Chapter 10 Process Communication

- Process 的溝通方式

	Shared Memory	Message Passing
Def.	Process 之間透過對共享變數(shared variable)之存取達到溝通之目的	Process 溝通，遵循下列 steps: 1. 建立 Communication link 2. 訊息相互傳輸 3. 傳輸完後，釋放 link
OS是否需提供支援	No	Yes (e.g. link 之管理，例外處理, etc.)
Programmer 負擔	負擔較大 (因為必須提供對共享變數之互斥存取的機制，防止 race condition 發生)	無啥負擔

- 名詞解釋

Race Condition	Synchronization	Busy-waiting
在 Share memory 溝通方式下，若未對共享變數(shared variable)提供任何互斥存取控制機制時，則"共享變數之最終值會依 Process 之間交錯執行的順序不同而有所不同"，此一稱之	Process 執行過程中，因某些事情的發生，必須等待其他 process do something 之後才會繼續進行	Process 透過 loop 不斷測試執行達到強迫等待之效果 ● Process 被 wait, 直到條件為 False, 才離開 while, 網下執行

- Producer & Consumer

Def.	✓ Producer： 此 process 生產 info 供其他 Process 使用 ✓ Consumer: 此 process 消耗其他 process 的 info	
在 Share memory 的溝通下，再細分成兩類	Bounded-Buffer Producer / Consumer Problem ➢ 當 Buffer 滿 -> producer 被迫 wait ➢ 當 Buffer 空 -> consumer 被迫 wait	Unbounded-Buffer Producer / Consumer Problem ➢ Producer 無須 wait (Buffer 容量無限大) ➢ Consumer 仍需 wait, if Buffer 為空
	[Algorithm 1]	[Algorithm 2]
共享變數宣告	✓ Buffer : array [0..n-1] of item ✓ in, out : int = 0;(初值)	同 Algo. 1 外 ✓ count : int = 0; 記錄 Buffer 中的資料數
Code (Producer)	repeat *produce an item in nextp;*	repeat *produce an item in nextp;*

	while (in+1) mod n == out do no-op; Buffer[in] = nextp; In = (in+1) mod n; …… until False	while (count == n) do no-op; Buffer[in] = nextp; In = (in+1) % n; count = count + 1; …… until False
Code (Consumer)	repeat while (in == out) do no-op; out = (out+1) %n; nextp = Buffer[out]; …….. *Consume the item in nextc;* …….. until False	repeat while (count == 0) do no-op; out = (out+1)%n; nextp = Buffer[out]; count = count - 1; …….. *Consume the item in nextc;* until False
分析	最多只能用 n-1 格(in 所指那格不用)	此 Algo. 雖然正確，也充分利用到 n 格 但是若未對 count 此共享變數提供任何互斥機制，則 count 值會發生 Race Condition，導致不對之結果出現。

- 解決 Race Condition 問題的兩大策略
  - Disable interrupt
  - Critical Section Design

- Disable Interrupt

作法	Process 在對共享變數進行存取之前，先 Disable Interrupt 防止其他 Process 搶奪 CPU，直到敘述完成之後，才 Enable interrupt，如此，可保證對共享變數存取之敘述執行是 atomically executed，即執行過程不被 interrupt ● 即可防止 Race Condition	
例子	Producer *Disable interrupt* count = count+1; *Enable interrupt*	Consumer *Disable interrupt* Count = count-1; *Enable interrupt*
優點	1. Easy implementation 2. Uniprocessor 下使用	
缺點	不適用於 Multiprocessor 環境 因為必須 Disable All Processs 之中斷才有用，然而此舉會導致： 1. System Performance 變差(因為 message 太多) 2. 風險性太高	

	(96 交大) Multiprocessor 使 C.S Region 來解決 Race Condition 所以大多系統禁止此種方法。

- Critical Section Design

程式架構	C.S. Design 必須滿足 3 個性質		
repeat Entry Section C.S. Exit Section R.S. until False	Mutual Exclusion	Progress	Bounded waiting
	任何時間點，最多只允許一個 process 在他自己的 C.S.內活動，不可同時多個 Process 在各自的 C.S.活動	1. 不想進入 C.S.之 Processes(或在 R.S.中活動的 Process)，不能夠阻礙其他 process 進入 C.S.(or 不能夠參與進入 C.S.與否之決策) 2. 必須在有限時間內，自那些想進 C.S.之 process 中決定出哪個 process 可進入 C.S.(即 No Deadlock) [原文]: If no process is executing in its critical region and some processes wish to enter their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot postpone in infefintely.	自 process 提出申請進入 C.S.到它獲得進入的這段等待時間是有限的(即 No Starvation) ● n 個 process 皆想進入 C.S.，則任一 process 至多等 n-1 次後必可進入 C.S.

Note: (1) C.S.(Critical section) : Process 中對共享變數進行存取的敘述集合(or 程式碼片段)

(2) R.S (Remainder section) : 即 C.S.之外的區域

- Critical Section Design 的方法
  - Software Solution
  - HW instruction support
    - Test and Set
    - SWAP
  - Semaphore

- Monitor
- Critical Region

- Software Solutions

2 個 Process 之 C.S. Design			
	[Algorithm 1] (wrong)	[Algorithm 2] (wrong)	[Algorithm 3] (Right)
共享變數宣告	✓ turn : i or j only (初值無所謂) ✓ 權杖, 誰拿到誰就可進入 C.S.	✓ flag[i…j] of Boolean, 初值設為 False ✓ flag[i] = True : 表 Pi 有意願 False: 表 Pi 無意願	✓ turn : i or j ✓ flag[i…j] of Boolean, 初值設為 False
P_i 程式	repeat while (turn≠i) do no-op; C.S. turn = j; R.S. until False	repeat flag[i] = True; while(flag[j]) do no-op; C.S. Flag[i] = False; R.S. until False	repeat flag[i] = True; turn = j; while(flag[j] && turn == j) do no-op; C.S. Flag[i] = False; R.S. until False
P_j 程式	repeat while (turn≠j) do no-op; C.S. turn = i; R.S. until False	repeat flag[j] = True; while(flag[i]) do no-op; C.S. Flag[j] = False; R.S. until False	repeat flag[j] = True; turn = j; while(flag[i] && turn == i) do no-op; C.S. Flag[j] = False; R.S. until False
Mutual	OK	OK	OK

exclusion	因為 turn 不會同時為 i 且為 j，讓 Pi, Pj 同時進入 C.S.		若 Pi, Pj 皆想進入 C.S.(表 flag[i], flag[j] 分別設為 true) 當 Pi, Pj 執行到 while 測試時， 代表 Pi, Pj 已分別執行 turn = j 及 turn = i 因為 turn 值之設定只會為了 i or j 所以 Pi(or Pj)只有一個能進入 C.S.
Progress	違反 假設 Pi 在 R.S.內活動且目前 turn 為 i 若此時 Pj 想進入 C.S.，則無法進入 (因為 Pj 被 Pi 所阻擋，則 Pj 必須等到 Pi 進入 C.S.後，將 turn 改為 j，Pj 才可進入)	違反 ✓ 可能會發生 Deadlock T1: flag[i] = True; T2: flag[j] = True; T3: while(flag[j]) do no-op; -> Pi wait T4: while(flag[i]) do no-op; -> Pj wait	OK (i) 若 Pi 不想進入 C.S.(即 flag[i] ==false) 且 turn == i, 若此時 Pj 想進入 C.S.(因為 flag[i] ==false)，未被 Pi 阻礙 (ii) 若 Pi, Pj 皆想進入 C.S.,則在有限的時間內必可決定出 turn = i or turn = j, 而讓 Pi or Pj 進入 C.S. ✓ No Deadlock
Bounded Waiting	OK Pf: 令 Pi 已進入 C.S.，且 Pj 正在 wait 若 Pi 離開 C.S.，又企圖立刻再進入 C.S. 則 Pi 必定會將 turn = j, Pi 必定無法進入 C.S.，一定會讓 Pj 進入 所以，Pj 至多 wait 一次, No Starvation	OK	OK 假設 Pi 已經進入 C.S.，而 Pj 正在 wait 若 Pi 離開 C.S.之後又企圖立刻再進入 C.S. 則 Pi 必定 turn = j, 使 Pi 不得進入 Pj 進入 C.S., 所以 Pj 至多等一次 ✓ No Starvation
n 個 Processes (n＞2) 之 C.S. Design			

	[Algorithm 4](McGuire's Solution)	[Algorithm 5](Bakery's Algo.)
共享變數 宣告	✓ turn : 0..n-1; (權杖): 初值 no care ✓ flag[0..n-1] : of (idle, want-in, in-cs) idle : 表無意願進入 C.S. want-in : 表有意願進入 C.S. in-cs : Process 已在 C.S 中，其 flag 必為 in-cs but, flag == in-cs, 不代表一定能進入 C.S.(只代表優先權較 want-in 高)	● 觀念： ■ 客人要進店，須先取號碼牌 ■ 號碼最小，優先進入 ■ 萬一同號，則以客戶 ID 最小者優先進入 ✓ choosing : [0..n-1] of Boolean，初值為 False choosing[i] = True : 表 Pi 正在取號碼牌當中 False : 表初值 or Pi 已取得號碼牌 ✓ number : [0..n-1] of int，初值為 0 number[i] = 0 : 表 Pi 無意進入 C.S. > 0 : 表 Pi 有意進入 C.S. Note: ✓ MAX : 取最大值 ✓ (a, b) < (c, d) 成立條件為 (i) a < c 或 (ii) a == c 且 b < d
P_i 程式	repeat repeat flag[i] = want-in; j = turn; while (j ≠ i) do if (flag[i] ≠idle) then j = turn; else j = (j+1)%n; flag[i] = in-cs;	repeat choosing[i] == true; number[i] == MAX(number[0], … …, number[n-1]) + 1; choosing[i] == False; for j≠0 to (n-1) do { while choosing[j] do no-op; while(number[j]≠0 &&

<table>
<tr>
<td></td>
<td>
j = 0;
[while (j < n) and (j == I or flag[j] ≠in-cs) do
j = j+1;
until (j ≧ n) and (turn == I or flag[turn] == idle)

C.S.
j = (turn + 1)% n;
[while(flag[j] == idle) do
j = (j + 1) % n;
turn = j;
flag[i] == idle;
R.S.
until False
</td>
<td>
(number[j], j) ＜ (number[i], i)) do no-op;
}
C.S.
number[i] = 0;
R.S.
until False
</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td>
OK
Pi 可以進入 C.S.的條件為：
(i) 確保只有一個 Pi process 之 flag 為 in-cs 且
(ii) turn 值為 i 或 flag[turn] == idle

因此 由(i) 知，唯一性確保→互斥
</td>
<td>
OK
Case1: 若各 process 之 number 值皆為 unique，則具有最小 number 值可進入 C.S.
● Min number 只有一個，唯一性，互斥成立

Case2: 若各 process 之 number 值不 unique，則以 Process ID 最小者進入
● Process ID 唯一
</td>
</tr>
<tr>
<td>Progress</td>
<td>
OK
(i) 若 Pi 不想進入 C.S.(即 flag[i] == idle)，且 turn 值為 i，
則若 Pj(j≠i)想進入 C.S.，則 Pj 必通過 until (j≧n) and (turn == j or flag[turn] == idle)，測試而進入 C.S.，不會被 Pi 阻礙
</td>
<td>
OK
(i) 若 Pi 不想進入 C.S.，則 number[i] == 0，
若 Pj(j≠i)想進入 C.S.，則絕不會被 Pi 阻礙
</td>
</tr>
</table>

	(ii) 在有限的時間內，必可決定出一個 $process_i$，其保證只有他是 in-cs 且(turn 為 i 或 flag[turn]==idle)，得以進入 C.S. ✓ No Deadlock	(ii) 在有限的時間內，必可選出具最小 number 值或最小 Process ID 值者，進入 C.S.
Bounded Waiting	OK 若 P0 ~ Pn-1 等 n 個皆想進入 C.S.，令 turn 值目前為 i，則 Pi 可先進入 C.S.，當 Pi 離開後，會依序把 turn 值設給 $P_{(i+1)\%n}$，使其進入，依此類推，Process 會依下列 FIFO 進入 C.S. ● Pi, ……, $P_{(i+1)\%n}$ ● Process 至多等 n-1 後必可進入 C.S.	OK 若 n 個 process 皆想進入 C.S.， 令 Pj 具最大 number 值為 k 所以 Pi(i≠j)皆可於 Pj 先進入 C.S.，而 Pj 再等待 若其他先進去的 process，離開 C.S.後又企圖再進入 C.S.，則其 number[i]必大於 k，故不會搶先於 Pj 進入 C.S. ● Pj 最多等 n-1 次，即可進入 C.S.

- Hardware instruction support for C.S. design

		Test-and-Set	SWAP
Def.		int Test-and-Set (int *Lock) { int *temp = *Lock; *Lock = 1; return temp; }	void SWAP (int *a, int *b) { int temp = *a; *a = *b; *b = temp; }
Algo1 (錯)	共享變數宣告	✓ Lock : Boolean = False; (初值)	✓ Lock : Boolean (初值為 False)
	Code	repeat	區域變數 = key:Boolean

		while Test-and-Set (Lock) do no-op; C.S. Lock = False; R.S. until False	repeat key = True; repeat SWAP(Lock, key); until (key == False) C.S Lock = False; R.S. until False
	Mutual Exclusion	OK 因為第一個搶到 Test-and-Set(Lock)執行的 process 才能進入 C.S. 而無法搶到第一次執行者，會被迫等待	OK
	Progress	OK (i) 不想進入 C.S.之 progress 不會與其他 process 搶奪 Test-and-Set 之執行 (ii) 有限的時間，必可決定出一個出一個 process 優先執行到 Test-and-Set 而進入 C.S.，所以 No Deadlock	OK
	Bounded Waiting	違反 若 Pi 已在 C.S.中，且 Pj 正在 wait 進入 C.S.，若 Pi 離開 C.S.後又企圖進入 C.S.，則 Pi 有可能再度優先於 Pj 執行 Test-and-Set。 所以 Pj maybe Starvation	違反
Algo2	共享變數	✓ Lock : Boolean = False; (初值)	

<table>
<tr><td rowspan="4">(對)</td><td>宣告</td><td>✓ waiting[0..n-1] of Boolean (初值為 False)
➢ waiting[i] = True : 表 Pi 有意願且正在等待進入 C.S.
False: 初值 or Pi 已進入 C.S.</td><td></td></tr>
<tr><td>Code</td><td>區域變數 = ｛ j : int, key : Boolean ｝
repeat
waiting[i] = True;
key = True;
while (waiting[i] && key) do
key = Test-and-Set(Lock);
waiting[i] = False;
C.S.
j = (i+1) % n;
while (j≠i) and (not waiting[j]) do
j = (j+1) % n;
if (j==i) then
Lock = False;
else
waiting[j] = False;
R.S.
until False</td><td>仿照 Test-and-Set 下列修改
while (waiting[i] && key) do
key = SWAP(Lock, key);</td></tr>
<tr><td>Mutual Exclusion</td><td>OK
Pi 可進入 C.S.之條件有二：
(i) key 為 False
(ii) waiting[i]為 False</td><td></td></tr>
<tr><td>Progress</td><td>OK</td><td></td></tr>
</table>

		(i) 不想進入 C.S.之 process 其 waiting[i]必為 False，所以 Pi 不會同其他 process 爭奪 Test-and-Set 之執行，且從 C.S. 離開之 Process 也不會去設定 Pi 之 waiting 值為 False ➢ 所以 Pi 不會參與進入 C.S.的決策 (ii)在有限的時間內，必可決定出第一個搶到 Test-and-Set 執行的 Process 而進入 C.S.	
	Bounded Waiting	若 n 個 process 皆想進入 C.S.，則他們的 waiting 值必為 True 若 Pi 先行進入 C.S.，則當 Pi 離開後他必設定 wait[(i+1)%n] 為 False，選擇下一個 Process 進入 C.S. 所以任一 process 至多等 n-1 次後進入 C.S.	

- Semaphore
  - Def. 是一個用來解決 C.S. Design 及 Synchronization Problem 之資料型態

    令 S 為一個 semaphore type 變數，則在 S 上提供兩個 atomic operations：wait(s) & signal(s)

    其中意義如下：

    wait(s)：while s ≦ 0 do no-op;

    s = s-1;

    signal(s)：s = s+1;

  - 用在 C.S. Design 上
    - 共享變數宣告：

      mutex：semaphore = 1; (初值)
    - Pi 程式如下：

      repeat

```
        wait(mutex);
            C.S.
        signal(mutex);
            R.S.
    until False
```

- 檢查三個 criteria

- 簡單同步問題解決(課本)

- 著名同步問題解決(背)

	Producer – Consumer (1) (2)	Reader –Writer (1) (2)	Second Reader – Writer (1) (2)	Sleeping Barber (1)	Dining Philosophers
共享變數	(1) mutex : semaphore = 1; -> 提供互斥存取之用 (2) empty : semaphore = n; -> 代表 Buffer 中空格數目，即若空格數為 0，表 buffer 滿 (3) full : semaphore = 0; ->代表 buffer 中提入資料的格數，即若填入資料格數為 0，代表 buffer 空	(1) wrt : semaphore = 1; -> 提供 R/W, W/W 之基本互斥控制，且兼對 writer 之不利控制 (2) readcount : int = 0; -> 統計目前 reader 個數 (3) mutex : semaphore = 1; -> 提供做為 readercount 之互斥控制，防止 race condition	(1) readcount : int = 0; (2) writecount : int = 0; (3) x : semaphore = 1; (4) y : semaphore = 1; (5) z : 對 reading section 入口控制 (6) wsem : semaphore = 1; ->R/W, W/W 之基本互斥控制 (7) rsem : semaphore = 1; -> 作為對 reader 不利之控制	(1) Customers: semaphore=0 ->強迫 Barber 等待 (2)Barber : semaphore = 0; ->強迫 customer 等 (3) waiting : int = 0; -> 等待之 customer 數目 (4) mutex : semaphore = 1;	✓ chopstick[0..4]: semaphore 初值為 1 ● 代表每根筷子之互斥控制之用
(1) 程式	repeat *produce an item in nextp;* wait (empty); wait (mutex); *add nextp into Buffer;* signal(mutex); signal(full); until False	wait(mutex); readcount = readcount+1; if readcount == 1 then wait(wrt); signal(mutex); *reading is performed;* wait(mutex); readcount = readcount–1; if readcount == 0 then signal(wrt); signal(mutex);	wait(z); wait(rsem); wait(x); readcount=readcount+1; if(readcount == 1) then wait(wsem); signal(x); signal(rsem); signal(z); *reading is performed;* wait(x); readcount = readcount -1; if(readcount==0) then signal(wsem);	repeat wait(Customers); wait(mutex); waiting = waiting -1; signal(Barber); signal(mutex); *cuthair(); //剪客人頭髮* until False;	Pi 之程式如下： repeat hungry; wait(chopstick[i]); wait(chopstick[(i+1)%5]); eating; signal(chopstick[i]); signal(chopstick[(i+1)%5]) thinking; until False; ✓ 此 solution 有錯， ● 可能發生"Deadlock"，若

			signal(x);		每個哲學家依序取得左邊筷子，然後每位哲學家皆在等待右邊的筷子，形成 Circular Waiting
(2) 程 式	repeat wait (full); wait (mutex); *retrieve an item in nextp from buffer;* signal(mutex); signal(empty); *consume the item in nextp;* until False	wait(wrt); *writing is performed;* signal(wrt); p.s. 如何改善 First Reader-writer problem 對 writer 的不公平? ● 藉由 timestamp 標記等待中的 process, 使之等待較久的 process 可以藉由 time 值的增加, 取得 CPU 的執行	wait(y); writecount=writecount+1; if(writecount==1) then wait(y); signal(y); wait(wsem); *writing is performed;* signal(wsem); wait(y); writecount=writecount-1; if(writecount==0) then signal(rsem); signal(y);	wait(mutex); if(waiting < n) then { waiting = waiting+1; signal(Customers); signal(mutex); wait(Barber); *haircut(); //被理髮* } else signal(mutex);	解上述 Deadlock 之方法 [1] 至多允許 4 個哲學家上餐桌 [2] 規定"除非哲學家可取得左右兩邊之筷子，否則不得持有任何快子" -> 可打破"Hold & Wait" [3] 用"Asymmetric"模式

- Semaphore 種類

	第一種區分角度：號誌值		第二種區分角度：有沒有使用 Busy waiting 技巧來定義 semaphore	
	Binary Semaphore	Counting Semaphore	Spinlock	Non-BusyWaiting
Def	✓ 號誌值中只有 0 & 1 兩種可能，不會為負值 ✓ 無法統計有多少 process 卡在 wait 敘述	✓ Sempahore 值可為負數，且若值為-N，則表示有 N 個 process 卡在 wait 敘述 ✓ 利用 Binary Semaphore 定義	使用 Busy Waiting 的技巧定義 Semaphore 稱為 SpinLock。反之則為 Non-BusyWaiting Spinlock 之優點：若 process 可以在極短的時間內(即小於一個 context-swithcing 的時間)決定出下一個進入 C.S.的 process,並離開	

		出 Counting Semaphore	busy waiting loop，則 spinlock 是有利的。 缺點：等待中的 process 仍會與其他 process 搶奪 CPU，將 CPU TIME 用在無意義的 loop 測試上，若將這些 CPU time 給其他可以往下執行的 process，則其 throughput 較好。	
共享變數	● S: semaphore = 1;	S1: Binary Semaphore = 1; ->提供對 C 值之互斥控制 C : int ->代表 Counting Semaphore 號誌值 S2: Binary Semaphore = 0; ->強迫 process 暫停之用		
wait (s)	while S ≦ 0 do no-op; S = S – 1;	wait(S1); C = C – 1; if(C<0) then { signal(S1); wait(S2); } else signal(S1);	While (S <= 0) ; // do nothing S = S – 1;	S.val = S.val – 1; If(S.val < 0) { //Add process p to S.QUEUE Block(p); }
signal (s)	S = S + 1;	wait(S1); C = C + 1; if(C ≦ 0) then	S = S + 1;	S.val = S.val + 1; If(S.val <= 0) { //remove a process P from S.Queue

		singal(S2); singal(S1);		Wakeup(p); }

※Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems?

Ans: 因為 spinlock 的條件式需要靠另一個不相同的 processes 執行，改變其條件值的 state，也就是在 uniprocessor system 之架構下，除非該 process 放棄 processor，否則其餘 processes 沒有機會執行它們的 program.

In multiprocessor system, other processes executed on other processors and thereby modify the program state in order to release the first process form spinlock.

- Semaphore 的製作

	Non-Busy waiting	Spinlock
Disable interrupt	wait(s): *Disable Interrupt* S.value = S.value – 1; if(S.value < 0) { Add process to S.Queue; *Enable Interrupt* Block(p); } else *Enable Interrupt*	wait(s): *Disable Interrupt* while (S≤0) do { *Enable Interrupt* //do nothing *Disable Interrupt* } S = S – 1; *Enable Interrupt*
	signal(s): *Disable Interrupt* S.value = S.value + 1; if(S.value ≤ 0) {	signal(s): *Disable Interrupt* S = S + 1; *Enable Interrupt*

	remove process form S.Queue; *Enable Interrupt* wakeup(p); } else *Enable Interrupt*	
C.S. Design	將 Algo1 & Algo3 中的 Disable interrupt 改成 Entry Section Enable interrupt 改成 Exit Section	

- Semaphore V.S. Monitor

Semaphore	Monitor
-> A synchronization tool do not use busy waiting -> A semaphore S is a variable -> Semaphore is accessed only through two standard atomic operations "wait()" and "signal()" -> Semaphore is a Data type to solve C.S. design and synchronization	是一個用來解決 synchronization Problem 的高階資料結構，分為下列三大部分 (1) shared variable declarations (2) procedures (3) initialization Area
本身未確保 Mutual Exclusion	Mutual Exclusion 已確保，programmer 不需刻意提出
Programmer 對 signal 及 wait 的誤用仍會引起 Mutual Exclusion 違反，及 deadlock 的問題	Programmer 只需專心對付同步問題的解決，不必擔心違反 Mutual Exclusion

- 利用 Monitor 解決同步問題

CASE1: 沒有優先權考量的同步問題

The Dining Philosophers Problem

```
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; /*allow neighbors to eat
        // 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)) {
```

[恐習題] 試說明 Monitor 與 Semaphore 中的之 signal() operation 的差異?

(1) Monitor 中之 signal():
其 signal operation 在 waiting queue 中有 process 時執行，會取出 waiting queue 中之一 process 恢復其作用，若 waiting queue 中無 process 時執行，則此 operation 無作用

(2) Semaphore 中之 signal():
當 Semaphore 執行 signal() operation 的時候，無論是否有 waiting process(or thread)的存在，都會將其 semaphore value+1, 以便將來 wait operation 的執行成功

```
            state[i] = eating;
            self[i].signal();
        }
}

void initialization_code
{       for (int i = 0, i < 5; i++)
        state[i] = thinking; }}
do{
        hungry;
        dp.pickup(i);
        eating;

        dp.putdown(i)}
```

CASE2: 考慮優先權值

例子	設計一個 monitor，負責資源分配的工作，且使用資原時間最小的 process 優先分配	一個 file 多個 process 共享，規定 priority 最小的優先權最高，且 access 此 file 之所有 process 之優先權值和必須 < n	3 部列表機被 n 個 process 共享，設計 monitor 去支配三部列表機依照 process 之 priority number(值越大優先權值越高)
定義 Monitor	monitor resource{ boolean busy;	monitor allocate{ int sum;	monitor printers { int num_avail ;

```
condition x;

void Request( int time )
{
      if busy then x.wait(time);
      busy = TRUE;
}

void Release()
{
      busy = FALSE;
      x.signal();
}

void initialization_code
{
      busy = FALSE;
}
```

```
condition x;

void Request( int i )
{
      while(sum + i ≧ n) do x.wait(i);
      sum = sum + i;
}

void Release(int i)
{
      sum = sum – i;
      x.signal();
}

void initialization_code
{
      sum = 0;
}
```

```
   int waiting [MAX PROCS];
   int num_waiting;
   condition c;

void request printer(int proc_number)
{
    if (num avail > 0) {
       num avail--;
       return;
    }
    waiting[num_waiting] = proc_number;
    num_waiting++;
    sort(waiting);
    while (num avail == 0 ||
    waiting[0] != proc_number) do
       c.wait(proc_number);
    waiting[0] = waiting[num_waiting-1];
    num_waiting--;
    sort(waiting);
    num_avail--;
}
void release printer() {
   num avail++;
   c.singal();
}
void initialization_code()
```

			{ num_avail = 3; } }
Monitor 之使用	Resource.Request(time); //use the resource Resource.Release();	allocate.Request(time); //access the file allocate.Release();	
優先權值之使用方式	此 monitor 的 x condition 變數之 waiting Queue 及 monitor 的 entry Queue 是 priority Queue，規定 time 值最小的 process 優先自 Queue 中刪除	此 monitor 的 x condition 變數之 waiting Queue 及 monitor 的 entry Queue 是 priority Queue，規定 priority NO.最小的 process 優先自 Queue 中刪除	

- Monitor 的三種種類

[Hoare] monitor (最佳, 恐用此版本) P wait Q until Q finish or Q is blocked again	Q wait P until P finishes or P is blocked	P exits monitor and let Q resume execution (current C / PASCAL 採用)
優點: ● 保證可以恢復 Q 執行 ● 因為 Hoare 之 monitor 在 P 執行 signal 時，是將它放置在 waiting Queue 中，而不是 exit monitor, 所以在一進一出之間若執行多次 signal，則可以恢復多個 Q	缺點: Q 可能來被真正解救，且在等待 P 完成的過程中，Q 有可能其恢復的條件值改變以致於 Q 再次 blocked	優點: ➢ Q 可立即被 resume ➢ 相對於 Hoare, 製作較簡單 缺點: 在 P 的一次週期內(i.e., P 一進一出 monitor 期間), 頂多只恢復一個 Q

- Monitor 與 Semaphor 之相互製作

利用 Monitor 製作 Semaphor

```
monitor Semaphore{
    int value;
    condition x;

    void wait()
    {
        value--;
        if value  <  0
            x.wait;
    }

    void signal()
    {
        value++;
        if value ≦ 0
            x.singal;
    }
}
```

利用 Semaphor 製作 Monitor

針對 monitor 內的每一個 procedure 必須在 body 前後加入以下程式碼:

```
wait(mutex);
    //body of F;
if next-count>0   then signal(next)
        else signal(mutex);
```

針對 condition 變數

- x.wait

```
    x-count++;
    if (next-count > 0)
        signal(next);
    else
        signal(mutex);
    wait(x-sem);
    x-count--;
```

- x.signal

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

※Write a monitor that implements an alarm clock that enables a calling program to delay itself for a specified number of time units(ticks). You may assume that existence of a real tick in your monitor at regular intervals.

Ans:

```
monitor alarm{
        condition c;
        void delay (int ticks)
        {
                int begin_time = real_clock();
                while( real_clock() < begin_time + ticks )
                        c.wait();
        }
        void tick()
        {
                c.signal();
        }
}
```

※adaptive mutex: protect access to every critical data item. On multiprocessor system, adaptive mutex starts as a standard semaphore implemented as a spinlock. 若一個 data 被 lock 住,而占用的是一個 running state 的 thread, 則 thread 等待 lock 成為可用;但若占用的是一個 waiting state 的 thread, 則 thread 將被 block 直到 lock 被 release 才 wake up

※turnstile: is a queue structure containing threads blocked on a lock. Solaries 用它來排序等待的 adaptive mutex 或 reader-writer lock 的 thread

- Message passing 的兩種模式

Direct Communication	Indirect Communication

收、送雙方要互相指名對方 ID 才能建立 communication link	收、送雙方要有共享的 mailbox 才能建立 communication link
communication link 是屬於溝通雙方，不可共享	communication link 可被多個 process 共享
溝通雙方只存在一條 communication link，不可多條	可有多條 links 存在溝通之雙方(唯每一條皆有對應的 mailbox)