行程間溝通

為何 Process 之間需要溝通

- 一般會有好幾個 Process 同時存在系統中並行執行 (Concurrent Execution)
- 系統中並行執行的 Process 可以分成兩類
  - **Independent Process 獨立行程** 無法影響其他 Process 也不受其他 Process 影響，**不會共享任何資料**
  - **Cooperating Process 合作行程** 能夠影響其他 Process 或是受到其他 Process 影響，**會共享資料**，需要資訊交換和協調管道

Process Communication 方式

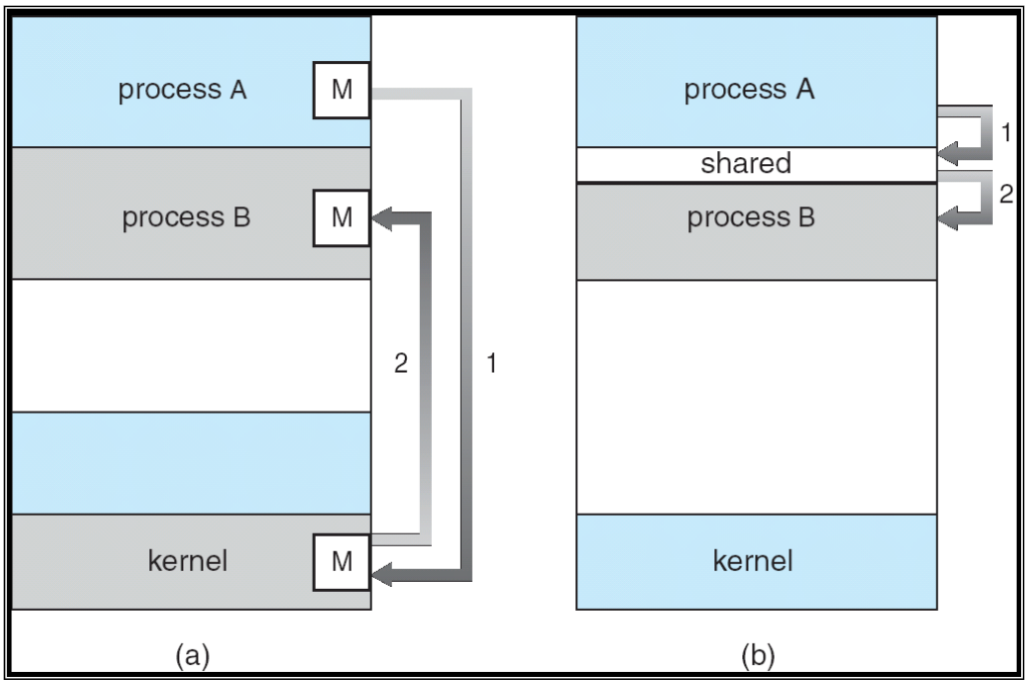
電腦系統溝通方法有兩種

- **Share Memory**
  - 各 Process **共享記憶體(共享變數 Shared Variables)**

- **互斥存取控制由 Programmer 負責**，OS 只負責共享記憶體空間，部提供額外資源

- **Messange Passing**
  - Process 溝通步驟
    - **建立 Communication Link**
    - **互傳訊息**
    - **傳輸完畢 Release Link**
  - **OS 需要提供額外支援**，Programmer 不需要額外負擔
    - e.g. Link Management, Link Capacity 管控, Messange Lost 處理...etc
  - 有**直接**和**間接**兩種方式

	Share Memory	**Message Passing**
定義	Processe 共享變數存取	1. 建立 Link 2. 互傳訊息 3. 釋放 Link
共享性	所有 Process 都可以存取共享變數	Link 專屬於溝通雙方 其他人不能共用
OS 支援	OS 提供 Share Memory 空間 部提供額外支援	OS 會提供額外支援 e.g. Link Management
Programmer 負擔	Programmer 負擔重 OS 負擔輕	Programmer 負擔輕 OS 負擔重
處理機制	須提供共享變數的互斥控制權	須提供 Link Creation, Link Capacity 控制, Message Lost...etc



(a) Message Passing (b) Shared Memory

Share Memory 共享記憶體

Race Condition 競爭情況

- Share Memory 若**未對共享變數提供互斥存取**，會因為 Process 的**執行順序不同**，導致**不可預期的錯誤**
- **Example**

  Process: P1, P2

  Shared Var: Count

  ```
  P1                    P2
  ------------------    ------------------
  Count = Count + 1     Count = Count - 1
  ```

  - Cocurrent 執行可能順序為

    T1: 執行 Count+1=6 但尚未 Assign 給 Count

    T2: 執行 Count-1=4 但尚未 Assign 給 Count

    T3: 將 T1 結果 Assign 給 Count

    T4: 將 T2 結果 Assign 給 Count

    Count=4

    若 T3, T4 對調 Count=6
- 需要對共享變數存取進行管制，同一時間只能有一個行程使用該變數
- 解決競爭情況的策略
  - Disable Interupt 停止使用中斷-->鎖住 CPU
  - Critical Section Design 臨界區間設計-->鎖住共享變數

Disable Interupt 停止使用中斷

- **執行過程中不會中斷**
  - 執行前 Disable Interupt
  - 執行後 Enable Interupt

  ```
  P1                    P2
  ------------------    -------------------
  Disable Interupt      Disable Interupt
  Count = Count + 1     Count = Count + 1
  Enable Interupt       Enable Interupt
  ```
- **優點** 簡單，易於實施
- **缺點** 只適用於 Single Processor 環境，Multiprocessor 系統的 Performance 差，且風險增加

- Multiprocessor 環境下，要關掉全部 CPU 的中斷功能才有用
- Process 需發出 Disable Interupt 給所有 CPU，且須要等所有 CPU 回覆 Disable Interupt 才能繼續執行，工作完成要在發出 Enable Interupt 給所有 CPU
- Disable Interupt 的做法會導致更緊急的工作無法執行

Critical Section Design 臨界區間設計

- **共享變數存取動作的互斥控制**，確保資料的正確性
- **臨界區間**
  - **對共享變數進行存取的指令**所組成的集合

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

  - **剩餘區間 Remaining Section** 非 C.S. 的指令集合
- 主要設計**入口**和**出口**的程式片段

  Entry Section　**進入區間** 管制 Process 是否可以進入 C.S.

  C.S.

  Exit Section　**離開區間** 離開 C.S. 後解除入口管制

  R.S.

```
P1                          P2
----------------------      ----------------------
...                         ...
Entry Section     (1)       Entry Section     (2)Busy Waiting
Count = Count + 1           Count = Count - 1 (4)
Exit Section      (3)       Exit Section
...                         ...
```

Critical Section 必須滿足的三個性質

Mutual Exclusion 互斥

- 任何一個時間點，**最多只允許一個 Process 進入 C.S.**

Progress 行進

- 不想進入 C.S. 的 Process 不可以阻礙其他 Process 進入 C.S.

- 必須在**有限時間內**，從想要進入C.S.的Process中，挑選出一個Process進入C.S.(No Deadlock)

Bound Waiting 有限等待

- 提出進入C.S.到獲准進入的等待時間必須有限
  - n個Process想進入C.S.，任一個Process等待時間至多n-1次(No Starvation)

C.S.設計的方法

Algorithm Based

2 Processes - Algorithm 1

- Process P_i, P_j
- 共用變數 **turn** 存放i或j，表示誰有權利進入C.S.
- P_i的程式設計如下

  [Entry] `while(turn!=i){no-op}` -->空迴圈 Busy Waiting

  C.S.

  [Exit] `turn = j;`

  R.S.
- **分析**
  - **滿足Mutual Exclusion**
    - turn不能同時為i且j
  - **不滿足Progress條件1**
    - 假設turn=i但i不想進入C.S.，導致P_j想進入C.S.但無法進入
  - **滿足Bounded Waiting**
    - 若P_i離開C.S.後又企圖立刻進入C.S.，會被卡住，因為P_i離開時turn已經設定為j，對P_j而言，至多等待一次就可以進入C.S.

2 Processes - Algorithm 2

- Process P_i, P_j
- 共享變數 bool **flag[n]**表示進入C.S.的意願
  - n = i or j
  - true表示有意願; false表示無意願
  - 初始值為false
- P_i的程式設計如下

  [Entry] `flag[i] = true;` //表示想進入C.S.

  `while(flag[j]){no-op}` //看對方想不想進C.S，如果對方想進C.S.則等待

C.S.

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

R.S.

- **分析**
  - **滿足 Mutual Exclusion**
  - **不滿足 Progress**
    - 會產生 Deedlock，**互相等待對方進入 C.S.**

      T1: flag[i] = true;

      T2: flag[j] = true;

      T3: P_i 卡在 while loop

      T4: P_j 卡在 while loop
  - **滿足 Bounded Waiting**

2 Processes - Algorithm 3

- Process P_i, P_j
- 共享變數
  - bool **flag[n]**
  - **turn**
- P_i 的程式設計如下

[Entry] `flag[i] = true;`

`turn = j;`

`while(flag[j] && turn==j){no-op}`

C.S.

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

R.S.

- **分析**
  - **滿足 Mutual Exclusion**
    - turn 控制由哪一個 Process 進入 C.S.
  - **滿足 Progress**
    - 若 flag[i]=false，此時 P_j 想要進入 C.S.一定可以跳出 busy waiting
    - 若 flag[i]=true, flag[j]=true，在有限時間內一定會執行到 turn=i, turn=j(No Starvation)
  - **滿足 Bounded Waiting**
    - 假設 Pi 離開 C.S.，一定會執行 turn=j，保證 P 可以至多等待一次就進入 C.S.

N Processes - Bakery's Algorithm

- 麵包店發號碼牌，向要進入 C.S.必須抽一張號碼牌，號碼重複依照 Process ID 決定，ID 小的先進去
- bool **Choosing[0,...,n-1]** 初始值為 False

- True 想要進入 C.S.，**正在挑號碼牌**
- False 1)不想進入 C.S.或 2)有意願且**拿完號碼牌**

- int **Number[0,...,n-1]**初始值為 0 (代表號碼牌)
  - Number[i] = 0 表示 Pi 不想進入 C.S.
  - Number[i] > 0 表示 Pi 想進入 C.S. (號碼牌號碼)
- 使用函式
  - Pi Number[i] = a; ID = b

    Pj Number[j] = c; ID = d
  - if(a > c || (a == c && b > d)) --> (a, b) > (c, d)
    - 號碼相同比較 Process ID
- P_i的程式設計如下

```
//正在挑號碼牌
Choosing[i] = true;

//抽號碼牌，等待人數+1 就是抽到的號碼
Number[i] = Max(Number[0],...,Number[n-1]) + 1;

//挑完號碼牌
Choosing[i] = false; /
```

[Entry]

```
for(j = 0 to n-1){
   //等其他人抽號碼牌
   while(Choosing[j]){no-op}

   //對方想要進入 C.S. 且 對方的號碼/ID 比較小
   while((Number[j]!=0)&&(Number[j],j)<(Number[i],i)){no-op}
}
```

```
C.S.
```

[Exit]

```
Number[i]=0; //表示不想進入 C.S.
```

```
R.S.
```

為何會取得相同的號碼牌

	Pi	**Pj**
T1	Choosing[i] = true	
T2		Choosing[j] = true;

T3	抽號碼牌 Max(...)+1=k+1 尚未 Assign 給 Number[i]	
T4		抽號碼牌 Max(...)+1=k+1 Assign 給 Number[j]
T5	k+1 Assign 給 Number[i]	
Number[i] = Number[j] = k+1		

移除第一個 while 迴圈，結果是否正確

	Pi	**Pj**
T1	Choosing[i] = true	
T2		Choosing[j] = true
T3	抽號碼牌 Max(...)+1 但尚未 Assign 給 Number[i]	
T4		抽號碼牌 Max(...)+1 Number[j] = 1
T5		Choosing[j] = false
T6		第一個 while(Choosing[i])被移除 因為其他 Process 的 Number[] = 0，跳過第二個 while loop 進入 C.S.
T7	Number[i] = 1	
T8	第一個 while(Choosing[j])被移除 因為 Pi 和 Pj 的號碼牌一樣，Pi 的 ID 比 Pj 小，所以跳過第二個 while loop 進入 C.S	
不滿足 Mutual Exclusion		

證明 Bakery's Algorithm 正確性

- **滿足 Mutual Exclusion**
  - **號碼牌不同** 號碼較小的 Process 先進入 C.S.
  - **號碼牌相同** ID 小的 Process 先進入 C.S.
- **滿足 Progress**
  - 若 Pj 不想進入 C.S. --> Number[j] = 0; Choosing[j] = false

如果此時 Pi 想要進入 C.S.

- while(Choosing[j]){no-op} --> 可以跳過，不用等待其他 Process 抽號碼牌
- while((Number[j]!=0)&&(Number[j],j)<(Number[i],i)) --> 可以跳過，其他 Process 沒抽號碼牌

◦ 若多個 Process 同時想要進入 C.S.，由號碼牌和 ID 決定進入 C.S.順序

- **滿足 Bounded Waiting**
  - 假設有 n 個 Process，號碼牌依序為 1~n 且都不相同，當 P0 離開 C.S.必須重新抽號碼牌，號碼為 n+1
  - 假設有 n 個 Process，號碼牌為 k 且都相同，當 P0 離開 C.S.必須重新抽號碼牌，號碼為 k+1
  - 上述兩種情況所有的 Process 至多等 n-1 次後，就可以進入 C.S.

HW 指令支援

系統直接提供 Atomic 特性的指令，讓程式碼可以在低一時間點被完成，不會中途被插斷

Test and Set 指令

- 此指令為 Atomically Executed
  - **在單位時間內可以順利做完，不被任何中斷干擾**
- **定義**

```
// 回傳 Target 的舊值，將 Target 設定為 True
Int Test_and_Set(int *Target){
   int temp;
   temp = *Target;
   *Target = 1;
   return temp;
}
```

- **應用**
  - bool Lock; 初始值為 false

```
[Entry] while(Test_and_Set(Lock)){no-op}
        C.S.
[Exit]  Lock = false;
        R.S.
```

- **範例**

	Pi	**Pj**

T1	Lock 初始值 = false Pi 先搶到 Test_and_Set(Lock) **Test_and_Set 回傳 false** **Lock = true** Pi 進入 C.S.	
T2		Pj 執行 Test_and_Set(Lock) 回傳值為 true Lock = true 卡在 while loop

- **分析**
  - 滿足 Mutual Exclusion
  - 滿足 Progress
    - 不用 C.S.的 Process 不會去搶 Test_and_Set 指令
    - 有限時間內一定有人搶到 Test_and_Set
  - 不滿足 Bounded Waiting
    - 若 Pi 離開 C.S.企圖馬上再進入 C.S.，Pi 很有可能再度先執行 Test_and_Set 指令

Swap 指令

- Atomatically Executed 指令
- **定義**

```
Procedure SWAP(bool *a, bool *b){
   bool temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

- **應用**
  - bool Lock; 初始值為 False

[Entry]

```
key = true;
while(key == false){
   SWAP(lock, key);
}
```

C.S.

[Exit]

```
Lock = false;
```

R.S.

- **範例**

	Pi	**Pj**
T1	執行 key = true Lock 初始值 = false Pi 先搶到 SWAP(Lock, key) Lock = true; key = false; Pi 進入 C.S.	
T2		執行 key = true; 此時 Lock = true; 執行 SWAP(Lock, key) key = Lock = true; 卡再 while loop

- **分析**
  - **滿足 Mutual Exclusion**
  - **滿足 Progress**
    - 不用 C.S.的 Process 不會去搶 SWAP
    - 有限時間內一定會有人搶到 SWAP
  - **不滿足 Bounded Waiting**
    - 若 Pi 離開 C.S.企圖馬上再進入 C.S.，Pi 很有可能再度先執行 SWAP 指令

Semaphore 號誌

Data Type and Abstract Data Type

- 包含
  - a set of **data**
  - 資料型態的**相關操作**

- **定義** 用來解決**C.S.設計**和**同步問題**的資料型態
- 假設變數 S 為 Semaphore，其為 **Integer** 通常初始值為 1
- 額外提供 Atomic 操作 Wait(S), Signal(S)
  - **Wait(S)**

    ```
    while(S <= 0){no-op}
    S = S - 1;
    ```

  - **Signal**

```
S = S + 1;
```

- **共享變數**
  - Semaphore **mutex = 1;** **(互斥控制)**
    (型態) **(變數)** **(初始值)**
- Pi 程式片段如下

```
[Entry]  Wait(mutex);
         C.S.
[Exit]   Signal(mutex);
         R.S.
```

簡單的同步問題

- 假設有 P1 和 P2 兩個 Current Execution 的 Process

```
P1          P2
------      ------
...         ...
S1          S2
...         ...
```

現在規定 S1 必須在 S2 之前執行，請利用 Semaphore 滿足此問題

  - 宣告一個共享變數 Semaphore S = 0

```
P1            P2
------        ------
...           ...
...           Wait(S)
S1            S2
Signal(S)     ...
...           ...
```

  - 號誌初始值設定
    - 互斥控制 1
    - 強迫暫停 0

生產者&消費者問題

- **生產者行程** 用來生產資訊
- **消費者行程** 用來消耗生產者所生產的資訊

- **Buffer** 存放生產的資訊，讓消費者消耗

- 必須**同步**
- Buffer Empty 消費者還可以進去消費
- Buffer Full 生產者還可以繼續存放資訊

- **緩衝區 Buffer**
  - **無限緩衝區** 空間容量沒有限制
    - Buffer Empty 消費只需要等待
    - 生產者可以無限制生產資訊
  - **有限緩衝區** 空間容量有限制
    - Buffer Empty 消費只需要等待
    - Buffer Full 生產者需要等待
- **範例** 假設生產者消費者問題共享 Buffer 有 n 個儲存空間
  - Semaphore mutex = 1; //初始值 1,對 Buffer 存取提供互斥控制
  - Semaphore empty = n; //初始值 n,紀錄 Buffer 現有空格數
  - Semaphore full = 0; //初始值 0,紀錄 Buffer 有資料存在的格數
  - **Producer 程式片段**

```
produce an item in nextp;  // 生產
wait(empty);               // 每執行一次 empty 減 1，即空格少 1
                           // empty 為 0 表示 Buffer 已滿，生產者需等待
wait(mutex);               // Buffer 存取必須互斥控制
   Add nextp to Buffer;    // 資料加入 Buffer
signal(mutex);             // 解鎖互斥控制
signal(full);              // full 加 1 表示 Buffer 中有填資料的格數加 1
```

  - **Consumer 程式片段**

```
wait(full);                // 每執行一次 full 減 1，即有資料的格數減 1
                           // full 為 0 表示 Buffer 已空，消費者需等待
wait(mutex);               // Buffer 存取必須互斥控制
   Remove item from Buffer;
signal(mutex);
signal(empty);             // empty 加 1 表示 Buffer 中的空格數加 1
```

哲學家晚餐問題

- 有 5 位哲學家和 5 枝筷子
  - 哲學家思考時不用餐
  - 想吃番時要依序取用左右兩邊的筷子，才能用餐
- 利用 Semaphore 設計
  - Semaphore chopstick[4] = {1, 1, 1, 1}; //初始值為 1
  - 某位科學家 Pi 的程式片段如下

```
wait(chopstick[i]);          // 拿左邊筷子，如果沒有筷子 --> waiting
wait(chopstick[i+1] % 5);    // 拿右邊筷子，如果沒有筷子 --> waiting
   eating;                   // 拿到一雙筷子，可以吃飯
signal(chopstick[i]);        // 放下左手邊筷子
signal(chopstick[i+1] % 5);  // 放下右手邊筷子
```

  - **分析** 此程式不正確，會有 Dead Lock 問題
    - 當每哲學家拿都拿起自己左邊的筷子後，都拿不到右邊的筷子
- 解法一 最多允許 4 位哲學家同時用餐，即限制用餐人數避免死結
  - 利用 Dead Lock Avoidence 定理
    - 已知 m = 5, Maxi = 2
    - 滿足條件 1 --> 1 <= (Maxi = 2) <= 5
    - 欲滿足條件 2 --> 可得 2n < 5 + n --> n < 5 --> n 最大值為 4
    - 會有飢餓問題
- 解法二 除非哲學家可以同時取得左右兩邊的筷子，否則不准拿筷子
  - 打破 Hold and Wait
- 解法三 採用非對稱作法
  - 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
  - 打破 Circular Wait
- 解法四 採用 Monitor

Semaphore 的缺點

- Programmer 誤用 wait() 和 signal()
  - 違反 Mutual Exclusion
  - 造成 Dead Lock
- 範例 1 Semaphore S = 1

```
signal(S);  // 會讓一堆 Process 進入各自的 C.S.，違反 Mutual Exclusion
   C.S.;
wait(S);
   R.S.
```

- 範例 2 Semaphore S = 1

```
wait(S);
   C.S.;
wait(S); // 會讓其他卡在第一個 wait(S)，造成 Dead Lock
   R.S.
```

- 範例 3 Semaphore S = 1

```
P1                          P2
---                         ---
```

```
T0:   wait(S);              T1:   wait(Q);              S = 0, Q = 0
T2:   wait(Q); //卡住       T3:   wait(S); //卡住       產生 dead lock
        Do something;               Do something;
      signal(S);                  signal(Q);
      signal(Q);                  signal(S);
-->形成 Dead Lock
```

Monitor

- 定義 為解決同步問題的高階資料結構，由三部分組成
  - **一組 Procedures(Operations)** 供外界呼叫使用，一個 Operation 完成一個工作
  - **共享資料區** 宣告一些共享變數，只提供給 Monitor 內部 Procedure 共用，外部不能直接使用
  - **初始區** 設定某些共享資料變數的初始值
- Monitor 本身以確保**互斥**的性質
  - 每次只允許一個 Process 在 Monitor 內活動
    - 某個 Process 在執行 Monitor 內的 Procedure 時，其他 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
    - **不允許 2 個以上 Process 在 Monitor 區域內活動**
    - 保證在共享資料區內的共享變數**不會有 Race Condition**
- **優點** Programmer 不需花費額外負擔處理互斥問題，可專注於同步問題
- **流程**
  - 定義 Monitor
  - 利用 Monitor 宣告一個變數
  - 撰寫 Process i 程式片段

定義 Monitor

- 資料區
- 程序區
- 初始區

- 同步事件發生時，Process 需要暫停和喚醒的工作
  - 舉例 Buffer 滿 --> 生產者暫停; Buffer 空 --> 消費者暫停
  - Monitor 提供一種特殊的變數公 Programmer 使用: **Condition 型態變數**
- **宣告 Condition x** 提供 2 個 Atomic Operations(可宣告多個 Condition x, y...)
  - **x.wait** 強迫 Process 停止，並把 Process 放入 Waiting Queue 中(Shared data)
  - **x.signal** 如果有 Process 卡在 x.wait 的 Waiting Queue，此操作會從該 Waiting Queue 中將第一個 Process 喚醒

利用 Monitor 解決哲學家晚餐問題

資料區

- 每隔哲學家會有三種不同狀態 `thinking, hungry, eating`
- 當哲學家發生搶不到筷子的情況時的**同步處理**

初始區

- 每個行程的初始狀態都設定為 `thinking`
- 不需針對 `Condition` 型態變數設定初始值，用來卡住哲學家

程序區

訊息傳遞

- 定義 兩個 `Process` 溝通需要遵守下列步驟
  - 建立 `Communication Link`
  - 互傳訊息
  - 傳輸完畢 `release link`
- 需要 `OS` 提供額外支援，`Programmer` 不需額外負擔
- 提供操作
  - `send()`
  - `receive()`

直接聯繫 Direct Communication

- **OS 自動產生** `Communication Link`，`Process` 需要知道對方 `ID` 即可
- 一條 `Link` **只連接兩個 `Process`**
- 進行通訊的兩個 `Process`，**只能有一條 `Link`**

Symmetric

- 接收者或傳送者在聯繫時必須互相指名(位址對稱)
- `send(P, message) -->` 傳送一個 `Message` 給 `Process P`
- `receive(Q, message) -->` 從 `Process Q` 接收一個 `Message`
- 兩個指令的參數無法搭配，則 `Link` 無法成功建立(必須相互指名)

Asymmetric

- 發送者須確定接收者的名稱，接收者不需指出發送者的名稱(位址不對稱)
- `send(P, message) -->` 傳送一個 `Message` 給 `Process P`

- receive(Id, message) --> 從任何 Process 接收一個 Message，收到 Message 後，Id 設定為發送者的 Id
- 接收方不在意是誰發送訊息過來

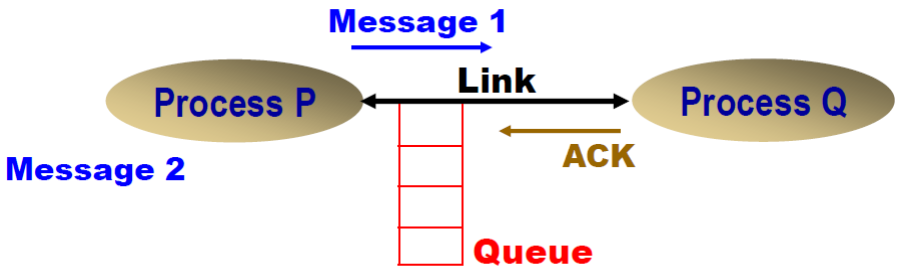
- 不論是 Symmetric 或 Asymmetric，當一個 Process 改變名稱，就需要對所有 Process 檢查，必須同步改為新名稱

範例 利用Direct Communication(Symmetric)解決生產者-消費者問題

```
Producer                    Consumer
--------------------        --------------------
Produce an item nextp       receive(Producer, nextc)
send(Consumer, nextp)       Consume the item nextc
```

- Message passing 會有類似同步的問題，是由 Link Capacity 引發

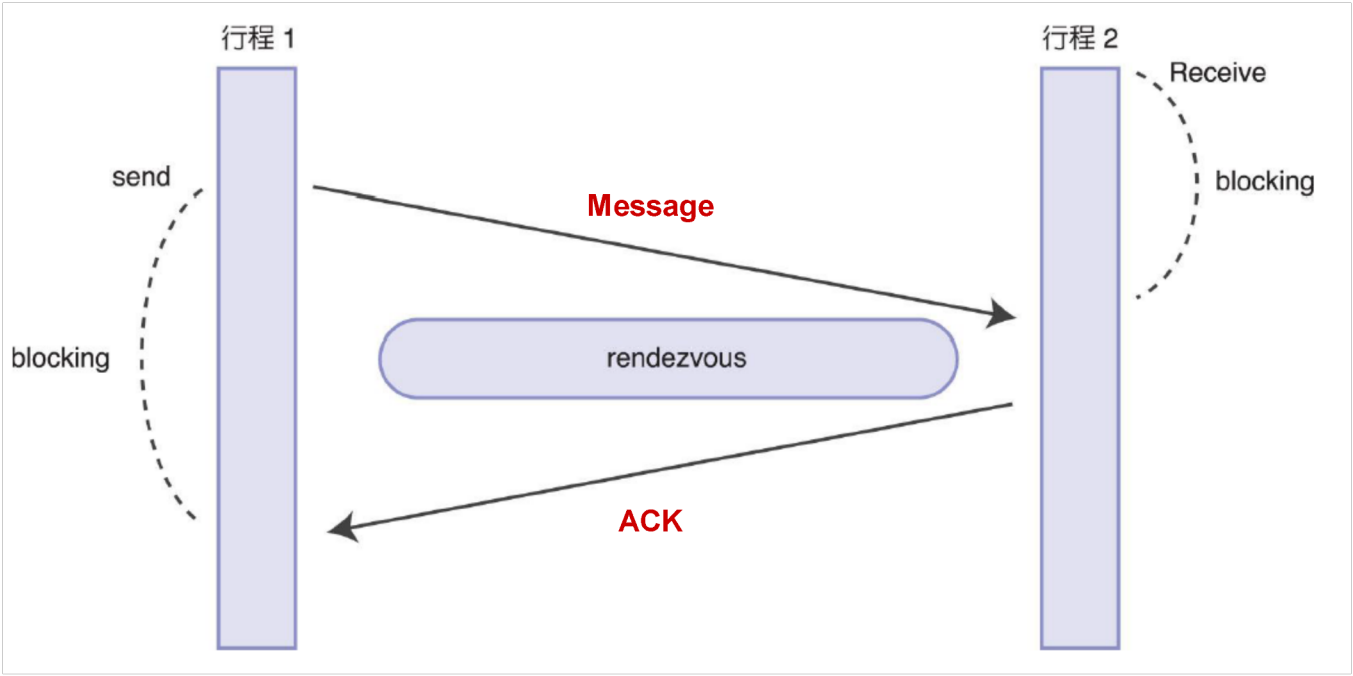
Link Capacity



- Queue 容量
  - Zero Capacity --> 沒有 Queue --> **同步狀況**
  - Bounded Capacity --> Queue 容量有限--> **同步狀況**
  - Unbounded Capacity --> Queue 容量無限

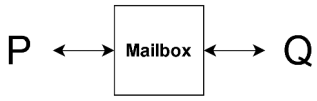
同步化

- 由於 Messaage Passing 的 Link Capacity，產生 **Blocking** 或 **Nonblocking**
  - **Blocking Send** 傳送者會等到接收者或 Mailbox 收到訊息為止
  - **Nonblocking Send** 傳送者送出訊息後繼續運作，不用等待
  - **Blocking Receive** 接收者會等到收到訊息才往下執行
  - **Nonblocking Receive** 接收者不管收到有效或無效訊息都可以往下執行
- 當傳送者和接收者都在**等待**，則傳送者和接收者之間就有**約會 (Rendezvous)**
  - Zero Capacity 為了保持同步，必須互相等待，這種架構叫做**約定 (Rendezvous)**

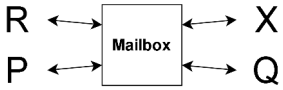


間接聯繫 Indirect Communication

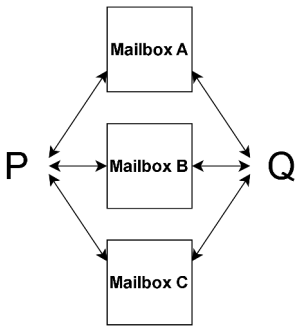
- 透過 **Mailbox (信箱)** 傳送接收訊息 **(透過 OS)**
- `send(A, message)` --> 傳送一個 `Message` 到 `Mailbox A`
- `receive(A, message)` --> 自 `Mailbox A` 接收一個 `Message`
- **特性**
  - 一對 `Process` 必須共用 `Mailbox` 才能建立 `Link`



  - 一條 `Link` 可以連到兩個以上的 `Process`
    - 不同對 `Process` 的 `Links` 可以共用 `Mailbox`



  - 進行通訊的兩個 `Process` 可以有多條不同的 `Link`，每個 `Link` 都對應一個 `Mailbox`



Message Passing 的例外狀況處理

- Message Passing 是由 OS 負責相關控制，需考慮例外狀況的處理

Process Terminates 行程結束

- 接收者 Q 或傳送者 P 在**訊息處理完成之前都有可能結束**，造成收不到或是無法傳送訊息
- 處理方法
  - **狀況 1** 接收者 Q 等待**已被中止的行程 P** 的訊息，行程 Q **無限期停滯**
  - **處理 1**
    - OS 可終止行程 Q
    - OS 通知行程 Q：行程 P 已經結束
  - **狀況 2** 傳送者 P 傳送一個訊息給**被終止的行程 Q**
    - 自動緩衝 對行程 P 無影響
    - 沒有緩衝 行程 P 需等待行程 Q 的回應訊息才可繼續工作，行程 P **無限期停滯**
  - **處理 2**
    - OS 可終止行程 P
    - OS 通知行程 P：行程 Q 已經結束

Message Lost 訊息遺失

- 硬體故障造成訊息遺失
  - **偵測是否遺失?**
  - **確認遺失，是否重送?**
- 處理方式
  - **OS** 負責偵測是否遺失，若有需要 **OS** 負責重新傳送訊息 --> **OS** 負擔過高
  - **傳送者**負責偵測是否遺失，若有需要**傳送者**負責重新傳送訊息 --> **傳送者**負擔過高
  - **OS** 負責偵測是否遺失，若有需要 OS 告知**傳送者**重新傳送訊息 --> 較常用

偵測訊息遺失

- 使用 **Time-Out(限時)**
- Message 發出後會有一個確認訊息(Acknowlegement, ACK)送回來
- OS 或行程規第一個時間間隔 T，時間間隔 T 未收到 ACK 訊息(>2T 時間內沒收到)，OS 或行程假定訊息遺失
- 網路傳輸時間比規定時間間隔長，但訊息未遺失，會造成有好幾份訊息拷貝
  - 接收者必須區分這些訊息的不同備份(判斷是否為同一次發送的訊息)