

CH6 、 Synchronization

Process 之同步問題

目錄：

Process Communication(Shared Memory 與 Message Passing)

1. Shared Memory：

Race Condition Problem 解決方式之兩大策略

Disable Interrupt

Critical Section Design

CS Design 必須滿足的 3 個 Criteria

Mutual Exclusive、Progress、Bounded Waiting

CS Design 方法(架構)

Hardware Instruction Support(Test-and-Set、SWAP)

Semaphore(號誌)

定義

應用： CS Design、Synchronization Problem Solution

種類： Binary vs Counting Semaphore、

Spinlock vs Non-Busy Waiting Semaphore

製作

Monitor

解決有名的同步問題

Producer-Consumer Problem

Reader-Writer Problem (First/Second type)

The Sleeping Barber Problem

The Dinner-Philosopher Problem

2. Message Passing 溝通方式

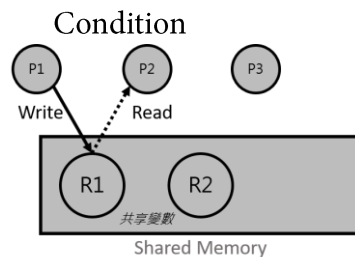
Process Communication 的 2 種方式

[1] Shared Memory

一、Def：Process 透過對共享變數(Share variables)之存取(Write/Read)，達到溝通(Information exchange)之目的

二、分析：

1. 適用於大量 Data(Message)傳輸之狀況
2. 傳輸速度較快(因為不需 kernel 介入干預/支持)
3. 不適合用於 Distributed System
4. Kernel 不需提供額外的支援(頂多供應 Shared Memory Space)
5. 是 Programmer 的負擔 => 必須寫額外的控制程式碼，防止 Race Condition



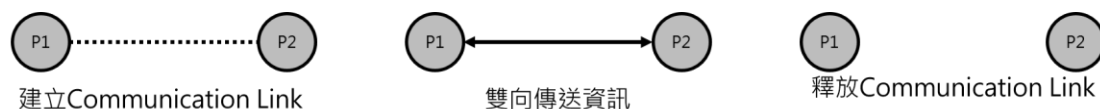
[2] Message Passing

一、Def：Process 雙方要溝通，必須遵循下列步驟：

1. 建立 Communication Link
2. 訊息可以雙向傳輸
3. 傳輸完畢，釋放 Communication Link

二、分析

1. 不適用於大量 Data 傳輸
2. 速度較慢(因為需 kernel 介入)
3. 適合 Distributed System
4. Kernel 需提供額外支援：例：Send/Receive 之 System Call、Communication Link 管理，Message Lost 之偵測，例外狀況處理
5. Programmer 沒什麼負擔(會使用 OS 提供的 API 即可)



Process Communication = Process Synchronization = Inter Process Communication(IPC)

Race Condition Problem

一、Def：In shared memory Communication，若未對共享變數存取提供任何互斥存取控制之 Synchronization 機制，則會造成”共享變數的最終結果值，會因為 Process 之間的交錯執行，有不同的結果值”，此種 data inconsistency 情況稱之

二、例：c 是共享變數，初值為 5。有 P1、P2 兩個 Concurrent execution processes

P1	P2
...	...
c=c+1;	c=c-1;
...	...

若 P1、P2 各執行一次，c 的結果為何？

4、5、6 皆有可能，說明如下：

P1、P2 中的實際運算指令，執行上是需要拆成三條組合語言程式碼：

c=c+1;	T1 : load R1, c	c=c-1;	T2 : load R2, c
	T3 : INC R1		T4 : Sub R2
	T5 : STD R1, c		T6 : STD R2, c

而當執行順利分別為以下 3 種情況，c 之運算值亦有 3 種不同結果

1. T1(R1=5) -> T2(R2=5) -> T3(R1=6) -> T4(R2=4) -> T5(c=6) -> T6(c=4) => c=4
2. T1(R1=5) -> T3(R1=6) -> T5(c=6) -> T2(R2=6) -> T4(R2=5) -> T6(c=5) => c=5
3. T1(R1=5) -> T2(R2=5) -> T3(R1=6) -> T4(R2=4) -> T6(c=4) -> T5(c=6) => c=6

因此需要解決此 Race Condition 之狀況

考試型態：

1. 解釋名詞、舉例
2. 搭配 fork(程式，考追蹤題)
3. 給予多個 Process 之片段敘述，問執行結果

例 1：x, y 是共享變數，初值為 x=5、y=7，Pi、Pj 程式碼如下，求 Pi、Pj 各執行一次之(x, y)結果可能值？

Pi	Pj
...	...
x=x+y;	y=x*y;
...	...

結果可能值為：(12, 84)、(40, 35)、(12, 35)

例 2：x 是共享變數，初值為 x=0，求 Pi、Pj 各執行一次之(x, y)結果可能值？

Pi	Pj
for(i=1;i<=3;i++) x=x+1;	for(i=1;i<=3;i++) x=x-1;

結果可能值為：-3、-2、-1、0、1、2、3

解決 Race Condition 的 2 大策略

1. Disable Interrupt：對 CPU 下手(還沒做完時不可被搶先)
2. Critical Section Design[誤用：Spinlock = Busy-waiting]：對共享資料下手(可以被切來切去，但對 data 要保護)

記憶方式：Disable Interrupt 的”D”與『單』CPU 同音；Spinlock 的”s”與『多』CPU，皆表示複數之意思，故 Disable Interrupt 適合單一 CPU、Spinlock 則適合多 CPU

[1] Disable Interrupt

一、Def：Process 在對共享變數存取之前，先 Disable Interrupt，等到完成共享變數存取後，才 Enable Interrupt，可保護 Process 存取共享變數期間，CPU 不會被 Preemptive，即此一存取是”Atomically Executed”(不可分割的執行)。

二、優點：

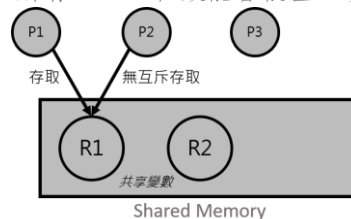
1. Simple, easy to implement
2. 適用於 Uniprocessor System(單一 CPU)

缺點：

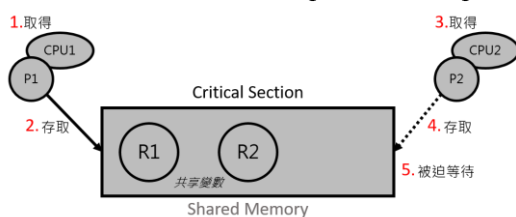
1. 不適用於 Multiprocessors System 中：因為只 Disable 一顆 CPU 的 Interrupt，是無法防止 Race Condition(其他 CPU 上執行的 Process，仍有存取共享變數的可能)，必須 Disable 『全部』的 CPUs Interrupt，才可防止 Race Condition，但此舉會大幅降低 performance(因為無法平行執行)
2. 風險很高：必須信任 User process 在 Disable Interrupt 後，在短時間內可以再 Enable Interrupt，否則 CPU never come back to kernel，產生極大風險。

Note：通常 Disable Interrupt 作法是不會放給 User Process 的，它通常只存在於 kernel 的製作中(只有 OS Developers 可以用)

P1 在 CPU1 執行，如果只 Disable CPU1 的 Interrupt，P2 依然可以到 CPU2 執行並存取共享變數，故多處理器需要一次 Disable 所有 CPU，但效能會很差，故不適用



[2] Critical Section Design (CS Design，臨界區間)

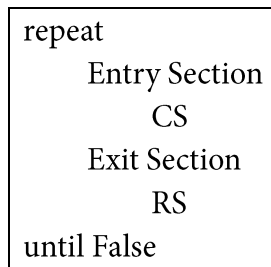


一、Def：對”共享變數”之存取，進行管制。當 Pi 取得共享變數存取權利，在它尚未完成共期間，任何其他 Process 無法存取共享變數，即使它們取得 CPU

二、Critical Section：Process 中對共享變數進行存取敘述之集合

Remainder Section：Process 中，除了 CS 之外的區間

Process 內容：每個 CS 的前及後，Programmer 需設計/加入額外的控制程式碼叫 Entry Section 與 Exit Section



而 CS Design 是在設計 Entry/Exit Section Code.

優點：適用 Multiprocessors System

缺點： 1.設計較為複雜
2.較不適合用在 Uniprocessor

Busy-Waiting Skill (Spinlock)

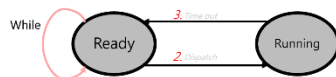
一、Def：透過使用 looping 相關敘述(ex：for、while、repeat...until)達到讓 Process 暫時等待之效果

例：

while(條件式) do no-op; 或者： while(條件式);

當條件式為真，Process 就被卡在 while 中，無法離開 while，如此達到 Process 暫停之效果，直到條件式變 False，Process 才會離開往下執行

1.



2. [恐]誤用：因為在 CS Design 的 Entry Section 中，經常使用 Busy-Waiting(Spinlock)技巧，所以[恐]也會把 Busy-Waiting(Spinlock)當成 CS Design，來與 Disable Interrupt 比較(但其實這是誤用)

二、Busy-Waiting 技巧

缺點：等待中的 Process 會與其他 Process 競爭 CPU，將搶到的 CPU Time 用在毫無實質進展的迴圈測試上，因此，若 Process 要等待長時間才能 Exits 迴圈，則此舉非常浪費 CPU Time

優點：若 Process 卡在 loop 的時間很短(i.e.,小於 Context Switching Time)，則 Spinlock 十分有利

三、另一種 Non-Busy-Waiting Skill

Def：當 Process 因為同步事件被卡住，且如果要卡很久的時間，則可以使用 Block(P)的 System all，將 P 暫停，即在 Blocked state，如此，P 就不會與其他 Process 競爭 CPU，直到同步事件發生了，才用 wakeup(P)的 System Call 將 P 從 Blocked state 切回到 Ready state

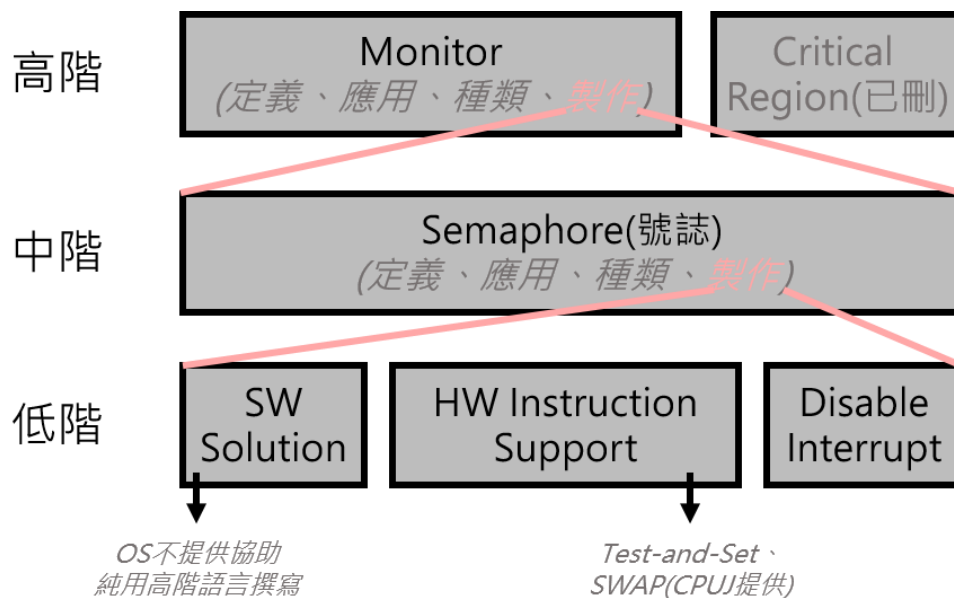
優點：等待中的 Process 不會與其他 Process 競爭 CPU、也不會浪費 CPU Time

缺點：額外付出 Context Switching Time

CS Design 應該滿足的 3 個性質

1. **Mutual Exclusion**：在任何時間點，最多只允許一個 Process 進入它自己的 CS，不可有多個 Process 分別進入各自的 CS
2. **Progress**：
 - (1) 不想進入 CS 的 Process(或在 RS 內活動的 Process)不能阻礙其他 Process 進入 CS(或不參與進入 CS 之決策)
 - (2) 從那些想進入 CS 的 Process 中，決定誰可以進入 CS 的決策時間，是有限的(不可以無窮)，即 No Deadlock(不可以 Waiting forever，不可以大家皆無法進入 CS)
3. **Bounded Waiting**：自某 Process 提出申請，到核准進入 CS 的等待時間(次數)是有限的。即若有 n 個 Process 想進入 CS，則這一個 Process 至多等待 $(n-1)$ 次後，即可進入 CS，也就是 No-Starvation，需公平對待

CS Design 架構圖



Software Solution

2 個 Process(P_i 、 P_j 或 P_0 、 P_1)	n 個 Process($n > 2$)
演算法 1(X) 演算法 2(X) 演算法 3(O) => Peterson's Solution	Peterson's Solution(n 個 Process)(不重要) Bakery's 演算法([恐]已刪，但重要)

2 個 Process 之 CS Design(Pi、Pj)

[演算法 1：Turn]

Global int: turn=i or j (only); 意義：權杖，turn 的值為 i，就是只能讓 Pi 進入(有資格進入)	
Pi: repeat while(turn!=i) do no-op; CS turn=j; //權杖給對方 RS until False	Pj repeat while(turn!=j) do no-op; CS turn=i; //權杖給對方 RS until False

分析

[1] Mutual Exclusion(O)：turn 值不會同時為 i 且為 j，只會為 i 或 j 之其中一個值，因此只有 Pi 或 Pj 一者進入 CS，不會 2 個同時進入 CS

[2] Progress(X)：假設目前 Pi 在 RS(Pi 不想進入 CS)，且 turn 平為 i，若此時 Pj 想進入 CS，將會無法進入，因為被 Pi 阻礙，因此唯有仰賴 Pi，才能將 turn 設為 j，Pj 才能進入 CS，但此時 Pi 並不想、也不會進入 CS、去執行 turn 的更改

[3] Bounded Waiting(O)：假設目前 turn 值為 i，且 Pi 已先於 Pj 進入 CS，而 Pj 等待中，當 Pi 離開 CS，Pi 會將 turn 設為 j，因此就算是 Pi 又立刻想再進入 CS，也因為 turn 值為 j，一定是 Pj 進入，而無法先於 Pj 進入 CS，因此 Pj 至多等一次後，就可以進入 CS

[演算法 2：Flag]

Global Boolean: flag[i,j] = False; 意義：旗誌，True 則有意進入 CS；False 則無意進入 CS	
Pi repeat T1: flag[i]=True; T4: while(flag[j]) do no-op; CS flag[i]=False; RS until False	Pj repeat T2: flag[j]=True; T3: while(flag[i]) do no-op; CS flag[j]=False; RS until False

分析

[1] Mutual Exclusion(O)：flag[i]與 flag[j]值不會同時為 True，一次只會有一個 flag 為 True，因此只有 Pi 或 Pj 一者進入 CS，不會 2 個同時進入 CS

[2] Progress(X)：當執行順序如上表程式碼中的 T1、T2、T3、T4 時，便會造成 Deadlock 的狀況出現、無法進入 CS，因此違反 Progress

[3] Bounded Waiting(O)：假設目前 flag[i] 值為 True，且 Pi 已先於 Pj 進入 CS，而 Pj 等待中，當 Pi 離開 CS，Pi 會將 flag[i] 值為 False，因此就算是 Pi 又立刻想再進入 CS，也因為 flag[i] 值為 False，一定是 Pj 進入，而無法先於 Pj 進入 CS，因此 Pj 至多等一次後，就可以進入 CS；反之 Pj 亦是如此

[演算法 3：Turn 與 Flag]

Global Boolean: flag[i:j] = False; int: turn=i or j (only); 意義：旗誌，True 則有意進入 CS；False 則無意進入 CS 意義：權杖，turn 的值為 i，就是只能讓 Pi 進入(有資格進入)	
Pi repeat flag[i]=True; //表達有意願 turn=j; //權杖給對方 while(flag[j] && turn==j) do no-op; //對方有意、且權杖在對方身上 CS flag[i]=False; //無意願 RS until False	Pj repeat flag[j]=True; //表達有意願 turn=i; //權杖給對方 while(flag[i] && turn==i) do no-op; //對方有意、且權杖在對方身上 CS flag[j]=False; //無意願 RS until False

分析

[1] Mutual Exclusion(O)：若 Pi、Pj 皆想進入 CS，表 flag[i] 及 flag[j] 皆為 True，當雙方皆作到 while 測試時，表示雙方分別皆執行過 turn=i 及 turn=j，只是順序先後不同，因此 Turn 值只會為 i 或 j 其中之一，不可能同時兩者，因此只有 Pi 或 Pj，其中一個可以進入 CS

[2] Progress(O)：

1. 假設 turn 值目前為 i，且 Pi 不想進，代表 flag[i] 為 False，若此時 Pj 想進，則 Pj 必可離開 while(因為 flag[i]=False)，而進入 CS，所以 Pi 不會阻礙 Pj 進入 CS
2. 若 Pi、Pj 皆有意進入 CS，則在有限的時間內，必可決定出 turn 值為 i 或 j，讓 Pi 或 Pj 進入 CS。兩者都不會 waiting forever.

[3] Bounded Waiting(O)：假設 turn=i，Pi 已先於 Pj 進入 CS，而 Pj 等待進入中，flag[i]==flag[j]==True，若 Pi 離開 CS 後，必定會將 turn 設為 j，因此就算 Pi 又立刻想要再進入 CS，也一定是 Pj 先進入 CS，無法先於 Pj 進入 CS，因此 Pj 至多等一次，就可以進入 CS

n 個 Process 之 CS Design(n>2)

[Peterson's 演算法]已刪略過

Bakery's 演算法(麵包店取號碼牌)

一、觀念：

1. 客人要先取得號碼牌才可以進入店內
2. 店內一次只有一位客人
3. (1)號碼牌最小的客人、或
(2)同為最小號碼的多位客人之中，ID 最小的客人，得以入店

二、共享變數

1. Boolean choosing[0: n-1] = False;
(意義：Pi 正在取號碼牌為 True、Pi 取得號碼牌為 False)
2. int Number[0: n-1]=0;
(代表 n 個 Process 之號碼牌數值，為正代表有意願進入 CS：0 表示無意願)

三、數學函數

1. MAX()：取最大值
2. (a, b) < (c, d)：必須滿足(1)a<c、或(2)a=c 且 b<d

四、程式

```
Pi
repeat
    choosing[i]=True;
    Number[i]=MAX(Number[0]~Number[n-1])+1;
    choosing[i]=False;
    for(j=0;j<n-1;j++)
    {
        while(choosing[j]) do no-op;
        while(Number[j]>0 && (Number[i], j) < (Number[j], i)) do no-op;
    }
    CS;
    Number[i]=0;
    RS;
until False
```

五、經典問題：

Q1：為何會有多個 Process 取得相同的號碼牌？

A1：目前 MAX(Number[0]~Number[n-1])之值為 k，Pi、Pj2 個 Process 之交錯執行順序如下：

T1：執行 MAX()+1，但未 assign 回 Number[i] T3：k+1 assign 回 Number[i] => Number[i]=Number[j]=k+1	T2：Number[j]=MAX()+1=k+1 <i>Race Condition</i>
--	---

Q2：正確性證明：

1. Mutual Exclusion：

- (1) 假設 Number 值皆不同(>0)，則具有最小的 Number 值之 Process，得以優先進入 CS，其餘 Process 要等待最小 Process 完成才能進入，故 Mutual Exclusion

- (2) 有多個 Process 具有相同 Number 值，則以 Process ID 最小者，得以進入 CS，且其他 Process 需等到它完成才得以進入，再加上因為 Process ID 是唯一的，故 Mutual Exclusion
 由(1)與(2)可得知：唯一性確立，Mutual Exclusion

2. Progress :

- (1) 假設 Pj 不想進，代表 Number[j] 為 0，此時若 Pi 想進，則 Pi 檢測到 Pj 時，Pi 必定不會被 Pj 所阻礙，可以跳出 for 中的第二個 while 迴圈(因為 while(Number[j]>0) 是 False)
- (2) 若 P0~Pn-1，這 n 個 Process 皆想進入 CS，則在有限的時間內，必有一個 Process(其 Number 最小、或多個相同 Number 中、ID 最小的 Process)，可以順利離開 For 迴圈、進入 CS，並不會有 Deadlock，故 Progress
3. Bounded Waiting：假設 P0~Pn-1，n 個 Process 皆想進入 CS，令 Pi 具有最大的 Number 值 k(Number[i]=k)，因此其他 n-1 個 Process，都必定先於 Pi 進入 CS，若其中一個 Process Pj 離開 CS 後，又立刻想要進入 CS，則 Pj 取得之號碼牌 Number[j] 值，一定會大於 k，因此 Pj 不會再度先於 Pi 進入 CS，因此 Pi 頂多等(n-1)次，就可以進入 CS 了

Q3：Remove 第一個 while 是否正確？

A3：會違反 Mutual Exclusion

例：令目前 Number[0]~Number[n-1] 之值為 0，Pi 與 Pj 兩個 Process(i≠j) 皆想進入 CS，且假設 Process ID 是 i<j

Pi	Pj
T1 : choosing[i]=True; MAX()+1=0+1=1; <i>但尚未 assign 給 Number[i]</i> T3 : assign 回 Number[i] T4 : Pi 也可以順利跑完 for loop <i>因為 Number[i]=Number[j]，但 Process ID 是 i<j，所以 Pi 也進入 CS，因此違反 Mutual Exclusion</i>	T2 : choosing[j]=True Number[j]=MAX()+1=1 Choosing[j]=False <i>可順利離開 for loop，不會被 Pi 阻礙</i>

Hardware (CPU) Instruction Support

若 CPU 有提供下列指令之一，則 Programmer 可運用在 CS Design 上：

1. Test-and-Set(Lock)
2. SWAP(a, b)

以上兩者基本功能相同，只有關鍵指令執行方式不同而已

Test-and-Set(Lock)指令

一、Def：此 CPU Instruction 之功能為：傳出 Lock 參數值，且將 Lock 參數為 True(1)且 CPU 保證此指令為”Atomically” executed

若以 c 語言說明此指令功能：

```
int Test-and-Set(int *Lock)
{
    int temp = *Lock;
    *Lock=1;
    return temp;
}
```

二、用在 CS Design 上

演算法 1：(X)

1. 共享變數：

Boolean: Lock=False;

2. 程式：

```
repeat
    while(Test-and-Set()) do no-op;
    CS;
    Lock=False;
    RS;
until False
```

比喻：紳士的追求

今天有許多位紳士，想要追求同一位氣質女孩(任何狀況下都不劈腿，Mutual Exclusion)但因為不確定對方是否單身，因此會有禮貌地傳書詢問女孩的管家(Test-and-Set())：是否有榮幸可以與她交往看看呢？，該管家都會回復，但若是回復內容為：她目前有交往對像(回傳值為1)，則紳士們會很紳士地不打擾(女孩)，是我(們)的溫柔(do no-op)，不過雖然不會打擾女孩，但依然可以每隔一陣子再次傳書詢問管家；若是詢問結果變成：女孩已回復單身(回傳值為0)，則收到回復的紳士會立刻開始與女孩交往(進入CS)。

至於是哪一位紳士可以成為下一任交往對像，就要看是哪一位紳士，最先詢問到女孩回復單身的那一位紳士了(最先搶到 Test-and-Set()回傳值為0)

3. 分析：

(1) Mutual Exclusion：(O)

(2) Progress：(O)

(3) Bounded Waiting：(X)

假設 Pi 已先於 Pj 進入 CS，且 Pj 等待中，當 Pi 離開 CS 後，又立刻想要再進入 CS，則 Pi 有可能再度先於 Pj 搶到 Test-and-Set 之執行，再度先於 Pj 進入 CS，因此 Pj 可能會 Starvation，違反 Bounded Waiting

演算法 2 : (O)

1. 共享變數 :

(1) Boolean: Lock=False;

(2) Boolean: Waiting [0: n-1]=False

意義 : True 表 P_i 有意願進入 CS , 且正在等待中 ; False 表不用等待 , 可以進入 CS

2. 程式 :

```
區域變數 :
    Boolean: key
    int: j;

repeat
    waiting[i]=True;
    key==True;
    while(waiting[i] and key)
        key=Test-and-Set(Lock);
    waiting[i]=False;
    CS;
    j=(i+1)%n;
    if(j==i) then
        Lock=False;
    else
        waiting[j]=False;
    RS;
until False
```

比喻：紳士的追求

與演算法 1 概念相同，唯一差別在於：追求的紳士們會有一個順序，每一位剛分手的前男友，都會通知排序下一位追求者，第一時間來詢問管家，進而使其能成為下一位交往對象

3. 分析 :

(1) Mutual Exclusion : (O)

P_i 可進入 CS 之條件有 2 種可能 :

- i. $key=False$ 代表 P_i 是第一個搶到 Test-and-Set 的執行者，如此才能把 key 改為 False，否則絕不可能，故唯一性成立
- ii. 任何本來在等待的 Process (P_i)，是無法將自己的 $waiting[i]$ 改為 False。只有當要離開 CS 的 Process (P_j)，才會修改其他在等待 Process 們的其中一個(P_i)，將 $waiting[i]$ 改為 False。尤於一次只有一個 Process 進入 CS，一次也只有一個 Process 離開 CS，則等待中，會被修改的 Process，也只會有一個

由 i、ii 可知，Mutual Exclusion

(2) Progress : (O)

- i. 若 P_i 不想進入 CS，其 $waiting[i]$ 為 False，且 P_i 不會跟其他 Process 競爭 Test-and-Set 之執行，且從 CS 離開之 Process 也不會改變 P_i 之 $waiting[i]$ 值，因此 P_i 不會參與誰要進 CS 之決策

- ii. 若 n 個 Process 都想進入 CS，則在有限的時間內，必定會決定出一個 Process，取得 Test-and-Set 的執行而進入 CS，等它從 CS 離開後，也會在有限的時間內，讓下一個想進 CS 之 Process 取得 Test-and-Set、進入 CS，

(3) Bounded Waiting : (X)

假設 $P_0 \sim P_n$ 等 n 個 Process 皆想進入 CS，表示 $waiting[0] \sim waiting[n-1]$ 皆為 True，令 P_i 是第一個搶到 Test-and-Set 執行之 Process、率先進入 CS，當 P_i 離開 CS 後，會將 $P(i+1)\%n$ 之 $waiting$ 值改為 False，讓 $P(i+1)\%n$ 進入 CS，依此類推，Process 會讓 P_i 、 $P(i+1)\%n \dots$ 以 FIFO 順序、依序進入 CS，而不會有 Starvation，故 Bounded Waiting

SWAP(a, b)指令

一、Def：此 CPU 指令將 a, b 兩值互換，且 CPU 保證它是” Atomically” executed.

```
void SWAP(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

二、用在 CS Design 上
演算法 1：(X)

1. 共享變數：
Boolean: Lock=False;
2. 程式：

```
區域變數：
Boolean: key

repeat
    key = True;
    SWAP(Lock, key);
until(key==False)
    CS;
    Lock=False;
    RS;
until False
```

比喻：紳士的追求

今天有許多位紳士，想要追求同一位氣質女孩(任何狀況下都不劈腿，Mutual Exclusion)但因為不確定對方是否單身，因此會有禮貌地傳一枚金幣(a)女孩的管家(SWAP())：表示想與她交往看看？，若是女孩目前有交往對象，就會回傳原本的金幣(回傳值為 a)，則紳士們會很紳士地不打擾(女孩)，是我(們)的溫柔(do no-op)，不過雖然不會打擾女孩，但依然可以每隔一陣子再次傳送金幣詢問管家；若是女孩已回復單身，則會收下金幣，並回贈一枚銀幣表示同意(回傳值為 b)，則收到回復的紳士會立刻開始與女孩交往(進入 CS)。但若是紳士與女孩緣分已盡、結束感情關係時，女孩會請管家向紳士收回原本的銀幣(b)，同時也還回紳士的金幣(a)

至於是哪一位紳士可以成為下一任交往對象，就要看是哪一位紳士，最先詢問到女孩回復單身的那一位紳士了(最先 SWAP()回傳值為 b)

3. 分析：
 - i. Mutual Exclusion：(O)
 - ii. Progress：(O)
 - iii. Bounded Waiting：(X)

假設 P_i 已先於 P_j 進入 CS，且 P_j 等待中，當 P_i 離開 CS 後，又立刻想要再進入 CS，則 P_i 有可能再度先於 P_j 搶到 SWAP()之執行，再度先於 P_j 進入 CS，因此 P_j 可能會 Starvation，違反 Bounded Waiting

演算法 2 : (O)

1. 共享變數 :

(3) Boolean: Lock=False;

(4) Boolean: Waiting [0: n-1]=False

意義 : True 表 P_i 有意願進入 CS , 且正在等待中 ; False 表不用等待 , 可以進入 CS

2. 程式 :

```
區域變數 :
    Boolean: key
    int: j;

repeat
    waiting[i]=True;
    key==True;
    while(waiting[i] and key)
        SWAP(Lock, key);
    waiting[i]=False;
    CS;
    j=(i+1)%n;
    if(j==i) then
        Lock=False;
    else
        waiting[j]=False;
    RS;
until False
```

比喻：紳士的追求

與演算法 1 概念相同，唯一差別在於：追求的紳士們會有一個順序，每一位剛分手的前男友，都會通知排序下一位追求者，第一時間來提交金幣給管家，進而使其能成為下一位交往對象

3. 分析 :

(1) Mutual Exclusion : (O)

P_i 可進入 CS 之條件有 2 種可能：

- i. $key=False$ 代表 P_i 是第一個搶到 SWAP() 的執行者，如此才能把 key 改為 False，否則絕不可能，故唯一性成立
- ii. 任何本來在等待的 Process (P_i)，是無法將自己的 $waiting[i]$ 改為 False。只有當要離開 CS 的 Process (P_j)，才會修改其他在等待 Process 們的其中一個 (P_i)，將 $waiting[i]$ 改為 False。尤於一次只有一個 Process 進入 CS，一次也只有一個 Process 離開 CS，則等待中，會被修改的 Process，也只會有一個

由 i、ii 可知，Mutual Exclusion

(2) Progress : (O)

- i. 若 P_i 不想進入 CS，其 $waiting[i]$ 為 False，且 P_i 不會跟其他 Process 競爭 SWAP() 之執行，且從 CS 離開之 Process 也不會改變 P_i 之 $waiting[i]$ 值，因此 P_i 不會參與誰要進 CS 之決策

- ii. 若 n 個 Process 都想進入 CS，則在有限的時間內，必定會決定出一個 Process，取得 SWAP() 的執行而進入 CS，等它從 CS 離開後，也會在有限的時間內，讓下一個想進 CS 之 Process 取得 SWAP()、進入 CS，

(3) Bounded Waiting : (X)

假設 $P_0 \sim P_n$ 等 n 個 Process 皆想進入 CS，表示 $waiting[0] \sim waiting[n-1]$ 皆為 True，令 P_i 是第一個搶到 SWAP() 執行之 Process、率先進入 CS，當 P_i 離開 CS 後，會將 $P(i+1)\%n$ 之 $waiting$ 值改為 False，讓 $P(i+1)\%n$ 進入 CS，依此類推，Process 會讓 P_i 、 $P(i+1)\%n \dots$ 以 FIFO 順序、依序進入 CS，而不會有 Starvation，故 Bounded Waiting

Semaphore 內容架構

定義

用在 CS Design

用於著名同步問題之解決

種類：

角度一：Binary semaphore vs Counting semaphore

角度二：Spinlock vs Non-busy waiting semaphore

Semaphore 的製作

Semaphore(號誌)

一、Def：令 S 為 Semaphore Type 變數，架構在 Integer Type 上，針對 S ，提供 2 個"Atomic" Operations：wait(S)與 signal(S)定義如下：

wait(S)：while($s \leq 0$) do no-op; $s = s - 1$;

signal(S)： $s = s + 1$;

Note：因為 Atomic，所以 s 不會有 Race Condition

二、Semaphore 主要應用在 CS Design 及同步問題之解決

CS Design 使用如下：

1. 共享變數宣告如下：

Semaphore：mutex = 1;

2. 程式：

Pi
repeat
wait(mutex);
CS;
signal(mutex);
RS;
until False

Mutual Exclusion、Progress、Bounded Waiting 三條件皆滿足

解決簡單的同步問題

例 1：規定 A 必須在 B 之前執行，試用 Semaphore 達到此需求？

宣告一共享變數 $S=0$ (初值)

Pi	Pj
...	...
A;	wait(S);
signal(S);	B;
...	...

Note：Semaphore 的初值，有某些意義： $1 \Rightarrow$ 互斥控制使用； $0 \Rightarrow$ 強迫等待

例 2：以下程式碼的 A、B、C 執行順序為何？

$S1=S2=S3=0$

Pi	Pj	Pk
repeat	repeat	repeat
A;	wait(S1);	wait(S2);
signal(S1);	B;	C;
wait(S3);	Signal(S2);	Signal(S3);
until False	until False	until False

ABCABCABC...

例 3-1 : $c=3$

Pi	Pj
repeat $c=c*2$ until False	repeat $c=c+1$; until False

4、6、7、8

例 3-2 : $c=3$ 、 $S=1$

Pi	Pj
repeat wait(S); $c=c*2$; signal(S); until False	repeat wait(S); $c=c+1$; signal(S); until False

7、8

例 4 : $S1=1$ 、 $S2=0$

Pi	Pj	Pk
repeat wait(S1); A; signal(S2); until False	repeat wait(S2); B; signal(S1); until False	repeat wait(S1); C; Signal(S1); until False

CAB、ABC

Semaphore 的『誤用』所造成的問題

⇒ 違反互斥、或形成 Deadlock

例 1 :

Pi	Pj
signal(S) CS wait(S) RS	signal(S) CS wait(S) RS

⇒ 違反互斥

例 2 :

Pi	Pj
wait(S) CS signal(S) RS	wait(S) CS signal(S) RS

⇒ 形成 Deadlock

例 3 :

Pi	Pj
T1 : wait(S1)	T2 : wait(S2)
T4 : wait(S2)	T4 : wait(S1)
...	...
signal(S1)	signal(S2)
signal(S2)	signal(S1)
...	...

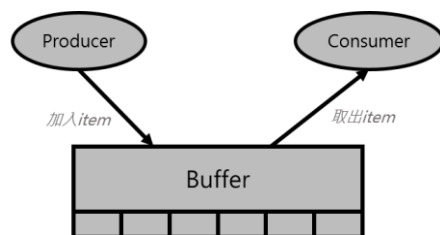
⇒ 可能形成 Deadlock，如果 Pi、Pj 依 T1~T4 交錯執行

著名的同步問題

Producer-Consumer Problem(生產者-消費者問題)

一、描述： Producer：此 Process 專門產生訊息，供別人使用

Consumer：此 Process 專門消耗別人產生的成果
在”Shared Memory”溝通方式下：



細分為 2 類型問題

1. Bounded Buffer：

同步條件

- (1) 當 Buffer 滿，則 Producer 被迫等待
- (2) 當 Buffer 空，則 Consumer 被迫等待

2. Unbounded Buffer：

- (1) Producer 無需等待
- (2) 當 Buffer 空，則 Consumer 被迫等待

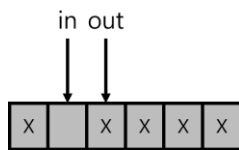
二、[演算法 1]：只能利用 n-1 格

1. 共享變數宣告如下：

- (1) Buffer[0: -1] of items;
- (2) In, out: int = 0;

2. 程式：

Producer	Consumer
<pre>repeat produce an item in nextp; while ((in+1)%n == out) do no-op; Buffer[in] = nextp; in = (in+1)%n; ... until False</pre>	<pre>repeat while (in==out) do no-op; nextc = Buffer[out]; out = (out+1)%n; ... consume the item in nextc; ... until False</pre>



此時 P 無法再加 item，因為 $(in+1)\%n == out$ ，即 Buffer 滿，故最多只能利用 $n-1$ 格

三、[演算法 2]：可充分利用 n 格

1. 共享變數宣告如下：

- (1) Buffer[0: -1] of items;
- (2) In, out: int = 0;
- (3) Count: int = 0;

2. 程式：

Producer	Consumer
<pre>repeat produce an item in nextp; while (Count==n) do no-op; //Buffer 滿 Buffer[in] = nextp; in = (in+1)%n; Count = Count+1; ... until False</pre>	<pre>repeat while (Count==0) do no-op; nextc = Buffer[out]; out = (out+1)%n; Count = Count-1; ... consume the item in nextc; ... until False</pre>

但是[演算法 2]Count 值，有可能會 Race Condition，因此並不完全正確

四、用號誌解生產者消費者問題

1. 共享變數宣告如下：

- (1) empty: semaphore= n ; //代表 Buffer 內空格數，若空格為 0，表滿了
- (2) full: semaphore=0; //代表 Buffer 中，填入 item 之格數，若為 0，表 Buffer 空
- (3) mutex: semaphore=1; //對 Buffer, in, out, Count 作互斥控制防止 Race Condition

思考哲學：1.滿足同步條件之方法變數(*empty*=卡生產者：*full*=卡消費者)
 2.互斥控制防止 Race Condition 之號誌變數(*mutex*=*Count* 之互斥控制)

2. 程式：

Producer	Consumer
repeat produce an item in nextp; wait(empty); //若無空格，則 P 被迫等待 wait(mutex); add nextp into Buffer; signal(mutex); signal(full); //填入 item 之格數被加 1 //maybe 拯救 Count until False	repeat wait(full); wait(mutex); remove item from Buffer in nextc; signal(mutex); signal(empty); consume the nextc; until False

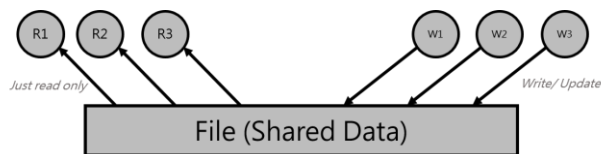
若是以下狀況，則可能 DL

Producer	Consumer
repeat wait(mutex); wait(empty); ... signal(full); signal(mutex); until False	repeat wait(mutex); wait(full); ... signal(empty); signal(mutex); until False

『重點』：先測同步、再測互斥

Reader-Writer Problem(讀者-寫者問題)

一、描述：



基本的同步條件：

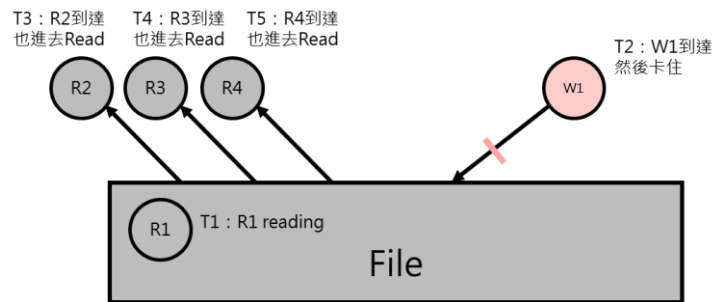
1. Reader、Writer 要互斥
2. Writer、Writer 要互斥

二、此外，此問題再細分為 2 型：

1. First Reader/Writer Problem：對 Reader 有利、對 Writer 不利；Writer 有可能 Starvation
2. Second Reader/Writer Problem：對 Writer 有利、對 Reader 不利；Reader 有可能 Starvation

First Reader/Writer Problem

一、何謂對 Reader 有利、對 Writer 不利：



源源不絕的 Reader 到達，則 W1 可能 Starvation

二、程式

1. 共享變數宣告如下：

- (1) wrt: semaphore=1; //提供 R/W、W/W 互斥控制，兼作對 Writer 不利之控制
- (2) readcnt: int=0; //統計目前 Reader 個數：Reader 到達則 readcnt+1；reader 離開，則 readcnt-1
- (3) mutex: semaphore=1; //對 readcnt 作互斥控制，防止 Race Condition

2. 程式：

Writer	Reader(上半部為進入、下半部為離開)
...	...
wait(wrt);	wait(mutex);
執行寫入工作;	readcnt=readcnt+1;
signal(wrt);	if(readcnt==1) then wait(wrt); //註 1
...	signal(mutex);
	執行讀取工作;
	wait(mutex);
	readcnt=readcnt-1; //註 2
	if(readcnt==0) then signal(wrt);
	signal(mutex);

註 1：成立，則表示目前你是第一個 Reader，負責偵測有無 Writer 在？

若有，則卡位；若無，則通過、也順便卡住 Writer

註 2：Reader 離開、readcnt-1，可解除 R/W

例：若目前 W1 已在寫入，則

1. R1 到達、R1 會被卡在__、此時 readcnt=__？
2. R2 到達、R2 會被卡在__、此時 readcnt=__？
3. R3 到達、R3 會被卡在__、此時 readcnt=__？

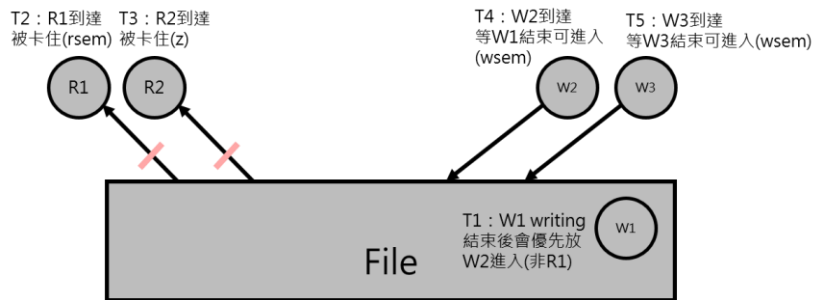
1. wrt、1
2. mutex、1
3. mutex、1

[考法]：

1. 程式
2. 說明程式為何對 Reader 有利、對 Writer 不利？
3. 上述例題

Second Reader/Writer Problem

一、何為對 Writer 有利、對 Reader 不利？



只要 Writer 離開時發現，尚有 waiting 的 Writer，就會優先放 Writer 進入，故 Reader 可能 Starvation

二、共享變數宣告如下：

1. readcnt: int=0; //統計 Reader 個數
2. wrtcnt: int=0; //統計 Writer 個數
3. x: semaphore=1; //作 readcnt 的互斥控制，防止 Race Condition
4. y: semaphore=1; //作 wrtcnt 的互斥控制，防止 Race Condition
5. z: semaphore=1; //作為對 Reader 之入口控制(卡多一些關卡，讓 Reader slower)
6. rsem: semaphore=1; //作為對 Reader 不利之控制
7. wsem: semaphore=1; //提供 R/W、W/W 互斥控制

三、程式：

Writer(上半部為進入、下半部為離開)	Reader(上半部為進入、下半部為離開)
<pre>wait(y); wrtcnt=wrtcnt+1; if(wrtcnt==1) then wait(rsem); signal(y); wait(wsem); 執行寫入工作; wait(y); wrtcnt=wrtcnt-1; if(wrtcnt==0) then signal(rsem); signal(wsem); signal(y)</pre>	<pre>wait(z); wait(rsem); wait(x); readcnt=readcnt+1; if(readcut==1) then wait(wsem); signal(x); signal(rsem); signal(z); 執行讀取工作; wait(x); readcnt=readcnt-1; if(readcnt==0) then signal(wsem); signal(x);</pre>

The Sleeping Barbers Problem(理髮型睡覺之問題)

一、描述：1 個理髮師，1 張理髮椅，n 張等待椅

- 客人行為：
- 1.等待椅滿(n 個等待的客人)，則不會進入店內
 - 2.等待椅未坐滿，入店內且坐在等待椅上
通知/喚醒理髮師
如果理髮師在忙，則客人睡覺；直到理髮師叫他起來理髮
理完離開
- 理髮師行為：
- 1.如果沒有客人，則理髮師睡覺；直到有客人叫醒/通知他
 - 2.叫醒客人剪髮
 - 3.剪完之後，若無客人則回到 1.
 - 4.剪完之後，若有客人則回到 2.

二、共享變數宣告如下：

1. Customer: semaphore=0; //用來卡住理髮師，if 無客人
2. Barber: semaphore=0; //用來卡住客人，if 理髮師忙碌中
3. Waiting: int=0 //坐在等待椅上的客人數目：客人入店坐上則+1；
理髮師叫客人起來理髮-1
4. Mutex: semaphore=1; //防止 waiting 值 Race Condition

三、程式

Barber	Customer
<pre>repeat wait(Customer); //若無客人則卡住 wait(mutex); waiting=waiting-1; signal(barber); signal(mutex); 剪客人頭髮(); until False</pre>	<pre>wait(mutex); if(waiting<n) //等待椅是否有空位 { waiting=waiting+1; signal(Customer); //叫醒/通知理髮師 signal(mutex); wait(Barbere); //客人卡住等理髮師被剪髮(); } else signal(mutex);</pre>

Note：客人沒有 repeat...until False，只有理髮師有

The Dining Philosophers Problem (哲學家晚餐問題)

一、描述：兩兩之間有一根筷子(chopstick)，哲學家若 hungry，他必須要能有同一時間取得左右兩根筷子，才可 eating。吃完後，放下左右兩筷，進入 Thinking mode

Note：

- 1.吃中餐：奇偶數位哲學家皆可
- 2.吃西餐：僅偶數位哲學家才可(刀、叉一副)

二、共享變數宣告如下：

Chopstick[0: 4] of semaphore; //初值皆為 1，對 5 根筷子作互斥控制

三、i 號(i: 0~4)哲學家：Pi 之程式

Pi
repeat hungry now; wait(chopstick[i]); wait(chopstick[(i+1)%5]); eating now; signal(chopstick[i]); signal(chopstick[(i+1)%5]); thinking now; until False

四、此 Solution 有誤(不完美)，可能導至 Deadlock

Ex：若每位哲學家依序取得左筷，則每位哲學家皆無法取得右筷，形成 circular waiting：皆卡於 wait(chopstick[(i+1)%5]); => 形成 Deadlock

五、解法：

[法一]：最多允許 4 位哲學家上桌 => Deadlock[定理]：m=5; Max=2

1. $1 \leq \text{Max}_i \leq m$ ，成立

2. $\sum \text{MAX}_i \leq n+m \Rightarrow 2n < n+5 \Rightarrow n < 5$ ，最多 4 位

保證 Deadlock free，可額外加入 No: semaphore，入口控制

Pi
wait(No); repeat hungry now; wait(chopstick[i]); wait(chopstick[(i+1)%5]); eating now; signal(chopstick[i]); signal(chopstick[(i+1)%5]); thinking now; until False signal(No);

[法二]：規定：除非哲學家可同時取得左、右 2 手子，才準許持有筷子，否則不得持有任何筷子 => 破除”Hold & Wait”條件

[法三]：規定：相鄰哲學家之取筷順序不同，創造”Asymmetric 非對稱”

Ex：奇數號：先取左、再取右 => 破除”circular waiting”條件

偶數號：先取右、再取左

Semaphore 種類

[分類一]：號誌值域作為區分

Binary semaphore(二元) vs Counting semaphore(計數)

一、Binary semaphore

Def：semaphore 之值只有 0 與 1 兩種(CS Design 正常使用下)

s: binary semaphore=1
wait(s): while(s<=0) do no-op; s=s-1;
signal(s): s=s+1;

二、Counting semaphore

1. Def：semaphore 值不限於 0 與 1，可以為負值，且若值為 1N，可知道(統計出)有 N 個 process 卡在 wait 中

2. 請用 Binary semaphore 定義出 Counting semaphore

(1) 共享變數宣告如下：

i. c: int; //代表 Counting semaphore 號誌值

ii. S1: Binary semaphore=1; //對 c 作互斥控制，防止 c 值 Race Condition

iii. S2: Binary semaphore=0; //強迫 process 暫停之用，當 c 值<0

(2) 程式

wait(s): wait(S1); c=c-1; if(c<0) then { signal(S1); wait(S2); } else signal(S1);	signal(s): wait(S1); c=c+1; if(c<=0) then signal(S2); signal(S1);
---	--

[分類二]：是否將用 Busy-Waiting(spinlock)來定義 semaphore？

Spinlock vs Non-Busy-Waiting semaphore

一、Spinlock(Busy-Waiting) semaphore

Def：令 s 為 semaphore 變數：

wait(s): while(s<=0) then do no-op; s=s-1;	signal(s): s=s+1;
--	----------------------

缺點：等待中的 Process 會與其他 Process 競爭 CPU，將搶到的 CPU Time 用在毫無實質進展的迴圈測試上，因此，若 Process 要等待長時間才能 Exits 迴圈，則此舉非常浪費 CPU Time

優點：若 Process 卡在 loop 的時間很短(i.e.,小於 Context Switching Time)，則 Spinlock 十分有利

二、Non-Busy-Waiting semaphore

Def：semaphore type 定義如下：

```
Struct semaphore
{
    int valuve; //號誌值
    Queue Q; //FIFO Queue
}
```

令 s 為 semaphore 變數：

<pre>wait(s): s.value=s.value-1; if(s.value<0) then { add process P to s.Q; Block(P); //P 之狀態改為 Block } State }</pre>	<pre>signal(s): s.value=s.value+1; if(s.value<=0) then { remove a process P from s.Q; wakeup(P); //將 P 改為 Ready } State; }</pre>
--	---

製作 semaphore

1. Disable Interrupt
2. SW solution / HW Instruction support

一、何謂製作 semaphore？

即是如何保證 semaphore 值不會 Race Condition。或：
如何確保 wait 及 signal 是”atomic” operation

二、4 個[演算法]或[作法]

製作方式\定義	Non-Busy-Waiting'semaphore	Spinlock semaphore
Disable Interrupt	[演算法 1]	[演算法 3]
CS Design 基礎 SW solution / HW Instruction Support	[演算法 2]	[演算法 4]

[演算法 1]

signal(s): Disable Interrupt s.value=s.value+1; if(s.value<0) then { remove a process from s.Q wakeup(P); } Enable Interrupt	wait(s) Disable Interrupt s.value=s.value-1; if(s.value<0) then { Enable Interrupt add process P into s.Q; block(P); } else Enable Interrupt
---	---

[演算法 2]：將[演算法 1]的 Disable Interrupt 換成 Entry Section；Enable Interrupt 換成 Exit Section。而 Entry/Exit Section 之控制碼，另外找個地方寫出來，且取決於題目要求使用 SW Solution(*Bakery 演算法*)或 HW Instruction Support(*Test & Set / SWAP 的演算法 2*)

signal(s): Entry Section s.value=s.value+1; if(s.value<0) then { remove a process from s.Q wakeup(P); } Exit Section	wait(s) Entry Section s.value=s.value-1; if(s.value<0) then { Exit Section add process P into s.Q; block(P); } else Exit Section
---	---

Test & Set / SWAP

區域變數： Boolean: key int: j; repeat waiting[i]=True; key==True; while(waiting[i] and key) key=Test-and-Set(Lock); waiting[i]=False; CS; j=(i+1)%n; if(j==i) then Lock=False; else waiting[j]=False; RS; until False	區域變數： Boolean: key int: j; repeat waiting[i]=True; key==True; while(waiting[i] and key) SWAP(Lock, key); waiting[i]=False; CS; j=(i+1)%n; if(j==i) then Lock=False; else waiting[j]=False; RS; until False
---	--

[演算法 3]

signal(s): Disable Interrupt s=s+1; Enable Interrupt	wait(s) Disable Interrupt while(s<=0) do { Enable Interrupt no-op; Disable Interrupt } s=s-1; Enable Interrupt
--	--

[演算法 4]

將[演算法 1]的 Disable Interrupt 換成 Entry Section；Enable Interrupt 換成 Exit Section。而 Entry/Exit Section 之控制碼，另外找個地方寫出來，且取決於題目要求使用 SW Solution(*Bakery 演算法*)或 HW Instruction Support(*Test & Set / SWAP 的演算法 2*)

signal(s): Entry Section s=s+1; Exit Section	wait(s) Entry Section while(s<=0) do { Exit Section no-op; Entry Section } s=s-1; Exit Section
--	--

Test & Set / SWAP

區域變數： Boolean: key int: j; repeat waiting[i]=True; key==True; while(waiting[i] and key) key=Test-and-Set(Lock); waiting[i]=False; CS; j=(i+1)%n; if(j==i) then Lock=False; else waiting[j]=False; RS; until False	區域變數： Boolean: key int: j; repeat waiting[i]=True; key==True; while(waiting[i] and key) SWAP(Lock, key); waiting[i]=False; CS; j=(i+1)%n; if(j==i) then Lock=False; else waiting[j]=False; RS; until False
---	--

Busy-Waiting 是否可以完全避免之？(avoid all together?)

⇒ 無法完全避免之：

以 semaphore 為例：

定義層次	可是製作層次
Busy-Waiting semaphore ⇒ Non-Busy Waiting semaphore(註)	使用 CS Design 手法，其 Entry Section 就有 Busy Waiting skill，因此沒有完全避免掉

註：使用 Disable Interrupt：沒有 Busy-waiting，但此方法風險過高，不適用 Multiprocessors，故不列入考慮

Monitor

定義、組成、特性(優點)

Condition Type 變數使用

解同步問題

Conditional Monitor 及其應用

種類(3 種)

用 semaphore 製作 Monitor

Monitor

一、Def：Monitor 是一個用來解決同步問題的高階結構，是一種 ADT(Abstract Data Type)，Monitor 之定義，主要有 3 個組成：

1. 共享變數宣告區
2. 一個 local functions (or procedures)
3. Initialization area(初始區)

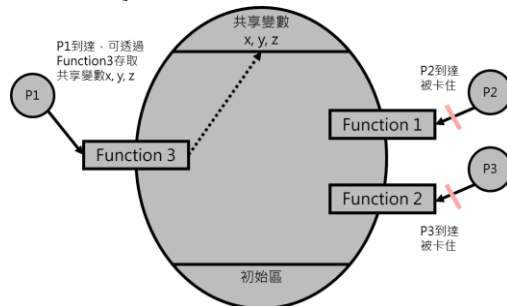
語法格式：

type [Monitor-Name] = Monitor Var [共享變數宣告];
procedure entry [funct1-Name(參數)] begin [Body]; end procedure entry [functx-Name(參數)] begin [Body]; end
begin [初始區]; end

二、特性：Monitor 本身已保證互斥(mutual exclusive)：即在『任何時間點，最多允許 1 個 process 在 monitor 內活動(active)』

白話：也就是說，在任何時間點，最多只允許一個 process 呼叫 monitor 的某一個 function 或 procedure 執行中，不可以有多個 process，同時呼叫 monitor 的 functions 執行。

Enable Queue：P1、P2、P3...



此一互斥性質帶來何種好處？

因為共享變數區之共享變數，只能被 monitor 的 local function 直接存取，外界不可直接存取，外界(process)只能透過呼叫 monitor 的 local function 來存取共享變數，而因為 Monitor 保障互斥特性，所以保障了共享變數不會發生 Race Condition，代表 programmer 無需煩惱 Race Condition 問題(不用撰寫額外的 Code、或使用 mutex semaphore)，只需專心解同步問題即可，此點優於 semaphore

例：semaphore 比 Monitor 容易使用，when solving synchronization problem(T/F)？

False，Monitor 較易使用，因為不用煩惱 Race Condition 與 Deadlock 的問題

Condition Type

Def：Condition 型別是用在 Monitor 中，提供給 Programmer 解決同步問題之用，令 x 是 Condition Type 變數，在 x 上提供 2 個 operations：x.wait 及 x.signal 定義如下：

1. x.wait：執行此運作的 process 會被 Blocked、且置入 Montior 內，x 所屬的 waiting Queue 中(預設是 FIFO Queue)
2. x.signal：如果先前有 process 卡在 waiting Queue 中，則此運作會從此 waiting Queue，移走一個 process，且恢復(resume)其執行，否則無任何作用

使用 Monitor 解決哲學家晚餐問題

一、先定義所需的 Monitor ADT

type Dining-ph = Monitor Var [共享變數宣告]; state[0:4] of {thinking, hungry, eating} self [0:4] of condition;
procedure entry pickup(i:0:4) //哲學家編號 begin state[i]=hungry; test(i); if(state[i]!=eating) then self[i].wait; end; procedure entry test (k:0:4) begin if (state[(k+4)%5]!=eating and state[k]==hungry and state[(k+1)%5]!=eating) then { state[k]=eating; self[k].signal; } end; procedure entry putdown (i:0:4) begin state[i]=thinking; test[(i+4)%5]; test[(i+1)%5]; end;
begin for(i=0;i<=4;i++) state[i]=thinking; end

二、使用方式

共享變數宣告如下：

dp: Dining-ph //變數名稱：Monitor type
Pi(i 號哲學家) repeat hungry now; dp.pickup(i); //在 monitor 內(not active) eating; //不在 monitor 內 dp.putdown(i); //在 monitor 內(not active) thinking until False

Conditional Monitor

- 一、緣由：Conditional 變數，ex：x，所付屬的 waiting Queue，一般是 FIFO Queue，(甚至 Monitor 的 Entry Queue，也是 FIFO, in general)；可是有時候我們需要”priority Queue”，優先移除高優先權的 process，恢復執行、或讓他進入 Monitor 內 active
- 二、語法改變：x.wait(c) //c 代表此 Process 的 priority information

用 Conditional Monitor 解決問題

例 1：使用 Monitor，解決互斥資源的配置，規定 Process ID 較小者，優先權較高、優先取得資源

解決哲學： 非優先權需求(互斥) => Monitor 定義去處理
 優先權需求(ID) =>只要告知老師說，你用的是 priority queue 的 monitor 即可

Ans：

1. 先定義 Monitor：

<pre>type ResourceAllocator = Monitor Var Busy: Boolean; //代表資源配置出去與否 x: condition;</pre>
<pre>Procedure entry Apply(pid:int) begin if(Busy) then x.wait(pid); Busy=True; end;</pre>
<pre>Procedure entry Release() begin Busy=False; x.signal; end;</pre>
<pre>begin Busy=False; end;</pre>

2. 使用方法：

共享變數宣告如下：

<pre>RA: ResourceAllocator; Pi(i 代表 process ID) PA.Apply(i); 使用資源 PA.Release();</pre>

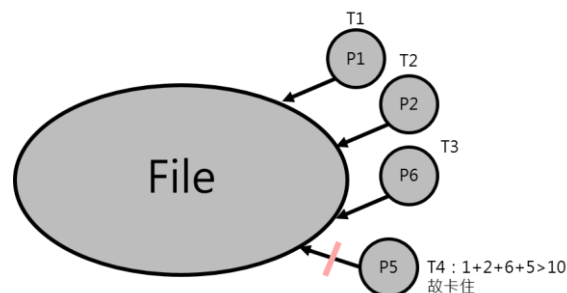
3. 此 Monitor 的 x condition 變數之 waiting queue 及 Monitor 的 entry queue 是 priority queue，且 Process ID 小者，優先權高、優先移出

例 2(P6-56-7)：有一個 file 可被多個 process 使用，每一個 process 有 unique priority No.，存取 file 需滿足下列限制：

(1) 所有正在存取此 file 的 process 之 priority No.加總，需 $<n$

(2) Process priority No 較小的優先權較高

設計此 Monitor：



Ans：

1. 先定義 Monitor：

<pre>Type FileAccess = Monitor Var sum:int; x:condition;</pre>
<pre>Procedure entry Access(i:priority No.) begin when(sum+i)>=n do x.wait(i); sum=sum+i; end;</pre>
<pre>Procedure entry Leave(i:priority No.) begin sum=sum-i; x.signal; end;</pre>
<pre>begin sum=0; end;</pre>

2. 使用方式：

共享變數宣告如下：

<pre>FA:File Access; Pi(i: process priority No.) FA.Access(i); 使用 File FA.Leave(i);</pre>

3. 此 Monitor 的 x condition 變數之 waiting queue 及 Monitor 的 entry queue 是 priority queue，且 Process ID 小者，優先權高、優先移出

例 3(P6-72-24)：有 3 部 printers 被 process 使用，且規定 process ID 較小者，優先權較高

1. 先定義 Monitor

<pre>type Allocator = Monitor Var P[0:2] of Boolean; x: condition;</pre>
--

<pre> int Acquire(i: process ID) { if(P[0] & P[1] & P[2]) then x.wait(i); if(!P[0]) then y=0; else if(!P[1]) then y=1; else y=2; P[x]=True; return y; } void Release(y:printer No.) { P[y]=False; x.signal; } begin for i=0 to 2 do P[i]=False; end;</pre>
--

2. 使用方法：

共享變數宣告如下：

<pre> PA: Allocator Pi(i: process ID) pno:printer No. pno=PA.Acquire(i); 使用 pno 號之列表機 PA.Release(pno);</pre>
--

3. 此 Monitor 的 x condition 變數之 waiting queue 及 Monitor 的 entry queue 是 priority queue，且 Process ID 小者，優先權高、優先移出

使用 Monitor 定義 semaphore

<pre> Type semaphore = Monitor Var value:int; x.condition;</pre>
<pre> Procedure entry wait() begin value=value-1; if(value<0) then x.wait; end;</pre> <pre> Procedure entry signal(); begin value=value+1; x.signal; end;</pre>
<pre> begin value=1; end;</pre>

Monitor 的種類(3 種)

- 一、區分角度(緣由)：假設 Process Q 目前卡住，x.condition 變數之 waiting Queue(因為 Q 先前執行了，x.wait)，目前 process P is active in the monitor。當 P 執行了 x.signal，P 會將 Q 救出，resume Q 之執行，此時代表 P 與 Q 同時 active in the monitor；但是這會違反 mutual exclusive 性質，因此只能讓 P 或 Q 其中一個 active，選 P 或 Q 呢？

這也就是區分 Monitor type 的角度。

二、3 種：

[Type 1]：P 等 Q，直到 Q 完成 function 或再度被 Blocked，又稱為 Hoare Monitor(Hoare 是 Monitor 的第一發明者)

[Type 2]：Q 等 P 直到 P 完成或 P 被 Blocked

[Type 3]：P 離開 Monitor，然後讓 Q 恢復執行，直到完成或再度被 Blocked，P 才再度進入 Monitor 執行。(Concurrent C/ PASCAL 等程式語言採用)

三、分析

1. [Type 2]：效能最差

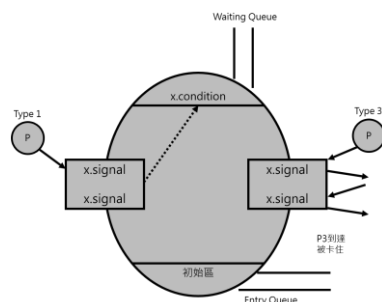
缺點：不保證 Q 一定可被恢復執行，因為允許 P 繼續往下執行的過程中，P 有可能改變了讓 Q 可以恢復執行的同步條件執，使得 Q 仍被卡住

優點(但稱不上是優點)：較為自然...

2. [Type 3]：

優點：保證 Q 一定可以立即被 Resume 執行

缺點：威力(效果)不若[Type 1]，因為”一進一出”Monitor 期間，頂多救一個 Q，然而[Type 3]可救多個 Q



3. [Type 1]：Hoare Monitor

- 優點：
1. 保證 Q 一定可以恢復執行
 2. 相對於[Type 3]更 powerful

四、[恐]現在分 2 類：

1. Signal-and-Wait：[Type 1] Monitor([Type 3])
2. Signal-and-Continue：[Type 2] Monitor

使用 semaphore 製作 Monitor

一、製作 Monitor 必須滿足 3 類需求：

1. 保證”互斥”之性質：1 最多一個 Process active in the monitor
2. [Hoare] Monitor 性質
3. Condition 變數(ex : x)的 x.wait 與 x.signal 製作

二、共享變數宣告如下：

```
1. mutex: semaphore=1;
2. next: semaphore=1;    //用以卡住 P, if P 執行 x.signal
3. next-count: int=0;     //統計 P 之個數
4. x-sem: semaphore=0;    //用以卡住 Q, if Q 執行 x.wait
5. x-count: int=0;        //統計 Q 之個數
```

三、製作碼

1. 確保互斥方面：在 Monitor 的每個 function 的 Body 之前、及之後，加入一些額外的控制碼如下：

```
Procedure entry pickup(i)
begin
    wait(mutex);
    Body
    if(next-count>P) then
        有救命恩人 P 存在
        signal (next);
    else
        signal(mutex);    //若沒 P，才放外人進入
end;
```

2. X.wait 之製作碼：

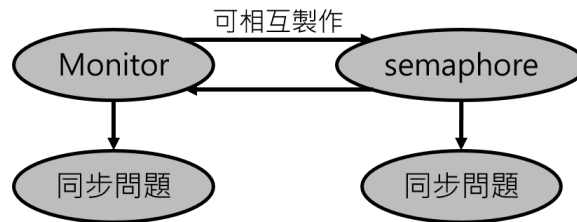
```
x-count=x-count+1;    //Q 個數+1
if (next-count>0) then
    signal(next);
else signal(mutex);
wait(x-sem);           //Q 自己卡住
x-count=x-count-1;    //當 Q 被救，Q 個數少 1
//(有人卡才有作用，否則無作用)
```

3. X.signal 之製作碼：

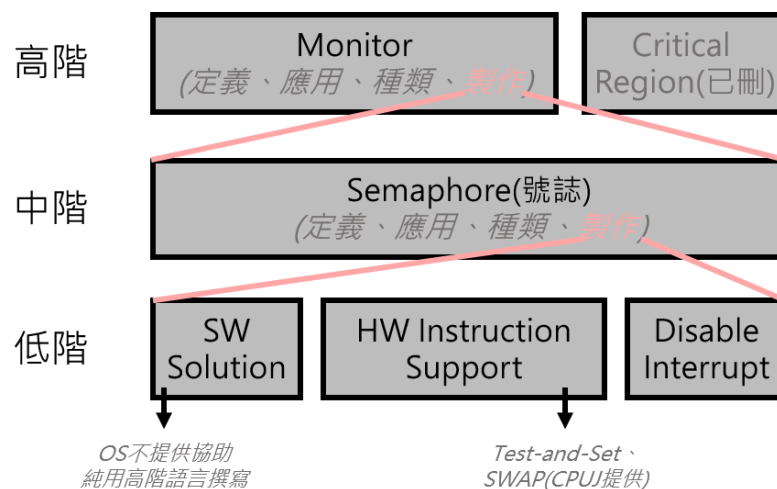
```
if(x-count>0) then
{
    next-count=next-count+1;    //P 個數加 1
    signal(x-sem);              //救 Q
    wait(next);                 //P 卡住[Hoare]
    next-count=next-count-1;    //當 P 被救，P 個數少 1
}
```

[恐]證明 Monitor 與 semaphore，解決同步問題的能力是相同的
(identical/equivalent)

Ans : Monitor 與 semaphore 可以相互製作，因此兩者解決同步問題之能力相同
(再將 M/S 之製作碼寫出)



Monitor 定義 semaphore	semaphore 定義 Monitor
<pre> Type semaphore = Monitor Var value:int; x.condition; Procedure entry wait() begin value=value-1; if(value<0) then x.wait; end; Procedure entry signal(); begin value=value+1; x.signal; end; begin value=1; end; </pre>	<pre> 1. mutex: semaphore=1; 2. next: semaphore=1; //用以卡住 P, if P 執行 x.signal 3. next-count: int=0; //統計 P 之個數 4. x-sem: semaphore=0; //用以卡住 Q, if Q 執行 x.wait 5. x-count: int=0; //統計 Q 之個數 Procedure entry pickup(i) begin wait(mutex); Body if(next-count>P) then 有救命恩人 P 存在 signal(next); else signal(mutex); //若沒 P，才放外人進入 end; end; x.wait: x-count=x-count+1; //Q 個數+1 if (next-count>0) then signal(next); else signal(mutex); wait(x-sem); //Q 自己卡住 x-count=x-count-1; //當 Q 被救，Q 個數少 1 //(有人卡才有作用，否則無作用) x.signal: if(x-count>0) then { next-count=next-count+1; //P 個數加 1 signal(x-sem); //救 Q wait(next); //P 卡住[Hoare] next-count=next-count-1; //當 P 被救，P 個數少 1 } </pre>



Message Passing 溝通方式

Direct vs Indirect Communication

解 Producer-Consumer Problem

同步意義如何呈現？

1. Link Capacity
2. Blocking/Non-blocking

Send/Receive 組合

Rendezvous 同步模式

例外狀況(exception handling)

Direct 與 Indirect Communication

一、Direct(直接)溝通

1. Symmetric：雙方需互相指令對方的 process ID，才能建立 Communication Link，OS 提供 send、receive 之 system call

例：

P 送	Q 收
...	...
send(Q, message)	recieve(P, mes)
...	...

2. Asymmetric：只有送方需指名收方 ID，但收方無需指名送方，即從任何 Process 皆可以(ex：Email)

例：

P 送	Q 收
...	...
send(Q, message)	recieve(id, mes)
...	...

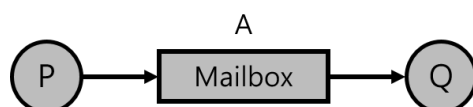
Q 收到後，會將送方 ID 記錄在 id 變數中

二、Indirect(間接)溝通

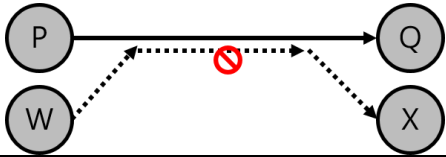
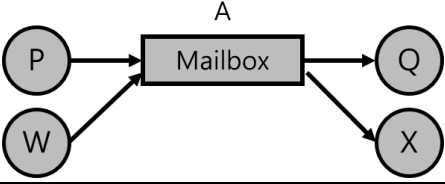
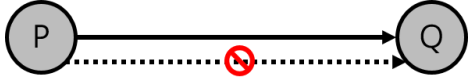
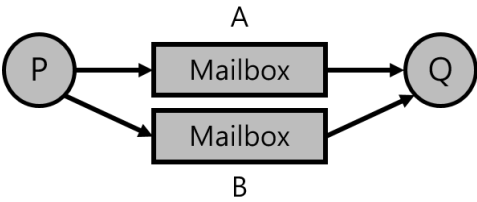
Def：收、送雙方是透過 Shared Mailbox(共享郵箱)才能建立 Communication Link

例：

P 送	Q 收
...	...
send(A, message)	recieve(A, mes)
...	...



三、比較表

Direct(symmetic)	Indirect
收、送雙方需相互指合對方 ID 才能建立 Communication Link	雙方透過 Shared Mailbox 才能健康 Communication Link
Communication Link 是專屬於雙方，不能與他人共享	Communication Link 可多組共享
	
溝通雙方最多只能有一條 Communication Link，不可多條	溝通雙方可同時存在多條 Communication Link，每一條皆需有 Shared Mailbox
	

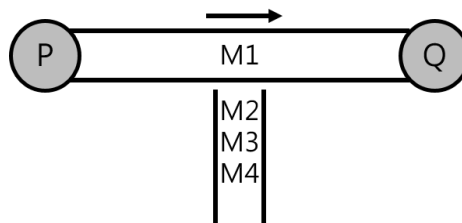
用 Message Passing 解 Producer-Consumer Problem

Producer 程式	Consumer 程式
<pre>repeat produce an item in nextp; send(Consumer, nextp); until False</pre>	<pre>repeat Recieve(Producer, nextc); consume the item in nextc; until False</pre>

同步意義之呈現

[法一]Link Capacity：假若收方一律是：“若未收到訊息則暫停，直到收到訊息後，才往下執行，而對於送方，就看 Link Capacity”

⇒ 在每條 Communication Link 皆附有一個 message Queue，用以保存除正在傳輸中的訊息以外之其他訊息。而 Queue size = Link Capacity



Link Capacity 有下列 3 種 size :

一、Zero capacity :

1. 送方送出訊息後則暫停，直到收方收到訊息後，送方才可往下執行。
2. 收方一律是”若未收到訊息則暫停，直到收到訊息後，才往下執行，而對於送方，就看 Link Capacity”

例：

P 送	Q 收
...	...
send(Q, mes)	recieve(P, m)
receive(Q, “ACK”)	send(P, “ACK”)
...	...

二、Bounded capacity : 當 Queue 滿了，則送方被迫暫停

三、Unbounded capacity : 送方無需被迫暫停

[法二]4 條指令之組合來表現不同同步模式

1. Blocking-send : 要等收方收到才繼續
2. Non-blocking-send : 不用等收方收到，可直接繼續
3. Blocking-receive : 有收到東西才繼續
4. Non-blocking-receive : 沒有收到東西也繼續

例 1 : rendezvous 模式 ?

P	Q
Blocking send(Q, mes);	Blocking receive(P, mes);

例 2 : 收方程式如下 :

```
Non-blocking-receive(A, mes);  
if(mes==NULL) then  
{  
    Blocking-receive(B, mes);  
    Blocking-receive(A, mes);  
}  
else  
    Blocking-receive(B, mes);
```

代表：

1. 從 A 或 B 收到訊息後，即可往下
2. 一定要從 A 及 收到訊息後，且要先 A 後 B，才可往下
3. 同 2，但 AB 順序無所謂

Exception Handling(例外情況)

1. 在 rendezvous 模式下，若收或送的一方已死亡，但另一方不知道，則另一方會永久停滯
 $P(X) \Rightarrow Q$ 或 $P \Rightarrow Q(X)$
解法 1：OS 通知另一方取消動作
解法 2：OS 也終止另一方
2. 在 Direct symmetric 下，當 new process 或 resume process，未通報其他 process，且 new process 也不知道其他人 ID，則無法溝通
3. Message 傳輸過程中，有可能 lost \Rightarrow OS 負責偵測 message 是否 lost，若 lost，OS 通知送方重送 message \Rightarrow 偵測 lost 之方法[Network 課程(例：Time-out 方式)]

比喻(校園生活)：

1. 某通識課需要兩兩一組才能修：A 邀請 B 一起修課，但後來 A 退選、或 B 退選，另一方(B/A)可能的解決辦法：1.由老師通知，請他再邀請 or 被邀請、2.老師通知其退選課程
2. 新同學 or 邊緣人不認識班上同學、班上同學也不認識他們，則正常情況下將永遠無法溝通，因為永遠也不會有誰去認識誰
3. 同 1.，A 發出修課邀請給 B，但邀請可能丟失，故老師負責確認是否有正確發出邀請，若無的話，老師會請 A 再次重送邀請，若只是速度較慢，則老師需要能通知 B 同學去忽略重新發出之邀請