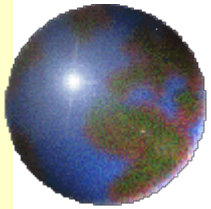
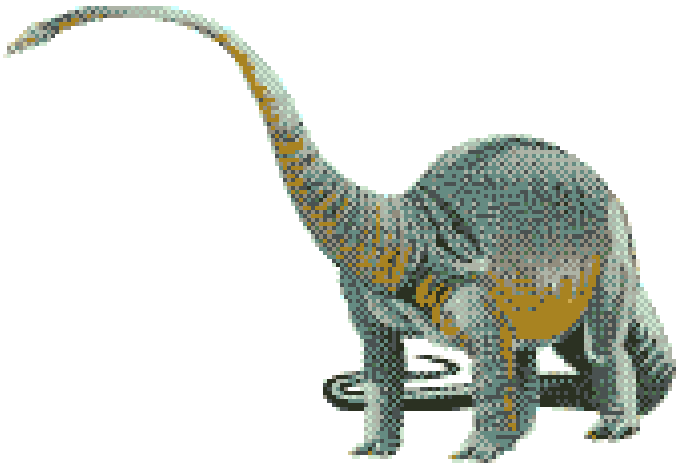


作業系統(Operating Systems)

Course 7: 行程間的溝通 (Process Communication)



授課教師:陳士杰

國立聯合大學　資訊管理學系

本章重點

- Process Communication 的方式
  - Share Memory
  - Message Passing
- Shared Memory溝通方式所產生的問題與解決策略
- Critical Section 定義與設計上所須滿足的三個性質
- Critical Section 設計方法
  - Software solution (Algorithm-Based)
    - 2個Process
    - n個Process
  - H.W. instruction
  - Semaphore
  - Monitor
- 著名的同步問題與解決方法
- Message Passing 溝通方式介紹

■為何Process之間需要溝通?

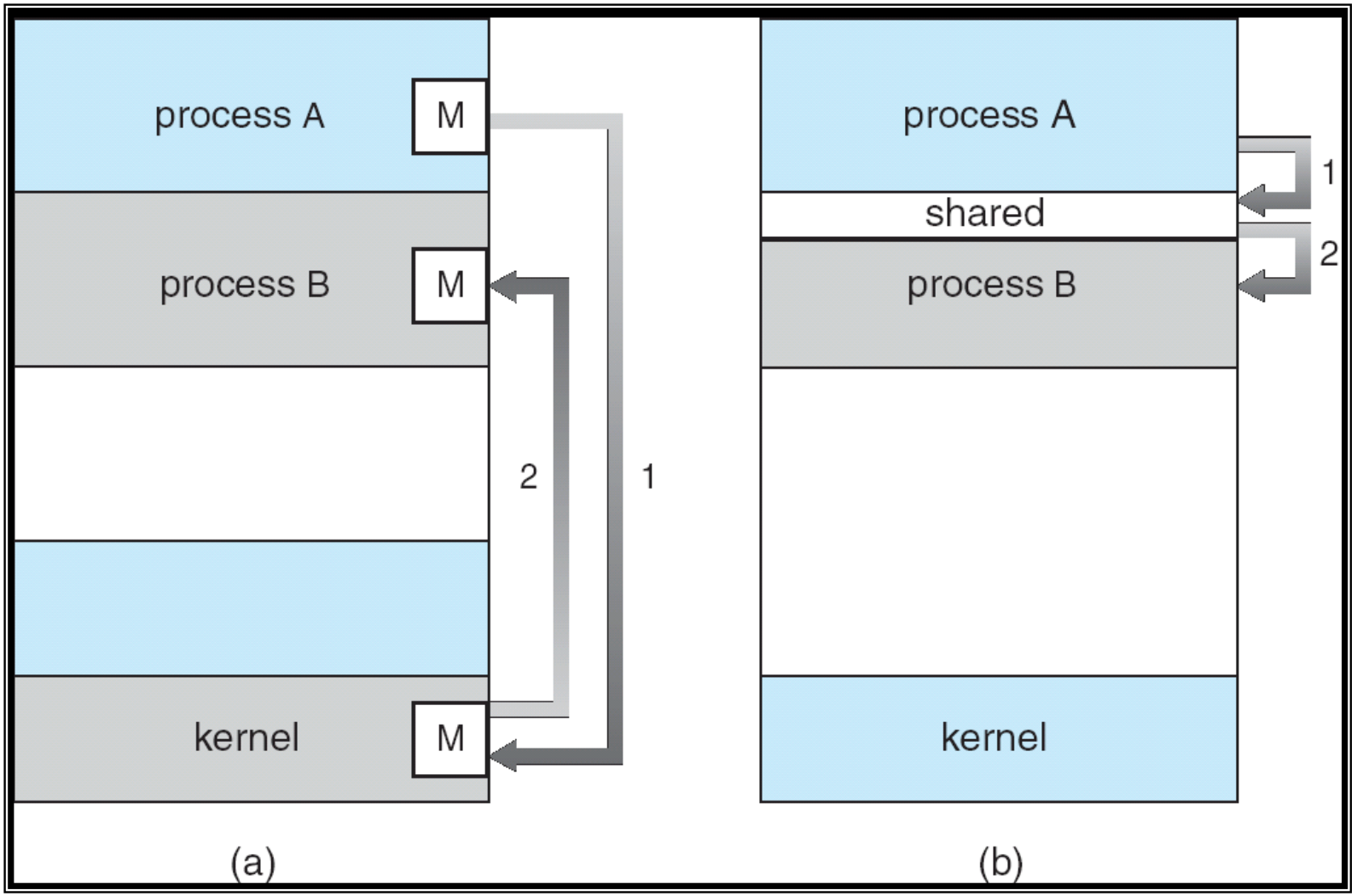
- 在現行的作業系統中，通常不會只有一個Process存在於O.S.內，一般會有好幾個Processes同時存在於系統中並行執行(Concurrent Execution)。
- 存在於系統中並行執行的Process可以分成兩大類：
  - **獨立行程(independent process)** – 如果一個Process無法影響其它Process的執行，同時它也不受其他Process的影響，該Process即為獨立行程。這一類的Process之間不會有任何共享的資料。
  - **合作行程(cooperating process)** – 如果一個Process能夠影響其它Process，或是受到其它Process的影響，它就是屬於合作行程。這些Process會共同完成一份工作，所以這一類的Process之間會有共享的資料，彼此之間需要有資訊交換與協調的管道。
- 獨立的行程因為行程之間並不會相互影響，而且行程之間也是各自行事，完成各自的工作。而合作行程就有較多可討論的東西。

Process Communication的方式

- 在電腦系統中讓兩個合作行程做溝通的方法有兩種:
  - **Share Memory** 的定義:
    - 各個Process利用對**共享的記憶體 (共享變數; Shared Variables)**的存取, 來達到彼此溝通、交換資訊的目的。
    - 提供對共享記憶體之互斥存取控制的責任是由**Programmer**來負責; 而**O.S.**只負責提供共享的記憶體空間, 不提供任何額外的資源。
  - **Message Passing** 的定義:
    - 兩個Process要溝通, 需遵守下列步驟:
      - 建立Communication Link
      - 互傳訊息 (Message)
      - 傳輸完畢, release link

      此法需要OS提供額外支援(e.g., Link Management, Link Capacity 管控, Message Lost處理...etc), 而Programmer不需額外負擔。
    - 有**直接**與**間接**兩種連繫方式。

	Share Memory	Message Passing
Def	Process彼此之間透過對共享變數的存取，藉由其值的變化，達到溝通的目的。	Process之間要溝通，須：①先建立Link，②建立後，才互相傳送訊息，③釋放Link
共享性	共享變數是所有Process皆可存取	Link是專屬於溝通雙方，不會被其它人共用
O.S.的支援	O.S.僅提供Share Memory空間，不提供任何額外支援	O.S.會提供如Link Management等額外支援
Programmer的負擔	Programmer負擔重，O.S.負擔輕	Programmer負擔輕，O.S.負擔重
須提供何種處理機制	須提供對共享變數之互斥存取控制，否則會有問題	須提供如Link Creation，Link Capacity控制，Message Lost等處理機制



(a) Message Passing

(b) Shared Memory

■共享記憶體 (Share Memory)

- 在Share Memory方式下可能之問題：**Race Condition (競爭情況)**
  - **Def:** 一組Processes在Shared Memory溝通方式下，若未對共享變數提供互斥存取等同步機制，則會造成“共享變數之最終結果值，會因Processes之間執行順序不同而有所不同”，導致不可預期之錯誤發生。
- 例：有兩個Processes: P1及P2，共享一個Count變數，且該變數目前的值為5。現在，P1及P2各作一次Count加1與減1之動作，則Count可能值為何?

P1	P2
:	:
Count = Count+1	Count = Count -1
:	:

Ans:

- P1與P2為Concurrent execution，且Count = Count+1與Count = Count-1並不保証在程式執行過程中，不會被任意中斷 (Interrupt)。

  ∴此兩條指令的可能執行順序為：

  T1: 執行Count+1(結果為6)，但尚未Assign給Count

  T2: 執行Count-1(結果為4)，但尚未Assign給Count

  T3: 將T1的計算結果Assign給Count

  T4: 將T2的計算結果Assign給Count

  此時Count最終值為4

- 若時間點T3與T4的動作對調，則Count值為6
- 若P1整個做完，再執行P2，則Count值為5

- 需要對共享變數之存取進行管制，即：同時段只能有一個行程使用該變數。
- 解決**Race Condition**之策略：
  - **Disable Interrupt (停止使用中斷)**
  - **Critical Section Design (臨界區間設計)**

Disable Interrupt (停止使用中斷)

- **Def:** 為了確保對共享變數進行存取之相關指令敘述可以“於執行過程中不會被中斷 **(Atomically Executed)**”，所以在該指令敘述執行之前，先**Disable Interrupt**，完成敘述後再**Enable Interrupt**。

P1	P2
:	:
(Disable Interrupt)	(Disable Interrupt)
Count = Count+1	**Count = Count -1**
(Enable Interrupt)	(Enable Interrupt)
:	:

- 優點:簡單，易於實施
- 缺點:只適用於Single Processor環境，不適用於Multiprocessor環境。(∵Performance變差，且風險性增加)
  - 在Multiprocessor環境下，關掉單顆CPU的中斷功能沒用，要全部CPU的中斷功能都關掉才有用。所以Process須發出Disable Interrupt給所有CPU，且須等到所有CPU均回覆 “Interrupt已Disable”的訊息方可繼續執行。且工作完成後須發出“Enable Interrupt”的訊息通知所有CPU。
  - Disable Interrupt的作法會導致其它更緊急的工作無法執行。

Critical Section Design (臨界區間設計)

- 目的：提供對共享變數之存取動作的互斥控制，確保資料的正確性。
- **Critical Section**
  - **Def:** 在程式中，針對共享變數進行存取之指令敘述所形成之集合

例：

P1		P2	
:		:	
C.S.	Count = Count+1	Count = Count -1	C.S.
:		:	

非**C. S.**之指令敘述集合稱為**Remainder Section (剩餘區間；R. S.)**

Process之程式架構

- 臨界區間的設計，主要是設計其“入口”與“出口”的程式片段。
- 程式架構：

repeat

Entry Section 進入區間：管制Process可否進入C.S.的程式片段

C.S.

Exit Section 離開區間：離開C.S.後之解除入口管制的程式片段

R.S.

until false.

P1	P2
(Entry Section)	(Entry Section)
Count = Count+1	Count = Count -1
(Exit Section)	(Exit Section)

Critical Section Design 必須滿足的三個性質

- **Mutual Exclusion (互斥)**
- **Progress (行進)**
- **Bounded Waiting (有限等待)**

- **Mutual Exclusion**
  - **Def:** 在任何一個時間點，最多只允許一個**Process**進入它自已的**C.S.**內活動，不允許多個**Process**同時進入各自的**C.S.**內活動。
  - 例：

P1	P2
(Entry Section)	(Entry Section)
Count = Count+1	Count = Count -1
(Exit Section)	(Exit Section)

- Progress
- Bounded Waiting

- Mutual Exclusion
- **Progress**
  - **Def: 必須同時滿足下面2個要件:**
    - **不想進入C.S.的Process不可以阻礙其它Process進入C.S. (即:不可參與進入C.S.之決策過程)**
    - **必須在有限的時間內, 自那些想進入C.S.的Process之中, 挑選出一個Process進入C.S. (隱含:No Deadlock)**
- Bounded Waiting

- Mutual Exclusion
- Progress
- **Bounded Waiting**
  - **Def: 自Process提出進入C.S.之申請，到它獲准進入C.S.之等待時間是有限的。即：若有n個Processes想進入C.S.，則任一Process至多等待n-1次，即可進入C.S. (隱含：No Starvation)**

C.S.設計的方法

- Algorithm Based
  - 2個Processes之設計方案
  - N個Processes之設計方案
- H.W.指令支援
  - Test-and-Set指令
  - SWAP指令
- Semaphore (號誌)
- Monitor

Algorithm-Base Solution to C.S. Design

- 自行設計程式
- 2個Processes之設計方案
  - Algorithm 1
  - Algorithm 2
  - Algorithm 3
- N個Processes之設計方案
  - Bakery Algorithm

2個Processes之設計方案 (假設有2個Process P_i & P_j)

- **Algorithm 1**
  - 共享變數宣告如下：
    - **turn:** 整數變數，存放的值為 i 或 j。初值設定為 i 或 j 皆可。
  - P_i 的程式設計如下：

Repeat

【Entry】 **while (turn ≠ i) do *no-op*;**

C.S.

【Exit】 **turn = j;**

R.S.

Until false.

"no-op"不做任何事情。此迴圈為Busy Waiting，即透過一個空迴圈不斷進行測試執行，以達到Process Waiting之效果。

- 分析：
  - **滿足 Mutual Exclusion**
    - turn的值不可能同時為 i 且為 j，所以 P_i 與 P_j 不會同時進入自已的C.S.中。
  - **不滿足 Progress** (不滿足Progress的條件 1)
    - 假設目前turn的值為 i，但是 P_i 不想進入C.S.，此時若 P_j 想進入自已的C.S.，則無法進入!!
    - P_j 被 P_i 阻礙。
  - **滿足 Bounded Waiting**
    - 若 P_i 與 P_j 皆想進入自已的C.S.，若此時turn為 i，則 P_i 先進入C.S.，而 P_j 就先等待。
    - 若 P_i 離開C.S.後又企圖立刻進入自已的C.S.，此時 P_i 會被卡住，因為 P_i 離開C.S.時已將 turn 設定為 j 了。所以對 P_j 而言，它至多等待一次即可進入C.S.。

Algorithm 2

- 共享變數宣告如下：
  - **flag[n]: Boolean陣列，其中 n 的值為 i 或 j 。當布林值為 true，表示該 Process有意願進入自已的C.S.；若為 false，則表示無意願進入。**
  - **flag[i] 與 flag[j] 的初始值皆設為false。**
- **P_i 的程式設計如下：**

Repeat

```
flag[i] = true;
while (flag[j]) do no-op;
```

【Entry】

C.S.

```
flag[i] = false;
```

【Exit】

R.S.

Until false.

- 分析:
  - **滿足 Mutual Exclusion**
  - **不滿足 Progress**
    - 會產生Deadlock。

    說明:

    T1: P_i 設定 flag[i] = True

    T2: P_j 設定 flag[i] = True

    T3: P_i 卡在while loop

    T4: P_j 卡在while loop

    造成兩個Process皆無法進入自已的C.S.，因而產生死結。
  - **滿足 Bounded Waiting**

Algorithm 3

- 共享變數宣告如下：
  - **flag[n]: Boolean陣列，其中 n 的值為 i 或 j。flag[i] 與 flag[j] 的初始值皆設為false。**
  - **turn: 整數變數，存放的值為 i 或 j。初值設定為 i 或 j 皆可。**
- P_i 的程式設計如下：

Repeat

```
flag[i] = true;
turn = j;
while (flag[j] and turn = j) do no-op;
```

【Entry】

C.S.

```
flag[i] = false;
```

【Exit】

R.S.

Until false.

- 分析：
  - **滿足 Mutual Exclusion**
    - 若P_i與 P_j 皆想進入自已的**C.S.**，代表 **flag[i] = flag[j] = true**，而且會分別執行到 **turn = i** 及 **turn = j** 之設定 (只是先後次序不同)。因此，**turn** 值僅會是 **i** 或 **j**，絕不會兩者皆是。∴Mutual Exclusion得以確保。
  - **滿足 Progress**
    - 若 P_i不想進入**C.S.**，則表示**flag[i] = false**。此時若 P_j 想進入自已的**C.S.**，必可通過 **while(flag[i] and turn = i) do no-op** 這個空迴圈而進入其**C.S.**，不會被 P_i阻礙。
    - 若P_i與 P_j 皆想進入自已的**C.S.**，則在有限的時間內，P_i與 P_j 會分別執行到**turn = i** 及 **turn = j** 之設定 (只是先後次序不同)，**turn**值必為 **i** 或是 **j**。∴ P_i (或 P_j) 會在有限的時間內進入自已的**C.S. (No Deadlock)**。
  - **滿足 Bounded Waiting**
    - 假設 P_i與 P_j 皆想進入自已的**C.S.**，且此時P_i已先進入**C.S.**，而 P_j 卡在**while loop** 等待。若 P_i離開**C.S.**後又企圖立刻進入自已的**C.S.**，此時 P_i一定會執行 **"turn = j"** 之設定，使得P_i無法再搶先於P_j 進入自已的**C.S.**而被卡在**while loop** 等待。所以對 P_j而言，它至多等待一次即可進入**C.S.**。

N個Processes之設計方案

Bakery's Algorithm

- **運作概念**：麵包店發號碼牌，有意進入店內**(C.S.)**的人要抽一張；但號碼可能會重覆，此時要憑**Process** 的 **ID**大小來決定，小的先進去。
- 共享變數宣告如下：
  - **Choosing[0, …, n-1] of Boolean:** 初值皆為**False**，且：
    - **True:** 表示有意願進入**C.S.**，且正在挑號碼牌
    - **False:** 可能是 ① 毫無意願 **(**若**False**是初值**)**，或是②有意願且拿完號碼牌 **(**若**False**是初值以外**)**
  - **Number[0, …, n-1] of Integer:** 初值皆為 **0**。
    - 若**Number[i] = 0**，表示 P_i 無意願進入**C.S.**
    - 若**Number[i] > 0**，表示 P_i 有意願進入**C.S.**

- 使用的數學函式：
  - (a, b) < (c, d)
    - If (a<c) or (a = c and b < d)
    - a, c 為號碼牌；b, d 為Process的ID。比較時，以號碼牌為準。
  - Max(a_0, a_1, …, a_{n-1}) = 取出 a_0 ~ a_{n-1}之最大值

- P_i 的程式設計如下：

Repeat

Choosing[i] = True; //正在挑號碼牌

Number[i] = Max(Number[0], …, Number[n-1]) +1;

Choosing[i] = False; //挑完號碼牌

【Entry】

```
for j = 0 to (n-1) do  //Process i與其它的Process比大小
   {
    while (Choosing[j]) do no-op;
    while ((Number[j] ≠ 0) and [(Number[j], j) < (Number[i], i)]) do no-op;
   }
```

C.S.

【Exit】 **Number[j] = 0;** //表明無意願

R.S.

Until false.

從“所有Process所持有之號碼牌中，挑出號碼最大者”，再加上1即為Process i 的號碼

欲比較的對象若還在抽號碼，則等待它。

- **Q1: 為何會有許多 (≥2) Processes會取得相同的號碼牌?**
  - 指令Number[i] = Max(…) + 1可分成三個動作。(①挑最大值，②加1，③將結果Assign給Number[i])
  - 假設目前有P_i、P_j兩個Process，且目前最大的Number值為k，依下列順序執行：

	P_i	P_j
T0	Choosing[i] = True	
T1		Choosing[j] = True
T2	執行Max(…) + 1，得出k+1，但尚未Assign給Number[i]	
T3		執行Max(…) + 1，得出k+1，且Assign給Number[j]，故Number[j] = k+1
T4	執行 k+1 Assign給Number[i]，故Number[i] = k+1	

  - ∴ P_i、P_j具有相同的Number值

- **Q2: 移除for 迴圈中的第一個while迴圈敘述，則結果是否正確?請說明。**
  - 不正確，會違反互斥 **(Mutual Exclusion)**
  - 假設目前**Choosing[0] ~ Choosing[n-1]** 值皆為 **False**，且所有號碼牌的值**Number[0] ~ Number[n-1]**皆為**0**。現有兩個**Process** P_i 及 P_j 欲進入**C.S.**，且**Process**的**ID**大小分別為：**i < j**。
  - 若兩個**Process**的執行順序如下：

	P_i	P_j
T0	Choosing[i] = True	
T1		Choosing[j] = True
T2	執行Max(...) + 1，得出結果為 1，但尚未Assign給Number[i]	
T3		執行Number[j] = Max(...) + 1，∴Number[j] = 1
T4		設定Choosing[j] = False
T5		順利執行完for loop不會被 P_i 或任何Process卡住，(∵Number[i] = 0，P_j認為其它所有的Process皆無意願)。∴ P_j進入C.S.。
T6	執行 Number[i] = 1的 Assign動作	
T7	設定Choosing[j] = False	
T8	順利執行完for loop不會被 P_j 或任何Process卡住，(∵Number[i] < Number[j]，但Process ID: i < j)。∴ P_i進入C.S.。	

Q3: 証明其正確性。

- 滿足Mutual Exclusion。
  - 考慮底下兩個CASE:
    1) 若所有Process的號碼牌皆不相同，則具有最小號碼值的Process必唯一且只有它可以進入C.S.。
    2) 若取得最小號碼值的Process不只一個，但因為Process ID為Unique，所以會由具有最小ID的Process進入C.S.。
  - 上述的兩個CASE中，其它未進入的Process都是被while ((Number[j] ≠ 0) and [(Number[j], j) < (Number[i], i)]) 卡住，∴可確保最多只有一個Process得以進入C.S.，互斥得以確保。

- 滿足Progress。

  1) 若Pj不想進入C.S.，其Numbewr[i] = 0且Choosing[i] = False。若此時Pi想進入C.S.時，則不會被Pj卡在下述兩個迴圈中：

     - while (Choosing[j]) do *no-op*;
     - while ((Number[j] ≠ 0) and [(Number[j], j) < (Number[i], i)]) do *no-op*;

     ∴ Pj不會妨礙Pi進入C.S.。

  2) 若有多個Process同時想進入各自的C.S.，則必定可以在有限的時間內，找出一個①具有最小號碼牌值的Process；或是②同為最小號碼牌值但ID最小的Process，進入臨界區間 。

- 滿足Bounded Waiting。

  1) 若P0~Pn-1皆想進入自已的C.S.，假設它們所抽到的號碼牌(即：Number[0]~Number[n-1])分別為1~n，且皆不相同。

     – 此時，P0得以先進入C.S. (∵號碼牌最小)。若P0離開後又企圖馬上進入C.S.，則因為他所抽的新號碼一定會大於n，所以P0不會馬上再進入C.S.。此時，應該是輪到P1進入C.S.，接著是P2, P3, ..., Pn-1依序進入自已的C.S.。

  2) 若P0~Pn-1皆想進入自已的C.S.，假設它們所抽到的號碼牌(即：Number[0]~Number[n-1]) 皆相同為k。

     – 此時，P0得以先進入C.S. (∵Process ID最小) 。若P0離開後又企圖馬上進入C.S.，則因為他所抽的新號碼一定會大於k，所以P0不會馬上再進入C.S.。此時，應該是輪到P1進入C.S.，接著是P2, P3, ..., Pn-1依序進入自已的C.S.。

  ∴不論是上述兩種情況的哪一種，Pn-1至多等(n-1)次後，必可進入C.S.。滿足Bounded Waiting。

H. W. 指令支援 to C.S. Design

- 系統直接提供具有Atomic特性的指令，讓程式碼可以在單一時間點被完成，且不會中途被插斷。
  - Test and Set
  - Swap

Test and Set指令

- 此指令為Atomically Executed
  - 即:在單位時間內可以順利做完，不受任何中斷干擾
- 功能定義如下:

```
int Test_and_Set (int *Target)
{
   int temp;
   temp = *Target;
   *Target = 1;
    return temp;
}
```

程式解釋:

- 將傳入的Target之舊值再回傳出去，當成該函數的傳回值，隨後並將Target設成True。
- 此段程式一定是Atomic Execution

- 如何用在C.S. Design上？
  - 共享變數 Lock : Boolean，初值設為False
  - [Algorithm] Pi的程式片段如下：

Repeat

【Entry】 **while (Test_and_Set(Lock)) do *no-op*;**

C.S.

【Exit】 **Lock = False;**

R.S.

Until false.

- 例：假設Pi和Pj皆想進入自已的C.S.:

	P_i	P_j
T0	Pi先搶到Test_and_Set(Lock)指令執行，此時Test_and_Set的傳回值為False且設定Lock為True，∴Pi進入C.S.。	
T1		Pj執行while(Test_and_Set(Lock)) do no-op，此時Test_and_Set的傳回值為True，且設定Lock為True，∴Pj被卡在此迴圈。

- 上例說明了該演算法合乎Mutual Exclusion (互斥)。
- 該演算法也合乎Progress:
  - 不用C.S.的Process不會去搶Test_and_Set指令
  - 有限的時間內，一定會有人搶到Test_and_Set，所以一定會有人進入C.S.
- 該演算法不滿足Bounded Waiting:
  - Pi已進入C.S.，而Pj卡在while loop中等待進入C.S.，此時若Pi離開C.S.後又馬上企圖進入C.S.，則Pi很有可能再度先於Pj執行Test_and_Set指令，而又再度進入C.S.。∴Pj可能Starvation。

Swap指令

- 此指令為Atomically Executed
- 功能定義如下 :

```
Procedure SWAP(var a, b: Boolean)
   var temp: Boolean;
   begin
        temp = a;
        a = b;
        b = temp;
   end
```

- 如何用在C.S. Design上？
  - 共享變數 Lock : Boolean，初值設為False
  - [Algorithm] Pi的程式片段如下：

```
Repeat
【Entry】  key = true;
           repeat
               SWAP(Lock, key);
           until key = False;
                    C.S.
【Exit】   Lock = False;
                    R.S.
Until false.
```

- 例：假設Pi和Pj皆想進入自已的C.S.:

	P_i	P_j
T0	Pi先搶到SWAP(Lock, key)指令執行，此時Lock與key的值互換，故Lock = True，key = False，∴Pi進入C.S.。	
T1		Pj執行SWAP(Lock, key)，此時因為Lock = True，故執行Lock與key的值互換時，Lock = True且key = True∴Pj被卡在內層的repeat ... until迴圈中 (∵ key = False不成立)。

- 上例說明了該演算法合乎Mutual Exclusion (互斥)。

- 該演算法也合乎Progress:
  - 不用C.S.的Process不會去搶SWAP指令
  - 有限的時間內，一定會有人搶到SWAP，所以一定會有人進入C.S.
- 該演算法不滿足Bounded Waiting:
  - Pi已進入C.S.，而Pj卡在while loop中等待進入C.S.，此時若Pi離開C.S.後又馬上企圖進入C.S.，則Pi很有可能再度先於Pj執行SWAP指令，而又再度進入C.S.。∴Pj可能Starvation。

Semaphore (號誌)

- **Def: 用來解決C.S. Design及同步問題 (Synchronization Problem)的一種資料型態 (Data Type)**
  - 假設變數S為Semaphore，其為Integer，通常初值為1.
  - 在S上提供兩個Atomic Operations，分別是Wait(S) (或P(S)) 及Signal(S) (或V(S))，其定義如下：
    - Wait(S): while (S≤0) do *no-op*;
      S = S-1;
    - Signal(S): S = S+1;

利用Semaphore設計C.S.

- 共享變數的宣告：
  - mutex: Semaphore = 1;
    (變數)　(型態)　(初值)
- Pi的程式片段如下：

```
Repeat
    wait(mutex);        【Entry】
        C.S.
    Signal(mutex);      【Exit】
        R.S.
Until false;
```

簡單的同步問題

- 假設有P1和P2兩個Current Execution的Processes如下：

P1	P2
:	:
S1	S2
:	:

今規定 “S1的敘述一定要在S2敘述之前執行”，請利用Semaphore滿足此一問題設計。

Ans:

- 宣告一個共享變數 S: Semaphore，初值為 0。

P1	P2
:	:
	Wait(S)
S1	S2
Signal(S)	
:	:

號誌初值設定之重要觀念：

- 作為互斥控制之用：1
- 作為強迫暫停之用：0

(∵while S≤0 do no-op)

著名的同步問題

- 生產者/消費者問題 (Producer/Consumer Problem)
- 哲學家晚餐問題 (Dining Philosophers Problem)

生產者/消費者問題 (Producer/Consumer Problem)

- 生產者-消費者問題主要是假設有兩個**Process**，分別是
  - **生產者行程**：用來產生資訊。
  - **消費者行程**：用來消耗生產者所產生的資訊。
- 為了能讓生產者行程和消費者行程能夠順利的執行，必須有一個**緩衝區 (Buffer)**，讓**生產者存放所產生的資訊**，而且可以**讓消費者去消耗裡面的資訊**。這兩者必需**同步**，才不會造成下列狀況：
  - 消費者在該緩衝區是**空**的時候還可以進去消費。
  - 當這個緩衝區已**滿**，生產者還一直生產資訊往這個緩衝區裡面放。

- 這個緩衝區可以分成兩種：
  - **無限緩衝區(Unbounded Buffer)**：它的**空間容量是沒有限制**的。
    - 消費者在當這個緩衝區沒有任何資訊時，需要暫停等待。
    - 生產者因為其空間容量沒有限制，可以無限制地產生資訊。所以對生產者是沒有影響的。
  - **有限緩衝區(Bounded Buffer)**：它的**空間容量是有限制的**。
    - 消費者在當這個緩衝區沒有任何資訊時，亦需要暫停等待。
    - 生產者因為其空間容量有其上限，在當緩衝區滿的時候，生產者需休息，以等待消費者來消耗這些資訊。

- 現假設生產者-消費者問題共享的Buffer有n個儲存格空間。
- 宣告三個共享的號誌變數:
  - **mutex: Semaphore**
    - 對Buffer存取提供互斥控制
    - 初值為 1
  - **empty: Semaphore**
    - 記錄Buffer內現有的空格數
    - 初值為 n
  - **full: Semaphore**
    - 記錄Buffer內有資料存在的格數
    - 初值為 0

Producer之程式片段

```
repeat
   produce an item in nextp;
      wait(empty);
      wait(mutex);
          Add nextp to Buffer;
      signal(mutex);
      signal(full);
until False;
```

- 每執行wait(empty)一次，empty就會減1，即Buffer中的空格數減 1
- 若empty為0，則表示Buffer已滿!!Producer就要wait!!!

- 要對Buffer做存取時，要先做互斥控制

- full 加 1，表示Buffer中有填資料的格子數加 1!!

Consumer之程式片段

```
repeat
      wait(full);
      wait(mutex);
          Remove item from Buffer;
      signal(mutex);
      signal(empty);
until False;
```

- 每執行wait(full)一次，full就會減1，即Buffer中有資料的格子數減 1
- 若full為0，則表示Buffer已空!!Consumer就要wait!!!

- 要對Buffer做存取時，要先做互斥控制

- empty 加 1，表示Buffer中的空格數加 1!!

哲學家晚餐問題 (Dining Philosophers Problem)

- 有5位哲學家和5枝筷子
  - 哲學家思考時，就不會用餐
  - 若哲學家想吃飯時，要依序取用位於左右兩邊的筷子，方可用餐

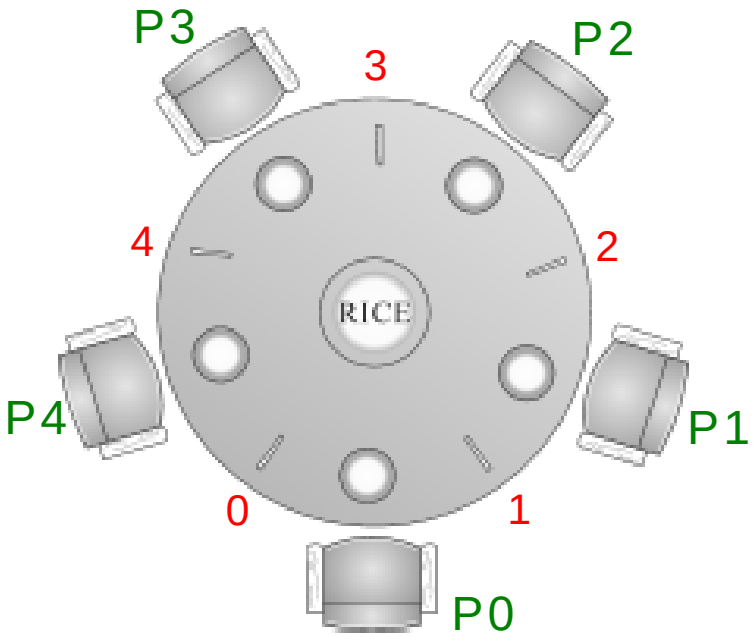


圖 6.16　哲學家吃飯

利用Semaphore設計

- 宣告變數:
  - **chopstick[0...4] of Semaphore**, 初值皆設為 **1**
- 某位哲學家**Pi**的程式片段如下:

```
repeat
    wait(chopstick[i]);
    wait(chopstick[i+1] mod 5);
        eating;
    signal(chopstick[i]);
    signal(chopstick[i+1] mod 5);
until False;
```

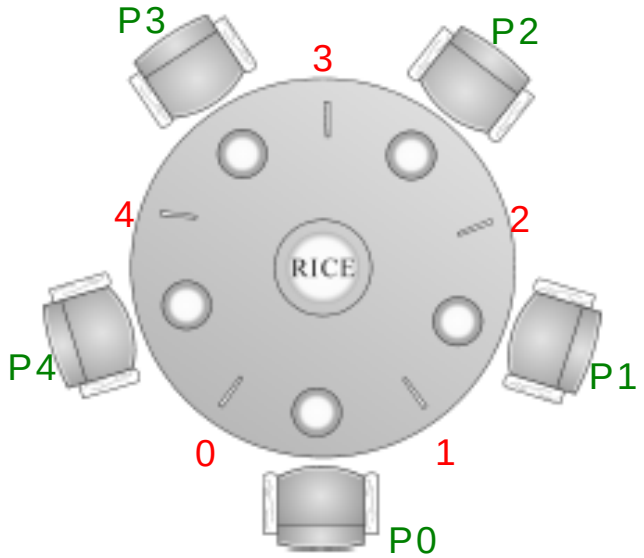


圖 6.16 哲學家吃飯

- 分析:此程式並不正確, 會有**Dead Lock**問題。
  - 即:當每位哲學家依序拿起位於自已左邊的筷子後, 則每位哲學家都拿不到右邊的筷子, 而呈現全部**wait**, 無人可**eating**的情況。

- 解法:
  - [法一]:最多允許 4 位哲學家同時用餐
    - 即:限制同時用餐的人數，以避免死結
    - **Dead Lock Avoidance**定理

      ① $1 \le Max_i \le m$

      ② $\sum_{i=1}^{n} Max_i < m + n$

Ans:

已知 $m = 5$，$Max_i = 2$

條件①滿足 ($\because 1 \le Max_i = 2 \le 5$)

欲滿足條件② (即:$\sum_{i=1}^{n} Max_i < m + n$)，則可得 $2n < 5+n \rightarrow n < 5$

$\therefore$n的最大值為 4。

- 此法雖無死結，但有**Starvation**。

- [法二]:規定除非哲學家可以順利取得左右兩邊的筷子，否則，哲學家不准拿任何筷子。
  - 即:打破**Hold and Wait**
- [法三]:採用非對稱作法。即:奇數號的哲學家先拿左、再拿右;偶數號的哲學家先拿右、再拿左。
  - 即:打破**Circular Wait**
- [法四]:採用**Monitor**來解

Semaphore的缺點

- Programmer可能會誤用Semaphore的wait及signal運作，導致錯誤的狀況發生：
  - 違反Mutual Exclusion
  - 造成Dead Lock
- 範例 1. S: Semaphore = 1，有一程式如下：

```
signal(S);
   C.S.;
wait(S);
   R.S.;
```

- 會讓一堆Process進入各自的C.S.。∴違反Mutual Exclusion。

- 範例 2. S: Semaphore = 1，有一程式如下：

```
wait(S);
   C.S.;
wait(S);
   R.S.;
```

- 會讓其它的Process被卡在第一個wait(S)指令。∴Dead Lock發生。

- 範例 **3. S, Q: Semaphore = 1**，有一程式如下：

P1：

T0: wait(S);
T2: wait(Q);**(卡住)**
Do something;
signal(S);
signal(Q);

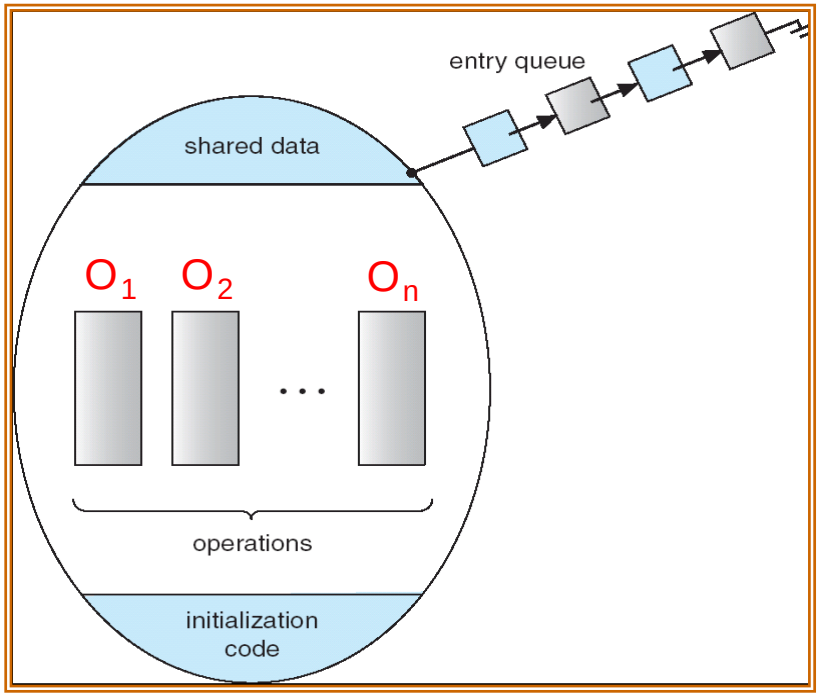
P2：

T1: wait(Q);
T3: wait(S);**(卡住)**
Do something;
signal(Q);
signal(S);

⇒形成 **Dead Lock**

Monitor

- **Def:**
  - 為一個解決同步問題的高階資料結構，由以下三部份組成：
    - 一組**Procedures (Operations)**：供外界呼叫使用
    - 共享資料區：此區域會宣告一些共享變數，且只提供給**Monitor**內各**Procedure**所共用，外界**Process**不得直接使用。
    - 初始區**:** 設定某些共享資料變數的初值



- Monitor本身已經確保互斥的性質，即：
  - 每次只允許一個Process在Monitor內活動。
    - 當某Process在執行Monitor內的某個Procedure時，其它Process不可呼叫/執行Monitor內的任何一個Procedure，須等到該Process執行完此Procedure、離開Monitor，或因同步條件成立/不成立而被Block為止。
    - 即不允許2個以上的Process在Monitor區域內活動。
    - 保証在共享資料區裡面的共享變數不會有Race Condition。
- 優點：Programmer不須花額外負擔來處理互斥問題，∴有更多心力可以用在同步問題的解決上。
- 利用Monitor解同步問題之流程:
  - 根據問題，定義Monitor
  - 使用Monitor
    - 利用所定義之Monitor，宣告一個變數
    - 撰寫Process i 之程式片段

定義Monitor

Type *monitor_name* = Monitor

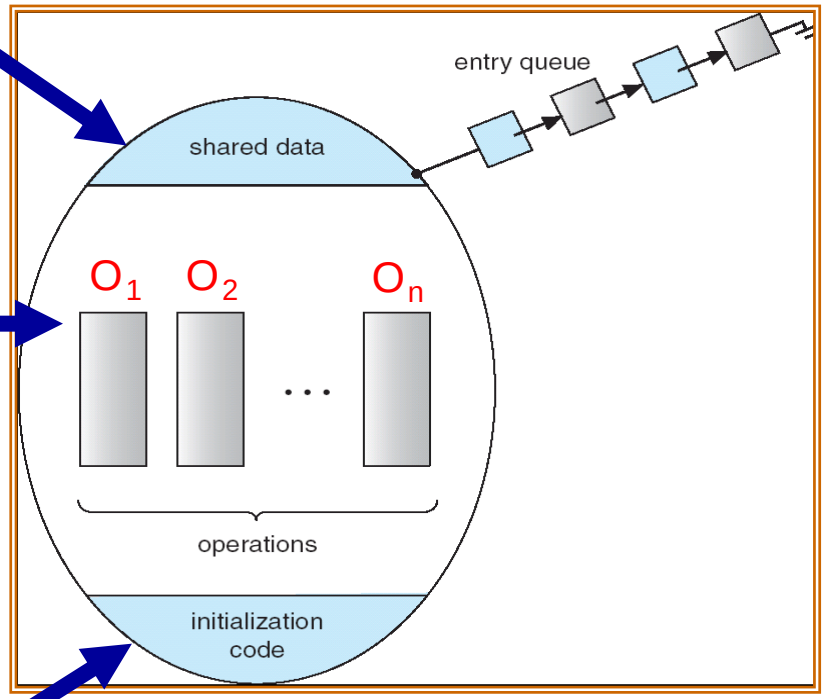
// 共享的資料變數宣告 **資料區**

procedure entry O_1
    begin
        …
    end
…
procedure O_n
    begin
        …
    end

程序區

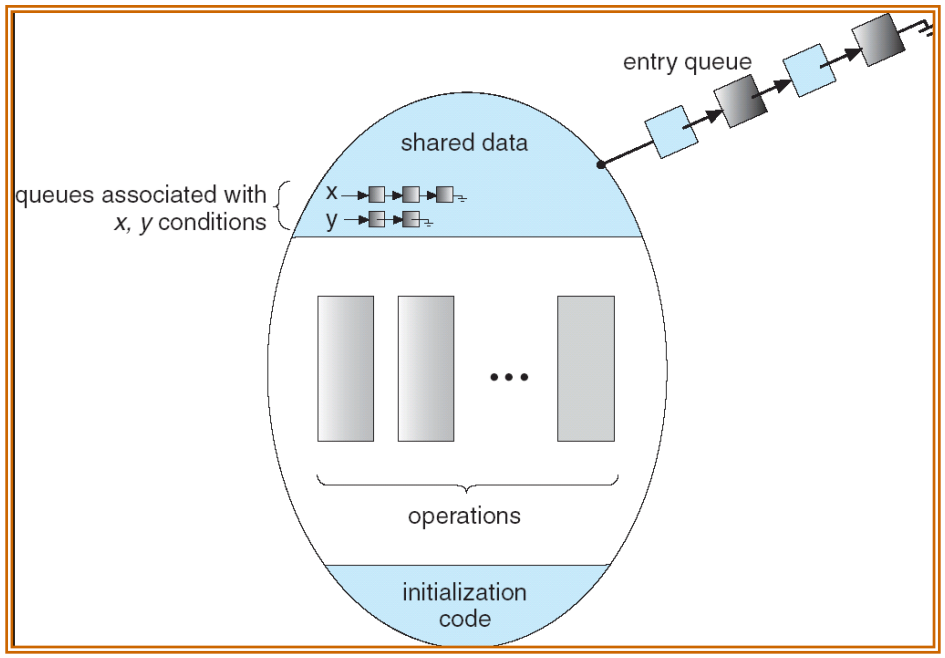
begin
        initialization
end

初始區



- 為解決同步問題，**Monitor**會提供一種特殊型態的變數供**Programmer**使用：**Condition型態變數**。
- 假設宣告x為**Condition**型態變數，

  x: Condition



則在x變數上，會提供2個**Atomic Operations:**

- **x.wait**：強迫**Process**暫停，並把該**Process**放入x 的**Waiting Queue**中。
- **x.signal**：如果有**Process**卡在**x.wait**的**Waiting Queue**中，則此運作會從該**Waiting Queue**中取出第一個**Process**，將其**wakeup**。否則，**x.signal**無任何作用。

利用Monitor解決哲學家晚餐問題

定義Monitor

```
type dinner-ph=monitor

var state: array[0…4] of (thinking, hungry, eating);
var self: array[0…4] of condition;

procedure entry pickup(i: 0…4)
    begin
        state[i] = hungry;
        test(i);
        if state[i] ≠ eating then self[i].wait;
    end;
procedure entry putdown(i: 0…4)
    begin
        state[i] = thinking;
        test(i+4 mod 5);
        test(i+1 mod 5);
    end;
```

```
procedure test(k: 0…4)
    begin
        if state[k+4 mod 5] ≠ eating and
          state[k] = hungry and
          state[k+1 mod 5] ≠ eating
         then begin
                  state[k] = eating;
                  self[k].signal;
              end;
    end;

begin
     for i = 0 to 4 do
     state[i] = thinking;
end;
```

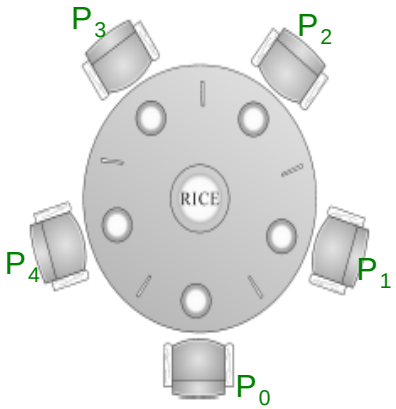


圖 6.16 哲學家吃飯

資料區

- 每個哲學家會有三種不同的狀態(State)：thinking, hungry, eating
- 當哲學家發生搶不到筷子的情況(Condition)時之同步處理

初始區

- 每個Process的初始狀態皆設定為thinking
- 不需針對Condition型態變數設定初值，因為它是用來卡住哲學家

程序區

- 每個Procedure的輸入參數皆為哲學家編號
- 哲學家 i 在拿起筷子(pickup)時：
  - 先將自已的狀態設成飢餓(hungry)
  - 再測試(test)左右兩邊的人是否正在eating；
    - 若是，代表搶不到兩枝筷子，則會將自已卡住(wait)而去睡覺。
    - 若否，
      - 則將自已設成eating狀態。
      - 若哲學家先前曾因搶不到筷子而被卡住(wait)，則將會被喚醒(signal)；否則signal指令無作用。【因為被卡住的哲學家不會喚醒自已！而是有別的哲學家因吃完飯、放下筷子(putdown)時嘗試將他喚醒以進行吃飯】
- 哲學家 i 在放下筷子(putdown)時：
  - 先將自已的狀態設成思考(thinking)
  - 再測試(test)左右兩邊的人是否因為我在eating而被卡住；若有，則解救他們，讓他們可以吃飯。

- 使用**Monitor**
  - 宣告一個變數 dp

    dp: dinner-ph;
  - Process i 的程式片段如下 :

```
repeat
    dp.pickup(i);
        ⋮
      eating
        ⋮
    dp.putdown(i);
until False;
```

■訊息傳遞 (Message Passing)

- **Def:** 兩個Process要溝通，需遵守下列步驟：
  - 建立Communication Link
  - 互傳訊息 (Message)
  - 傳輸完畢，release link

  此法需要OS提供額外支援(e.g., Link Management, Link Capacity 管控, Message Lost處理...etc)，而Programmer不需額外負擔。
- 假設有兩個Process要互相通訊，則它們必須能互相傳送與接收訊息，因為在它們之間需存在一個**通訊鏈 (Communication Link)**。
- 提供兩種操作：
  - **send ( )**
  - **receive ( )**
- 可分成
  - **Direct Communication (直接連繫)**
  - **Indirect Communication (間接連繫)**

直接連繫 (Direct Communication)

- 有以下特性:
  - 要相互連繫的Processes之間的Communication Link是自動產生的。要能進行通訊，兩Processes只需知道對方的身份(Identity; ID)即可。
  - 一條Link只連到兩個Processes 。
    - 只能讓連結雙方的兩個Processes共用，其它Process不能摻一腳。
  - 進行通訊的兩個Processes間，只有一條Link。
- Direct Communication又可分成兩種體系:
  - Symmetric (對稱)
  - Asymmetric (不對稱)

- 在**Symmetric**的**Direct Communication**中：
  - 每一個**Process**需先確定接收者或傳送者的名稱，即**接收者或傳送者在連繫時必須互相指名 (位址對稱)**。
  - **send**與**receive**的基本運算定義如下：
    - **send (P, message)** – 傳送一個**Message**給**Process P**。
    - **receive (Q, message)** – 自**Process Q**接收一個**Message**。

$$Q \xrightarrow{(Link)} P$$

    - 這兩個指令間相互指名的參數若無法搭配，則**Link**無法成功建立。
- 在**Asymmetric**的**Direct Communication**中：
  - 只有發送者**Process**需先確定接收者的名稱，而**接收者不需指出發送者的名稱 (位址不對稱)**。
  - **send**與**receive**的基本運算定義如下：
    - **send (P, message)** – 傳送一個**Message**給**Process P**。
    - **receive (id, message)** – 自任何**Process**接收一個**Message (收方不指定送方)**；當收到**Message**後，會將**id**變數設定為此**Sender** 的**id (只要有人送訊息來即可!!收方不在意是誰送的)**。
  - 只要有人送訊息過來即可，接收方不在意是誰送的。

- 不論是Symmetric或Asymmetric，當改變一個Process的名稱之後，可能就需要對其它所有的Processes進行檢查，若發現與此Process有關的舊名稱，就必須全都改成新的名稱，此即**Process的模組性**受到了限制。

❖範例❖

- 如何利用Direct Communication的Message Passing (Symmetric)解決Producer-Consumer Problem?

Producer	Consumer
Produce an item in nextp	Receive (Producer, nextc)
Send (Consumer, nextp)	Consume the item in nextc

是否面臨先前所說的同步問題？

間接連繫 (Indirect Communication)

- 溝通雙方需透過**Mailbox (信箱)**來傳送與接收訊息。
- send與receive的基本運算定義如下 (A: Mailbox):
  - **send (A, message)** – 傳送一個Message至Mailbox A。
  - **receive (A, message)** – 自Mailbox A接收一個Message。
- 有以下特性:
  - 只有**具有共用的Mailbox之一對Process**才能建立Link。
  - 一條Link可以**連到兩個以上的Process**。
    - 指**不同對的Processes之Links**可**共用同一個Mailbox**。
  - 進行通訊的兩個Processes間，可以**有多條不同的Link**，且每一個Link均對應一個Mailbox。

Direct Communication	Indirect Communication
• 雙方需相互指名，才可溝通	• 雙方需有共享的Mailbox才可溝通
• 溝通的Communication Link專屬於雙方，不得被其它Process共享	• Communication Link (即: Mailbox)可同時被多對Process共享
• 同一對Process只能有一條Link存在，不得有多條Links同時存在	• 溝通之雙方可同時存在多條Link (即: Mailbox)

Message Passing的例外狀況處理

- 由於Message Passing這種Process通訊的管理，是由O.S.負責提供相關控制，因此要考慮一些例外狀況 (Exception Condition)的處理。
  - Process Terminates (行程結束)
  - Message Lost (訊息遺失)

行程結束 (Process Terminates)

- 不論是接收者Q或是傳送者P，在**訊息處理完成之前，都可能會結束**。而使得接收者Q收不到訊息，或是傳送者P等不到接收者Q的確認訊號。
- Process Terminates所造成的狀況及其處理方法：
  - **狀況 1.**：接收者Q可能正等待一個來自**已被中止的Process P**的訊息。如果不對Q採取任何行動，則會造成Process Q **無限期的Blocking (停滯)**。**(P → Q，然而P已死，而Q不知)**
  - **處理**：**1.** O.S.可以終止Process Q，或者 **2.** O.S.通知Process Q說Process P已經結束。
  - **狀況 2.**：傳送者P可能正要送一個訊息給**被中止的Process Q**。若
    - 在**自動緩衝 (Automatic-Buffering)技巧**下，對Process P並無大礙。
    - 在**沒有緩衝**下，Process P需等待Process Q的回應訊息才可繼續工作，此時會造成Process P **無限期的Blocking (停滯)**。**(P → Q，然而Q已死，而P不知)**
  - **處理**：**1.** O.S.可以終止Process P，或者 **2.** O.S.通知Process P說Process Q已經結束。

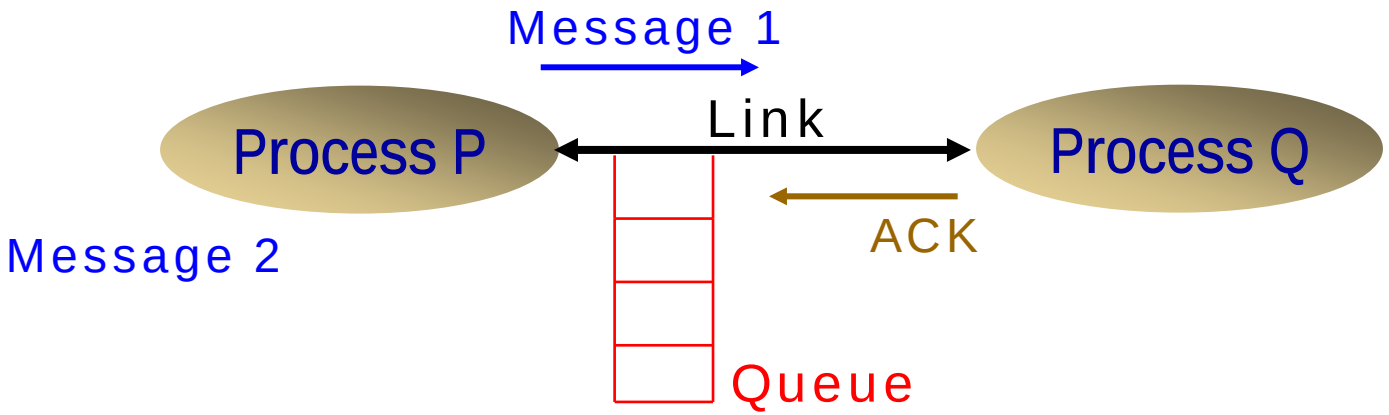
訊息遺失 (Message Lost)

- 由於硬體或通訊線路的故障，可能使得從傳送者P送往接收者Q的訊息遺失。
- 有兩個工作要做：
  - **偵測是否遺失 (Lost)？**
  - **遺失的話，是否要重送 (Resend)？**
- 處理這種事件有三種基本的方法 (即：上述兩個工作**是誰做**？)：
  - 由**O.S.**負責偵測訊息是否遺失。若有需要，由**O.S.**負責重新傳送此訊息。**(O.S. 負擔過高)**
  - 由**傳送的Process**負責偵測訊息是否遺失。若有需要，由**傳送的Process**負責重新傳送此訊息。**(Process負擔過高)**
  - 由**O.S.**負責偵測訊息是否遺失。若有需要，由O.S.告知傳送的Process，並由**傳送的Process**重新傳送此訊息。**(較常用)**

- 如何偵測訊息遺失？
  - 使用**Time-Out (限時)**法。
  - 當一個Message發送出去後，會有一個確認 (Acknowledgement; ACK) 訊息送回來。O.S.或Process可以規定一個**時間間隔T**，當在此段時間T未能收到ACK訊息 (如：> **2T**的時間內沒收到)，則O.S.或Process可以假定此訊息已經遺失了，此時P會再重送。
  - 不過，有時訊息並未遺失，只是**網路傳輸時間比所規定的時間間隔要長**。此時，同樣的訊息拷貝 (如：P的重送訊息) 會有好幾份在網路上流動。
  - 這時Receiver Q必須能區分這些訊息的不同備份。

補　充

Link Capacity

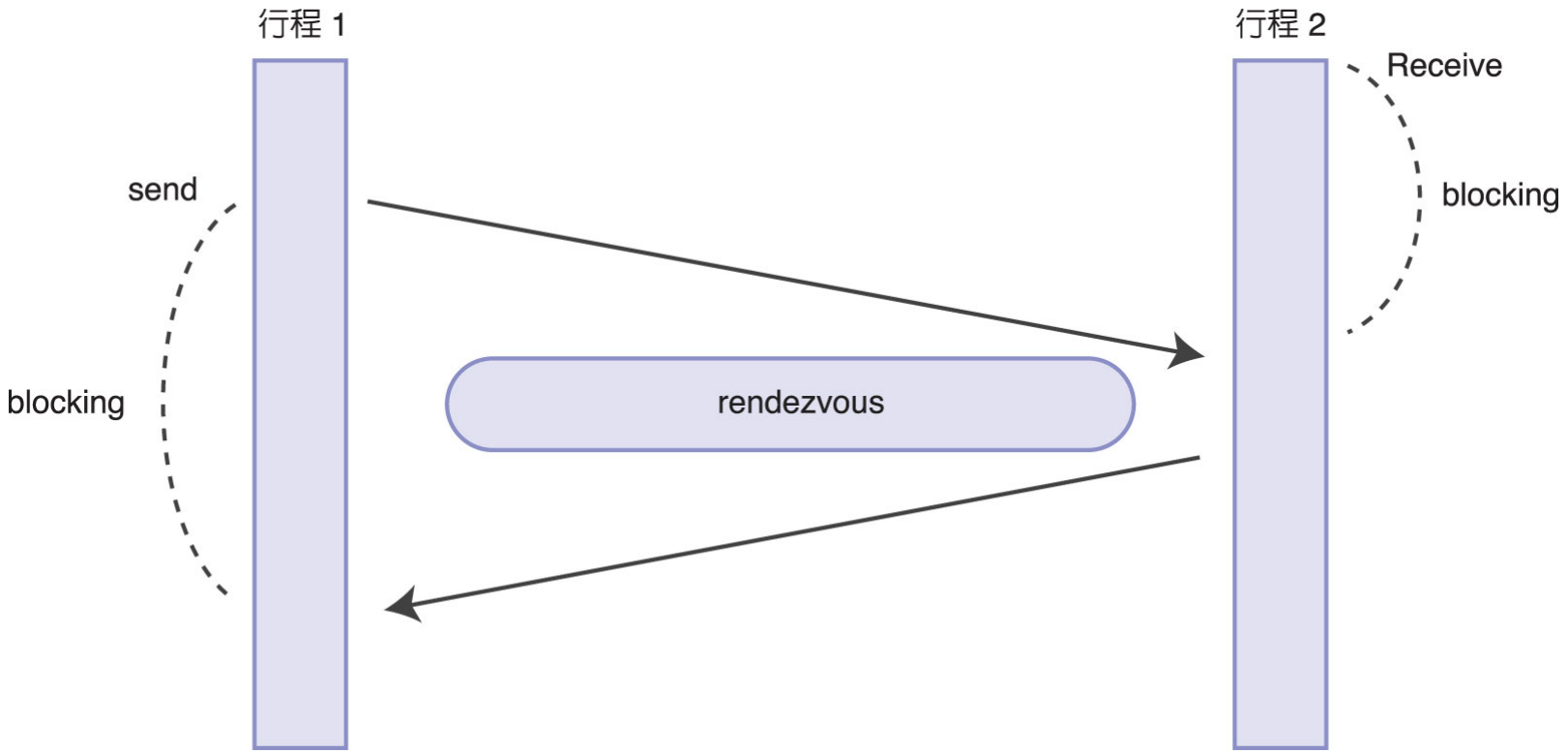


- **Queue的容量 (Capacity)**:
  - **Zero Capacity**
  - **Bounded Capacity**
  - **Unbounded Capacity**

同步化 (Synchronization)

- 由於Message Passing的Link Capacity，得知Message Passing可以是**等待 (Blocking)**或**非等待 (Nonblocking)**，也可稱為**同步 (Synchronous)**或**非同步 (Asynchronous)**。
  - **等待傳送 (Blocking Send)**：Sender會等待直到Receiver或Mailbox收到訊息為止。
  - **非等待傳送 (Nonblocking Send)**：Sender送出訊息後即繼續運作，不用等待。
  - **等待接收 (Blocking Receive)**：Receiver會等待，直到收到訊息才往下執行。
  - **非等待接收 (Nonblocking Receive)**：Receiver收到有效或無效訊息都可以往下執行。
- 當接收與傳送兩者都為**等待**時，則傳送者與接收者之間就有**約會 (Rendezvous)**。

- 在**Zero Capacity**中，為了要保持同步，必須要有互相等待的架構，這種架構又叫做**約定(Rendezvous)**。



Semaphore的類型

- Binary Semaphore (二元號誌)
- Counting Semaphore (計數號誌)

Binary Semaphore (二元號誌)

- **Def:** 在正常使用下，**Semaphore**的值只有**0**與**1**兩種，不會有負值存在。
- 假設**S**為**Semaphore**，初值為**1**，**Wait** 與 **Signal**的定義如下：
  - **Wait(S)**：

    while S ≤ 0 do *no-op*;

    S=S-1;
  - **Signal(S)**：

    S = S+1;

e.g.

S = 1

Wait(S);

C.S.

Signal(S);

R.S.

- 缺點：不知道有多少個**Process**卡在**Wait(S)**中。

Counting Semaphore (計數號誌)

- **Def: Semaphore的值不只可以為0與1，也可以有負值存在。若Semaphore的值為-n，則代表有n個Processes卡在Wait中。**
- **Wait 與 Signal的實作方式有二：**
  - **利用Binary Semaphore**
  - ***利用Suspended/Wakeup System Call + Queue***
- **利用Binary Semaphore實作Counting Semaphore**
  - **宣告共享變數**
    - **C**：存放整數型態的值，初始值為**1**，做為計數號誌的號誌值。
    - **S1**：存放二元號誌的值，初始值為**1**，對變數**C**進行互斥存取。
    - **S2**：存放二元號誌的值，初始值為**0**，當**C < 0** 時，用來強迫**Process**暫停。

- **Wait 與 Signal的定義如下 :**
  - **Wait(C) :**

```
Wait(S1);
C = C-1;
if C < 0 then { Signal(S1);
                Wait (S2); }
          else {Signal(S1);}
```

  - **Signal(C) :**

```
Wait(S1);
C = C+1;
if C ≤ 0 then {Signal(S2)};
Signal(S1);
```

e.g.

C = 1

Wait(C);

C.S.

Signal(C);

R.S.