

第六章 處理元之同步(Process Synchronization)

6.1 非同步並行處理元

- 並行處理元(Concurrent Processes)
 - Processes execute **concurrently**.
- 非同步並行處理元(Asynchronous Concurrent Processes)
 - 電腦內的處理元依自己的程式碼執行，它們都是**依照自己的步伐處理，而且彼此獨立自主**。
 - Processes proceed **at their pace, independent of one another**.

非同步並行處理元(Asynchronous Concurrent Processes)

- 共用資源(Shared Resources)
 - Shared logical address space
 - Shared data
 - Shared file
- 共用資源之互斥性(Mutual Exclusion)
 - 當一個共用資源被某一個處理元存取時，別的處理元不能並行存取，一定要等到存取完畢之後，才能輪由別的處理元使用。
 - When a process **manipulates shared resources**, other processes must be **excluded** from doing so simultaneously.

處理元排程產生之問題(1)

步驟1： process1() read

shared ;

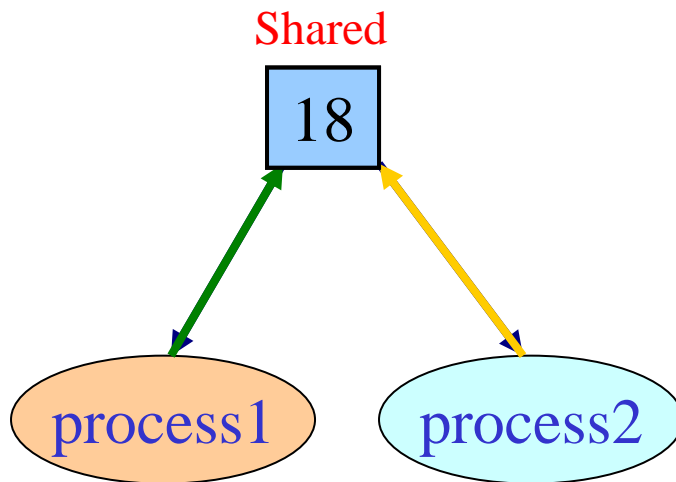
步驟2： process2() read

shared ;

步驟3： process1() add 1 to

shared ;

步驟4： process2() add 1 to shared ;



```
process1()
```

```
{
```

```
    read shared;
```

```
    add 1 to shared;
```

```
}
```

```
process2()
```

```
{
```

```
    read shared;
```

```
    add 1 to shared;
```

```
}
```

處理元排程產生之問題(2)

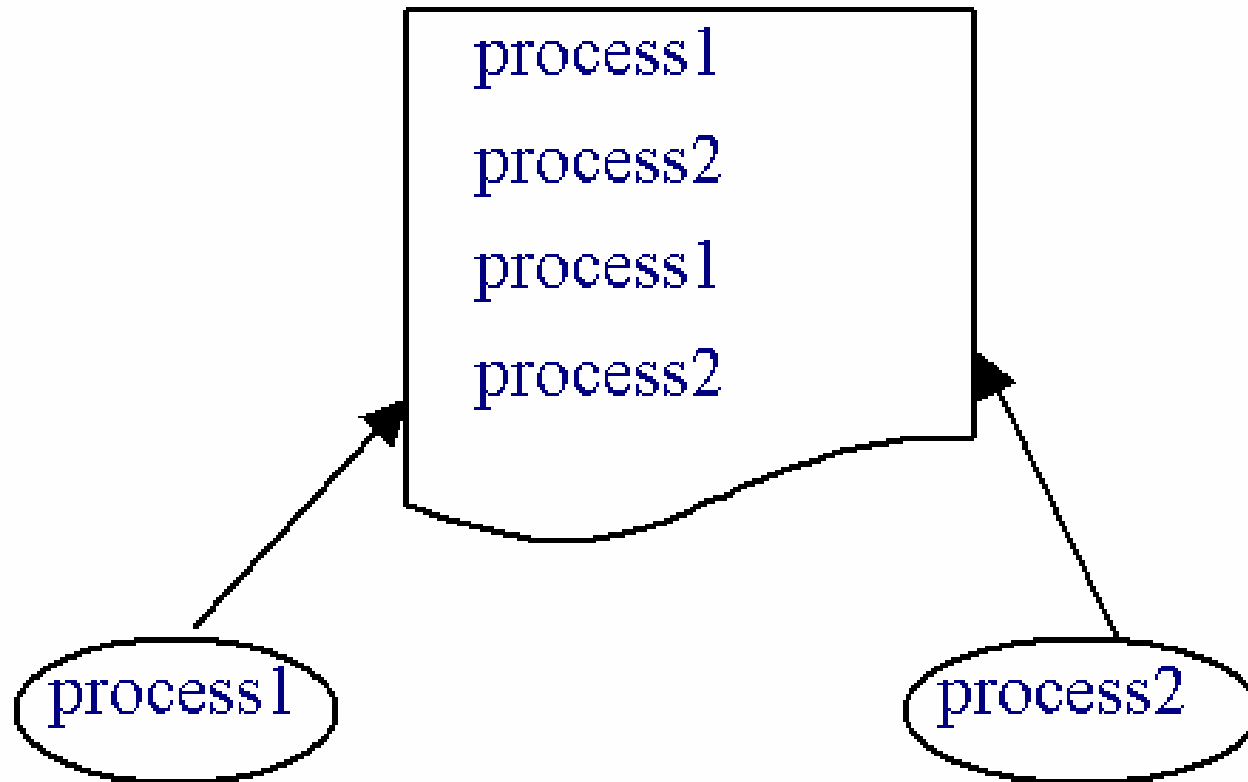


圖 4.13 二個處理元共用印表機印表

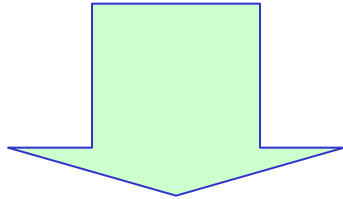
可以使用週邊設備線上同時工作（**Spooling**）來解決₄。

競賽情況 (Race Condition)

- 指多個處理元同時並行 (Concurrent) 存取共用資源，系統依排程次序執行，而造成資源內的資料不正確的問題發生。
- Several processes access and manipulate the shared resources concurrently , and the outcome of the execution depends on the particular order in which the access takes places.

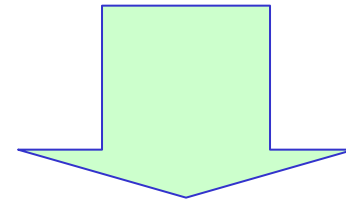
處理元排程產生之問題(3)

- Producer
- `counter = counter + 1;`



- 1 MOV AX, counter
- 2 ADD AX, 1
- 3 MOV counter, AX

- Consumer
- `counter = counter - 1;`



- 4 MOV AX, counter
- 5 SUB AX, 1
- 6 MOV counter, AX

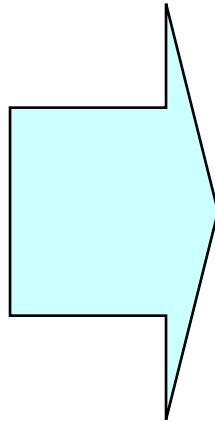
Context switch may occurs, execute in arbitrary order

處理元排程產生之問題(4)

- 1 MOV AX, counter
- 2 ADD AX, 1
- 3 MOV counter, AX

- 4 MOV AX, counter
- 5 SUB AX, 1
- 6 MOV counter, AX

- 1 2 3 4 5 6
- 1 4 2 5 3 6
- 1 2 4 3 5 6
- 1 4 5 2 3 6
- 1 4 2 3 5 6
- 4 5 6 1 2 3
- 4 1 5 2 6 3



- counter之初始值為5
- 5
- 4
- 4
- 4
- 4
- 5
- 6

循序執行 (Serial Execution)

- 當一些處理元等待被執行，而我們每執行完畢一個處理元之後，再依序處理下一個處理元，這種執行方式稱為循序執行 (Serial Execution)。
- Any execution of transactions one at a time in any order.
- 多個處理元依循序執行方式執行，結果是對的，假如有 n 個處理元，則有 $n!$ 種循序執行次序。
- 循序執行違反分時系統之初衷。

可循序化 (Serializable) (1)

- 處理元不是依循序執行，而是依中央處理器排程次序執行，其結果與循序執行相同，則此執行次序是可循序化 (Serializable) 的。
- 可循序化的排程 (Serializable Schedule) 。
- The concurrent execution of transactions must be equivalent to the case where these transactions executed **serially** in some arbitrary order.

可循序化 (Serializable) (2)



圖 4.15a 二個共用資源之處理元

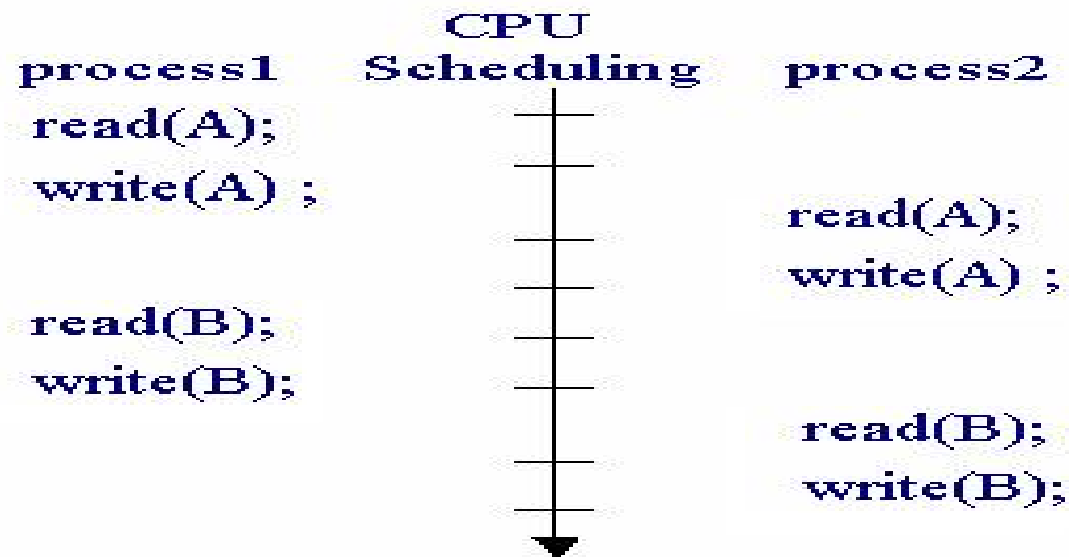
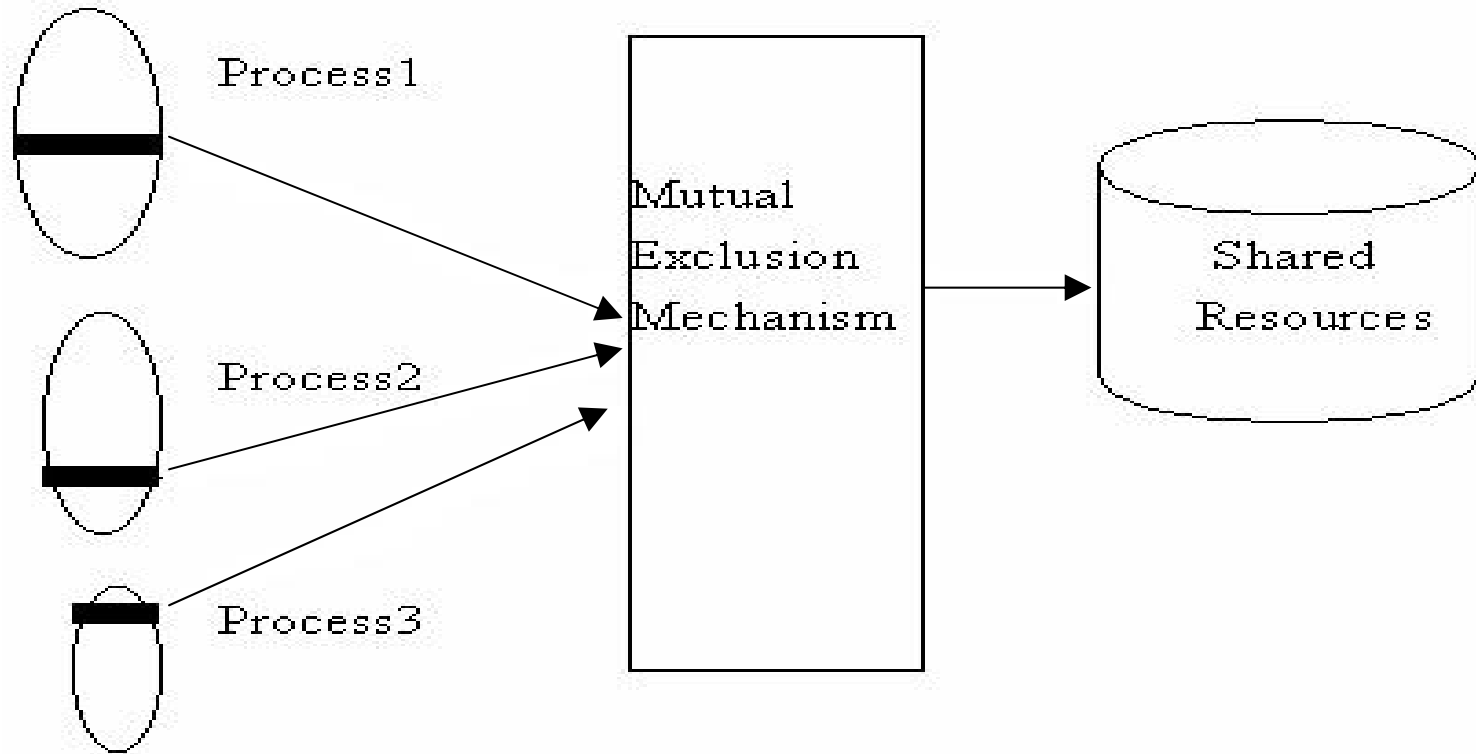


圖 4.15b

圖 4.15

中央處理器排程之次序
可循序化的執行範例

互斥機制(Mutual Exclusion Mechanism)



互斥機制

人工或自動

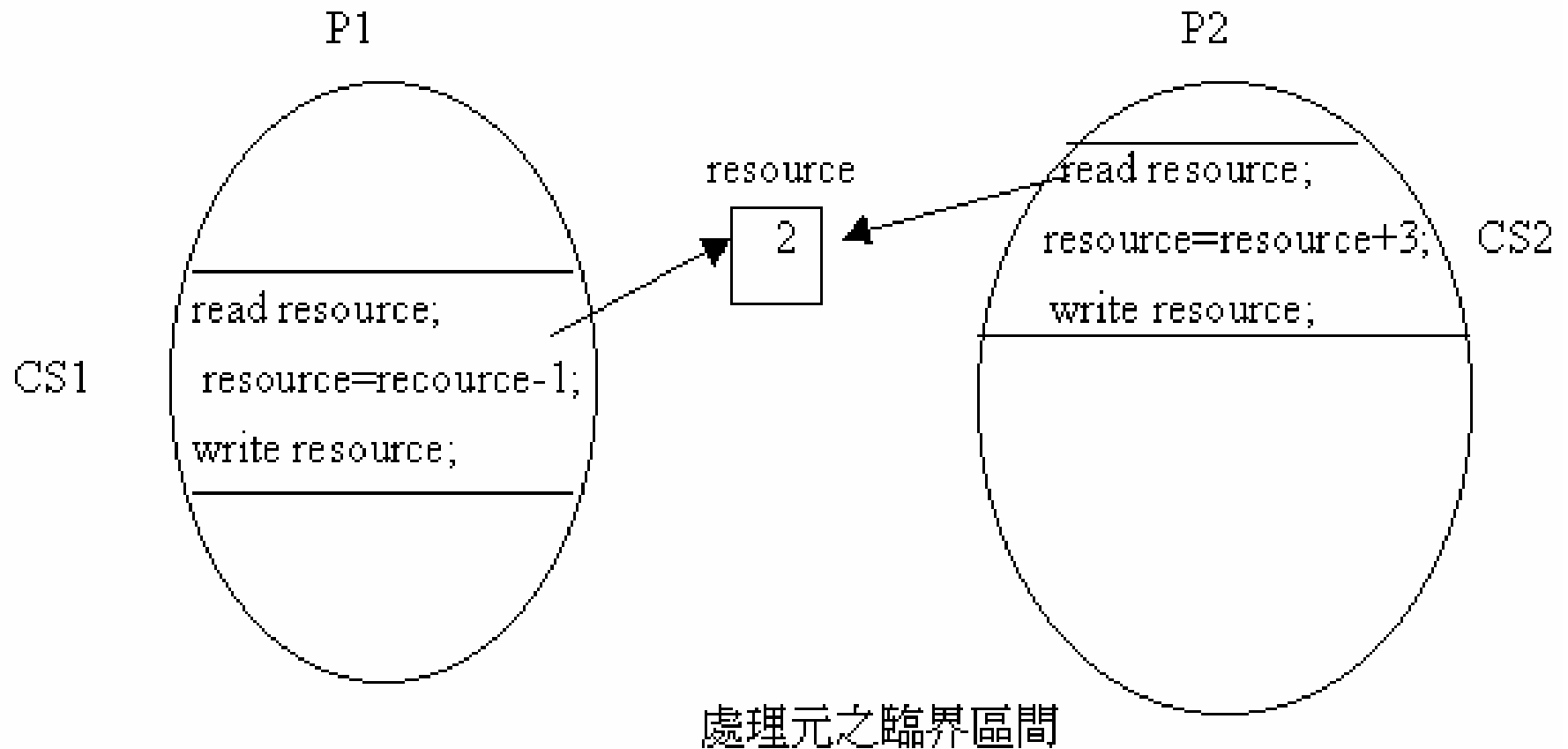
互斥 (Mutual Exclusion)

- When a process manipulates shared resources, other processes must be excluded from doing so simultaneously.

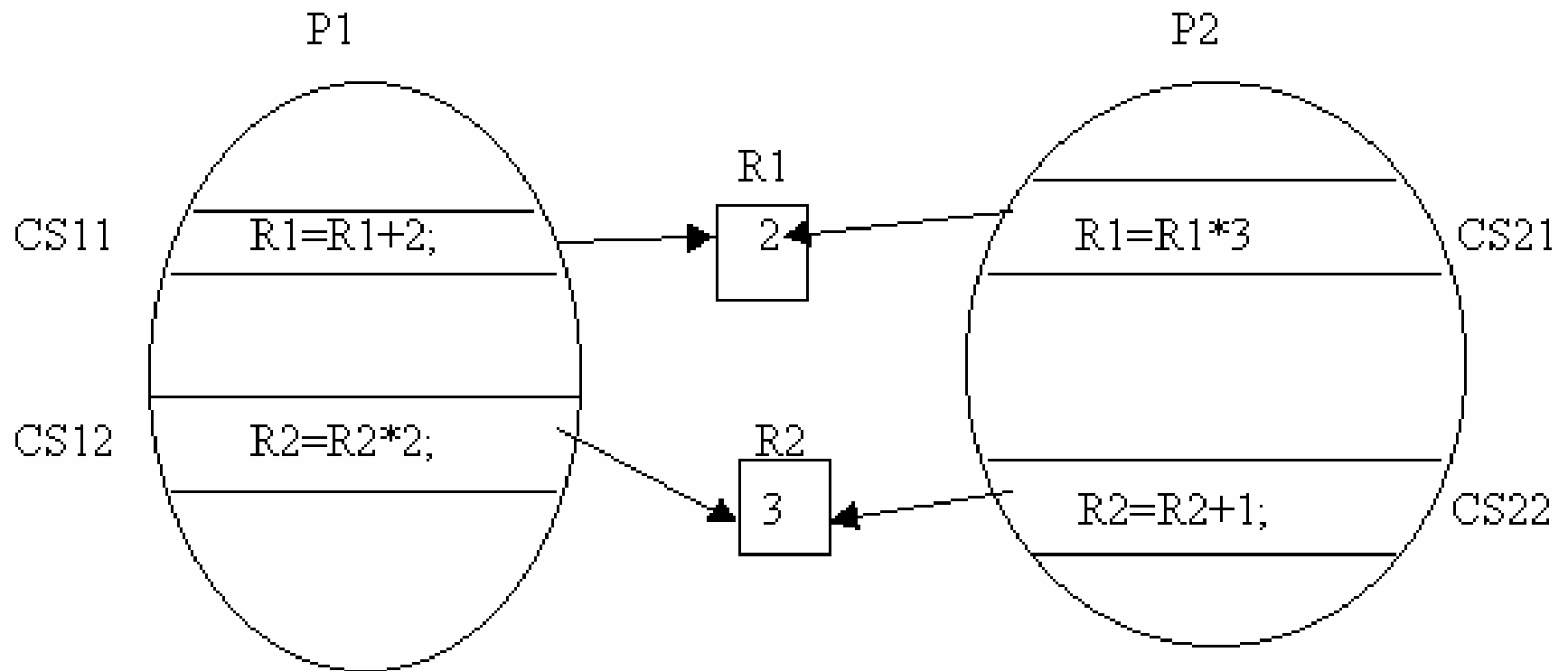
臨界區間(Critical Section)(1)

- 處理元去存取共用資源的那段程式碼。
- The procedural code that changes a set of shared resources is called critical section.
- 當一個 process 正在動用share resource 時，其餘 process 不能去動用。
- When one process is executing in its critical section , no other process is to be allowed to execute in its critical section.
- When a process manipulates shared resources, other processes must be excluded from doing so simultaneously.

臨界區間(Critical Section)(2)



臨界區間(Critical Section)(3)



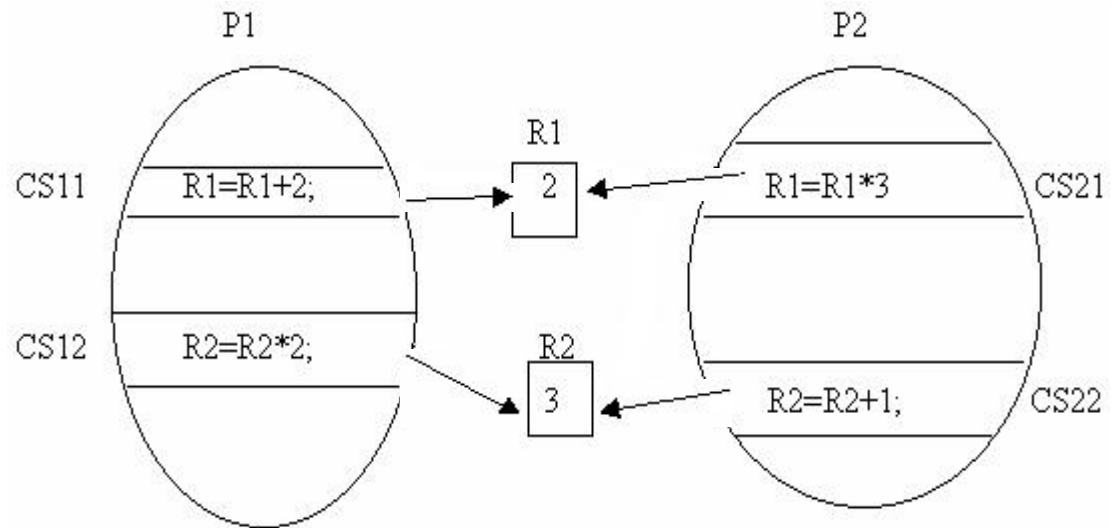
二個處理元因共享資源而同步之臨界區間

交易(Transaction)

- 所謂的交易，是一個最細小 (Atomic) 不可分割的單位，它包含一個或數個指令 (Instructions) 或運算 (Operations)，用來完成一個邏輯功能 (Logical Function)。
- A collection of instructions(operations) that performs a single logical function.
- 一個處理元可以視為一個交易，但有時又可以將處理元看成是有許多交易所組成。

交易的循序執行(1)

- 以下幾種交易的循序執行都是對的。
 - 1. CS11 , CS12 , CS21 , CS22
 - 2. CS11 , CS21 , CS12 , CS22
 - 3. CS11 , CS21 , CS22 , CS12
 - 4. CS21 , CS22 , CS11 , CS12
 - 5. CS21 , CS11 , CS22 , CS12
 - 6. CS22 , CS11 , CS12 , CS21

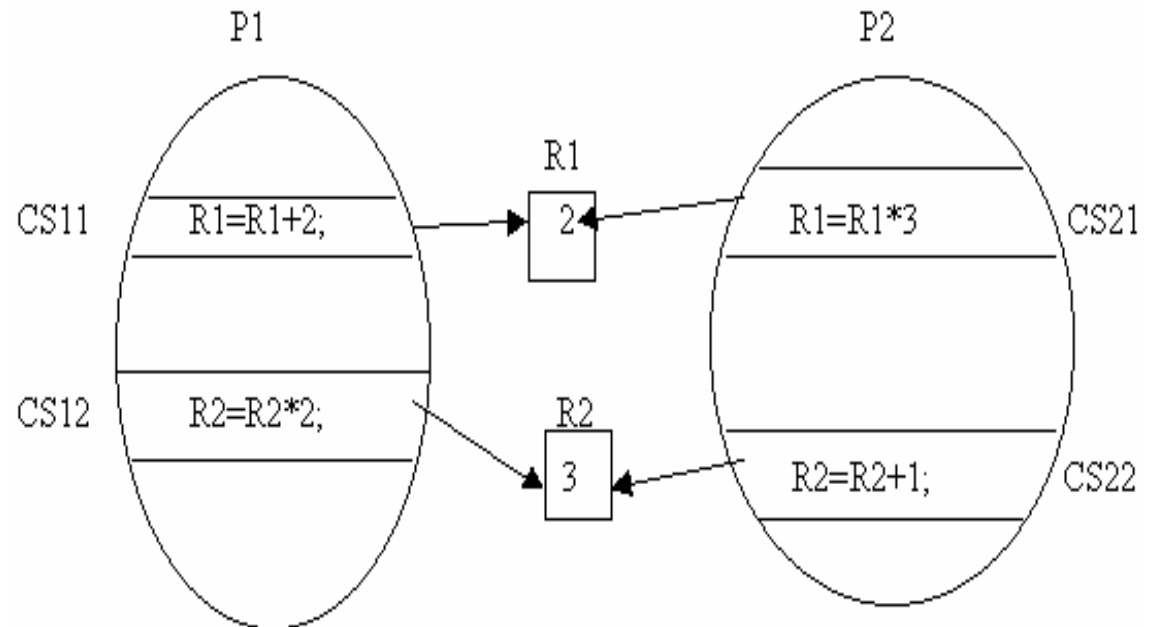


二個處理元因共享資源而同步之臨界區間

交易的循序執行(2)

- 若我們考慮將處理元視為一個交易，則執行結果僅有
 - 1. CS11，CS12，CS21，CS22
 - 2. CS11，CS21，CS12，CS22
 - 4. CS21，CS22，CS11，CS12
 - 5. CS21，CS11，CS22，CS12 是對的。

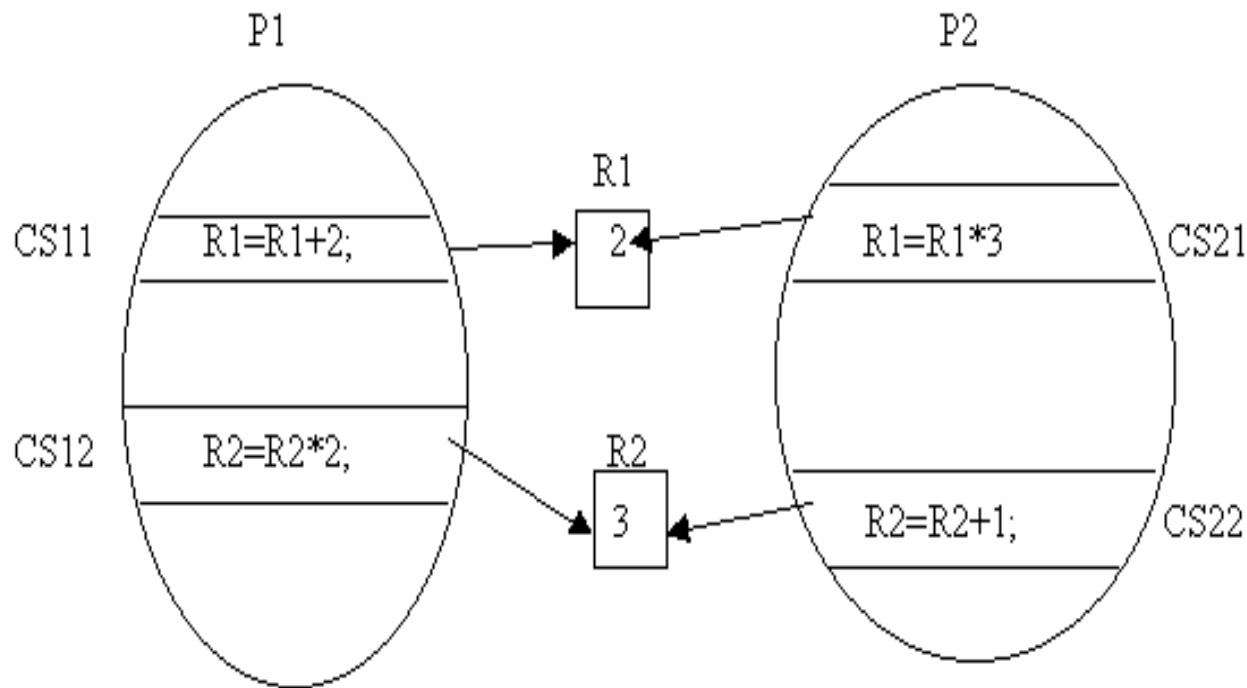
- 也就是 $R1=8$ 及 $R2=8$
- 或 $R1=12$ 及 $R2=7$ 。



二個處理元因共享資源而同步之臨界區間

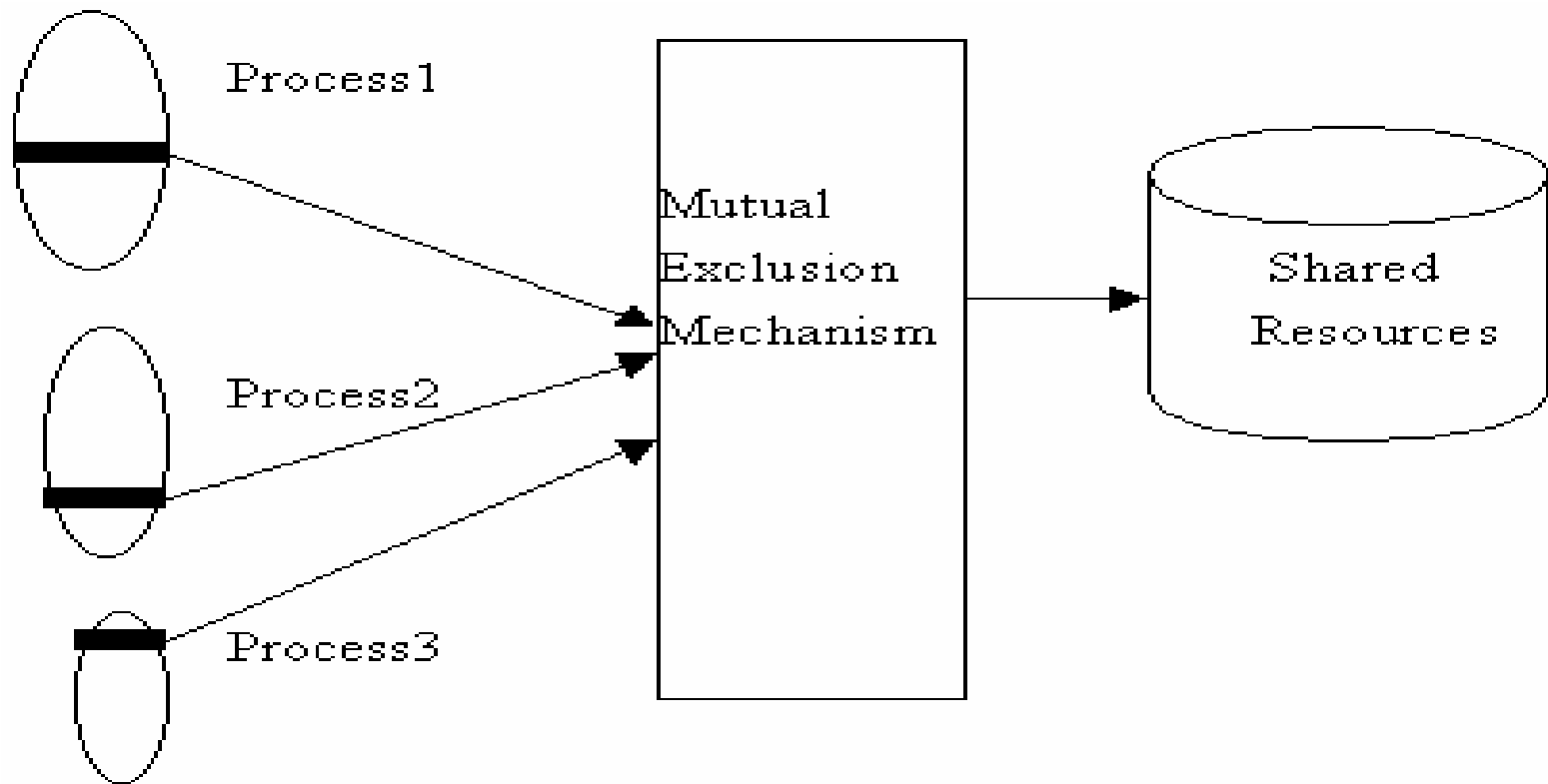
交易的循序執行(3)

- 以下幾種交易是不對的。
 - 3. CS11 , CS21 , CS22 , CS12 $R1=12, R2=8$
 - 6. CS21 , CS11 , CS12 , CS22 $R1=8, R2=7$



二個處理元因共享資源而同步之臨界區間

進入臨界區間的方法(1)



互斥機制

進入臨界區間的方法(2)

Entry Statement;

CRITICAL SECTION;

Exit Statement;

Remainder Code;

進入及離開臨界區間之方法

進入臨界區間的條件

6.2 臨界區間的問題

- 1. 互斥(Mutual Exclusion)
 - 當某個處理元在臨界區間內存取某些資源，則其他處理元不能進入臨界區間去存取相同資源。
- 2. 行進(Progress)
 - 當臨界區間內沒有處理元在執行，若有多個處理元要求進入臨界區間執行，則只有那些不在執行剩餘程式碼 (Remainder Code) 的處理元，才有資格被挑選為下一個進入臨界區間的處理元，而且這個挑選工作不能無限期的延遲下去。
- 3. 有限等待(Bounded Waiting)
 - 當某個處理元要求進入臨界區間，一直到它獲得進入臨界區間這段時間，允許其他處理元進入臨界區間的次數有限制。

以軟體解決進入臨界區間的方法(1)

- 禁止中斷法
 - 在進入臨界區間前，使用disable()系統呼叫，將系統內所有中斷均禁止，並於離開時使用enable()，啟動所有中斷。
 - 勉強在單一中央處理器環境下可以被使用，但在多處理器環境下並不可行

```
disable( );
```

```
CRITICAL SECTION;
```

```
enable( );
```

使用禁止中斷法

以軟體解決進入臨界區間的方法(2)

- 軟體解決方法1
 - 若二個處理元依序輪流進入臨界區間，將保證互斥，但是行進的條件無法滿足。

```
while turn < > i do no-op;
```

CRITICAL SECTION;

```
turn = j;
```

Process P_i

```
while turn < > j do no-op;
```

CRITICAL SECTION;

```
turn = i;
```

Process P_j



軟體解決方法 1

以軟體解決進入臨界區間的方法(3)

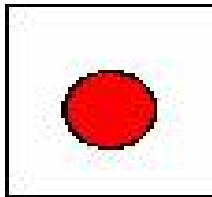
- 軟體解決方法2

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

CRITICAL SECTION;

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

Process P_i

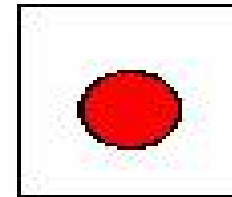


```
flag[j] = true;  
while flag[i] do no-op;
```

CRITICAL SECTION;

```
flag[j] = false;
```

Process P_j



軟體解決方法 2

以軟體解決進入臨界區間的方法(4)

- 處理元為非同步並行執行，而環境切換可能會造成：
 - 在時間T1時執行Pi的flag[i] = true。
 - 在時間T2時執行Pj的flag[j] = true。
- 二個處理元均將在迴圈中等待對方將flag設為false，但卻永遠無法進入臨界區間的問題。

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

CRITICAL SECTION;

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

Process Pi

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

CRITICAL SECTION;

```
flag[j] = false;
```

Process Pj

以軟體解決進入臨界區間的方法(5)

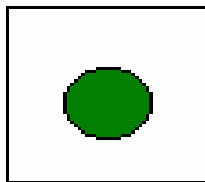
- 軟體解決方法3

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

CRITICAL SECTION;

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

Process P_i

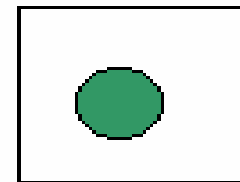


```
while flag[i] do no-op;  
flag[j] = true;
```

CRITICAL SECTION;

```
flag[j] = false;
```

Process P_j



軟體解決方法 3

以軟體解決進入臨界區間的方法(6)

- 可能遭遇到下列環境切換的問題：
 - 在時間T1時執行Pi的迴圈測試，得知flag[j]為false。
 - 在時間T2時執行Pj的迴圈測試，得知flag[i]為false。
- 這時二個處理元均在臨界區間，它們違反了互斥的條件，故此方法並不可行。

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

CRITICAL SECTION;

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

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

CRITICAL SECTION;

```
flag[j] = false;
```

以軟體解決進入臨界區間的方法(7)

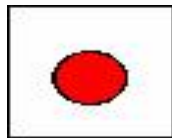
- 軟體解決方法4
 - 保證二個處理元進入臨界區間可以達到互斥、行進、及有限等待的條件，**而且在任何敘述間作環境切換均不受影響。**

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

CRITICAL SECTION;

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

Process P_i

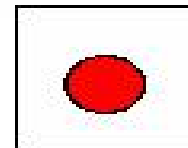


```
flag[j] = true;  
turn = i;  
while (flag[i] and turn = i)do no-op;
```

CRITICAL SECTION;

```
flag[j] = false;
```

Process P_j



解決二個處理元進入臨界區間的方法

以軟體解決進入臨界區間的方法(8)

- 互斥

- 若兩個處理元僅有一個的flag為true，則這個flag為true的處理元可以進入臨界區間，另一個處理元不需進入臨界區間，因此它們是互斥的。
- 當二個處理元均要求進入臨界區間，而且各別的flag也都是true時，因為turn的值只可能為i或j，若turn = i則Pi進入臨界區間，Pj在迴圈等待；若turn = j，則Pj進入臨界區間，Pi在迴圈等待。它們是互斥的。

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

CRITICAL SECTION;

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

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

CRITICAL SECTION;

```
flag[j] = false;
```

以軟體解決進入臨界區間的方法(9)

- 行進

- 當 P_i 與 P_j 均要求進入臨界區間，若 P_i 進入臨界區間， P_j 在迴圈等待。只要 P_i 離開臨界區間時，它會將 $flag[i]$ 設為false，turn設為j，因此 P_j 便可以進入臨界區間，這符合行進的條件。因此只要對方從臨界區間出來，另一方就可以進入臨界區間。

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

CRITICAL SECTION;

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

Process P_i

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

CRITICAL SECTION;

```
flag[j] = false;
```

Process P_j

以軟體解決進入臨界區間的方法(10)

- 有限等待
 - 正如同證明行進條件， P_j 只要等待 P_i 進入臨界區間一次，它便可以進入臨界區間，這符合有限等待之條件。
- 此方法可以滿足進入臨界區間的條件，但僅能提供解決二個處理元競爭共用資源。

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

CRITICAL SECTION;

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

Process P_i

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

CRITICAL SECTION;

```
flag[j] = false;
```

Process P_j

Bakery演算法(1)

- 保證達到互斥、行進、及有限等待條件。

```
choosing[i] = true;  
number[i] = max(number[0],number[1],...,number[n-1] )+ 1;  取最大號碼牌  
choosing[i] = false;  
for j = 0 to n-1  
    {  
        while choosing[j] do no-op;  
        while ( number[j] < > 0 and compare(number[j],number[i]) ) do no-op;  
    }
```

CRITICAL SECTION

```
number[i] = 0;
```

Bakery演算法(2)

- 保證達到互斥、行進、及有限等待條件。
- 就如同到銀行或商店接受服務一樣，任何一個顧客欲接受服務，均需依序取一張號碼牌，並依號碼牌由小至大依序接受服務；但是萬一很多人均有相同號碼牌時，則依客戶姓名排序依序接受服務。

Bakery演算法(3)

- $\text{compare}(\text{number}[j], \text{number}[i])$ 。 $\text{compare}()$ 函數用來比較二個 number 中那一個較小，若二個 number 相同，則看那個處理元編號較小。
 - 若 $\text{number}[j] < \text{number}[i]$ ，號碼牌 (number) 不同號碼。
 - 若 $\text{number}[j] = \text{number}[i]$ ，有多個處理元擁同號碼，則比較處理元編號。

Bakery演算法(4)

- 互斥

- 任何要求進入臨界區間的處理元，均會使用while迴圈查核 $\text{number}[j] < 0$ and $\text{compare}(\text{number}[j], \text{number}[i])$ 。
- 若最小號碼牌 (number) 的處理元僅有一個，則輪由這個處理元進入臨界區間，其他處理元只好繼續在迴圈內等待；
- 若有多個處理元擁有最小號碼牌 (均同號碼)，則因為每個處理元編號不同，故僅會有一個處理元進入臨界區間，其他處理元繼續在迴圈中等待。因此這些處理元進入臨界區間是互斥的。

Bakery演算法(5)

- 行進
 - 由互斥的證明中，我們可以知道只有一個處理元進入臨界區間，其他在迴圈等待之處理元因為各擁有自己的號碼牌，且之後欲進入臨界區間之處理元，其號碼牌均較大，所以處理元進入臨界區間的次序是**完全順序 (Totally Order)**，因此漸漸會有機會進入臨界區間。

Bakery演算法(6)

- 有限等待

- 當某個處理元要求進入臨界區間時，此處理元取得號碼牌具有完全順序之後，它在迴圈中等待時，若有別的處理元亦要進入臨界區間，則別的處理元取到的號碼牌一定比此處理元大，別的處理元不可能插隊在完全順序之內，因此這是一種**先到先服務 (First Come First Serve)**的排程，也就是在此處理元之前進入臨界區間的其他處理元，總共進入臨界區間的次數是有限的。

以硬體解決進入臨界區間的方法(1)

6.3 以硬體解決進入臨界區間的方法

- 最細小運算是利用硬體來實現此運算，使得此運算在執行過程中不會被中斷。

```
function test-and-set( target:boolean):boolean;  
{  
    test-and-set = target;  
    target = true;  
}
```

ts1 register, flag

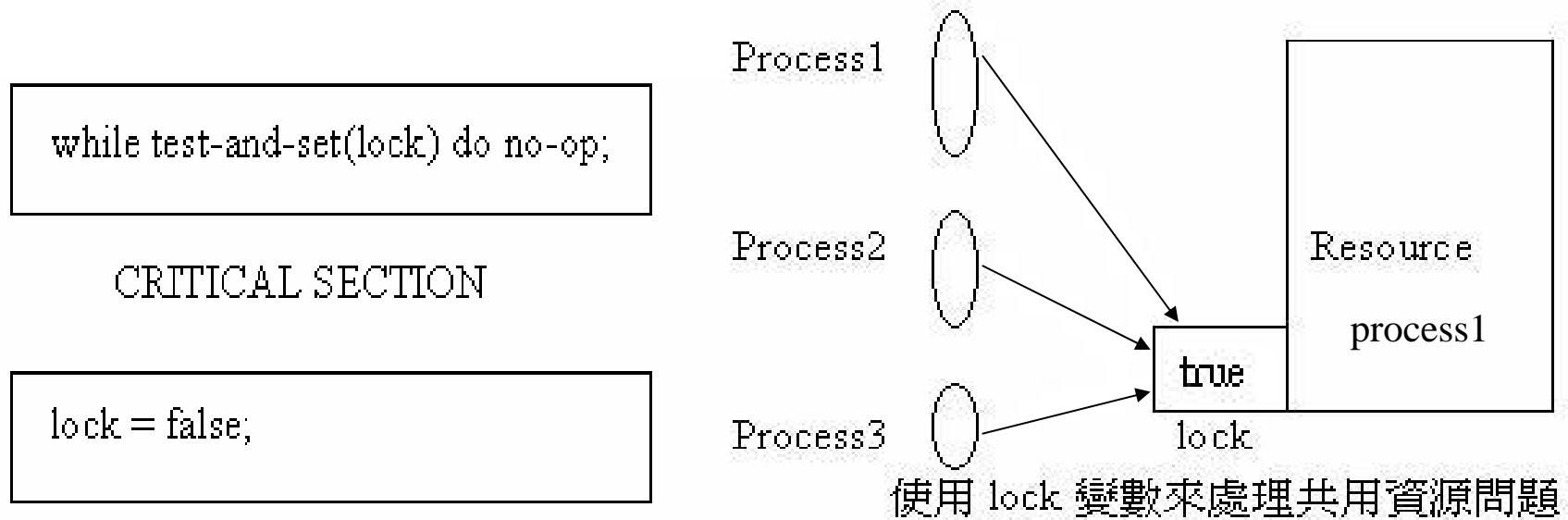
test-and-set()最細小運算

```
function swap( a,b:boolean);  
    boolean temp;  
{  
    temp = a;  
    a = b;  
    b = temp;  
}
```

swap()最細小運算

以硬體解決進入臨界區間的方法(2)

- 使用 test-and-set()
 - 保證互斥，不滿足行進及有限等待條件



使用 test-and-set () 解決進入臨界區間的方法

以硬體解決進入臨界區間的方法(3)

- 使用 swap()
 - 保證互斥，但不保證行進及有限等待

```
key = true;  
repeat  
    swap(lock, key);  
until key = false;
```

CRITICAL SECTION

```
lock = false;
```

使用 swap() 解決進入臨界區間的方法

以硬體解決進入臨界區間的方法(4)

- 符合互斥、行進及有限等待條件之硬體解決方法

```
waiting[i] = true;  
key = true;  
while(waiting[i] and key) do key = test-and-set(lock);  
waiting[i] = false;
```

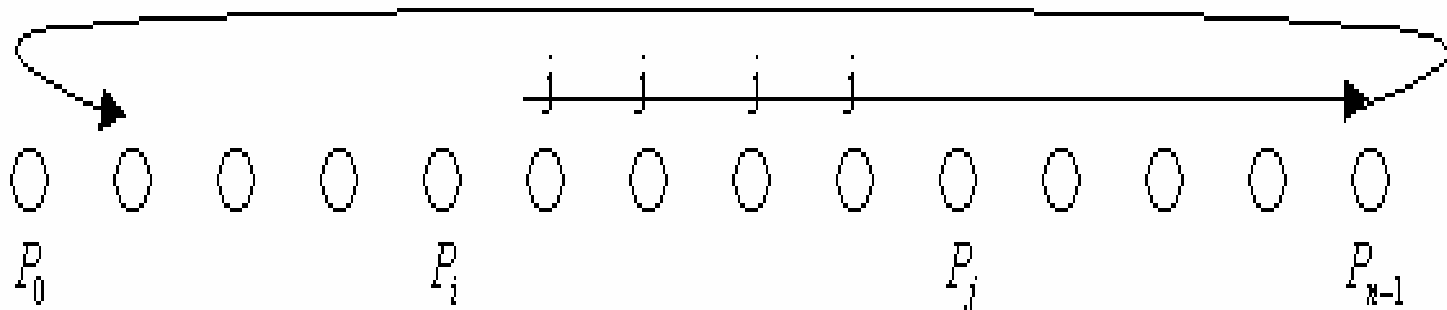
CRITICAL SECTION

```
j = i+1 mod n;  
while( (j < > i) and (not waiting[j])) do j = j+1 mod n;  找到在等待的處理元 j  
if j = i then  
    lock = false;      無人要進入  
else waiting[j] = false;  處理元 j 可以進入臨界區間
```

符合互斥、行進、及有限等待條件之硬體解決方法

以硬體解決進入臨界區間的方法(5)

- 離開while迴圈有二種情況，並各依情況進行不同處
 - 找到 P_j 想進入臨界區間，故將 $waiting[j]$ 設為false，它使得正在while迴圈等待之 P_j 處理元可以進入臨界區間。
 - 當 $P_j = P_i$ 時，代表找遍了所有處理元均不想進入臨界區間，此時亦沒有處理元使用臨界區間，故將lock設為false。



找尋下一個欲進入臨界區間之處理元 P_j

以硬體解決進入臨界區間的方法(6)

- 找到 P_j 想進入臨界區間，故將 $waiting[j]$ 設為false，它使得正在while迴圈等待之 P_j 處理元可以進入臨界區間。

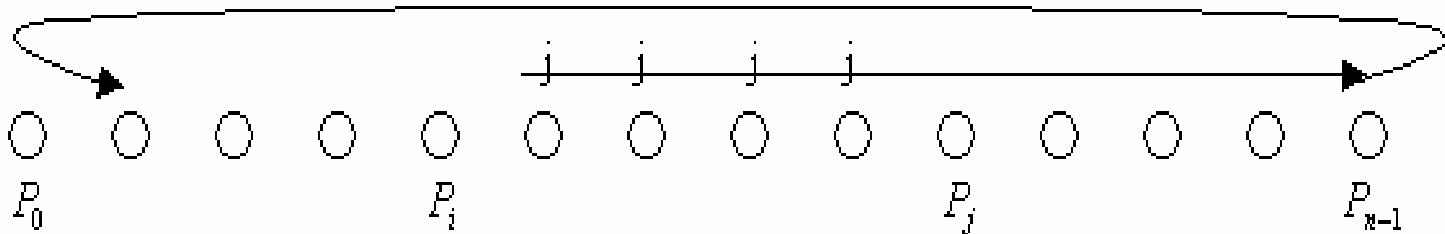
```
j = i+1 mod n;
```

```
while( (j < i) and (not waiting[j])) do j = j+1 mod n; 找到在等待的處理元 j
```

```
if j = i then
```

```
    lock = false;      無人要進入
```

```
else  waiting[j] = false; 處理元 j 可以進入臨界區間
```



找尋下一個欲進入臨界區間之處理元 P_j

以硬體解決進入臨界區間的方法(7)

- 當 $P_j = P_i$ 時，代表找遍了所有處理元均不想進入臨界區間，此時亦沒有處理元使用臨界區間，故將 `lock` 設為 `false`。

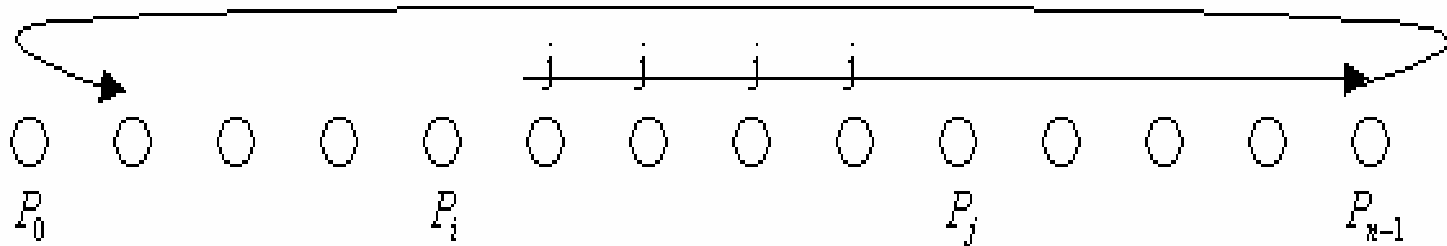
```
j = i+1 mod n;
```

```
while( (j < i) and (not waiting[j])) do j = j+1 mod n; 找到在等待的處理元 j
```

```
if j = i then
```

```
    lock = false;      無人要進入
```

```
else waiting[j] = false; 處理元 j 可以進入臨界區間
```



找尋下一個欲進入臨界區間之處理元 P_j

以硬體解決進入臨界區間的方法(8)

- 互斥
 - 由於lock是一個全域變數，所以當它為false時，僅允許一個處理元進入臨界區間，其他的處理元均因為lock成為true，而無法進入臨界區間。

以硬體解決進入臨界區間的方法(9)

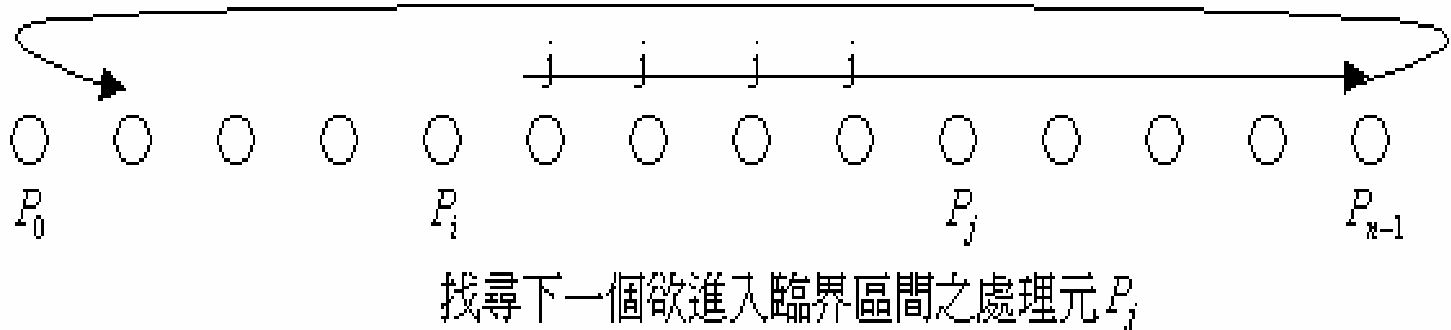
- 行進
 - 某個處理元 P_i 離開臨界區間時，它不是設定lock為false，就是繼續找下一個 P_j 處理元，將waiting[j]設為false；因此可以使得 P_j 處理元進入臨界區間，或使得所有處理元均有機會提出進入臨界區間的要求。

```
j = i+1 mod n;  
while( (j < > i) and (not waiting[j])) do j = j+1 mod n;  找到在等待的處理元 j  
if j = i then  
    lock = false;      無人要進入  
else waiting[j] = false; 處理元 j 可以進入臨界區間
```

以硬體解決進入臨界區間的方法(10)

- 有限等待

- 任何已在while迴圈內等待進入臨界區間的處理元皆會因為於 P_i 離開臨界區間時，依序從 $(i+1, i+2, \dots, n-1, 0, 1, \dots, i-1)$ 次序循環，逐一檢視waiting的值是否為true，找到某一個處理元 P_j 之waiting[j]為true，而讓其進入臨界區間。因此這些在while迴圈等待進入臨界區間之處理元，在最壞情況下，只需等待 $n-1$ 個處理元進入臨界區間之後，便可以輪到進入臨界區間。



號誌 (Semaphores) (1)

6.4 號誌 (Semaphores)

- 號誌被區分為二元號誌 (Binary Semaphore) 及計數號誌 (Counting Semaphore) 。
 - 二元號誌是指其保護變數的值可以為0、1或負值。
 - 計數號誌其保護變數之值可以為任何正負整數值。

號誌 (Semaphores) (2)

- 號誌S是一個保護變數 (Protected Variable) ，它透過P與V這二個最細小運算來達到處理元同步。
- A semaphore is a **protected variable** that can be operated upon only by the **synchronizing primitives P and V**.
- primitive means **atomic operation**, can not be interrupted.
- P與V這二個最細小運算執行時不會作環境切換。

號誌 (Semaphores) (3)

P(S);

CRITICAL SECTION

V(S);

wait(S);

CRITICAL SECTION

signal(S);

使用號誌進入及離開臨界區間

忙碌等待(Busy Waiting)

- 回轉鎖定(Spin Lock)是指不斷由中央處理器測試某個條件是否成立。
- Spin lock wastes CPU cycle.

wait(S)

```
while S<=0 do no-op;  
S=S-1;
```

signal(S)

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

使用 Spin Lock 完成之 wait(S)與 signal(S)

Block and Wakeup

- 保證互斥、行進、及有限等待，而且不使用忙碌等待。
- S變數是負值時，其值代表有多少個處理元在佇列內等待號誌。

wait(S)

```
S = S - 1;  
if S < 0  
{  
    add this process P to S queue;  
    Block(P);  
}
```

signal(S)

```
S = S + 1;  
if S <= 0  
{  
    remove a process P from S queue;  
    Wakeup(P);  
}
```

使用 Block()及 Wakeup()完成之 wait(S)及 signal(S)

號誌的實作問題(1)

- 號誌機制提供最細小運算，因此在執行wait(S)或signal(S)時，是不能作環境切換的。
- 萬一wait(S)及signal(S)是使用軟體來實現，則必須將禁止中斷disable()加在wait(S)及signal(S)的第一個敘述之前，將打開中斷enable()加在wait(S)及signal(S)的最後一個敘述。

號誌的實作問題(2)

- 假如wait(S)及signal(S)是以硬體來實現，則此類號誌不論在一個中央處理器系統或多處理器系統，均能夠完整發揮功能。
- 利用disable()及enable()所實現之wait(S)及signal(S)，在一個中央處理系統是完全沒有問題的；但是在多處理器系統就會發生問題。

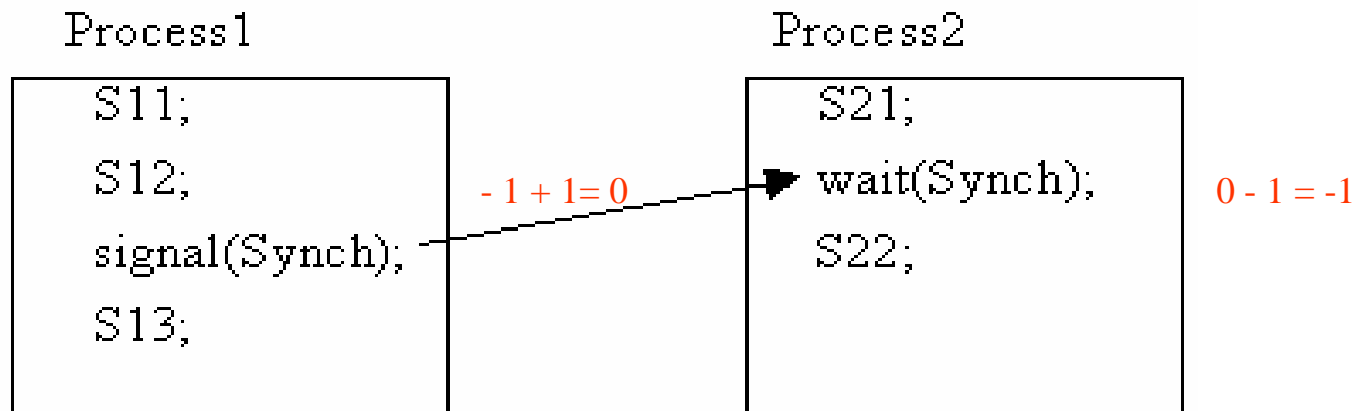
號誌的實作問題(3)

- 原則上wait(S)及signal(S)必須由硬體來實現。
 - 若用軟體來實現，則必須使用軟體的Bakery演算法。
 - 或是使用符合互斥、行進、及有限等待條件之硬體演算法，將wait(S)及signal(S)作互斥保護，才能夠在多處理器系統中使用。

非同步並行處理元進行同步通訊

6.5 號誌之應用及問題

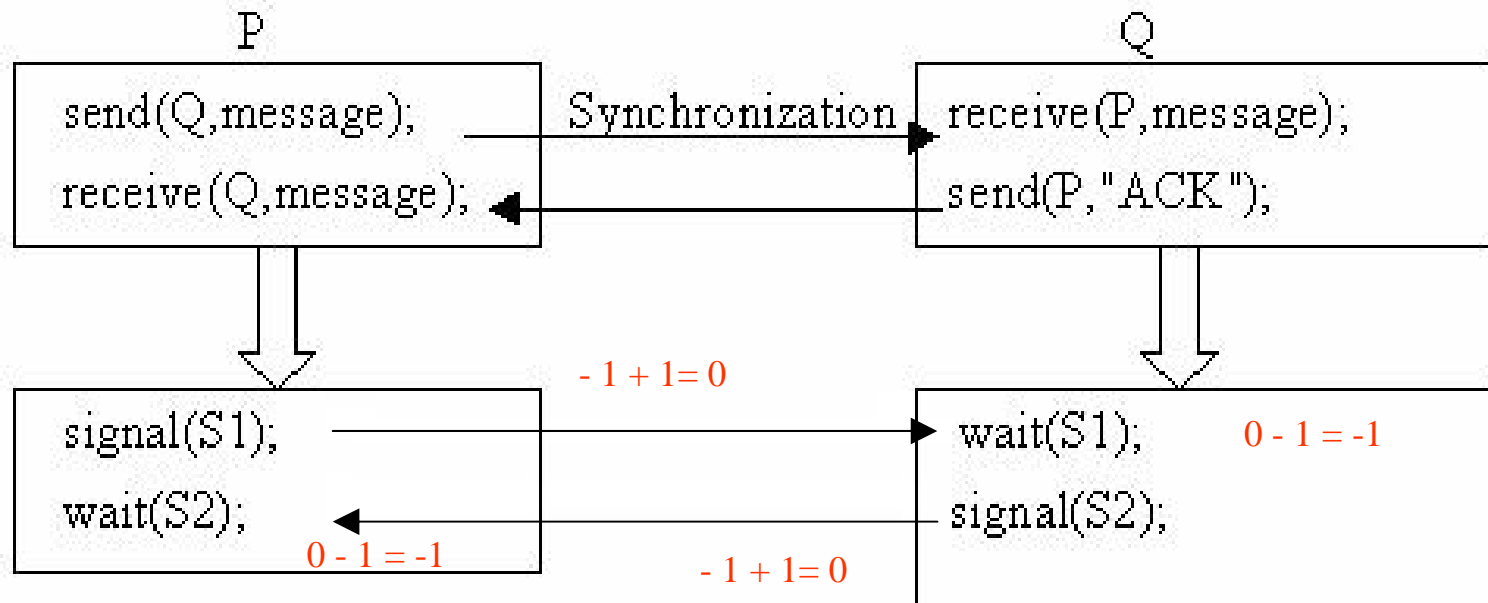
- $\text{Synch} = 0$



二個處理元使用號誌進行同步

處理元內部通訊 (Inter Process Communication)

- $S1 = S2 = 0$



處理元內部通訊示意圖

時間相依錯誤(Time Dependent Error)(1)

- 使用號誌時，wait()及signal()必須由程式撰寫者親自寫在程式中。
- 時間相依錯誤
 - 誤用wait()及signal()將造成資料錯誤的問題，這是因為wait()及signal()的執行時間不正確。

時間相依錯誤(Time Dependent Error)(2)

- Several processes are simultaneously active in their critical section.

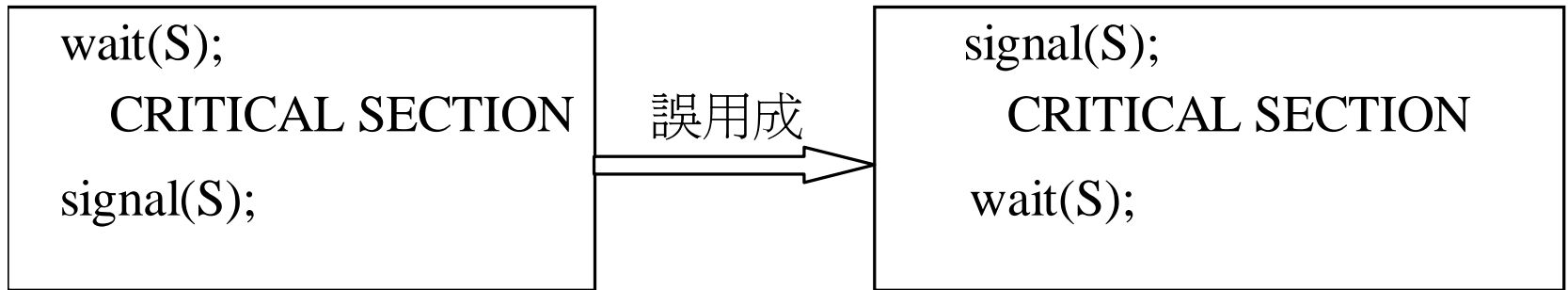


圖 5.23a 將 wait(S)及 signal(S)反過來使用

- 餓死(Starvation):** The process wait indefinitely.

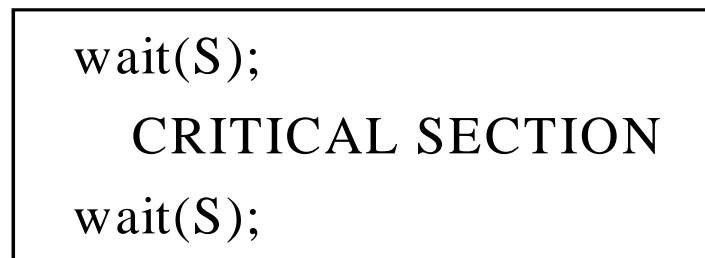


圖 5.23b 進入及離開臨界區間均使用 wait(S)

時間相依錯誤(Time Dependent Error)(3)

- 死結(Deadlock)
 - Two or more processes are waiting indefinitely for an event never occur.
- Deadlock 造成starvation, 但starvation並不一定會造成deadlock.

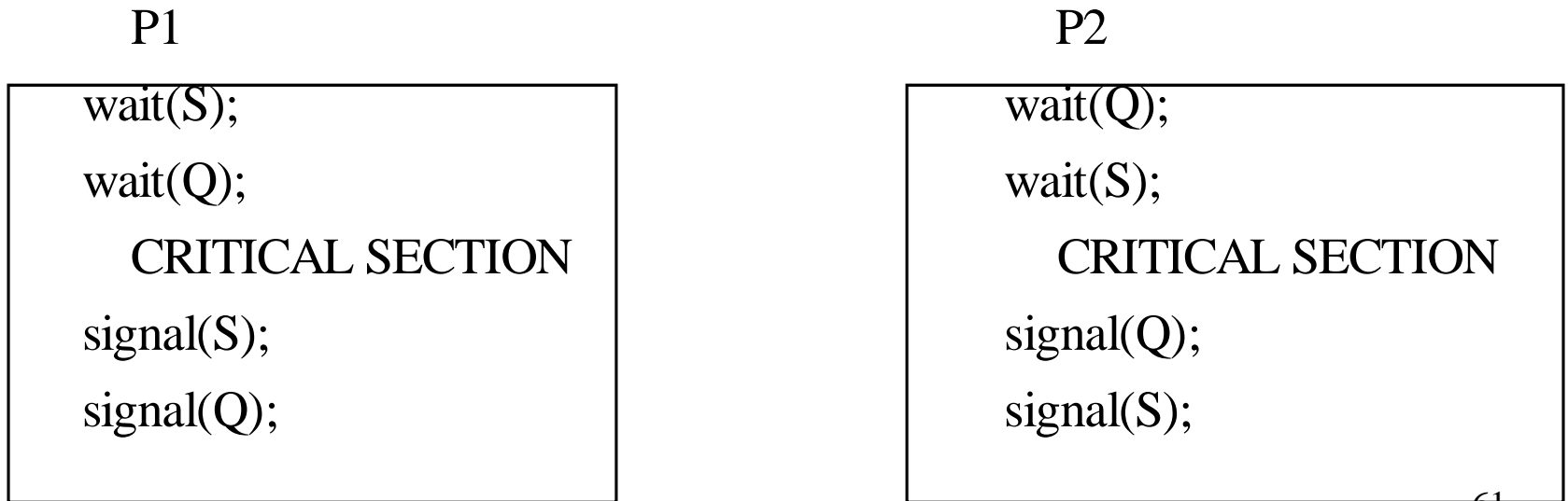
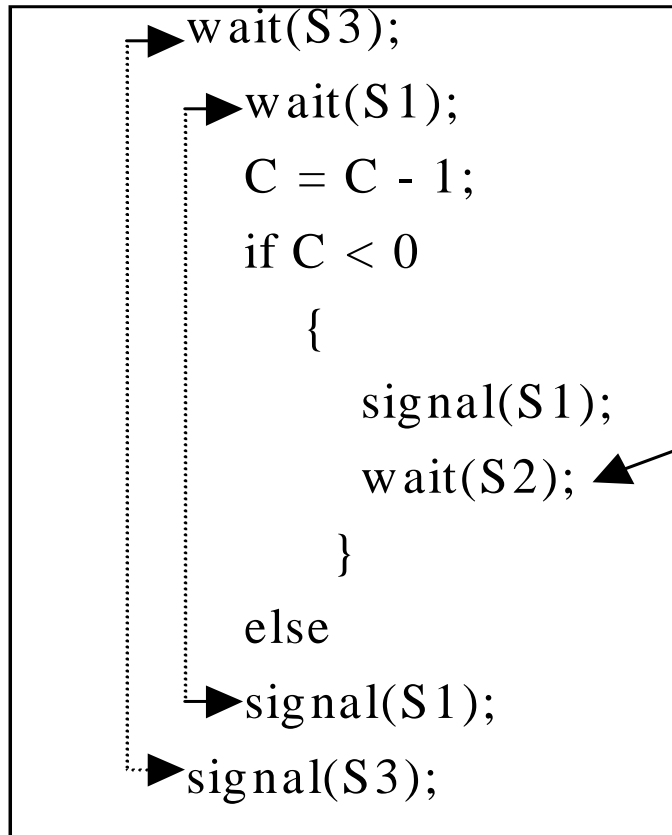


圖 5.23c 造成死結

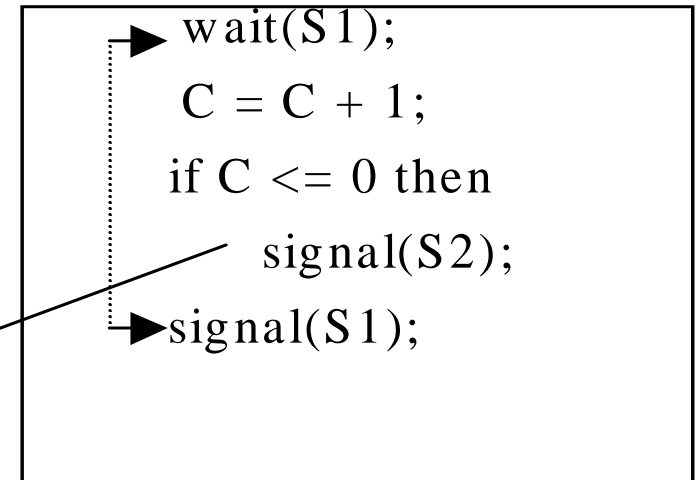
使用二元號誌製作計數號誌(1)

- 計數號誌可以直接以Block()及Wakeup()函數來完

成
 $\text{wait}(\overset{\circ}{C})$



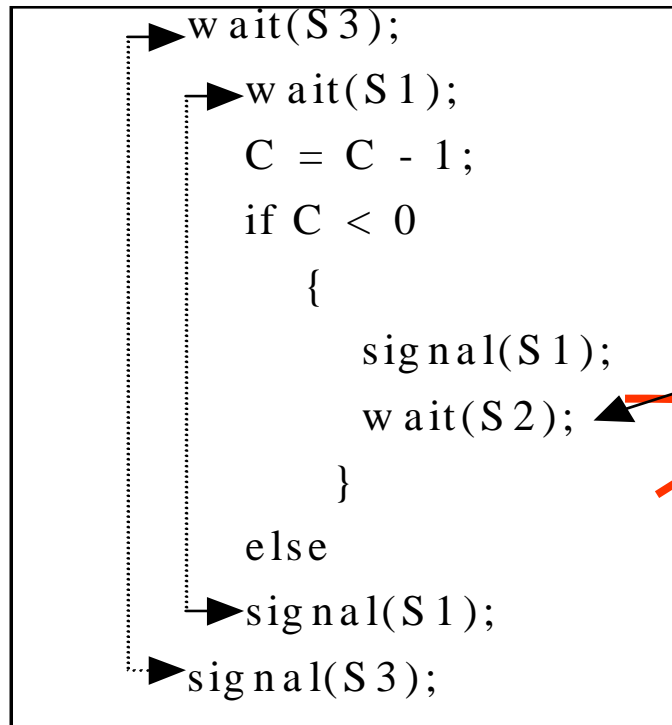
signal(C)



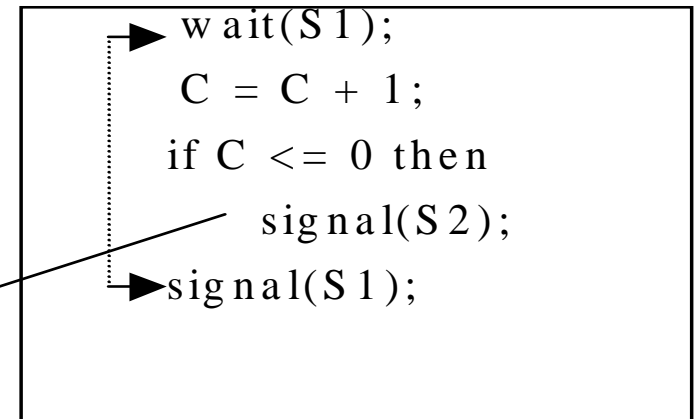
使用二元號誌製作計數號誌(2)

- $S1=1$, 保護C.
- $S3=1$, 保護wait(c).
- $S2=0$, Inter process communication, 用來block and wake up.

wait(C)

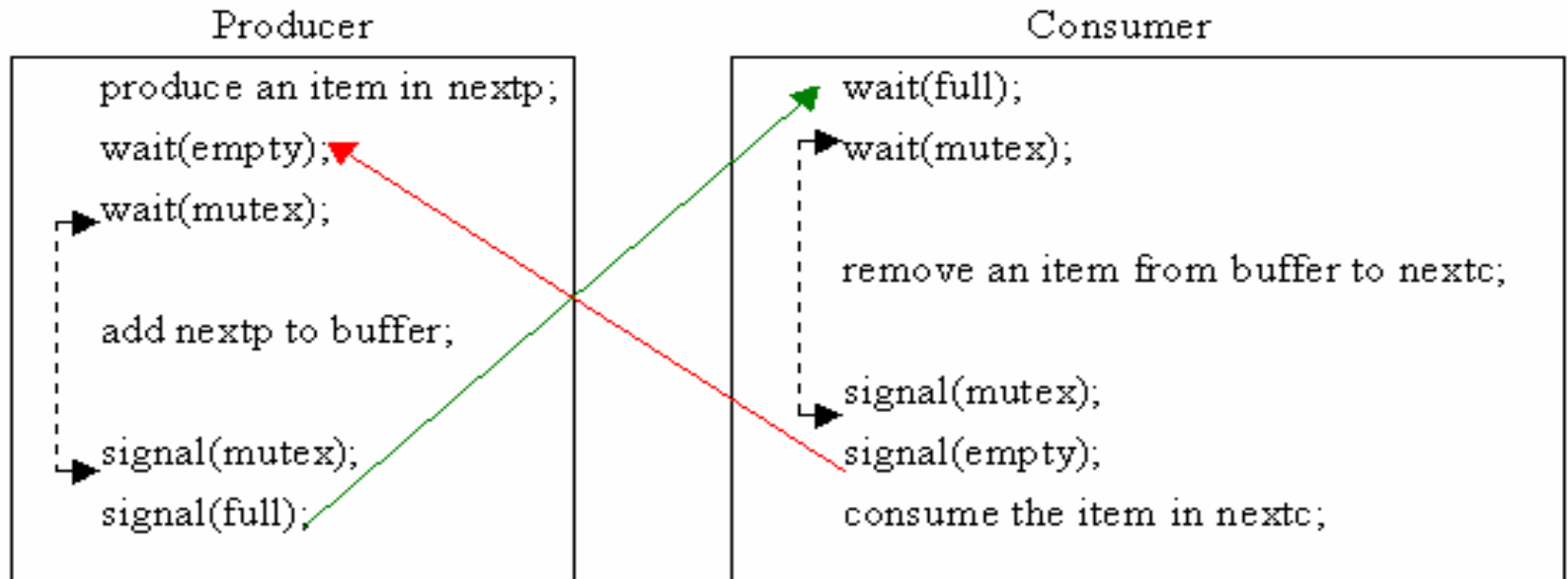


signal(C)



生產者與消費者共用緩衝區問題(1)

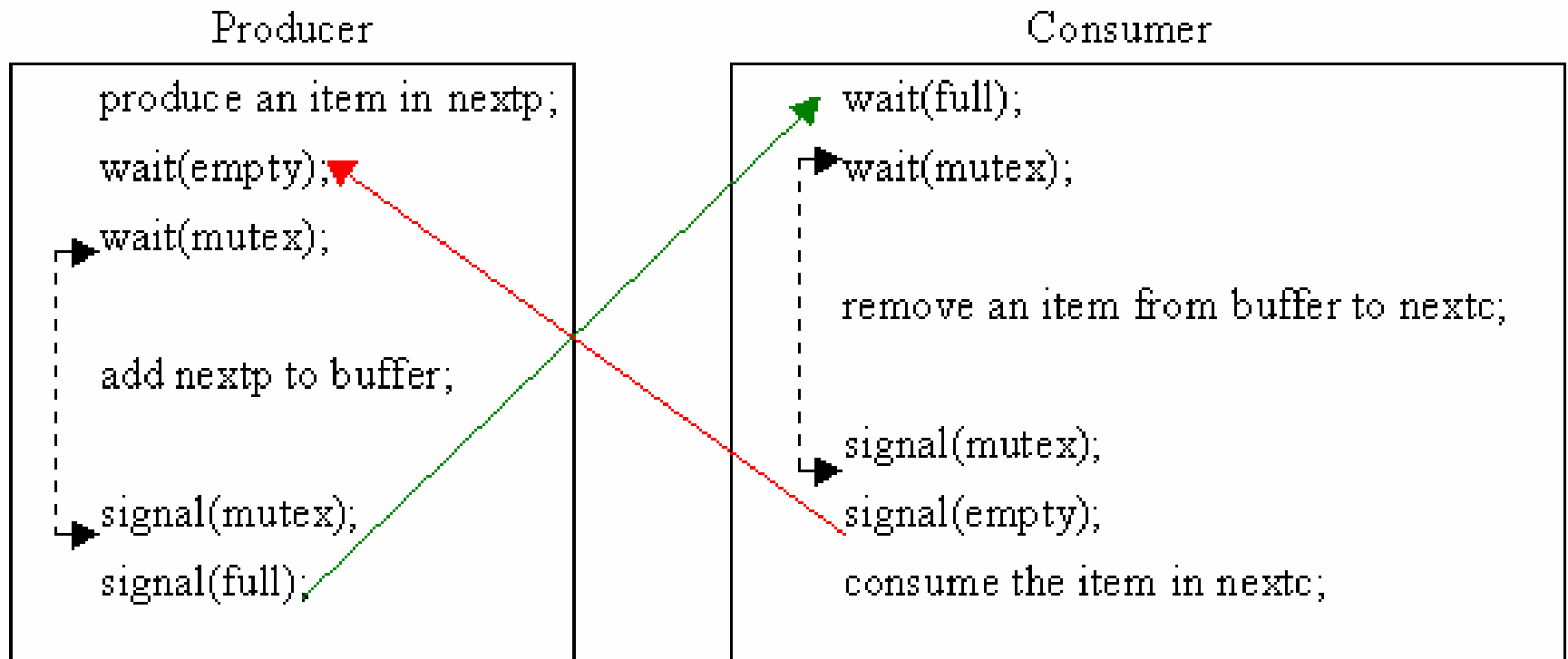
- 生產者處理元與消費者處理元共用 n 個緩衝區。
- 當生產者處理元填滿 n 個緩衝區時，它必須等待消費者處理元取走一些資料後，才能再將資料擺入緩衝區內。
- 若緩衝區已空，則消費者處理元必須等待生產者處理



生產者與消費者處理元共用緩衝區

生產者與消費者共用緩衝區問題(2)

- 使用保護變數mutex = 1，它是用來保護處理元存取緩衝區的互斥性。
- full初始值設為0，用來表示緩衝區已經填滿幾筆資料。
- empty初始值設為n，用來表示緩衝區內還有多少空



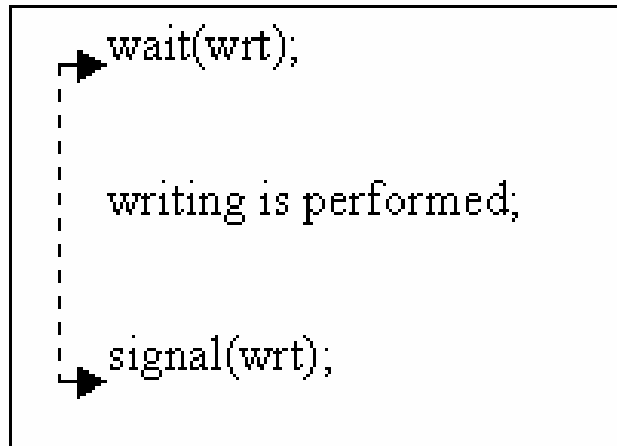
生產者與消費者處理元共用緩衝區

讀取 / 寫入問題(1)

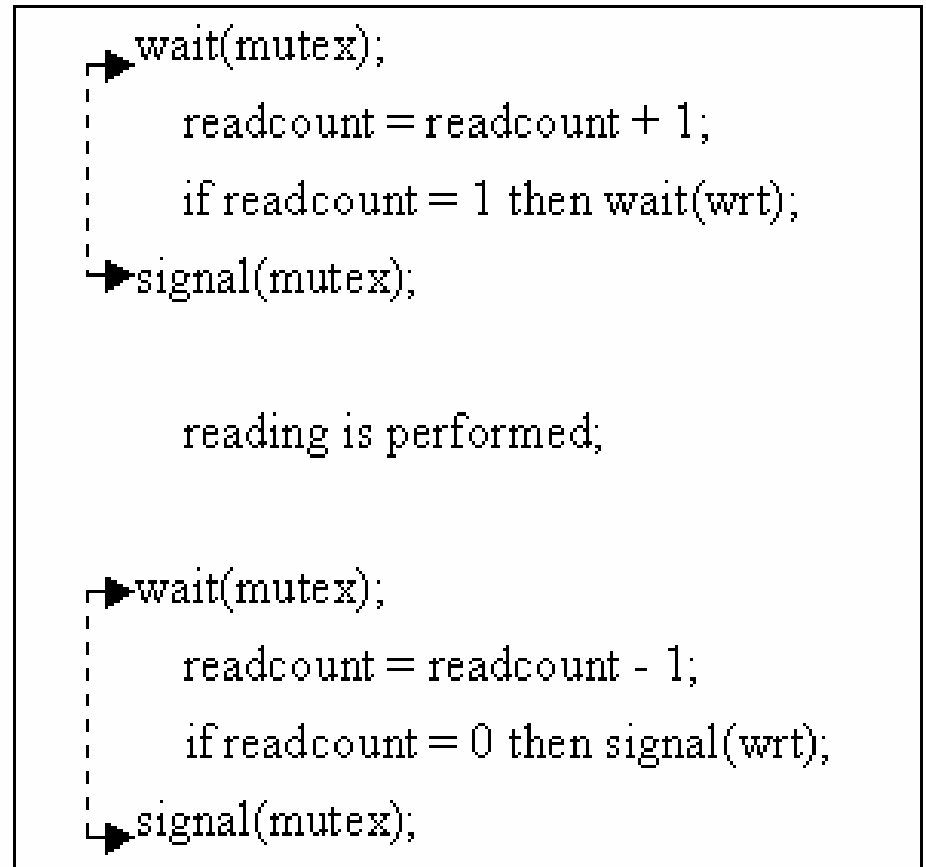
- 檔案是可以**共享讀取 (Shared Read)**，亦即多個處理元均能同時讀取檔案內容。
- 檔案是**寫入互斥 (Exclusive Write)**，亦即某個處理元正在寫入檔案時，其他處理元不能同時寫入。
- 若有一個處理元正在寫入檔案，其他處理元亦不能同時讀取此檔案。

讀取 / 寫入問題(2)

Writer



Reader

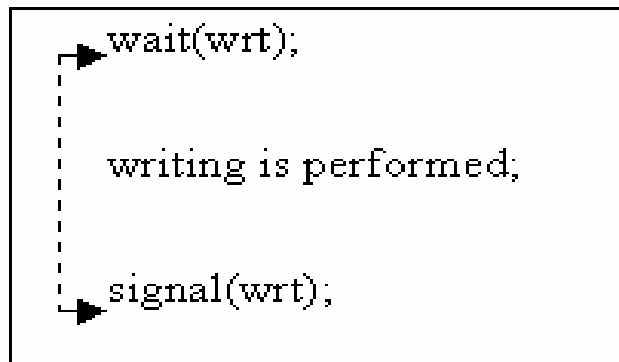


共享讀取 / 寫入互斥之程式範例

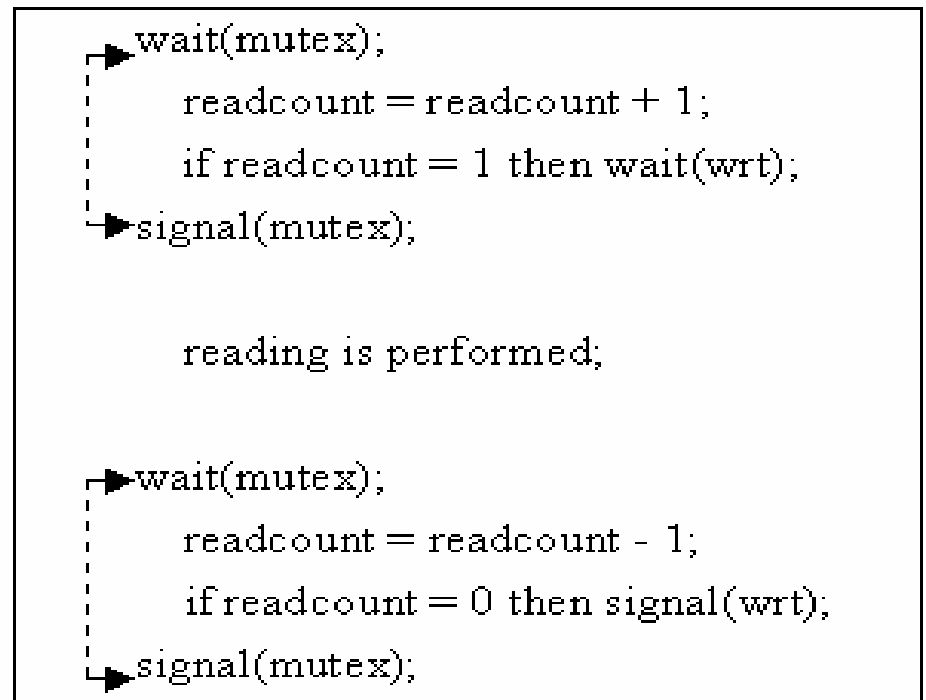
讀取 / 寫入問題(3)

- 同時多個處理元讀取資料
 - 第一個讀取資料處理元會呼叫wait(wrt)，建立寫入互斥條件，不會有其他寫入處理元可以進入臨界區間內。但可以同時有多個讀取資料處理元在臨界區間內。

Writer



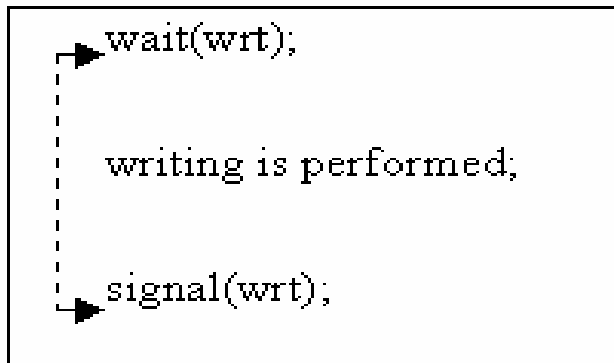
Reader



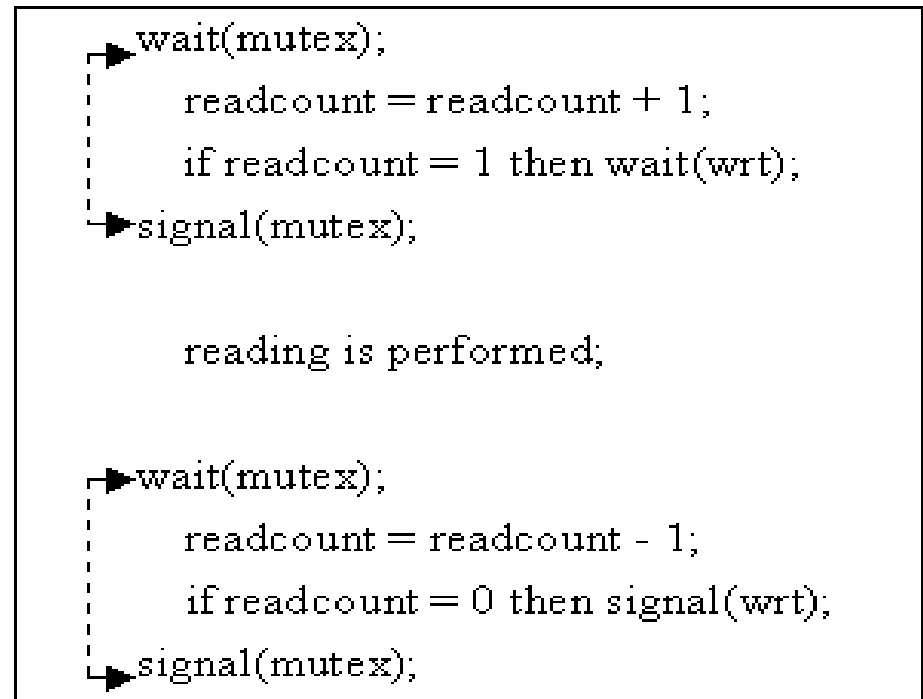
讀取 / 寫入問題(4)

- 一個處理元寫入資料
 - 當一個處理元正在寫入資料時，它利用wait(wrt)建立寫入互斥條件，故不會有第二個寫入資料處理元進入臨界區間，另外第一個讀取資料處理元在呼叫wait(wrt)時，必須等待寫入資料處理元離開臨界區間，才有機會進入臨界區間讀取資料。

Writer

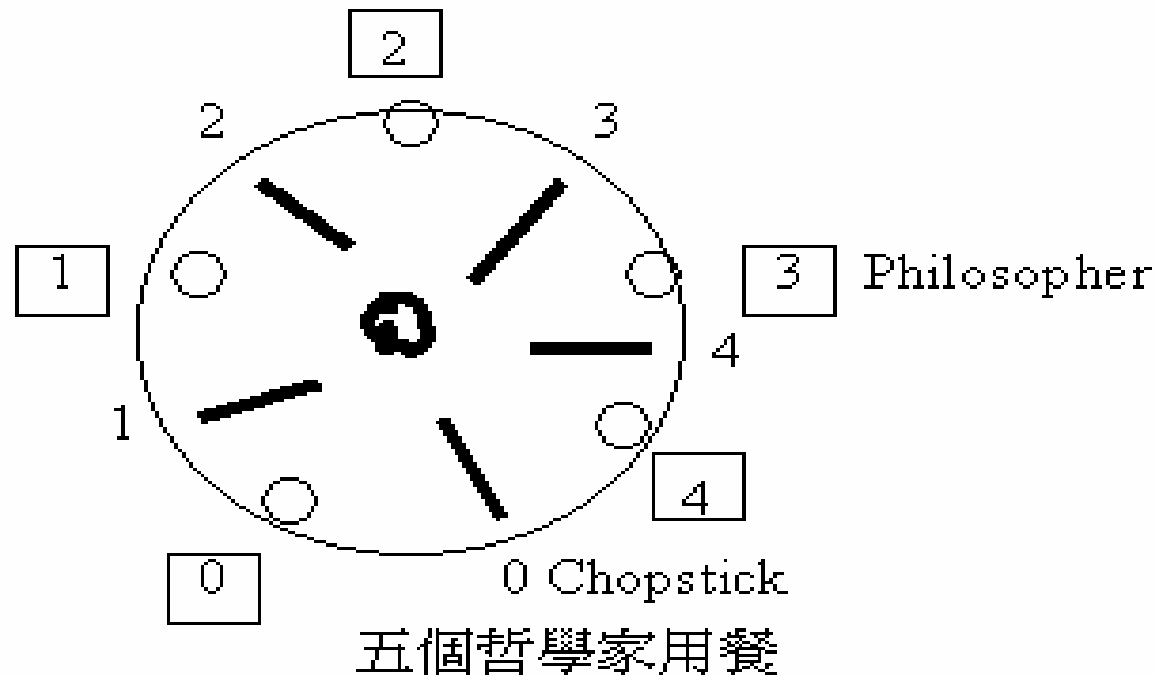


Reader



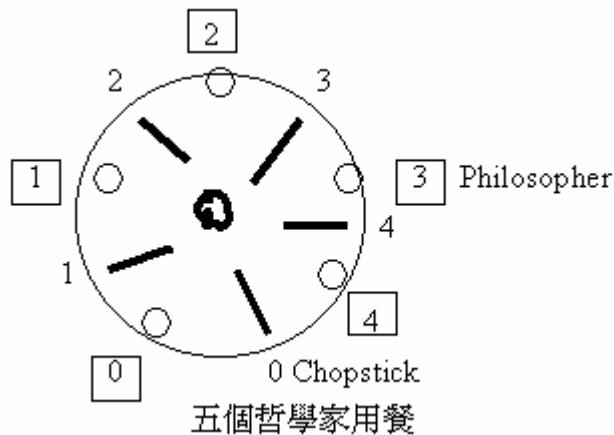
哲學家用餐(Dining Philosophers)問題(1)

- 五個哲學家坐在圓形餐桌前，每人面前有一碗飯，且左右各有一枝筷子，他們的動作只有吃飯與思考。當哲學家欲吃飯時，他必須拿取左右筷子，才能夠吃飯；問題是筷子只有五枝，必須與旁邊的哲學家共用筷子，所以我們必須解決共享資源問題。



哲學家用餐(Dining Philosophers)問題(2)

- chopstick[i]之初始值設為1。
- 若五個哲學家同時感覺飢餓，則可能造成每個哲學家各自呼叫wait(chopstick[i])，且取得右邊筷子，卻在呼叫wait(chopstick[i+1 mod 5])時無法取得左邊筷子，而造成死結(Deadlock)的問題。

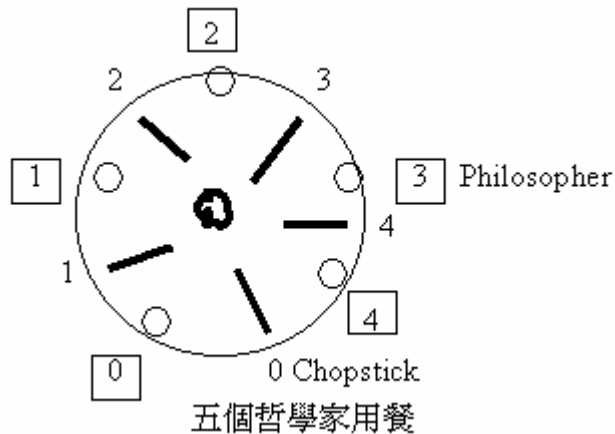


```
wait(chopstick[i]);  
wait(chopstick[i+1 mod 5]);  
  
eat;  
  
signal(chopstick[i]);  
signal(chopstick[i+1 mod 5]);  
  
think;
```

哲學家用餐(Dining Philosophers)問題(3)

- 死結 (Deadlock) 的問題
 - Waiting to get another resource
- Solution
 - Allocate two resources simultaneous(同時取用兩根筷子)。
 - Odd philosopher get left then right, and even philosopher get right then left.(編號為奇數的哲學家先取左邊筷子，再取右邊筷子，而編號為偶數的哲學家先取右邊筷子，再取左邊筷子)。
 - Give up one philosopher(放棄一個處理元不讓它執行)。

哲學家用餐(Dining Philosophers)問題(4)



```
state[i] = THINKING;
s[i] = 0;
pickup(i:integer)
{
    wait(mutex);           進入臨界區間
    state[i] = HUNGRY;      設定哲學家為饑餓
    test(i);                嘗試拿二根筷子
    signal(mutex);          離開臨界區間
    if state[i] <> EATING then wait(s[i]);  拿不到筷子則等待
}
putdown(i:integer)
{
    wait(mutex);           進入臨界區間
    state[i] = THINKING;   設定哲學家為思考
    test(i mod 5);         查看右邊的哲學家是否可以用餐
    test(i+1 mod 5);       查看左邊的哲學家是否可以用餐
    signal(mutex);         離開臨界區間
}
test(i:integer)
{
    if ((state[i] = HUNGRY) and (state[i mod 5] <> EATING) and
        (state[i+1 mod 5] <> EATING))  是否相鄰之哲學家均不用餐
    {
        state[i] = EATING;
        signal(s[i]);
    }
}
```

哲學家用餐(Dining Philosophers)問題(5)

- 餓死 (Starvation) 的問題
 - Wait resource indefinitely
- Solution
 - To use waiting queue(使用等待佇列，讓想用餐的哲學家依序註冊排隊)。
 - According waiting time, dynamic set process priority(讓等待很久一直無法用餐的哲學家，提升他的優先等級，讓用過多次餐的哲學家降低優先等級)。
 - Time out interrupt(讓等很久的哲學家，以時間逾時 (Timeout) 方式，優先獲得服務)。

哲學家用餐(Dining Philosophers)問題(6)

- 死結一定會餓死。
- 餓死並不一定由死結發生的。
- 解決死結並不一定解決餓死。

理髮師理髮問題

#define CHAIR 5	定義五張等待椅子
int customers = 0;	
int babers = 0;	設定 customers, barbers, mutex 之 semaphores
int mutex = 1;	
int waiting = 0;	有多少顧客在等待
barber()	
{	
while (TRUE)	在迴圈內等待服務
{	
wait(customers);	若顧客為零，則去睡覺
wait(mutex);	進入臨界區間
waiting = waiting - 1;	顧客人數減一
signal(barbers);	理髮師準備理髮
signal(mutex);	離開臨界區間
cut-hair();	理髮
}	
}	
customer()	
{	
wait(mutex);	進入臨界區間
if (waiting < CHAIRS)	測試是否有足夠椅子
{	
waiting = waiting + 1;	顧客人數加一
signal(customers);	叫醒理髮師
signal(mutex);	離開臨界區間
wait(barbers);	等待理髮師，若無理髮師，則在椅子上等待
get-haircut();	理髮
}	
else	
signal(mutex);	沒有椅子，離開臨界區間，放棄理髮
}	

臨界區域(Critical Region)(1)

6.6 臨界區域及監督器

- 臨界區域並不一定可以完全去除時間相依錯誤。

```
S=1;  
wait(S);  
    counter = counter + 1;  
signal(S);
```

圖 5.31a 使用 wait(S)及 signal(S)

```
shared S;  
region S  
    do counter = counter + 1;
```

圖 5.31b 改用臨界區域

```
shared S = 1;  
wait(S);  
    counter = counter + 1;  
signal(S);
```

圖 5.31c 使用臨界區域程式經過編譯後的程式碼

圖 5.31 使用臨界區域

臨界區域(Critical Region)(2)

- 利用臨界區域結構的語法，可以很容易的撰寫進入臨界區間的程式，而且直接透過編譯器編譯，便能夠達到在臨界區間內互斥性的需求。

Producer

```
region buffer when count <= n
do {
    pool[in] = nextp;
    in = in+1 mod n;
    count = count + 1;
}
```

Consumer

```
region buffer when count > 0
do {
    nextc = pool[out];
    out = out+1 mod n;
    count = count - 1;
}
```

以臨界區域結構撰寫生產者與消費者共用緩衝區的程式

監督器(Monitor)(1)

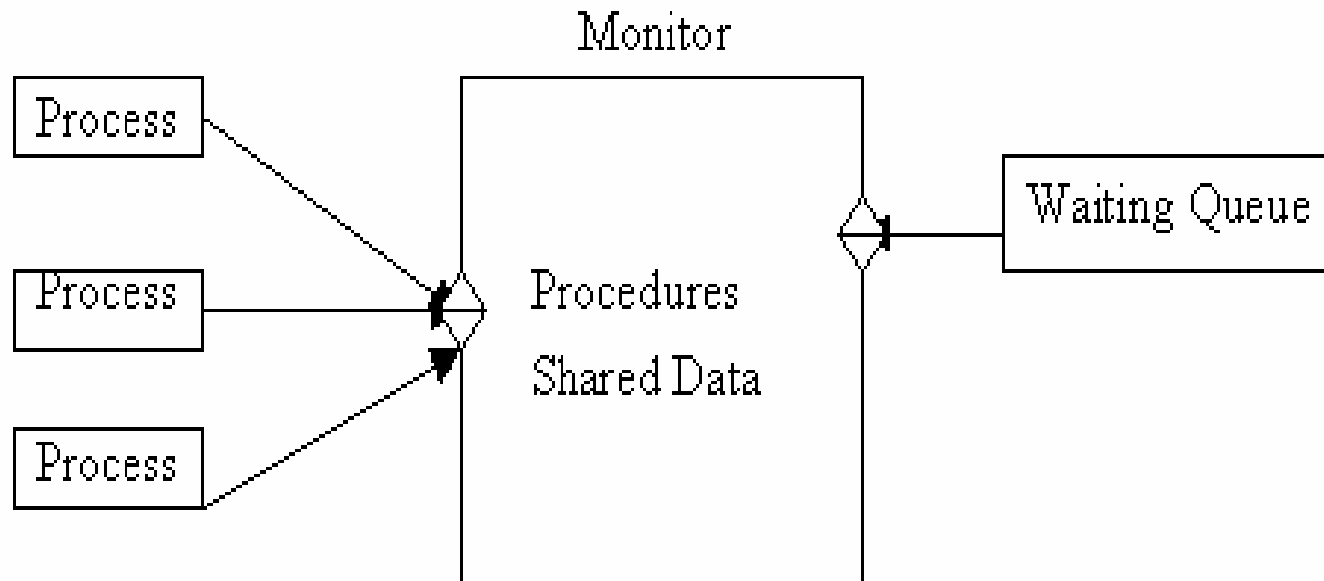
- 監督器是另一種以程式語言來達到處理元同步作業的機制。
- 可以說監督器是作業系統的一種結構，它裡面包含資料及程序（Procedure），並用來管理處理元之同步。
- 監督器保證任何存取其資料之處理元是互斥的。
- 使用懸置（Blocking）及喚醒（Wakeup）之機制來安排處理元進入臨界區間。

監督器(Monitor)(2)

- A monitor is an operating systems construct that contains both the **data and procedures** for handling **process synchronization**. It guarantees mutually exclusion access to **shared critical data**, and provides a convenient mechanism for **blocking and wake up** processes. Many processes may want to enter the monitor at various times, but **mutually exclusion** is rigidly enforced at the monitor boundary.

監督器(Monitor)(3)

- 任何要使用監督器內部資料及運算的處理元，必須經由監督器入口進入，且對資料的所有運算，均由監督器來執行，處理元是無法直接接觸到資料的。



監督器保護資料及程序的示意圖

監督器(Monitor)(4)

```

type dp = monitor
state: array[0..4] of (THINKING,HUNGRY,EATING); 初始值為 THINKING
self: array[0..4] of condition;
pickup(i:integer)
{
    state[i] = HUNGRY;      設定哲學家為饑餓
    test(i);                嘗試拿二根筷子
    if state[i] <> EATING then self[i].wait;  拿不到筷子則等待
}
putdown(i:integer)
{
    state[i] = THINKING;    設定哲學家為思考
    test(i mod 5);          查看右邊的哲學家是否可以用餐
    test(i+1 mod 5);        查看左邊的哲學家是否可以用餐
}
test(i:integer)
{
    if ((state[i] = HUNGRY) and (state[i mod 5] <> EATING) and
        (state[i+1 mod 5] <> EATING)) 是否相鄰之哲學家均不用餐
    {
        state[i]=EATING;
        self[i].signal;
    }
}
    
```

使用監督器撰寫哲學家用餐的程式

```

state[i] = THINKING;
s[i] = 0;
pickup(i:integer)
{
    wait(mutex);          進入臨界區間
    state[i] = HUNGRY;    設定哲學家為饑餓
    test(i);              嘗試拿二根筷子
    signal(mutex);        離開臨界區間
    if state[i] <> EATING then wait(s[i]);
}
putdown(i:integer)
{
    wait(mutex);          進入臨界區間
    state[i] = THINKING;  設定哲學家為思考
    test(i mod 5);        查看右邊的哲學家是否可以用餐
    test(i+1 mod 5);      查看左邊的哲學家是否可以用餐
    signal(mutex);        離開臨界區間
}
test(i:integer)
{
    if ((state[i] = HUNGRY) and (state[i mod 5] <> EATING) and
        (state[i+1 mod 5] <> EATING)) 是否相鄰之哲學家均不用餐
    {
        state[i] = EATING;
        signal(s[i]);
    }
}
    
```

監督器(Monitor)(5)

```
type PC = monitor
condition full, empty;
integer count = 0;
integer N = 10;
enter()
{
    if count = N then full.wait;
    enter-item;
    count = count + 1;
    if count = 1 then empty.signal;
}
remove()
{
    if count = 0 then empty.wait;
    remove item;
    count = count - 1;
    if count = N-1 then full.signal;
}
```

Producer Process

```
produce-item;
PC.enter;
```

Consumer Process

```
PC.remove;
consume-item;
```

使用監督器撰寫生產者與消費者共用緩衝區的程式

號誌及監督器之比較

- 當處理元使用號誌時，是各作各的，但彼此利用號誌來達到互斥及同步的目的。
- 監督器則總管其內部之程序及資料，任何使用其內部程序或資料的處理元，都必須經過監督器總管及安排，以達到互斥的需求。
- 使用監督器來撰寫處理元同步工作，相對容易很多。當然了，從作業系統的角度來看，監督器結構本身可以使用號誌來完成。

Java的同步機制

- Synchronized() 相當於
wait(mutex)
:
signal(mutex)
- wait()
notify()
相當於
wait(s)
signal(s)
但是Java沒有protect variable (s)
- 有Critical Region and Monitor機制

Java監督器(Monitor)

```
Public synchronized void enter(Object item) {  
    while (count == BUFFER_SIZE) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    notify();  
}
```

自動保護

```
Public synchronized Object remove() {  
    Object item;  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    notify();  
    return item;  
}
```

Java臨界區域(Critical Region)

```
Object mutexLock;  
:  
public void someMethod()  
{    SomeStatements();  
:  
    synchronized(mutexLock) {  
        criticalSection();  
:  
    }  
    SomeStatements();  
}
```

Java號誌(Semaphore)

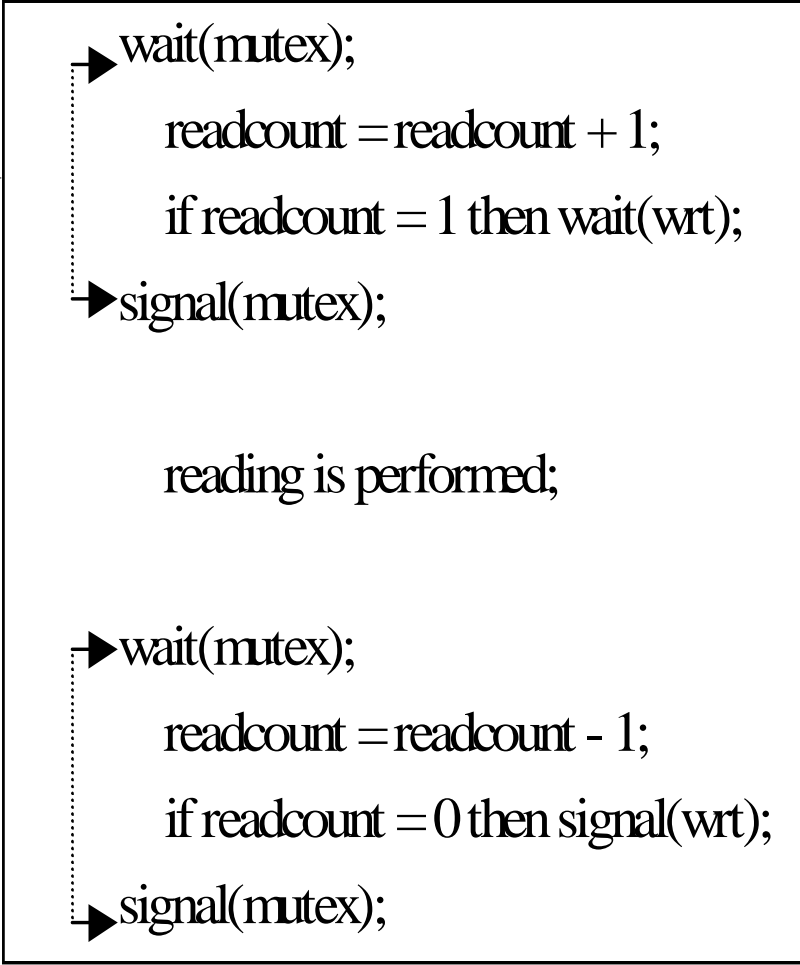
```
Public class Semaphore {  
    public Semaphore(int v) {  
        value = v;  
    }  
  
    Public synchronized void P() {  
        while (value <= 0) {  
            try {  
                wait();  
            } catch (InterruptedException e) {} }  
        value--;  
    }  
  
    Public synchronized void V() {  
        ++value;  
        notify();  
    }  
}
```


以Java實作讀取/寫入問題(1)

```
Public synchronized int startRead() {  
    while (Writing == true) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    ++readCount;  
    if (readCount == 1)  
        Reading = true;  
    return readCount;  
}
```

```
Public synchronized int endRead() {  
    --readCount;  
    if (readCount == 0)  
        Reading = false;  
    notifyAll();  
    return readCount;  
}
```

Reader



```
graph TD; A[wait(mutex);] --> B[readcount = readcount + 1;]; B --> C["if readcount = 1 then wait(wrt);"]; C --> A; A --> D[signal(mutex);]; D --> E[reading is performed;]; E --> F[wait(mutex);]; F --> G[readcount = readcount - 1;]; G --> H["if readcount = 0 then signal(wrt);"]; H --> F; F --> I[signal(mutex);]; I --> J[ ];
```

wait(mutex);

readcount = readcount + 1;

if readcount = 1 then wait(wrt);

signal(mutex);

reading is performed;

wait(mutex);

readcount = readcount - 1;

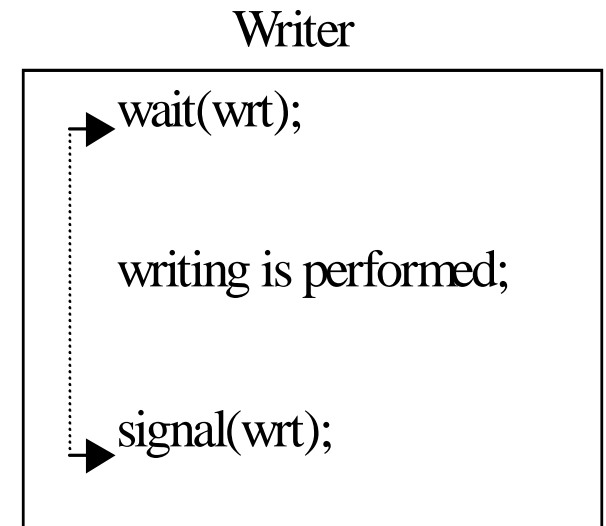
if readcount = 0 then signal(wrt);

signal(mutex);

以Java實作讀取/寫入問題(2)

```
Public synchronized int startWrite() {  
    while (Writing == true || Reading == true) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    Writing = true;  
}
```

```
Public synchronized int endWrite() {  
    Writing = false;  
    notifyAll();  
}
```



以號誌來實現監督器內的程序

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

圖 5.36a 任何一個程序 F 被編譯後的內容

```
wait(mutex);  
    state[i] = HUNGRY;  
    test(i);  
    if state[i] < > EATING then self[i].wait;  
if next-count > 0  
then  signal(next)  
else  signal(mutex);
```

圖 5.36b pickup()程序被編譯後，以號誌來實現的情形

圖 5.36 以號誌來實現監督器內的程序

以號誌來實現監督器內的條件變數x

```
x-count = x-count + 1;  
if next-count > 0  
then  signal(next)  
else  signal(mutex);  
wait(x-sem);  
x-count = x-count - 1;
```

圖 5.37a 以號誌實現 x.wait 的程式

```
if x-count > 0  
{  
    next-count = next-count + 1;  
    signal(x-sem);  
    wait(next);  
    next-count = next-count - 1;  
}
```

圖 5.37b 以號誌實現 x.signal 的程式

圖 5.37 以號誌來實現監督器內的條件變數 x

使用監督器所撰寫之處理元存取共用資源的程式範例(1)

- 此程式中已透過監督器保證共用資源互斥性；任何一個處理元欲使用資源僅需呼叫R.acquire，使用完畢之後僅需呼叫R.release。

```
type R = monitor
boolean busy = false;
condition x;
acquire(time:integer)
{
    if busy then x.wait(time);
    busy = true;
}
release()
{
    busy = false;
    x.signal;
}
```

以監督器撰寫處理元存取共用資源的程式範例

使用監督器所撰寫之處理元存取共用資源的程式範例(2)

- 臨界區域及監督器可以解決部份時間相依錯誤問題，但在實務上仍然有無法完全解決的困擾。
- 處理元不呼叫監督器，便直接去存取資源。
- 處理元取得資源後，忘記歸還資源。
- 處理元將R. acquire及R. release用反了！
- 處理元呼叫R. acquire二次，而沒有在第一次使用完畢後呼叫R. release。
- 處理元歸還某個資源，但未使用R. acquire 要求使用此資源。

Conflict Serializable

6.7 可循序化 (Serializable)

- 可循序化的排程 (**Serializable Schedule**)
 - 處理元不是依循序執行，而是依中央處理器排程次序執行，其結果與循序執行相同，則此執行次序是可循序化 (**Serializable**) 的。
 - The concurrent execution of transactions must be equivalent to the case where these transactions executed **serially** in some arbitrary order.
- Conflict Serializable
 - If a schedule S can be transformed into a serial schedule S' by a series of **swaps** of **non-conflicting operations**, we say that a schedule S is conflict serializable.

Locking Protocol

- Shared Lock(SLOCK)
 - 可以共享讀取 (Shared Read) ，亦即多個處理元均能同時讀取此內容。
- Exclusive Lock(XLOCK)
 - 是寫入互斥 (Exclusive Write) ，亦即某個處理元正在寫入此內容時，其他處理元不能同時寫入此內容，亦不能同時讀取此內容。
- Exclusive locking whole transaction can be used to enforce serializability. However, too restrictive.

Two Phase Locking

- Growing phase
 - A transaction may obtain locks, but may not release any lock.
- Shrinking phase
 - A transaction may release locks, but may not obtain any new locks.
- Two phase locking protocol ensures conflict serializability. It does not ensure freedom from deadlock.
- Time stamp ordering scheme ensures conflict serializability and deadlock.

Java的同步機制

- Synchronized() 相當於
wait(mutex)
:
signal(mutex)
- wait()
notify()
相當於
wait(s)
signal(s)
但是Java沒有protect variable (s)
- 有Critical Region and Monitor機制

Java監督器(Monitor)

```
Public synchronized void enter(Object item) {  
    while (count == BUFFER_SIZE) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    ++count;  
    buffer[in] = item;  
    in = (in + 1) % BUFFER_SIZE;  
    notify();  
}
```

自動保護

```
Public synchronized Object remove() {  
    Object item;  
    while (count == 0) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    --count;  
    item = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    notify();  
    return item;  
}
```

Java臨界區域(Critical Region)

```
Object mutexLock;  
:  
public void someMethod()  
{    SomeStatements();  
:  
    synchronized(mutexLock) {  
        criticalSection();  
:  
    }  
    SomeStatements();  
}
```

Java號誌(Semaphore)

```
Public class Semaphore {
    public Semaphore(int v) {
        value = v;
    }

    Public synchronized void P() {
        while (value <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {} }
        value--;
    }

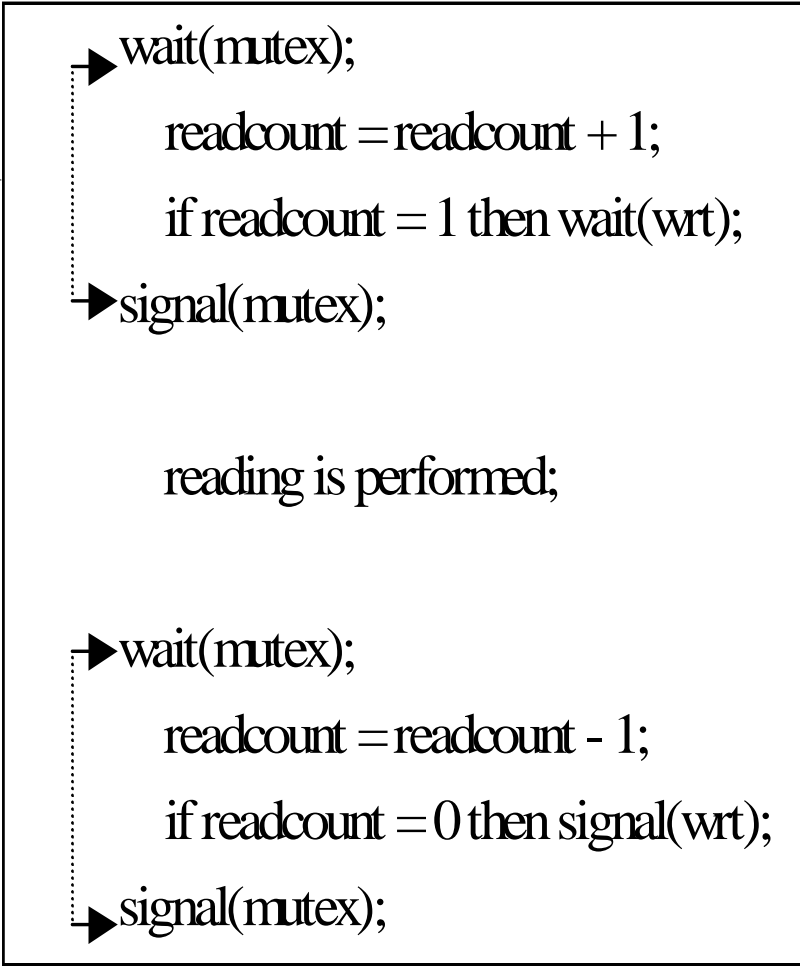
    Public synchronized void V() {
        ++value;
        notify();
    }
}
```

以Java實作讀取/寫入問題(1)

```
Public synchronized int startRead() {  
    while (Writing == true) {  
        try {  
            wait();  
        } catch(InterruptedException e) {}  
    }  
    ++readCount;  
    if (readCount == 1)  
        Reading = true;  
    return readCount;  
}
```

```
Public synchronized int endRead() {  
    --readCount;  
    if (readCount == 0)  
        Reading = false;  
    notifyAll();  
    return readCount }  
}
```

Reader



```
graph TD; A[wait(mutex);] --> B[readcount = readcount + 1;]; B --> C["if readcount = 1 then wait(wrt);"]; C --> A; A --> D[signal(mutex);]; D --> E[reading is performed;]; E --> F[wait(mutex);]; F --> G[readcount = readcount - 1;]; G --> H["if readcount = 0 then signal(wrt);"]; H --> F; F --> I[signal(mutex);]; I --> J[ ]; style J fill:none,stroke:none;
```

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);

以Java實作讀取/寫入問題(2)

```
Public synchronized int startWrite() {  
    while (Writing == true || Reading == true) {  
        try {  
            wait();  
        } catch (InterruptedException e) {}  
    }  
    Writing = true;  
}
```

```
Public synchronized int endWrite() {  
    Writing = false;  
    notifyAll();  
}
```

