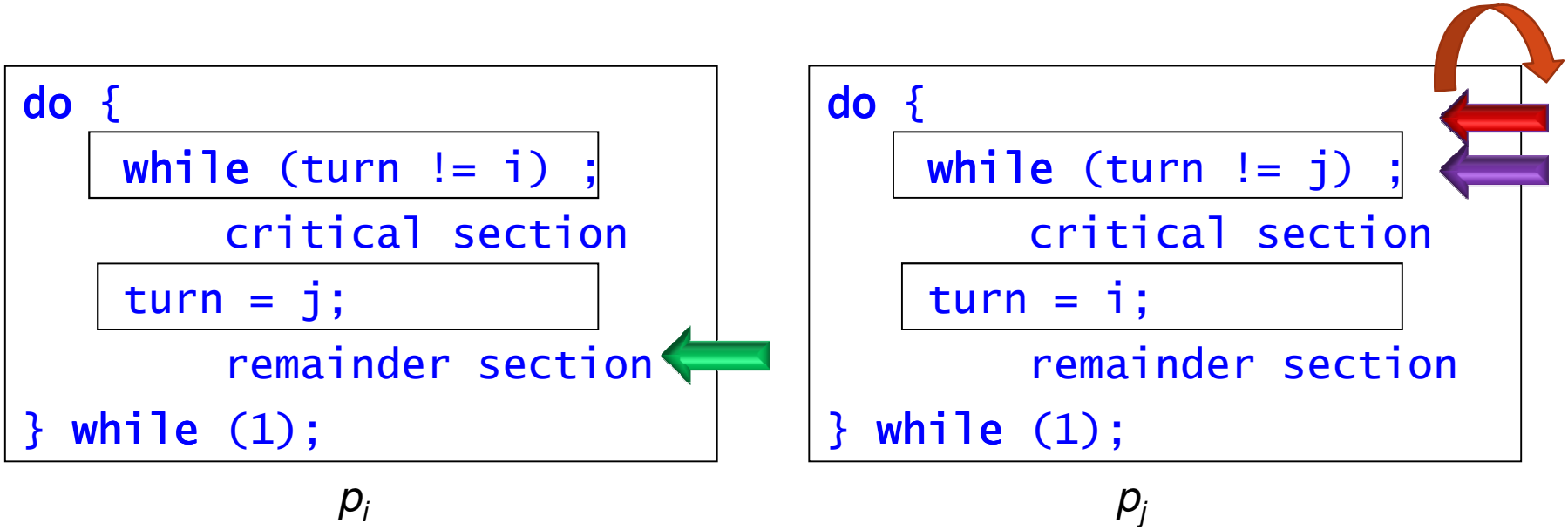


交替演算法 (Turn Algorithm)

- 兩個行程 P_i 及 P_j 之間的臨界區演算法。
  - 共用一個變數 turn，指出目前允許進入臨界區的是哪一個行程。
  - 只記錄系統目前的狀態，但是並不記錄行程個別的狀態。
  - 如果 turn = i，代表行程 P_i 可以進入臨界區之中。
  - 滿足臨界區**互斥**的條件，但是**不能滿足進行**的條件。

```
do {
    while (turn != i) ;
        critical section
    turn = j;
        remainder section
} while (1);
```

交替演算法 (續)



p_i p_j

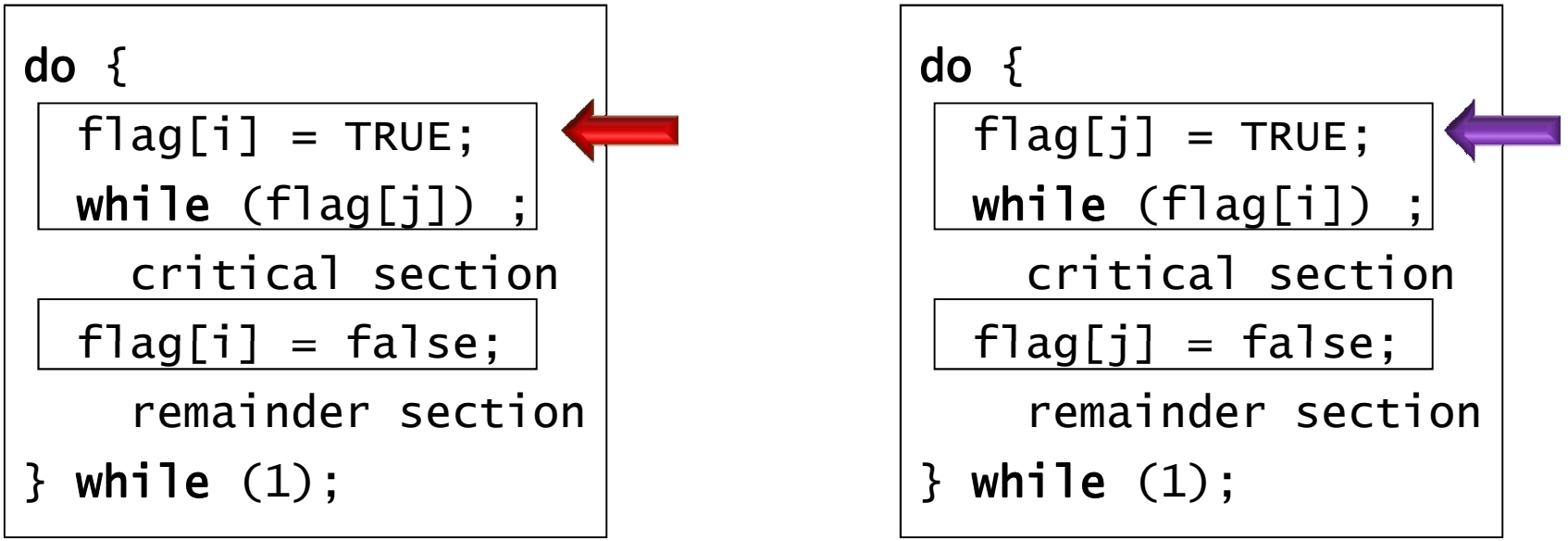
turn = j

turn = i

旗標演算法

- 兩個行程P_i 及 P_j之間的臨界區演算法。
  - 將交替演算法中**共有的變數** turn 改爲共有的**陣列** flag，**記錄**系統中**個別行程**的**狀態**。
  - 如果 P_i 要進入臨界區，則將 flag[i] 設爲 TRUE。
  - 當 P_i 離開臨界區後，再將 flag[i] 設爲 FALSE。
  - 滿足臨界區**互斥**的條件，但是仍然**無法滿足進行**的條件。

旗標演算法 (續)


do {
flag[i] = TRUE;
while (flag[j]) ;
critical section
flag[i] = false;
remainder section
} while (1);
do {
flag[j] = TRUE;
while (flag[i]) ;
critical section
flag[j] = false;
remainder section
} while (1);

p_i p_j

綜合演算法

- 兩個行程 P_i 及 P_j 之間的正確臨界區演算法。
  - 綜合**交替演算法**與**旗標演算法**。
  - 以 flag 陣列記錄個別行程是否想要進入臨界區；而 turn 變數指出目前系統允許哪個行程進入臨界區。
  - 同時滿足互斥、有限等待與進行三個條件。

綜合演算法 (續)

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j) ;
      critical section
   flag[i] = false;
      remainder section
} while (1);
```

p_i

p_j

```
do {
   flag[j] = TRUE;
   turn = i;
   while (flag[i] && turn == i) ;
      critical section
   flag[j] = false;
      remainder section
} while (1);
```

麵包店演算法

- **多個行程**間的臨界區演算法。
  - 以行程取到的**號碼牌**，**由小而大**地讓行程進入臨界區中。
  - 若有行程取到**相同號碼**，則以行程的 ID 大小決定先後順序，ID 較小的優先。
  - 同時滿足互斥、有限等待與進行三個條件。

麵包店演算法 (續)

- 行程 P_i 的程式結構如下：

P_i 是否在選號碼

P_i 選號碼

```
do {
   choosing[i] = TRUE;
   number[i] =
          max(number[0], number[1], …, number[n – 1]) + 1;
   choosing[i] = FALSE;
   for (j = 0; j < n; j++) {
     while (choosing[j]) ;
     while ((number[j] != 0) &&
          ((number[j], j) < (number[i], i))) ;
   }
     critical section
   number[i] = 0;
     remainder section
} while (1);
```

硬體支援

- 在單 CPU 的系統中，可以很簡單地讓行程在修改**共用的變數**時停止接受中斷而解決同步的問題。
- 但是在**多 CPU 的系統中**並不合適，因爲停止所有 CPU 的中斷除了很耗時間之外，也會增加行程進入臨界區所花費的時間。
- 除了利用程式技巧來達到同步之外，可以將**同步的機制**設計在**硬體**上。
  - 撰寫同步程式變得更加方便，並同時**提升系統效率**。
  - 許多機器實作了特殊的**硬體指令**，可以不被中斷地檢查並設定一個字元組的內容、或是交換兩個字元組的內容。
  - 利用這些指令可以很簡單地解決臨界區的問題。

Test-and-Set 硬體指令

- 在執行完整個指令之前都不會被中斷。
- Test-and-Set 指令會傳回參數 target 目前的值，並同時將 target 的值設爲 TRUE。

```
boolean Test-and-Set (Boolean &target) {
    Boolean rv = target;
    target = true;
    return rv;
}
```

- 可以利用 **Test-and-Set** 指令實作多行程的臨界區演算法。

```
do{
  while(Test-and-Set(lock));
     critical section
  lock = FALSE;
     remainder section
}while(1)
```

Swap 硬體指令

- 在執行完整個指令之前都不會被中斷。
- **Swap** 指令會交換參數 a 與 b 兩個字元組的內容。

```
Void Swap (Boolean &a, Boolean &b) {
    Boolean temp = a;
    a = b;
    b = temp;
}
```

```
do{
  key = TRUE;
  while(key == TRUE)
      Swap(lock, key);
    critical section
  lock = FALSE;
    remainder section
}while(1)
```

同步與死結

- 行程同步
- 臨界區
- 號誌
  - 計數號誌
  - 二元號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
- 死結避免
- 摘要

號誌 (Semaphore)

- 號誌是十分常用的同步工具，可以簡單地解決較爲複雜的**同步問題**。
  - 大部分的作業系統都已經實作了號誌，作爲行程間同步的工具。
  - 號誌包含一個數值，該值在初始化之後就只能經由 **signal()** 與 **wait()** 兩個不可被中斷的函式去存取。
  - 當一個行程在存取號誌的值時，其他行程無法存取同一個號誌的值。

```
Wait(S){
  while(S<=0);
    /* 等待 */
  S--;
}
```

```
signal(S){
  S++;
}
```

計數號誌 (1) (Counting Semaphore)

- 計數號誌的值像一個**計數器**，記錄著有多少行程可以再進入臨界區。
  - 號誌的值只能利用 **signal()** 與 **wait()** 存取。
  - **signal()** 會將號誌的值加 1。
  - **wait()** 則會先測試號誌的值，如果號誌的值大於零，會將該值減 1，否則 **wait()** 會等待到該值大於零再繼續執行。
  - 解決多行程臨界區的問題。
  - 達到**行程之間**的**同步**。

```
do{
   wait(mutex);
      critical section
   signal(mutex);
      remainder section
}while(1);
```

計數號誌 (2)

- 臨界區實作方法有使用**忙碌等待**的**缺點**。
  - 使用**忙碌等待**的**號誌**也被稱爲**旋轉鎖(Spin lock)**。
  - 如果已經有行程進入了臨界區，那麼其他想要進入臨界區的行程都會在入口區中一直地執行迴圈來等待。
  - 使用忙碌等待的行程因爲**不會被內文切換**，所以如果忙碌等待太久會浪費許多 CPU 的時間。
  - 但是如果忙碌等待的時間比內文切換的時間更短，反而可以省下內文切換的額外負擔。

計數號誌 (3)

- 爲了避免忙碌等待所造成的 CPU 資源浪費，可以修改 **wait()** 和 **signal()** 兩個函式。
  - 不再讓行程忙碌等待，直接讓行程將**自己阻隔起來**。
  - 新的號誌結構除了原有的整數可記錄號誌的值，還另外增加了一個**串列**，**記錄正在該號誌等待的行程**。
  - **wait()** 會將呼叫的行程放入該號誌的串列之中，並且將行程阻隔，而 CPU 排程器會另選出**就緒佇列**中的其他行程來執行。
  - 當 **signal()** 被呼叫之後，會由該號誌的串列中叫醒一個被阻隔的行程繼續執行。
  - 新的架構中，號誌的值可能會是**負值**，而這個負值的絕對值就是**號誌串列中被阻隔**的**行程數目**。

計數號誌 (4)

```
Void Wait(S){
  S.value--;
  if(S.value<0){
    將行程加入S.L中;
    block();
  }
}
```

```
void signal(S){
  S.value++;
  if(S.value <= 0){
    由 S.L 之中移除一個行程 P;
    wakeup(P);
  }
}
```

- 號誌中的串列如果使用不適當的實作方式，例如**後進先出串列**，可能會造成**無限阻隔**或是**飢餓現象**。
  - 可以使用**先進先出串列**來實作**號誌串列**。
  - 必須保證 **wait()** 和 **signal()** 在執行的過程中不會被中斷。
- 修改過後的號誌雖然不能完全不使用忙碌等待，但是大幅地縮短了忙碌等待的時間。
  - 由整個臨界區縮短到只有實作 **wait()** 和 **signal()** 的臨界區所造成的忙碌等待時間。
  - 可以提高系統的效能。

二元號誌

- 二元號誌的值只限定爲 0 或 1。
  - 利用**硬體**對二元數值的運算支援，二元號誌的實作要比**計數號誌簡單快速**得多。
  - 一個計數號誌 S 可以利用二元號誌及一個整數來實作計數號誌。
  - S1 初始爲1，S2初始爲0

```
Void wait(S){
  wait(S1);
  c--;
  if(c<0){
    signal(S1);
    wait(S2);
  }
  signal(S1);
}
```

```
Void signal(S){
  wait(S1);
  c++;
  if(c <= 0)
    signal(S2);
  else
    signal(S1);
}
```

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
  - 有限緩衝區問題
  - 讀取者與寫入者問題
  - 哲學家晚餐問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
- 死結避免
- 摘要

同步的經典問題

- 牽涉到了大型**並行控制**(Concurrency control)的領域。
- 經常拿來測試新的**同步機制**的正確性。
- 這些問題的解法當中都使用了號誌作爲同步的工具。

有限緩衝區問題

- 用**號誌**來實作可簡單解決許多問題。
- 假設緩衝區中有 n 個欄位，每個欄位可以存放一個項目。
- 使用 mutex 號誌確保**存取緩衝區時**的**互斥條件**成立，並初始化爲 1。
- empty 和 full 兩個號誌則分別用來計算緩衝區中空的與使用過的欄位數目。
  - empty 號誌初始化爲 n。
  - full 號誌初始化爲 0。

有限緩衝區問題 (續)

- 生產者和消耗者的程式碼如下：

生產者	消耗者

do {
…
產生一個新的項目放在 nextp
…
wait(empty);
wait(mutex);
…
將 nextp 加入到緩衝區中
…
signal(mutex);
signal(full);
} while(1);
	do {
	wait(full);
	wait(mutex);
	…
	將一個項目由緩衝區中
	移到 nextc
	…
	signal(mutex);
	signal(empty);
	…
	消耗放在 nextc 中的項目
	…
	} while(1);

讀取者與寫入者問題

- 一個系統中經常會有**數個行程共同分享同一份資料**物件
  - **只讀取**這份分享資料的行程稱爲**讀取者**。
  - **只更新**分享資料的行程稱爲**寫入者**。
  - 如果一個讀取者和一個寫入者同時存取所共享的資料，可能會發生錯誤。
  - 這種同步的問題稱爲讀取者與寫入者問題。
  - 以整數 readcount 記錄正在讀取資料的讀取者數目，初始值爲 0。
  - 利用初始化爲 1 的 mutex 和 wrt 兩個**號誌**形成**兩種臨界區**。

讀取者與寫入者問題 (續)

- 讀取者和寫入者問題的解法如下：

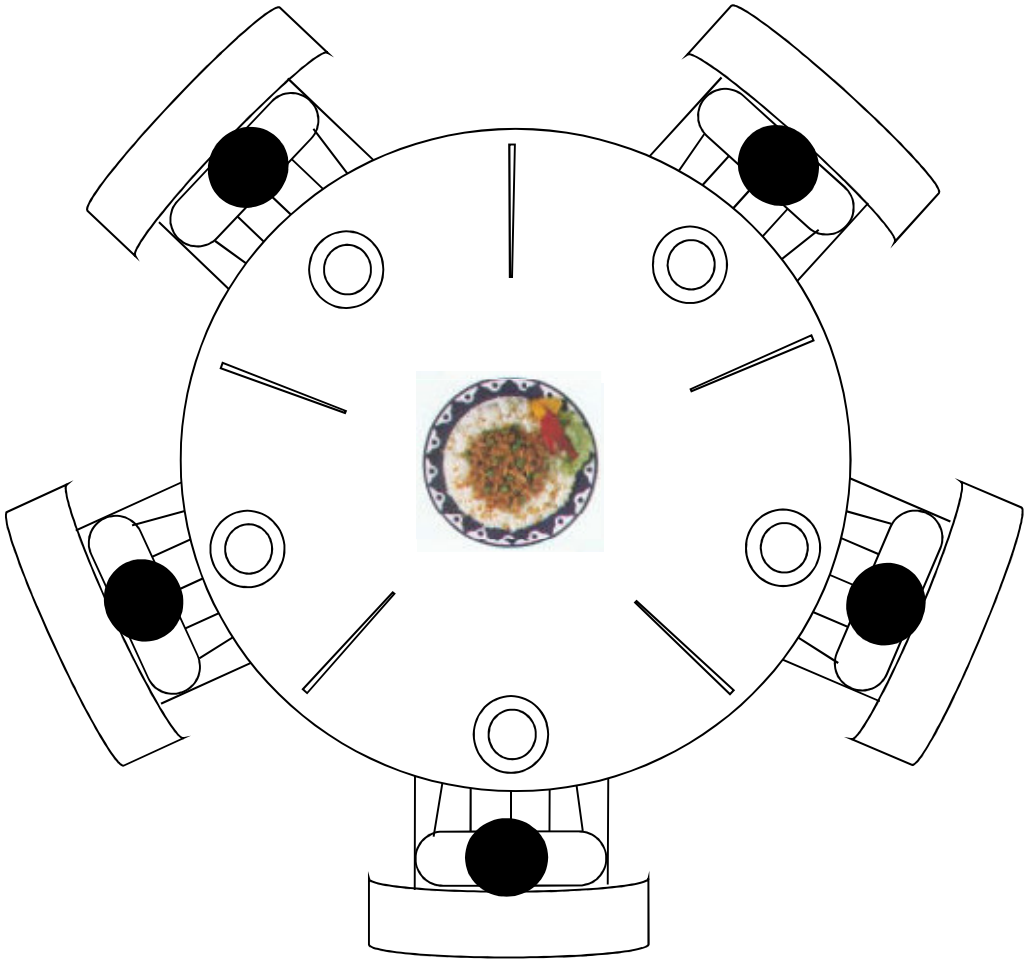
讀取者	寫入者

wait(mutex);
readcount++;
if (readcount == 1)
wait(wrt);
signal(mutex);
…
進行讀取
…wait(mutex);
readcount--;
if (readcount == 0)
signal(wrt);
signal(mutex);
	wait(wrt);
	…
	進行寫入
	…
	signal(wrt);

哲學家晚餐問題 (1)

- 哲學家們圍坐在一張圓桌一起共進晚餐。
- 桌上筷子數目與哲學家相等，每兩個哲學家之間共用一支筷子。
- 哲學家們會坐在餐桌上思考哲學問題。
- 當哲學家們想吃東西時會拿起左右各一支筷子進餐，拿齊兩支筷子的哲學家們可同時進餐，但**不能搶奪**別人手上的筷子。
- 當食物吃完後，哲學家會將兩支筷子都放下，並繼續思考哲學問題，其他哲學家可繼續用放下的筷子。

哲學家晚餐問題示意圖



哲學家晚餐問題 (2)

- 第 i 位哲學家的程式可寫成：

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i + 1) % 5];
        …
        進餐
        …
    signal(chopstick[i]);
    signal(chopstick[(i + 1) % 5];
        …
        思考
        …
} while(1);
```

哲學家晚餐問題 (2)

- 上述作法可能會導致**死結**的發生。
  - 假如每位哲學家都拿起了一支筷子而等待另一支筷子，則不會有任何一位哲學家可以拿到兩支筷子，所有的哲學家將會進入死結狀態。
- 下列作法可以**避免**哲學家晚餐問題發生死結：
  - 在圓桌上放置 n+1 支筷子或限制最多只有 n-1 位哲學家可以同時進餐。
  - 規定哲學家們必須要同時拿起左右兩支筷子。
  - 規定奇數座位的哲學家要先拿起左方的筷子再拿起右方的筷子；而偶數座位的哲學家則先拿起右方的筷子再拿起左方的筷子。

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
  - 臨界區域
  - 監督程式
- 死結簡介
- 死結預防
- 死結避免
- 摘要

臨界區域與監督程式

- 號誌是個非常方便而且有效率的同步工具。
  - 然而不正確地使用號誌，很容易產生行程之間同步上的錯誤。
  - 這類錯誤並不一定每次程式執行時都會發生，所以**很難除錯**。
- **臨界區域**(Critical Region)與**監督程式**(Monitor)是另外兩種較高階的常用同步工具
  - 使用起來較爲方便，而且不容易出錯。
  - 每個行程會有一些私有的局部變數，以及一些行程間的**共享變數**。
  - 行程不能夠直接存取其他行程的局部變數。

臨界區域

- 臨界區域的使用非常方便。
  - 以下宣告一個具有**共享變數** v 的臨界區域，在 B 條件式成立下，如果沒有其他行程在此臨界區域中執行，就會執行 S 敘述：

```
region v when B do S;
```

- 利用**臨界區域**來實作，程式設計師不用煩惱同步的問題，只要正確地把問題描述在臨界區域內。
- **有限緩衝區問題**可以用臨界區域來簡單地解決同步的問題。

臨界區域 (續)

- 臨界區域 **region** v **when** B **do** S 可利用 mutex、first_delay 及 second_delay 三個號誌實作。
  - mutex 號誌是用來確保臨界區的互斥條件成立。
  - 如果行程因爲 B 爲 FALSE 而無法進入臨界區，該行程將會在號誌 first_delay 等待。
  - 在號誌 first_delay 等待的行程重新檢查 B 值之前，會離開號誌 first_delay，而在號誌 second_delay 等待。
  - **分成first_delay 與 second_delay兩段式等待的原因，是爲了要避免行程持續忙碌地檢查 B 值**。
  - 當一個行程離開了臨界區之後，可能因爲執行了敘述 S 而改變了 B 的值，所以需要重新檢查。

監督程式

- 監督程式是另外一種高階的同步工具。
  - 對較複雜的同步問題提供了更一般性的實作工具。
  - 由一些變數宣告及函式所組成，變數的值定義了監督程式的狀態。
  - 保證只有一個行程在監督程式中執行所定義的函式。
  - 在監督程式中，程式設計師不需要撰寫有關同步的程式碼，但是可以利用條件變數來定義自己的同步機制。
  - 哲學家晚餐問題可以利用監督程式來實作。

```
monitor monitor-name{
        // shared variable declarations
        procedure P1 (…) { …. }
                …
        procedure Pn (…) {……}

  initialization code ( ….) { … }
}
```

監督程式

```
monitor DP {
    enum {THINKING; HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
            // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
    }
}
```

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
  - 死結特性
  - 死結偵測
  - 死結解除
- 死結預防
- 死結避免
- 摘要

死結簡介

- 許多行程會**共同競爭**系統中**有限的資源**。
- 當行程所要求的資源正被其他的行程所使用時，該行程就必須要**等待**。
- 等待的行程可能會因爲所要求的資源也被其他正在等待的行程所持有，而**無限期地等待**，這種情形稱爲**死結**。

死結特性 (1)

- 系統中行程共同競爭的有限資源可以分爲幾種不同的類型，而每種類型又會有不同數目的資源。
- 一個行程可以對一個資源進行**要求**、**使用**、**釋放**等 3 種動作。
- 行程不能要求比系統擁有數目更多的資源。
- 當行程要求了某種資源時，作業系統會先檢查系統的**資源配置表**，如果該種資源都正被其他行程所使用，作業系統會將目前要求的這個行程加入所等待資源的**等待串列**中。

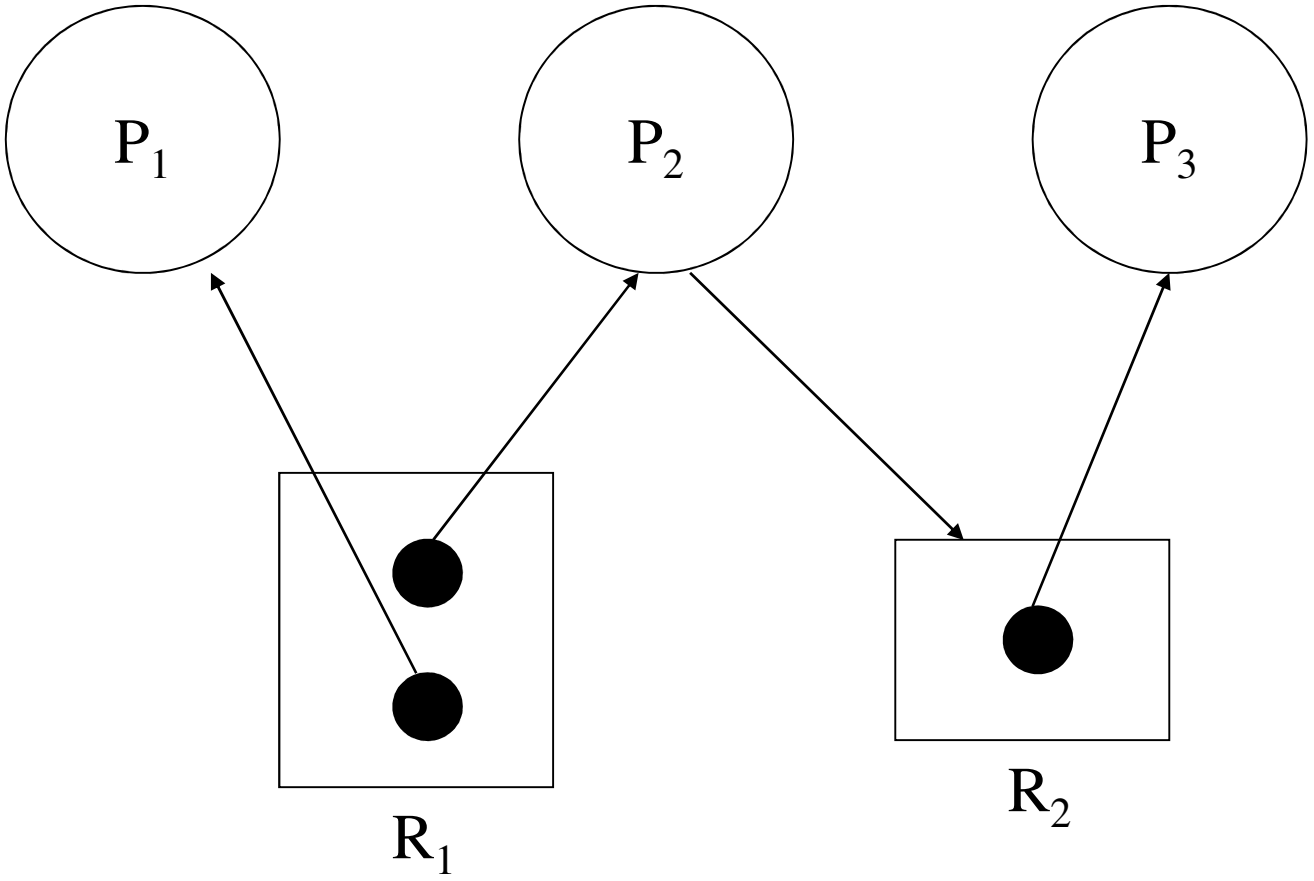
死結特性 (2)

- 死結的**定義**如下，**系統中的每一個行程都在等待著某些資源，而這些資源卻已經配置給其他正在等待的行程，因而所有的行程都進入無限期地等待而無法完成工作**。
- 死結只有在下列 4 種**條件**都**同時成立**下才會發生：
  - 互斥執行 (Mutual exclusion)
  - 佔用與等待 (Hold and wait)
  - 禁止搶先 (No preemption)
  - 循環等待 (circular waiting)

死結特性 (3)

- 我們可以使用**系統資源配置圖**來描述系統中**行程**與**資源**間的狀態。
  - **圓**代表系統中的一個**行程**，圓中寫著行程的名稱。
  - **方塊**代表系統中的一種**資源**，方塊中**黑點**的**數量**表示該種**資源的數目**。
  - **箭號**所代表的意義有兩種。
    - 由資源指向行程的箭號表示該資源目前被該行程所持有。
    - 由行程指向資源的箭號則代表該行程目前正在等待該項資源。

資源配置圖



死結偵測 (deadlock detection))

- 如果系統中**所有類型的資源**都**只有一項**的話，我們可以利用**資源配置圖**來**偵測死結**。
  - 利用偵測**迴圈**的演算法來檢查資源配置圖中是否有迴圈存在，就能知道目前系統中是否有死結發生。
  - 使用這種方法的系統必須要持續地更新資源配置圖，並且要**定期地執行偵測迴圈的演算法**以偵測死結。
- 如果系統中各類型**資源的數目不只一項**的話，可以使用**死結偵測演算法**來偵測死結。

死結解除 (Recovery from Deadlock)

- 當偵測到死結發生，有兩種方法可以**解除死結**。
  - 終止一些行程，**使循環等待的條件不成立**。
  - 由發生死結的行程中回收一些資源給其他行程，**使禁止搶先的條件不成立**。
- 終止行程的作法中，可以選擇終止所有行程或是一次只終止一個行程，直到死結的狀態解除。
  - 終止所有行程的作法雖然比較簡單，但是行程的工作必須重新或是部分執行，**會浪費較多的 CPU 時間**。
  - 一次終止一個行程所浪費的 CPU 時間較少，但是每終止一個行程之後都必須要執行一次偵測死結的動作來判定死結是否已經解除。

死結解除 (續)

- 用**回收資源**的作法來解除死結，必須持續地由一些行程回收資源，並將回收的資源配置給其他的行程，直到死結的狀態解除。這個作法有幾點需要注意：
  - **選定行程的方法**
  - **回溯** (rollback): 回收行程的資源後，必須將行程回溯到一個安全的狀態，要達到這樣的目的，作業系統必須收集更多資訊
  - **飢餓現象**

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
  - 互斥
  - 佔用與等待
  - 禁止搶先
  - 循環等待
- 死結避免
- 摘要

死結預防 (Deadlock Prevention)

- 死結的發生要 4 個條件同時成立。
  - 互斥
  - 佔用與等待
  - 禁止搶先
  - 循環等待
- 只要破除死結的任一個條件就可以預防死結的發生。

互斥執行

- 互斥執行的條件**只存在**於**不能共享的資源**上。
  - 如印表機不能同時列印兩份不同的文件。
  - 可以共享的資源能夠允許多個行程同時使用，因此可以共享的資源不可能造成死結的發生。
  - 但是，我們無法讓不可共享的資源變成可共享，因此無法利用資源的互斥條件來預防死結的發生。

佔用與等待

- 不讓佔用與等待的情形在系統中發生，必須要讓所有的行程在取得某項資源時不得先持有任何其他的資源。
  - 行程在執行前必須**要一次取得所有需要的資源**，但是這個作法比較沒有效率。
  - 或是行程在取得任何資源之前必須**要釋放所有持有的資源**。
  - 可能造成資源使用率降低或是發生飢餓現象。

禁止搶先

- 禁止搶先是指資源一旦配置給行程，就必須等到行程使用完，資源才會被釋放，要解除禁止搶先可以使用以下作法：
  - 當行程持有某些資源並要求新的資源，如果所要求的資源正被使用而必須等待，該行程必須釋放所有持有的資源。
  - 當行程 A 要求某些資源，如果所要求的資源可以使用，就立即配置；但是如果所要求的資源已經配置給其他的行程 B，檢查 B 是否正在等待其他的資源，如果是的話便將 A 所要求的資源由 B 回收並配置給 A。

循環等待

- 要使循環等待的條件不會發生，我們可以將系統中的所有**資源類型**都**事先編號**。
  - 規定所有行程必須按照編號順序（如由小而大）取得資源。
  - 行程必須要先釋放所持有編號較大的資源，才能要求編號較小的資源。
  - 會有資源使用率降低的問題。

同步與死結

- 行程同步
- 臨界區
- 號誌
- 同步的經典問題
- 臨界區域與監督程式
- 死結簡介
- 死結預防
- 死結避免
  - 安全狀態
  - 資源配置圖演算法
  - 銀行家演算法
- 摘要

死結避免 (Deadlock Avoidance)

- **預防死結**的方法大多會**降低資源的使用率**，而導致系統的效能降低。
- 死結避免雖然不完全排除死結發生的條件，但能有效地**偵測系統可能發生死結的狀態**，進而避免死結的發生。
  - 需要系統提供行程額外的資訊。
  - 當有行程要求資源，系統會利用這些資訊判斷是否要將資源配置給行程或者是讓行程等待以避免死結的發生。

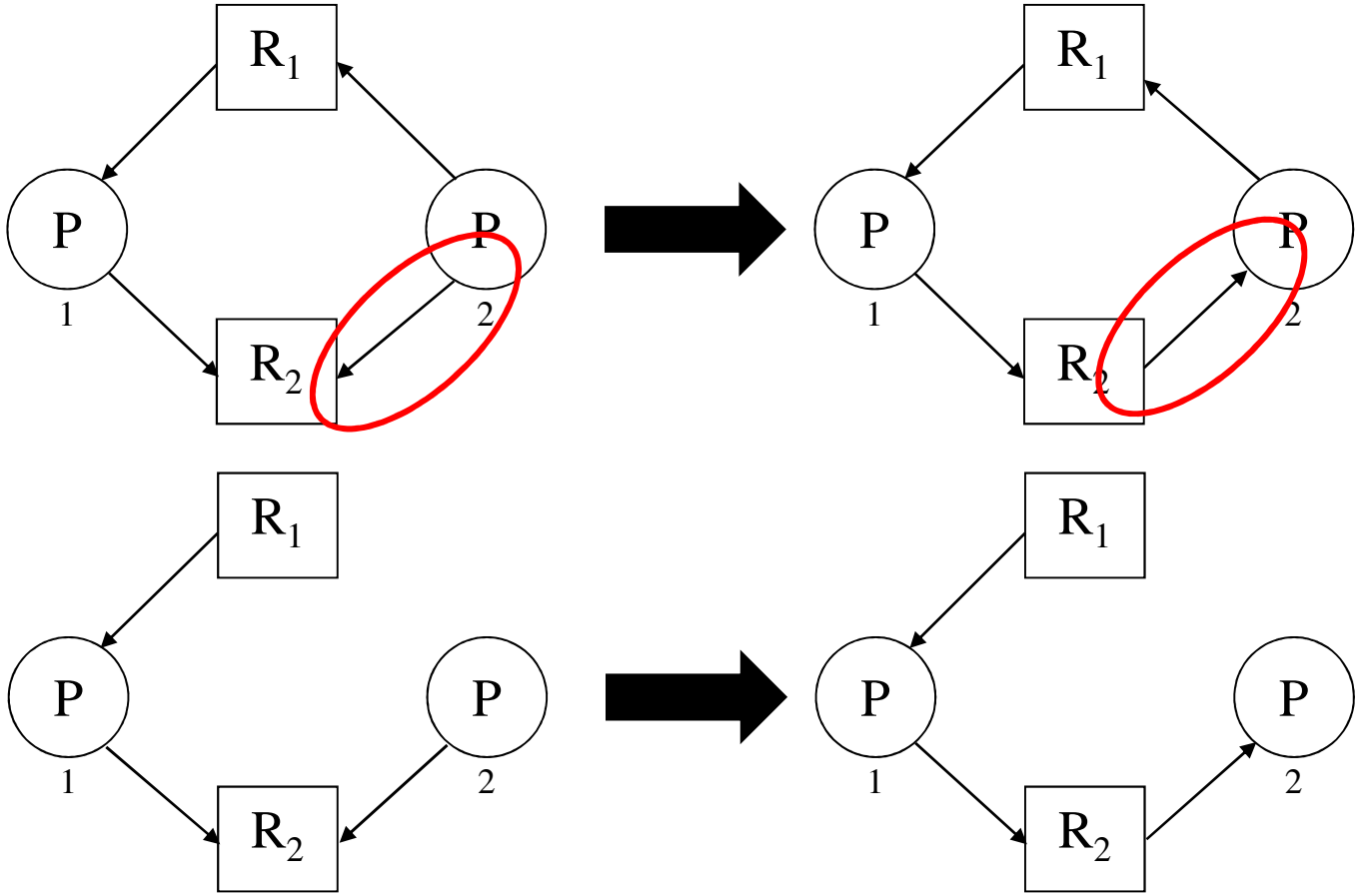
安全狀態

- 安全狀態是指**存在某種順序**，可以讓系統按照該順序將資源配置給所需要的行程，而不會發生死結。
  - 系統正處於安全狀態，存在**一組安全序列**。
  - 如果系統不存在一組安全序列，則表示系統正處於**不安全狀態**中，即系統**可能會發生死結**。
  - 不安全狀態不一定就會發生死結，但是處於安全狀態絕對不會發生死結，這是因爲系統使用資源的真正時間是無法確定的。

資源配置圖演算法

- 資源配置圖演算法在系統配置某項資源給行程之前
  - 先將配置圖中的**箭號反向**。
  - 然後使用偵測迴圈的演算法檢查新的配置圖中是否會出現**迴圈**。
  - 如果不會，則代表配置該資源後系統仍處於安全狀態，所以不會發生死結。
  - 反之，則代表配置該資源後，系統會進入不安全狀態而可能發生死結，因此系統不應該配置該項資源給該行程。
  - 不適用於有多項同種資源的系統之中。

(不)安全狀態的資源配置



銀行家演算法

- 每一個新進入到系統的行程必須要先**註冊**所需要的各種資源的**最大數量**。
- 當行程要求某些資源時，系統便判斷這樣的配置是否會導致系統進入不安全狀態。
- 使用**安全演算法**來測試系統**是否處在安全狀態**。
- 使用**資源要求演算法**來決定**是否允許資源的要求**。

安全演算法（Safety Algorithm）

1. 宣告兩個長度分別爲 m 與 n 的陣列 Work 與 Finish，並將 Work 初始化爲 Available，Finish 陣列中所有元素初始爲 FALSE。
2. 尋找 i 使得 Finish[i] = FALSE 而且 NEED[i] ≤ WORK[i]，如果找不到這樣的 i，執行步驟 4。
3. Work[i] = Work[i] + Allocation[i];

   Finish[i] = TRUE;

   執行步驟 2。
4. 如果 Finish 陣列中所有元素都爲 TRUE，則系統目前處於安全狀態中。

資源要求演算法
(Resource Request Algorithm)

1. 宣告 n × m 的 Request 陣列存放行程所要求各項資源的數量，如 Request[i, j] = 3 表示行程 P_i 要求 3 項資源 R_j。
2. 如果 Request[i] ≤ Need[i]，執行步驟 3；否則因爲行程要求過多的資源而發生錯誤。
3. 如果 Request[i] ≤ Available[i]，則執行步驟 4；否則因爲目前系統中尚未配置的資源不足，行程 P_i 必須等待。
4. 作以下的運算：

```
Available[i] = Available[i] – Request[i];
Allocation[i] = Allocation[i] + Requset[i];
Need[i] = Need[i] – Request[i];
```

使用安全演算法檢驗運算後的結果，如果處於安全狀態則允許配置該資源給 P_i；否則P_i 必須等待，並且回存步驟 4 執行前的結果。

行程與資源的配置

	Max			Allocation			Available		
行程	需要資源數目			持有資源數目			系統未配置資源數目		
	A	B	C	A	B	C	A	B	C
P_1	8	0	2	5	0	0	3	1	2
P_2	5	2	1	3	1	0			
P_3	1	2	2	0	1	2			
P_4	7	6	4	2	5	2			
P_5	3	3	5	0	2	3			

摘要 (1)

- 要讓數個行程共享一些資料變數，必須要避免資料的不一致。
  - 有一些不同的臨界區演算法可以滿足互斥的要求，不過這些解決方式因爲使用了忙碌等待而降低了系統的效能。
  - 使用號誌可以較有效率地解決大部分同步的問題。
  - 臨界區域與監督程式能較高階與方便地解決較複雜的同步問題。

摘要 (2)

- 同步的經典問題
  - 包括有限緩衝區問題、讀取者與寫入者問題、哲學家晚餐問題。
  - 牽涉到了大型並行控制的問題。
  - 被經常拿來測試新設計的同步機制。
- 數個行程共享一些系統資源，就可能發生死結。
  - 只有在互斥、佔有與等待、禁止搶先、循環等待等 4 個條件同時成立時才會發生。
  - 破除其中的任一個條件，可以預防死結的發生。

摘要 (3)

- 可以利用不同的演算法來避免死結的發生。
  - 避免死結比預防死結的演算法較不會降低系統資源的使用率，但是系統需要記錄較多有關行程的資訊。
  - 系統也可以提供偵測及解除死結的方法來處理死結，當系統偵測出死結發生的狀況
    - 可以利用終止行程。
    - 或是回收行程的資源。