同步與死結

資科系
林偉川

行程同步

- 多行程的系統，藉由互相合作以解決複雜的問題，使用執行緒來共享資料，同時存取共享的變數，可能會導致資料不一致，造成錯誤，因此需要同步機制來解決。
- 如何保證行程間執行次序的正確，與確保所共用資料內容一致的同步機制，即為行程同步主要的內容。

2

行程死結

- 行程也會競爭系統中數量有限的資源，當某行程所要求的資源正被其他行程使用時，該行程就必須要等待，而其他行程如果接著也等待該行程先前持有的資源，兩行程因此無限期地相互等待下去，此種情形即稱為死結。

3

行程同步

- 利用共享有限記憶體，最多可讓n-1個項目同時存放在緩衝區，若要完全利用n個項目，則每次新增一個項目就將counter變數加1，由緩衝區中取出一個項目，就將counter變數減1

4

生產者與消費者的例子

- 使用共享記憶體機制
- 生產者及消費者的行程共享了下列的變數：
  - #define BUF_SIZE n
  - int iIn = 0, iOut = 0
  - user_def_type buffer[BUF_SIZE]
- 這個例子中同時最多只能使用到 n-1 個緩衝區的空間why?

5

生產者與消費者的例子

- iIn代表緩衝區內下一個可放資料的位置，及生產者下一個可存放資料的位置。iOut代表緩衝區內最先可用位置。若iIn=iOut 代表緩衝區是空的，(iIn+1)%BUF_SIZE=iOut代表緩衝區已滿

6

生產者

```
do {
        …
        產生一個型別為 user_def_type 的資料並存放於 nextp，
         nextp為生產行程內的區域變數
        …

        /* 若緩衝區已滿則等待 */
        while ((iIn + 1) % BUF_SIZE == iOut) ;

        buffer[iIn] = nextp;
        iIn = (iIn + 1) % BUF_SIZE;
}
while (True);
```

7

修改後行程生產者程式

```
While (1) {
 // 產生一個新的項目放在 nextProduced
 while (counter == BUFFER_SIZE) ; //等待
 buffer[in]= nextProduced;
 in=(in+1) % BUFFER_SIZE;
 counter++;
}
```

8

消費者

```
do {
        while (iIn = iOut) ;   /* 若緩衝區為空則等待 */

        nextc = buffer[iOut];
        iOut = (iOut + 1) % BUF_SIZE;
                …
        將存於 nextc 的資料消耗掉，
         nextc為消費行程內的區域變數
                …
}
while (True);
```

9

修改後行程消耗者程式

```
While (1) {
 // 消耗放在 nextConsumed的項目
 while (counter == 0) ; //等待
 nextConsumed=buffer[out];
 out=(out+1) % BUFFER_SIZE;
 counter--;
}
```

10

行程同步

- 若此二程式同時執行，可能產生錯誤
- counter++; 轉成機器碼後需要3個指令：
  register2=counter; register2=register2+1;
  counter=register2;
- counter--; 轉成機器碼後需要3個指令：
  register1=counter; register1=register1-1;
  counter=register1;
- 因此二程式同時執行，此6個機器碼交錯執行的緣故，在不同執行的順序下，當counter=6時，將產生不同的結果5,6,7

11

行程同步錯誤分析(92nttu)

- 假設目前counter=6，執行一次生產及消費應仍為6

執行程式	執行指令	執行結果
消耗者	register1=counter;	register1=6
消耗者	register1=register1-1;	register1=5
生產者	register2=counter	register2=6
生產者	register2=register2+1;	register2=7
消耗者	counter=register1;	counter=5
生產者	counter=register2;	counter=7

12

行程同步錯誤分析(92tpu, 95tpu, 96tpu)

- 會得到這個不正確的狀態是因為允許兩個行程並行處理這個 counter變數。像這種數個行程同時存取和處理相同資料的情況，而且執行的結果取決於存取時的特殊順序，就叫競爭情況 (race condition)。

行程同步錯誤分析

- 發生錯誤原因是此二行程同時存取同一變數counter，在CPU的執行中，並不一定讓一條機器碼執行完後才執行下一條指令，而是讓所有指令交錯執行
- 由於多個行程同時去存取相同的資料時，會因為不同的指令執行順序，而得到不同的結果的現象，稱為競爭情形
- 為了避免競爭情況的發生
  - 同一時間只能讓一個行程去存取一個變數。
  - 行程之間需要互相同步，讓一個行程更改資料的動作不會影響到其他行程的執行結果。

14

臨界區(critical section)

- 臨界區是一段不能讓多個行程同時執行的程式碼。
  - 系統中的某個行程在執行臨界區的這段程式時，其他的行程不能在這段時間內進入同一個臨界區執行。
  - 一個行程在修改與其他行程所共有的資料，可以將修改共有資料的這段程式碼，寫在臨界區中解決
  - 可以解決因行程共享資料而造成資料可能不一致的問題。
  - 必須同時符合互斥、進行與有限等待三項條件。

15

臨界區

- 每個行程必須要先獲得許可後，才可以進入臨界區
- 在臨界區之前負責協調行程的程式碼稱之為入口區。
- 在臨界區之後會接著一個出口區，負責處理出臨界區後的動作。
- 而剩餘的程式部分則稱之為剩餘區。

16

臨界區演算法

```
do {
    entry section      入口區
      critical section   臨界區
    exit section       出口區
      remainder section  剩餘區
} while (1);
```

17

臨界區演算法

- 假設系統中有n個行程{p_1,p_2,…p_n}，這些行程共用一個臨界區，此臨界區必須符合以下3個條件：
  - 互斥：如果一個行程p_1正在臨界區中執行，則其他行程不可以進入此臨界區
  - 進行：如果有行程正等待進入臨界區，而且目前臨界區中沒有行程在執行，則等待行程之一必須能在有限時間內進入臨界區
  - 有限等待：一個行程請求進入臨界區到被獲准進入的期間，其他行程被獲准進入臨界區的次數是有限的，每個行程等待時間都是有限的

18

臨界區實作法：交替演算法

- 兩個行程 P_i 及 P_j 之間的臨界區演算法。
  - 共用一個變數 turn，指出目前允許進入臨界區的是哪一個行程。
  - 只記錄系統目前的狀態，但是並不記錄行程個別的狀態，故無法讓個別行程以不同的方式爭取資源
  - 如果 turn = i，代表行程 P_i 可以進入臨界區之中。
  - 滿足臨界區互斥的條件，但是不能滿足進行的條件。

19

交替演算法(94tpu, 97nttu)

- 行程 P_i 的程式結構如下：

```
do {
    while (turn != i) ;    入口區
        critical section
    turn = j;    出口區
        remainder section
} while (1);
```

- P_i已經在剩餘區執行時，turn=i又讓P_i可以進入臨界區，這使得P_j仍得繼續等待

20

臨界區實作法：旗標演算法

- 兩個行程P_i 及 P_j之間的臨界區演算法。
  - 將交替演算法中共有的變數 turn 改為共有的陣列 flag，記錄系統中個別行程的狀態。
  - 一開始將陣列所有元素都設為FALSE
  - 如果 P_i 要進入臨界區，則將 flag[i] 設為 TRUE，並檢查flag[j] ，若flag[j]為TRUE，則P_i 要等P_j執行完再進入臨界區
  - 當 P_i 離開臨界區後，再將 flag[i] 設為 FALSE。讓其他行程可以接著進入臨界區
  - 滿足臨界區互斥的條件，但是仍然無法滿足進行的條件。

21

旗標演算法(95tku)

- 行程 P_i 的程式結構如下：

```
do {
  flag[i] = TRUE;
  while (flag[j]) ;
    critical section
  flag[i] = false;
    remainder section
} while (1);
```

Pi:flag[i] = TRUE;
Pj:flag[j] = TRUE;
Pi: **while** (flag[j]) ;
Pj: **while** (flag[i]) ;

- 當P_i 與P_j皆將flag[i],flag[j]皆設為TRUE，則永遠跳不出while迴圈➔競爭情形

22

臨界區實作法：綜合演算法

- 將前述兩個演算法綜合起來，兩個行程 P_i 及 P_j 之間的正確臨界區演算法。
  - 綜合交替演算法與旗標演算法。
  - 以 flag 陣列記錄個別行程是否想要進入臨界區；而 turn 變數指出目前系統允許哪個行程進入臨界區。
  - 一開始將flag陣列所有元素都設為FALSE，turn 變數初始化為i或j
  - 若P_i 與P_j 皆想進入且將flag[i],flag[j]皆設為TRUE，則 turn 變數代表哪個行程可以進入
  - 同時滿足互斥、有限等待與進行三個條件??。

23

綜合演算法

- 行程 P_i 的程式結構如下：

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j) ;
      critical section
    flag[i] = false;
      remainder section
} while (1);
```

24

臨界區實作法：麵包店演算法

- 每位顧客(行程)到店中會領一張號碼牌，店家依號碼牌依序服務客戶，PID是唯一的
- 多個行程間的臨界區演算法。
  - 以行程取到的號碼牌，由小而大地讓行程進入臨界區中。
  - choosing與number陣列分別代表顧客P_i是否在選號碼，及所選號碼的大小
  - 若有行程取到相同號碼，則以行程的 ID 大小決定先後順序，ID 較小的優先。
  - 同時滿足互斥、有限等待與進行三個條件??。

25

麵包店演算法(93tpu)

- 行程 P_i的程式結構如下：
- (a,b)<(c,d)定義為a<c或a=c且b<d

```
do {
  choosing[i] = 1;  // TRUE
  number[i] = max(number[0], number[1], …, number[n – 1]) + 1; //取號碼
  choosing[i] = 0; // FALSE
  for (j = 0; j < n; j++) {
    while (choosing[j]) ;
    while ((number[j] != 0) && ((number[j], j) < (number[i], i))) ;
  }
    critical section
    number[i] = 0;
  remainder section
} while (1);
```

麵包店演算法

- 一開始將choosing及number陣列所有元素都設為0，若P_i正在臨界區，而P_j也想進入，因P_i已經領過號碼number[i]≠0，所以(number[i],i) < (number[j],j) ， P_j會在第二個while迴圈中等待，而無法進入臨界區，因此符合互斥條件
- 依號碼牌讓行程先到先做，符合有限等待的條件➔正確的多行程臨界區實作法

27

硬體支援

- 在單 CPU 的系統中，可以很簡單地讓行程在修改共用的變數時，停止接受中斷而解決同步的問題。
- 但是在多 CPU 的系統中並不合適，因為停止所有 CPU 的中斷除了很耗時間之外，也會增加行程進入臨界區所花費的時間。
- 除了利用程式技巧來達到同步之外，可以將同步的機制設計在硬體上。
  - 撰寫同步程式變得更加方便，並同時提升系統效率。
  - 許多機器實作了特殊的硬體指令，可以不被中斷地檢查並設定一個記憶體的內容、或是交換兩個記憶體的內容。
  - 利用這些指令可以很簡單地解決臨界區的問題。

28

TestAndSet 硬體指令

- TestAndSet 指令會傳回參數 target 目前的值，並同時將 target 的值設為 TRUE。
- 在執行完整個指令之前都不會被中斷。

```
boolean TestAndSet (Boolean &target) {
   Boolean rv = target;
   target = true;
   return rv;
}
```

- 可以利用 **TestAndSet 機器**指令實作多行程的臨界區演算法，來滿足互斥條件。

29

TestAndSet 硬體指令

- 利用lock變數並將初始值設為false，程式結構如下：滿足互斥，不滿足有限等待

```
do {
   while (TestAndset(lock)) ;
      critical section
   lock = false;
      remainder section
} while (1);
```

30

Swap硬體指令(96tpu,96nttu)

- **Swap** 指令會交換參數 a 與 b 兩個記憶體的內容。
- 在執行完整個指令之前都不會被中斷。

```
Void Swap (Boolean &a, Boolean &b) {
    Boolean temp = a;
    a = b;
    b = temp;
}
```

- 可以利用**Swap** 指令實作多行程的臨界區演算法，來滿足互斥條件。

31

Swap硬體指令

- 利用lock變數並將初始值設為false，每一個行程有個私有變數key來與lock做交換，程式結構如下：

```
do {
    key = TRUE;
    while (key == TRUE)  swap(lock, key);
      critical section
    lock = false;
      remainder section
} while (1);
```

32

TestAndSet硬體指令另一方法

- 皆利用lock變為FALSE進入臨界區，同時lock又設為TRUE來防止其他行程進入。 (boolean waiting[n], lock;)

```
do {  Pi
    waiting[i] = TRUE;  key=TRUE;
      while (waiting[i] && key) key=TestAndSet(lock);
      waiting[i]=FALSE;
Critical section
    j=(i+1) % n;
      while ((j != i) && !waiting[j]) j=(j+1) % n;
      if (j == i) lock=FALSE;  else waiting[j]=FALSE;
Remainder section } while (1);
```

TestAndSet硬體指令另一方法

- 行程會共用waiting陣列，紀錄哪些行程正在等待進入臨界區，lock變數代表目前臨界區是否可以進入，兩者皆初始化為FALSE，若任一行程進入臨界區後，會將lock變數設為TRUE，其他想要進入臨界區的行程會等待在入口區的while迴圈。當臨界區中的行程離開後，會在出口區將等待進入臨界區的其中一個行程的waiting陣列值設為FASLE，下一個行程才能進入臨界區，因此符合互斥條件

34

TestAndSet硬體指令另一方法

- 若沒有行程等著進入臨界區，要離開臨界區的行程會將lock變數設為FALSE，也會將某行程的waiting陣列值設為FASLE，讓只要有等待進入臨界區的行程，一定可以進入，因此滿足了進行的條件
- 若系統中有n個行程，離開臨界區行程設定waiting陣列值的方法，是循序檢查到行程的waiting陣列值為TRUE，將其設為FALSE，所以每個行程最多只會等待n-1次，就可以進入臨界區，所以滿足有限等待的條件。

35

號誌 (Semaphore)

- 號誌是十分常用的同步工具，可以簡單地解決較為複雜的同步問題。
  - 大部分的作業系統都已經實作了號誌，作為行程同步的工具。
  - 號誌包含一個數值，該值在初始化之後就只能經由 **signal()** 與 **wait()** 兩個不可被中斷的函式去存取。
  - 當一個行程在存取號誌的值時，其他行程無法存取同一個號誌的值。
  - 利用臨界值不可被中斷的函式是解決同步問題最重要的關鍵➔號誌即可達此目的

36

計數號誌

- 計數號誌的值像一個計數器，記錄著有多少行程可以再進入臨界區。
  - 號誌的值只能利用 **signal**() 與 **wait**() 存取。
  - **signal**() 會將號誌的值加 1。
  - **wait**() 則會先測試號誌的值，如果號誌的值大於零，則將該值減 1，否則 **wait**() 會等待到該值大於零再繼續執行。
  - 解決多行程臨界區的問題。
  - 達到行程之間的同步。

```
Wait(S){
While (S <=0) ;
S--;
}
```

```
Signal(S){
S++;
}
```

37

Wait() & Signal()指令

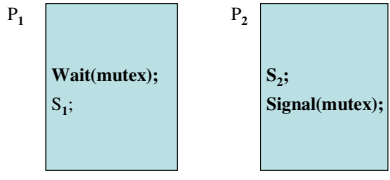
- **Wait() & Signal()**程式結構如下，行程共用一號誌mutex，並將其初值設為1

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```

38

計數號誌

- 利用號誌來達成行程間的同步，P_1要等到P_2執行完敘述S_2，才能執行S_1，可共用一個號誌mutex，將其初值設為0



39

計數號誌(93tpu , 95nttu)

- 臨界區實作方法有使用忙碌等待的缺點。
  - 如果已經有行程進入了臨界區，那麼其他想要進入臨界區的行程都會在入口區中一直地執行迴圈來等待。
  - 使用忙碌等待的號誌也被稱為旋轉鎖。
  - 使用忙碌等待的行程因為不會被內文切換，所以如果忙碌等待太久會浪費許多 CPU 的時間。
  - 但是如果忙碌等待的時間比內文切換的時間更短，反而可以省下內文切換的額外負擔。

40

計數號誌

- 為了避免忙碌等待所造成的 CPU 資源浪費，可以修改 **wait**() 和 **signal**() 兩個函式。
  - wait()呼叫時其值<=0，不讓行程忙碌等待，直接讓行程將自己阻隔起來。
  - 新的號誌結構除了原有的整數可記錄號誌的值，還另外增加了一個串列，記錄正在該號誌等待的行程。

```
typedef struct {
  int value;
  struct process *L; //該號誌等待的行程
} semaphore;
```

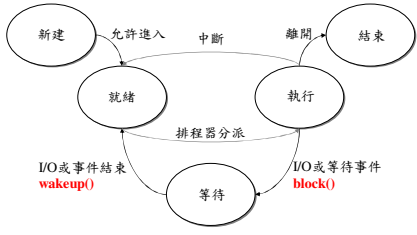
41

計數號誌

- **wait**() 會將等待的行程放入該號誌的串列之中，並且將行程阻隔，而 CPU 排程器會另選出就緒佇列中的其他行程來執行。➔不浪費CPU資源
- 當 **signal**() 被呼叫之後，會由該號誌的串列中叫醒一個被阻隔的行程繼續執行。
- 新的架構中，號誌的值可能會是負值，而這個負值的絕對值就是號誌串列中被阻隔的行程數目。
- block()和wakeup()函式是大部分OS提供的基本系統呼叫，block()會將現行行程狀態更改為等待狀態，並將之移置就緒佇列中。叫醒的動作則由wakeup()，將被叫醒之行程由等待改為就緒狀態

42

行程狀態圖



43

計數號誌

```
typedef struct {
    int value;
    struct process *L; //該號誌等待的行程
  } semaphore S;
```

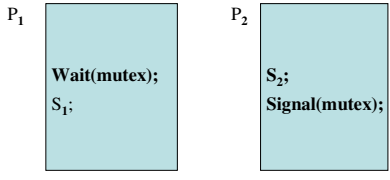
```
void wait(S) {
 S.value- -;
 if (S.value<0){
  將行程P加入S.L中
  block(P);
 }
}
```

```
void signal(S) {
 S.value++;
 if (S.value<=0){
 由S.L中移除一個行程P
  wakeup(P);
 }
}
```

44

計數號誌

- 利用號誌來達成行程間的同步，P_1要等到P_2執行完敘述S_2，才能執行S_1，可共用一個號誌mutex，將其初值設為0



45

計數號誌

- 號誌中的串列如果使用不適當的實作方式，例如後進先出串列，可能會造成無限阻隔或是飢餓現象。
  - 可以使用先進先出串列來實作號誌串列。
  - 必須保證 **wait()** 和 **signal()** 在執行的過程中不會被中斷(不可分割運算)。
- 修改過後的號誌雖然不能完全不使用忙碌等待，但是大幅地縮短了忙碌等待的時間。
  - 由整個臨界區縮短到只有實作 **wait()** 和 **signal()** 的臨界區所造成的忙碌等待時間。
  - 可以提高系統的效能。

46

死結和飢餓

- 使用等待佇列製作訊號可能導致有二個或以上的行程等待一項僅能由等待行程所引發事件的情形。此事件是指一個 signal()運算的執行;而當上述情形發生時，稱這些行程被打死結(deadlocked)。

爲了進一步說明，我們首先考慮一個包含有行程 P_0 和 P_1 的系統，P_0 和 P_1 都能存取兩個號誌 S 和 Q，都被設定成 1：

```
        P0              P1
wait(S);        wait(Q);
wait(Q);        wait(S);
   .               .
   .               .
   .               .
signal(S);      signal(Q);
signal(Q);      signal(S);
```

47

二元號誌

- 二元號誌的值只限定為 0 或1。
  - 利用硬體對二元數值的運算支援，二元號誌的實作要比計數號誌簡單快速得多。
  - 可以利用二元號誌來實作計數號誌。(兩個二元號誌及一個整數，S_1=1，S_2=0，C為計數號誌S的初值)

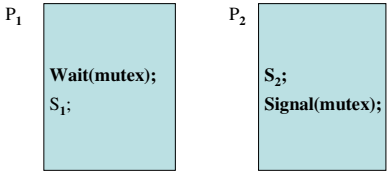
```
void wait(S) {
  wait(S1); C--;
 if (C<0){
   signal(S1); wait(S2);
 }
 signal(S1);
}
```

```
void signal(S) {
  wait(S1); C++;
 if (C<=0){
   signal(S2);
 else
   signal(S1);
 }
}
```

48

計數號誌

- 利用號誌來達成行程間的同步，P_1要等到P_2執行完敘述S_2，才能執行S_1，可共用一個號誌mutex，將其初值設為0



49

同步的經典問題

- 牽涉到了大型並行控制的領域。
- 經常拿來測試新的同步機制的正確性。
- 這些問題的解法當中都使用了號誌作為同步的工具。

50

生產者

```
do {
        …
        產生一個型別為 user_def_type 的資料並存放於 nextp，
         nextp為生產行程內的區域變數
        …

        /* 若緩衝區已滿則等待 */
        while ((iIn + 1) % BUF_SIZE == iOut) ;

        buffer[iIn] = nextp;
        iIn = (iIn + 1) % BUF_SIZE;
}
while (FALSE);
```

51

修改後行程生產者程式

```
While (1) {
 // 產生一個新的項目放在 nextProduced
 while (counter == BUFFER_SIZE) ; //等待
 buffer[in]= nextProduced;
 in=(in+1) % BUFFER_SIZE;
 counter++;
}
```

52

消費者

```
do {
        while (iIn = iOut) ;   /* 若緩衝區為空則等待 */

        nextc = buffer[iOut];
        iOut = (iOut + 1) % BUF_SIZE;
                …
        將存於 nextc 的資料消耗掉，
         nextc為消費行程內的區域變數
                …
}
while (FALSE);
```

53

修改後行程消耗者程式

```
While (1) {
 // 消耗放在 nextConsumed的項目
 while (counter == 0) ;  //等待
 nextConsumed=buffer[out];
 out=(out+1) % BUFFER_SIZE;
 counter--;
}
```

54

有限緩衝區問題

- 用號誌來實作可簡單解決許多問題。
- 假設緩衝區中有 n 個欄位，每個欄位可以存放一個項目。
- 使用 mutex 號誌確保存取緩衝區時的互斥條件成立，並初始化為 1。
- empty 和 full 兩個號誌則分別用來計算緩衝區中空的與使用過的欄位數目。
  - empty 號誌初始化為 n。
  - full 號誌初始化為 0。

55

有限緩衝區問題(96tpu,97tku)

- 生產者和消耗者的程式碼如下：

生產者	消耗者

do {
…
產生一個新的項目放在 nextp
…
wait(empty); //n-1,n-2➔0
wait(mutex);//1➔0
…
將 nextp 加入到緩衝區中
…
signal(mutex);//0➔1
signal(full);//1,2➔n
} **while**(1);
	do {
	wait(full);//1➔0
	wait(mutex);//1➔0
	…
	將一個項目由緩衝區中移到 nextc
	…
	signal(mutex);//0➔1
	signal(empty);//n-1,n
	…
	消耗放在 nextc 中的項目
	…
	} **while**(1);

56

讀取者與寫入者問題

- 一個系統中經常會有數個行程共同分享同一份資料物件
  - 只讀取這份分享資料的行程稱為讀取者(不改內容)。
  - 只更新分享資料的行程稱為寫入者。
  - 如果一個讀取者和一個寫入者同時存取所共享的資料，可能會發生錯誤。
  - 這種同步的問題稱為讀取者與寫入者問題。
  - 以整數 readcount 記錄正在讀取資料的讀取者數目，初始值為 0。 int readcount=0;
  - 利用初始化為 1 的 mutex 和 wrt 兩個號誌形成兩種臨界區。 Semaphore mutex=1,wrt=1;

57

讀取者與寫入者問題

- 讀取者和寫入者問題的解法如下：
  (91tku,94tpu, 93tku,96tku,97tku,90tku,90nttu)

讀 取 者	寫 入 者

wait(mutex); // 1→0
readcount++; // 0→1
if (readcount == 1)
  wait(wrt); //1→0
signal(mutex); //0→1
  …
進行讀取IN, OUT
wait(mutex);//1→0
readcount--;
if (readcount == 0)
  signal(wrt); // 0→1
signal(mutex); //0→1
	wait(wrt); // 1→0
	  …
	進行寫入IN, OUT
	  …
	signal(wrt); //0→1

58

讀取者與寫入者問題

- 若有一寫入者在臨界區進行寫入，有n個讀者在入口區等待，只會有一個讀者在等wrt號誌，其餘n-1個讀者都會在等mutex號誌
- 只當所有讀者都讀完後，才會釋放wrt號誌給寫入者
- 當寫入者執行signal(wrt)，是由排程器決定進入臨界區的是讀取者還是寫入者

59

讀取者與寫入者問題

- Condition x,y; Condition型別的變數只有透過wait和signal指令才能運作
- X.wait()執行這運作的行程X必須暫時等待，直到執行X.signal()，而X.signal()每次只能恢復一個等待行程的動作，如果沒有任何等待的行程，則此signal運作將不產生任何作用
- Non_Empty(OK_to_Write) is true if someone is waiting to write
- Non_Empty(OK_to_Read) is true if someone is waiting to Read

60

Readers and Writers

```
monitor Reader_Writer_Monitor
Readers : Integer := 0;
Writing : Boolean := False;
OK_to_Read, OK_to_Write : Condition;

procedure Start_Read
begin
  if Writing or
     Non_Empty(OK_to_Write) then
     OK_to_Read.wait();
  end if;
  Readers := Readers + 1;
  OK_to_Read.signal();
end Start_Read;

procedure Start_Write
begin
  if Readers <> 0 or
     Writing then
     OK_to_Write.wait();
  end if;
  Writing := True;
end Start_Write;
```

61

Readers and Writers

```
procedure End_Read
begin
 Readers := Readers - 1;
  if Readers = 0 then
    OK_to_Write.signal();
  end if;
end End_Read;

procedure End_Write
begin
   Writing := False;
   if Non_Empty(OK_to_Read) then
      OK_to_Read.signal();
   else
      OK_to_Write.signal();
   end if;
end End_Write;

end Reader_Writer_Monitor;
```

62

Readers and Writers

- The monitor has two status variables and two condition variables.
- The status variables are :
  - Readers : A counter of successful readers which have successfully passed Start Read and are currently reading.
  - Writing : A boolean flag which is true when a process is writing.

63

Readers and Writers

- The condition variables are :
  - OK to Read to suspend readers.
  - OK to Write to suspend writers.
- A reader is suspended if some process is currently writing or if some process is waiting to write. This gives priority to a suspended writer over the readers.
- A writer is suspended if other readers are currently reading or some writer is writing.

64

Readers and Writers

- End Read executes OK to Write.signal() if there is no other reader.
- End Write gives priority to suspended readers if any, otherwise, it signals suspended writers.

65

Readers and Writers

- The purpose of the OK_to_Read.signal() in Start_Read is to allow other readers to start reading. So, the readers get access in a cascading fashion.
- If there are suspended writers, a new reader is required to wait until the termination of (at least) the first write.
- If there are suspended readers, they will all be released before the next write.

66

Readers and Writers

- CASE 1(96TKU)
  While R1 is reading, the requests arrive in the order：R2, W1,R3, and while W1 is writing, the new requests arrive in the order：W2, R4, W3, and while W2 is writing , the new requests arrive in the order：R5, W4. (r1r2r3w1r4w2r5w3w4)
- CASE 2(93TKU)
  While R1 is reading, the requests arrive in the order：R2, W1,R3, W2, W3, R4, W4, and while W2 is writing, the new requests arrive in the order：R5. (r1r2w1r3r4w2w3w4)

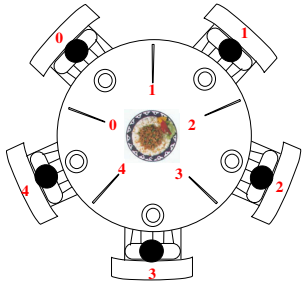
67

哲學家晚餐問題

- 哲學家們圍坐在一張圓桌一起共進晚餐。
- 桌上筷子數目與哲學家相等，每兩個哲學家之間共用一支筷子(不考律衛生問題)。
- 哲學家們會坐在餐桌上思考哲學問題。
- 當哲學家們想吃東西時會拿起左右各一支筷子進餐，拿齊兩支筷子的哲學家們可同時進餐，但不能搶奪別人手上的筷子。
- 當食物吃完後，哲學家會將兩支筷子都放下，並繼續思考哲學問題，其他哲學家可繼續用放下的筷子。

68

哲學家晚餐問題示意圖



69

哲學家晚餐問題(97tpu 3)

- 第 i 位哲學家的程式可寫成：
  semaphore chopstick[5]={1,1,1,1,1};

```
do {
   wait(chopstick[i]);1->0
   wait(chopstick[(i + 1) % 5]);1->0
      …
      進餐
      …
   signal(chopstick[i]); 0->1
   signal(chopstick[(i + 1) % 5]); 0->1
      …
      思考
      …
} while(1);
```

70

哲學家晚餐問題

- 上述作法可能會導致死結的發生。
  - 假如每位哲學家都拿起了一支筷子而等待另一支筷子，則不會有任何一位哲學家可以拿到兩支筷子，所有的哲學家將會進入死結狀態。
- 下列作法可以避免哲學家晚餐問題發生死結：
  - 在圓桌上放置 n+1 支筷子或限制最多只有 n-1 位哲學家可以同時進餐(why??)。
  - 規定哲學家們必須要同時拿起左右兩支筷子。
  - 規定奇數座位的哲學家要先拿起左方的筷子再拿起右方的筷子；而偶數座位的哲學家則先拿起右方的筷子再拿起左方的筷子。

71

臨界區域與監督程式

- 號誌是個非常方便而且有效率的同步工具。
  - 然而不正確地使用號誌，很容易產生行程之間同步上的錯誤。
  - 這類錯誤並不一定每次程式執行時都會發生，所以很難除錯。
- 臨界區域與監督程式是另外兩種較高階的常用同步工具。
  - 使用起來較為方便，而且不容易出錯。
  - 每個行程會有一些私有的局部變數，以及一些行程間的共享變數。
  - 行程不能夠直接存取其他行程的局部變數。

72

臨界區域

- 臨界區域的使用非常方便。
  - 以下宣告一個具有共享變數v 的臨界區域，在 B 條件式成立下，如果沒有其他行程在此臨界區域中執行，就會執行 S 敘述：v: shared T;

```
region v when B do S;
```

- 利用臨界區域來實作，程式設計師不用煩惱同步的問題，只要正確地把問題描述在臨界區域內。

```
region v when B do S1;
region v when B do S2;  不保證其執行次序
```

73

生產者

```
do {
        …
        產生一個型別為 user_def_type 的資料並存放於 nextp，
        nextp為生產行程內的區域變數
        …

        /* 若緩衝區已滿則等待 */
        while ((iIn + 1) % BUF_SIZE == iOut) ;

        buffer[iIn] = nextp;
        iIn = (iIn + 1) % BUF_SIZE;
}
while (True);
```

74

消費者

```
do {
        while (iIn = iOut) ;   /* 若緩衝區為空則等待 */

        nextc = buffer[iOut];
        iOut = (iOut + 1) % BUF_SIZE;
                …
        將存於 nextc 的資料消耗掉，
         nextc為消費行程內的區域變數
                …
}
while (True);
```

75

有限緩衝區問題

- 有限緩衝區問題可以用臨界區域來簡單地解決同步的問題。
- struct buffer { item pool[n]; int count, in, out; };

生產者	消耗者
region buffer when (count < n) { **pool[in]=nextp;** **in=(in+1)%n;** **count++;** **}**	**region buffer when (count > 0) {** **nextc=pool[out];** **out=(out+1)%n;** **count--;** **}**

76

臨界區域實作**region** v **when** B **do** S

```
wait(mutex); // mutex=1; first_delay=0; second_delay=0;
while (!B) { // B=False則loop
  first_count++;
  if (second_count > 0) signal(second_delay);  else signal(mutex);
  wait(first_delay);  first_count--; second_count++;
  if (first_count > 0) signal(first_delay);  else signal(second_delay);
  wait(second_delay); second_count--;
}

S;
if (first_count > 0) signal(first_delay);
else if (second_count > 0) signal(second_delay); else signal(mutex);
```

77

臨界區域(Critical Region)

- 臨界區域 **region** v **when** B **do** S 可利用 mutex 初值為1、first_delay初值為0及second_delay初值為0三個號誌實作。
  - mutex 號誌是用來確保臨界區的互斥條件成立。
  - 如果行程因為條件 B 為 FALSE 而無法進入臨界區，該行程將會在號誌 first_delay 等待。
  - 在號誌 first_delay 等待的行程重新檢查 B 值之前，會離開號誌 first_delay，而在號誌 second_delay 等待。

78

臨界區域

- 程式中用first_count初值為0與second_count初值為0兩個變數分別紀錄正在等待號誌first_delay與second_delay的行程數目
- 當一個行程離開臨界區後可能因執行S而改變了B的值，因此必須讓正在等待號誌first_delay與second_delay的行程重新檢查B的值
- 若B的值為TRUE，該行程就可進入臨界區，其餘的行程必須繼續在號誌first_delay與 second_delay等待
- 分成first_delay 與 second_delay兩段式等待的原因，是為了要避免行程持續忙碌地檢查 B 值。

79

臨界區域實作**region** v **when** B **do** S

```
void x.wait() {
  x_count++;
  if (next_count>0)
    signal(next);
  else
    signal(mutex);
  wait(x_sem);
  x_count--;
}
```

```
void x.signal(S) {
  wait(S1); C++;
 if (C>0){
    signal(S2);
 else
    signal(S1);
 }
}
```

80

臨界區域使用號誌製作監督程式

- 對每一個監督程式提一個號誌mutex(初始值為 1)。每一個行程在進入監督程式之前，都必須先執行wait(mutex)的動作，並在離開監督程式之後，執行signal(mutex)的動作。

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

wait(mutex)

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

signal(mutex)

81

監督程式(Monitor)

- 監督程式是另外一種高階的同步工具。
  - 對較複雜的同步問題提供了更一般性的實作工具。
  - 由一些變數宣告及函式所組成，變數的值定義了監督程式的狀態。變數值定義監督程式的狀態
  - 保證只有一個行程在監督程式中執行所定義的函式。
  - 在監督程式中，程式設計師不需要撰寫有關同步的程式碼，但是可以利用條件變數來定義自己的同步機制。 condition x, y;
  - 哲學家晚餐問題可以利用監督程式來實作。

82

監督程式

```
monitor monitor_name {
   //共享變數宣告
   procedure body P1(…) { … }
   procedure body P2(…) { … }
…
   procedure body Pn(…) { … }
  { 初始化程式碼 }
}
```

83

哲學家晚餐的監督程式

```
monitor DP {
  enum { thinking, hungry, eating} state[5];   condition self[5];
  void pickup(int i) {
    state[i]=hungry; test(i); //i=1
    if (state[i] != eating) self[i].wait();
  }
  void putdown(int i) {
    state[i]=thinking; 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)) { state[i]=eating; self[i].signal(); }
  }
  void init() { for (int i=0; i<5; i++) state[i]=thinking; }
}
```

哲學家晚餐的監督程式

- 和號誌一樣，條件變數也利用wait()與signal()兩個函式，來暫停或是叫醒行程 ➔ X.wait() 或 Y.signal()
- 每個哲學家呼叫DP.pickup()取得筷子之後開始用餐，用餐後哲學家呼叫DP.putdown()放回筷子，同步的問題都由監督程式解決，不用程式設計師煩惱

  DP.pickup(i)

  …用餐 …

  DP.putdown(i)

85

臨界區間的互斥性

- 臨界區間的互斥性保證了臨界區間的執行之不可分割。如果兩個臨界區間並行地執行，結果相當於它們以某種未知的順序循序地執行。雖然這項性質在許多應用領域很有用，依然有許多情況下，工作是完整的執行完，或完全不執行。
- 資料的一致性(連同資料的儲存和取回)，時常與資料庫系統有關。引發應用資料庫系統的技術在作業系統的興趣。作業系統可以視為資料的處理者，可以從資料庫研究的進階技術和模型得到一些幫助。

86

臨界區間的互斥性

- 單一邏輯函數的指令操作，稱為交易，處理交易主要事項為不管電腦各種失效可能，依然維持其不可分割性質
- 交易只是一系列的讀和寫的動作，最後以交付(commit)或中止(abort)操作結束。交付操作指出交易已成功地結束其執行，而中止操作則指示交易由於一些邏輯錯誤而停止其正常執行。

87

臨界區間的互斥性

- 一個成功完成其執行的交易是被交付，否則就被中止。
- 中止的交易可能已經修改它所存取的各種資料，為確保不可分割性，被中止的交易對已修改的資料必須不影響其狀態，必須恢復到交易執行前的狀態，稱為此交易被撤回(rolledback)，這是系統的部份責任，以確保此性質。

88

臨界區間的互斥性

- 為了決定系統該如何確保不可分割的性質，首先需要識別用來儲存被交易存取各項資料之裝量的特性。不同型態的儲存媒體是由它們的速度、容量和失效彈性來區分。
  - 揮發性儲存體：儲存在揮發性儲存體的資訊在系統當掉時通常不存在。這種儲存體的例子有主記憶體和快取記憶體。對揮發性儲存體的存取非常快，因為記憶體存取本身的速度，可以直接在揮發性儲存體存取任何資料項。

89

臨界區間的互斥性

- 非揮發性儲存體：儲存在非揮發性儲存體的資訊在系統當掉之後通常還存在。這種儲存體的例子有磁碟和磁帶。非揮發性儲存體的速度比揮發性儲存體的速度慢好幾個級次。因為磁碟和磁帶裝置是電子機械式，並且需要實體移動以存取資料。
- 穩定儲存體：在穩定儲存體的資訊絕不會遺失 。為了製作近似這種的儲存體，需要複製資訊到數個非揮發性的儲存體快取 (通常是磁碟)，每一個有各自的失效模式，並且以控制的方式更新資訊 。

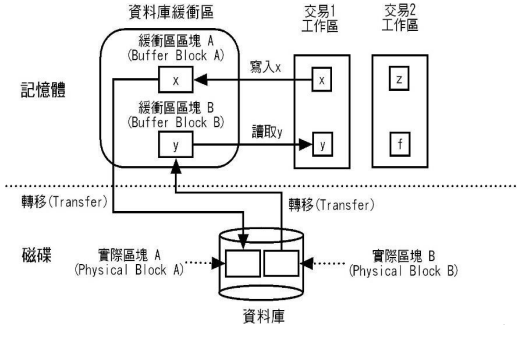
90

資料存取的基礎(陳會安)

- 資料庫系統的資料存取是從儲存裝置磁碟的資料庫存取資料，取得執行交易所需的資料。資料存取會影響資料庫系統故障時，可能損失的資料，回復處理就是在救回這些遺失的資料。
- 資料庫系統並不是直接從儲存裝置存取資料，而是透過資料庫緩衝區和主記憶體中各交易的工作區。

91

資料存取的基礎



以記載簿為基礎的復原

- 確保不可分割的方法是系統在穩定儲存體上維持一份登錄(log➔記載簿)，每一筆登錄記載著一筆寫入交易的操作，並有以下各欄
  - 交易名稱：執行寫入操作之交易的唯一名稱。
  - 資料項名稱：寫入資料項的唯一名稱。
  - 舊值：寫入前的資料項數值。
  - 新值：寫入後的資料項數值。

93

以記載簿為基礎的復原

- 在交易T_i開始執行前，記錄<T_i starts>就寫入記錄簿，在執行期間任何T_i做的write操作都先行在記錄簿寫入一筆適當的新記錄。當T_i交付時記錄<T_i commits>就寫入記錄簿
- 要求在write(X)操作被執行前，相對於X的記錄簿之紀錄被寫入穩定儲存體
- 使用記錄簿，系統可以處理任何不會造成非揮發性儲存體資料遺失之失效。復原演算法使用兩項步驟：
  - undo (T_i)，恢復所有被交易 T_i 更新過的資料爲舊有值。
  - redo (T_i)，設定所有被交易 T_i 更新過的資料爲新值。

94

以記載簿為基礎的復原

- 如果交易T_i中止，只要執行記錄undo(T_i)就可以恢復它更新過之資料狀態。藉由參考記錄簿就可決定那些交易需要重做，那些需要復原。如何區分則由以下方式完成：
  - 如果記錄簿包含了記錄<T_i starts>，但沒有<T_i commits>，則交易T_i都要被復原(舊值)
  - 如果記錄簿同時包含了記錄<T_i starts>和<T_i commits>，則交易T_i都要被重做(新值)

95

欄位名稱	說明
交易編號(Transaction ID)	識別屬於哪一個交易
資料表	影響資料庫的哪一個資料表
記錄(Row ID)	影響資料表的哪一筆記錄
屬性(Attribute)	影響記錄的哪一個欄位
操作(Operation)	在前述資料表相關記錄和欄位執行的指令，可能是：(start)BEGIN TRAN、INSERT、DELETE、UPDATE、COMMIT TRAN(commit)或 ROLLBACK TRAN
原始值(Before)	更新前的欄位值，當回復交易（Rollback）時，就是回復成此值
更新值(After)	更新後的欄位值，回復處理需要重作交易，就是使用更新值來執行資料庫單元操作，更改資料庫的資料

96

交易記錄(陳會安)

- 例如：在SQL Server開始執行交易T時，資料庫管理系統指定其交易編號為001，新增一筆記錄到交易記錄，如下所示：
  <T001, BEGIN TRAN>
- 上述交易記錄編號是001，指令BEGIN TRAN開始執行交易。接著將交易執行的所有INSERT、UPDATE和DELETE指令都新增一筆相同交易編號的記錄，如下所示：
  <T001, UPDATE, table, record#1, field_name1, before, after>
  <T001, UPDATE, table, record#1, field_name2, before, after>

97

系統故障的回復處理

- **UNDO復原**：將交易影響的資料庫資料回復到執行交易前的狀態，如果使用立即資料庫更新，因為已經將資料真正寫入資料庫，所以需要參考交易記錄，一一將更新資料回存成交易記錄的原始值
- **REDO重作**：如果交易已經確認交易，但是尚未真正寫入資料庫，回復處理需要重新執行這些交易，將資料庫更新成交易後的狀態，也就是參考交易記錄，一一將交易更新的所有資料項目都指定成交易記錄的更新值

98

檢查點(checkpoint)

- 系統失效要如何決定那些交易要被重作或復原？原因應搜尋整個記錄簿。為降低此額外負擔，引入檢查點。系統週期地執行檢查點，要求下列動作發生：
  - 將所有記錄簿中目前在揮發性儲存體(通常是主記憶體)的記錄輸出到穩定儲存體。
  - 將所有在揮發性儲存體的已修改資料輸出到穩定儲存體。
  - 輸出一筆記錄簿記錄(checkpoint)到穩定儲存體。

99

檢查點(checkpoint)

- 考慮一筆交易T_i在檢查點之前交付執行，<T_i commits>出現在<checkpoint>記錄前，則復原時就不用對T_i執行redo操作
- 改良原先之復原法則，在最近發生之檢查點所開始執行最近交易T_i，先由往後搜尋記錄簿以找出第一筆<checkpoint>記錄，再往後找出接下來的<T_i starts>紀錄，如此就可以找出此筆交易。

100

檢查點(checkpoint)(93tku 2)

- 一旦交易T_i找到後，redo和undo操作只要對交易T_i及交易T_i之後開始執行的所有交易T_j做動作，集合T表示這些交易，所需復原操作如下：
  - 對於在T中的交易T_k，如果出現<T_k commits>在記錄簿中，則執行redo(T_k)
  - 對於在T中的交易T_k，如果沒有出現<T_k commits>在記錄簿中，則執行undo(T_k)

101

並行的不可分割交易(96tpu 9)

- 每一筆交易都是不可分割，交易的並行執行就相當於以某種任意順序依序地執行這些交易。這項性質(叫做可串列化，serialization)可以藉著在一個臨界區只執行一項交易來維持。
- 所有的交易都共享一個共用號誌 mutex，此號誌被設定成1的初值。當一項交易開始執行時，它的第一個動作時執行wait (mutex)。在此交易交付或中止時，它就執行signal (mutex)。

102

可串列化

T_0	T_1
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

圖 6.24 排班 1：串列排班，T_0 而後 T_1

T_0	T_1
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

圖 6.25 排班 2：並行串列化排班

- 交換 T_0 的 read (B) 操作和 T_1 的 read (A) 操作。
- 交換 T_0 的 write (B) 操作和 T_1 的 write (A) 操作。
- 交換 T_0 的 write (B) 操作和 T_1 的 read (A) 操作。

UNDO/REDO的回復處理

- 資料表Test第1筆記錄的A欄位、第2筆記錄的B和C欄位初值都是1000，交易T1和T2的單元操作步驟，如下圖所示：

交易T1

```
1: Read A
2: A = A - 500
3: Write A
4: Read B
5: B = B + 500
6: Write B
```

交易T2

```
1: Read C
2: C = C - 300
3: Write C
```

104

UNDO/REDO的回復處理

交易記錄（一）

- 如果系統當機發生故障時的交易記錄內容，如下所示：

<T1, BEGIN TRAN>
<T1, UPDATE, Test, 1, A, 1000, 500>
<T1, UPDATE, Test, 2, B, 1000, 1500>
<T2, BEGIN TRAN>
<T2, UPDATE, Test, 2, C, 1000, 700> ← 系統故障

- 上述交易記錄的交易T1和T2都沒有<COMMIT TRAN>記錄，表示交易T1和T2尚未確認交易，所以沒有REDO程序。皆為1000

105

UNDO/REDO的回復處理

交易記錄（一）

<T1, BEGIN TRAN>
<T1, UPDATE, Test, 1, A, 1000, 500>
<T1, UPDATE, Test, 2, B, 1000, 1500>
<T2, BEGIN TRAN>
<T2, UPDATE, Test, 2, C, 1000, 700> ← 系統故障

- UNDO交易T1：將Test資料表第1筆記錄的欄位A回存原始值1000，第2筆記錄的欄位B回存原始值1000。
- UNDO交易T2：將Test資料表第2筆記錄的欄位C回存原始值1000。

106

UNDO/REDO的回復處理

交易記錄（二）

<T1, BEGIN TRAN>
<T1, UPDATE, Test, 1, A, 1000, 500>
<T1, UPDATE, Test, 2, B, 1000, 1500>
<T1, COMMIT TRAN>
<T2, BEGIN TRAN>
<T2, UPDATE, Test, 2, C, 1000, 700> ← 系統故障

- REDO交易T1：將Test資料表第1筆記錄的欄位A改為更新值500，第2筆記錄的欄位B改為更新值1500。
- UNDO交易T2：將Test資料表第2筆記錄的欄位C回存原始值1000。

107

UNDO/REDO的回復處理

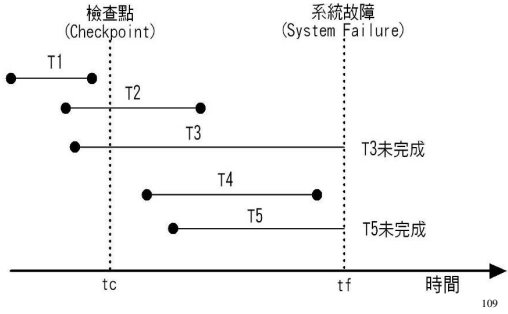
交易記錄（三）

<T1, BEGIN TRAN>
<T1, UPDATE, Test, 1, A, 1000, 500>
<T1, UPDATE, Test, 2, B, 1000, 1500>
<T1, COMMIT TRAN>
<T2, BEGIN TRAN>
<T2, UPDATE, Test, 2, C, 1000, 700>
<T2, COMMIT TRAN> ← 系統故障

- REDO交易T1：將Test資料表第1筆記錄的欄位A改為更新值500，第2筆記錄的欄位B改為更新值1500。
- REDO交易T2：將Test資料表第2筆記錄的欄位C改為更新值700。

108

使用檢查點執行回復處理



109

使用檢查點執行回復處理

- T1交易：在檢查點tc前開始和完成交易。
- T2交易：在檢查點tc前開始，tf系統錯誤前完成交易。
- T3交易：在檢查點tc前開始，直到tf系統錯誤都沒有完成交易。
- T4交易：在檢查點tc後開始，tf系統錯誤前完成交易。
- T5交易：在檢查點tc後開始，直到tf系統錯誤都沒有完成交易。

110

使用檢查點執行回復處理

- 在重新啟動資料庫系統，回復處理針對上述各種交易所執行的處理，如下表所示： (93TKU)

交易	執行處理程序
T1	不處理
T2	REDO
T3	UNDO
T4	REDO
T5	UNDO

111

上鎖協定

- **共用** (shared)：如果交易 T_i 取得資料 Q 的一個共用模式鎖 (表示成 S)，則 T_i 可以讀取這項資料，但不能寫入 Q。
- **互斥** (exclusive)：如果交易 T_i 取得資料項 Q 的一個互斥模式鎖 (表示成 X)，則 T_i 可以讀或寫 Q。

有一種協定可以保證可串列化，那就是**雙相上鎖協定** (two-phase locking protocol)。此協定要求每一筆交易以兩種相位發出上鎖和解鎖要求。

- **成長相位** (growing phase)：一筆交易可以獲得鎖，但不能釋放任何鎖。
- **收縮相位** (shrinking phase)：一筆交易可以釋放鎖，但不能獲得任何新鎖。

112

避免死結的並行控制二階段鎖定法

二階段鎖定法（Two-phase Locking）

- 交易使用二階段來鎖定與解除資料鎖定，如下：
  - **第一階段為擴展階段(Expanding or Growing Phase)**：在交易A執行資料庫單元操作前，交易A請求所有需要存取資料的互斥鎖定，如果有資料已經被交易B鎖定，就等待直到交易B解除資料鎖定。
  - **第二階段為縮減階段(Shrinking Phase)**：當交易A執行完操作後，就解除鎖定交易A所有鎖定的資料。

113

時間戳記為基礎的協定

1. 使用系統時鐘的數值做為時間戳記:一筆交易的時間戳記等於此交易進入系統的時鐘數值。這種方法對於發生在分隔系統間的交易，或沒有共用同一時鐘的處理器之交易無效。
2. 使用邏輯計數器做為時間戳記:交易的時間戳記等於此交易進入系統時的計數器數值。當設定一個新的時間戳記之後，計數器就加1。

- W-timestamp(Q) 表示任何執行 write(Q) 成功的最大時間戳記。
- R-timestamp(Q) 表示任何執行 read(Q) 成功的最大時間戳記。

避免死結的並行控制時間戳記法

時間戳記法（Time-Stamp Order）

- 時間戳記法是指並行控制的多個交易，依其執行先後順序，給予每一個交易一個遞增的時間戳記
- 當交易存取資料時，就將交易和資料的時間戳記進行比較，如下所示：
  - 如果交易的時間戳記大於或等於資料的寫入或讀取時間戳記，就執行資料存取，並且更新資料的時間戳記成為交易的時間戳記。
  - 如果交易的時間戳記小於資料的寫入或讀取時間戳記，交易將回復交易且重新啟動交易，以便讓交易可以得到更大的時間戳記來完成交易。

115

時間戳記為基礎的協定

時間戳記排序協定確保了任何衝突的 `read` 和 `write` 操作根據時間戳記的順序執行。此協定的操作如下所示：

- 假設交易 T_i 發出 `read`(Q)：
  - 如果 TS(T_i) < W-timestamp()，則這狀態暗示了，T_i 需要讀取已經被複寫過的 Q 值。因此，read 操作被拒絕，而 T_i 則被撤回。
  - 如果 TS (T_i) ≥ W-timestamp (Q)，則此 `read` 操作被執行，而且 R-time-stamp (Q) 設定成 R-timestamp (Q) 和 TS (T_i) 兩者中較大的值。
- 假設交易 T_i 發出 `write`(Q)：
  - 如果 TS (T_i) < R-timestamp (Q)，則這狀態暗示了，T_i 產生的 Q 值先被需要，而 T_i 假設此數值將永遠不會被產生。因此，這個 `write` 操作被拒絕，而 T_i 則被撤回。
  - 如果 TS (T_i) < W-timestamp (Q)，則這個狀態暗示了，T_i 試圖寫入 Q 的舊值。因此，此 `write` 操作被拒絕，而 T_i 則被撤回。
  - 否則，此 `write` 操作就被執行。

時間戳記為基礎的協定

T_2	T_3
read (B)	
	read (B)
	write (B)
read (A)	
	read (A)
	write (A)

圖 6.26 排班 3：在時間戳記協定下的一種可能排班

爲了說明此協定，我們來看圖 6.26 中包含交易 T_2 和 T_3 的排班 3。我們假設一筆交易在它的第一個指令之前就被設定一個時間戳記。因此，在排班 3 中 $TS(T_2) < TS(T_3)$，而且在時間戳記協定下，此排班是可能的。

Conflict Actions

- Two or more actions are said to be in conflict if:
  - The actions belong to different transactions.
  - At least one of the actions is a write operation.
  - The actions access the same object (read or write).
- Conflict example
  - T1:R(X), T2:W(X), T3:W(X)
- Not conflict example
  - T1:R(X), T2:R(X), T3:R(X)
  - T1:R(X), T2:W(Y), T3:R(X)

118

死結簡介

- 許多行程會共同競爭系統中有限的資源。
- 當行程所要求的資源正被其他的行程所使用時，該行程就必須要等待。
- 等待的行程可能會因為所要求的資源也被其他正在等待的行程所持有，而無限期地等待，這種情形稱為死結。

119

死結特性

- 系統中行程共同競爭的有限資源可以分為幾種不同的類型，而每種類型又會有不同數目的資源。
- 一個行程可以對一個資源進行要求(Request)、使用(Allocation)、釋放(Release)等 3 種動作。
- 行程必須在使用資源前先要求該種資源，必須在使用資源後將資源釋放。
- 行程不能要求比系統擁有數目更多的資源。

120

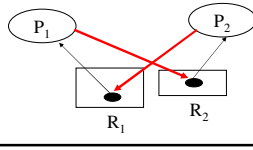
死結特性(93nttu,96tpu 3, 92tpu 1(c))

- 當行程要求了某種資源時，作業系統會先檢查系統的資源配置表，如果該種資源都正被其他行程所使用，作業系統會將目前要求的這個行程加入所等待資源的等待串列中。
- 死結的定義如下，系統中的每一個行程都在等待著某些資源，而這些資源卻已經配置給其他正在等待的行程，因而所有的行程都進入無限期地等待而無法完成工作。
- P_1和P_2目前分別持有R_1和R_2，當P_1要求R_2且P_2要求R_1時，P_1和P_2都會無限期地等待而發生死結。

121

死結特性

- 死結只有在下列 4 種條件都同時成立下才會發生：
  - 互斥：系統中至少有一種資源是不允許共享，若有某個行程正在使用該資源，其餘想要使用的行程必須等待該資源被釋放後才能使用。
  - 佔用與等待：系統中至少存在一個行程持有某項資源，並正在等待別項正被其他行程所持有的資源。



122

死結特性

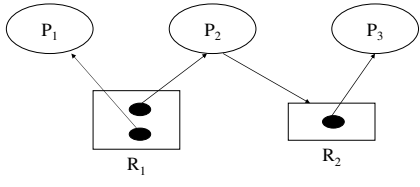
- 禁止搶先：資源一旦配置給某個行程後，一定要等到該行程將其釋放，其他行程才可使用。
- 循環等待：有N個行程P_1、…P_n，P_1正等待一項P_2持有R_2資源；P_2正等待一項P_3持有R_3資源；最後P_n正等待一項P_1持有R_1資源。

- 可以使用系統資源配置圖來描述系統中行程與資源間的狀態。
  - 圓代表系統中的一個行程，圓中寫著行程的名稱。
  - 方塊代表系統中的一種資源，方塊中黑點的數量表示該種資源的數目。

123

死結特性

- 箭號所代表的意義有兩種。
  - 由資源指向行程的箭號表示該資源目前被該行程所持有。
  - 由行程指向資源的箭號則代表該行程目前正在等待該項資源。



124

資源配置圖

- 集合P、R、E
  - P={P_1, P_2, P_3}
  - R={R_1, R_2, R_3 , R_4}
  - E={P_1→R_1, P_2→R_3 , R_1→P_2, R_2→P_2, R_2→P_1, R_3→P_3 }
- R_1有1個
- R_2有2個
- R_3有1個
- R_4有3個

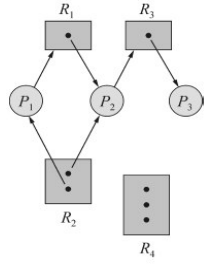


圖 7.2 資源配置圖

資源配置圖

- E={P_1→R_1, P_2→R_3 , R_1→P_2, R_2→P_2, R_2→P_1, R_3→P_3 , P_3→R_2 } P_3 要求1個 R_2
- P_1、P_2、P_3產生死結，因為系統會產生下列兩個循環
  - P_1→R_1→P_2→R_3→P_3→R_2→P_1
  - P_2→R_3→P_3→R_2→P_2

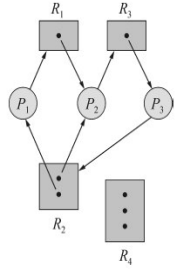


圖 7.3 含死結的資源配置圖

資源配置圖

- E={P_1→R_1, P_3→R_2, R_1→P_2, R_1→P_3, R_2→P_1, R_2→P_4}
- 雖然系統會產生循環，但是P_1、P_2、P_3、P_4不會產生死結，因為可將P_4所佔用的R_2資源分配給P_3
  - P_1→R_1→P_3→R_2→P_1

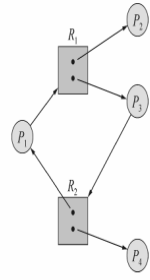


圖 7.4 含循環但無死結現象之資源配置圖

死結偵測

- 如果系統中所有類型的資源都只有一項的話，可以利用資源配置圖來偵測死結。
  - 利用偵測迴圈的演算法來檢查資源配置圖中是否有迴圈存在，就能知道目前系統中是否有死結發生。
  - 使用這種方法的系統必須要持續地更新資源配置圖，並且要定期地執行偵測迴圈的演算法以偵測死結。
- 如果系統中各類型資源的數目不只一項的話，可以使用死結偵測演算法來偵測死結。

128

死結偵測演算法資料結構

- n個行程及m種資源，偵測演算法要以下資料結構(slice 125)
  - Available陣列：一長度為m的陣列，存放目前未配置的各種資源數目。Available[i]=3表示目前R_i資源仍有3項未配置給任何行程
  - Allocation陣列：一長度為n*m的陣列，記錄每個行程目前所持有的各種資源的數量。Allocation[i,j]=2表示目前P_i正持有2項R_j資源。
  - Request陣列：一長度為n*m的陣列，記錄目前每個行程對每種資源所要求的數量。Request[i,j]=5表示P_i要求5項R_j資源，以完成工作。

129

死結偵測演算法

1. 宣告兩個長度分別為m(資源種類)與n(行程數)的陣列Work[m]與Finish[n]，並將Work初始化為Available陣列；若Available[i]≠0則Finish[i]=False，否則Finish[i]=True。
2. 尋找i使得Finish[i]=False且Request[j,i]≤Work[i]，若找不到這樣的資源i，執行步驟4。
3. Work[i]=Work[i]-Request[j,i]; Finish[i]=True; 執行步驟2。
4. 若Finish陣列中有任何的元素為False，表示系統目前發生死結，若Finish[i]=False，表示P_i目前處於死結狀態。

130

範例

Request資源 / 行程	R1	R2	R3
P1	3	0	4
P2	0	2	5
P3	7	3	0

資源	Available[]
R1	10
R2	5
R3	8

Process i=1, allocation[1,1]=3, allocation[1,3]=4,
work[1]=7,work[2]=5,work[3]=4, finish[1]=true;
Process i=3, allocation[3,1]=7, allocation[3,2]=3,
work[1]=0,work[2]=2,work[3]=4, finish[3]=true;
Process i=2, allocation[2,2]=2, allocation[2,3]=?,
work[1]=0,work[2]=0,work[3]=?, finish[2]=false;

131

死結解除

- 當偵測到死結發生，有兩種方法可以解除死結。
  - 終止一些行程，使循環等待的條件不成立。
  - 由發生死結的行程中回收一些資源給其他行程，使禁止搶先的條件不成立。

132

死結解除

- 終止行程的作法中，可以選擇終止所有行程或是一次只終止一個行程，直到死結的狀態解除。
  - 終止所有行程的作法雖然比較簡單，但是行程的工作必須重新或是部分執行，會浪費較多的 CPU 時間。
  - 一次終止一個行程所浪費的 CPU 時間較少，但是每終止一個行程之後，都必須要執行一次偵測死結的動作，來判定死結是否已經解除。

133

死結解除

- 用回收資源的作法來解除死結，必須持續地由一些行程回收資源，並將回收的資源配置給其他的行程，直到死結的狀態解除。這個作法有幾點需要注意：
  - 選定行程的方法：選擇花費最少的行程，例如選擇已執行時間最短、或是持有資源最少的行程。
  - 回溯：回收資源後，必須要將行程回溯到一個安全的狀態，或將行程由一個安全的狀態重新開始執行，但必須系統記錄所有行程的資訊才能達成。

134

死結解除

- 飢餓現象：不能每次都選定同一行程回收資源，如此會使某些行程無法完成工作，若以資源浪費較少做為選定行程的考量，很容易出現飢餓的情形，系統可設定每個行程被選中的次數上限，才能避免飢餓現象

135

死結預防

- 死結的發生要 4 個條件同時成立。
  - 互斥
  - 佔用與等待
  - 禁止搶先
  - 循環等待
- 只要破除死結的任一個條件就可以預防死結的發生。

136

互斥

- 互斥的條件只存在於不能共享的資源上。
  - 如印表機不能同時列印兩份不同的文件。
  - 可以共享的資源能夠允許多個行程同時使用，因此可以共享的資源不可能造成死結的發生。
  - 但是，我們無法讓不可共享的資源變成可共享，因此無法利用資源的互斥條件來預防死結的發生。

137

佔用與等待

- 不讓佔用與等待的情形在系統中發生，必須要讓所有的行程在取得某項資源時不得先持有任何其他的資源。
  - 行程在執行前必須要一次取得所有需要的資源，但是這個作法比較沒有效率。例如某行程P要求兩小時後才用到的資源！
  - 或是行程在取得任何資源之前必須要釋放所有持有的資源。
  - 可能造成資源使用率降低(前者)或是發生飢餓現象(後者)。

138

禁止搶先

- 禁止搶先是指資源一旦配置給行程，就必須等到行程使用完，資源才會被釋放，要解除禁止搶先可以使用以下作法：
  - 當行程持有某些資源並要求新的資源，如果所要求的資源正被使用而必須等待，該行程必須釋放所有持有的資源。
  - 當行程 A 要求某些資源，如果所要求的資源可以使用，就立即配置；但是如果所要求的資源已經配置給其他的行程 B，檢查 B 是否正在等待其他的資源，如果是便將 A 所要求的資源由 B 回收並配置給 A。

139

循環等待

- 要使循環等待的條件不會發生，我們可以將系統中的所有資源類型都事先編號。
  - 規定所有行程必須按照編號順序(如由小而大)取得資源。P須取得R3才能要求R5！
  - 行程必須要先釋放所持有編號較大的資源，才能要求編號較小的資源。
  - 會有資源使用率降低的問題。

140

死結避免

- 預防死結的方法大多會降低資源的使用率，而導致系統的效能降低。
- 死結避免雖然不完全排除死結發生的條件，但能有效地偵測系統可能發生死結的狀態，進而避免死結的發生。
  - 需要系統提供行程額外的資訊。
  - 當有行程要求資源，系統會利用這些資訊判斷是否要將資源配置給行程或者是讓行程等待以避免死結的發生。

141

安全狀態

- 安全狀態是指存在某種順序，可以讓系統按照該順序將資源配置給所需要的行程，而不會發生死結。
  - 系統正處於安全狀態，存在一組安全序列。滿足行程P_i所要求的資源數目必須小於系統目前尚未配置資源數目與所有行程所持有資源數量的總和。當某個行程完成工作後，必須要將所持有的資源釋放。
  - 如果系統不存在一組安全序列，則表示系統正處於不安全狀態中，即系統可能會發生死結。
  - 不安全狀態不一定就會發生死結，但是處於安全狀態絕對不會發生死結，這是因為系統使用資源的真正時間是無法確定。

142

範例

資源 行程	需要數目 Request	已持有數目 Allocation	還需數目 Need	未配置數目
P1	10	5	5	(13-10)3
P2	5	3	2	
P3	6	2	4	

<p2、p3、p1>or<p2、p1、p3>是安全序列，所以目前系統正處於安全狀態而不會發生死結(假設一種資源共有13項)
若此時P3要求2則會進入不安全狀態，所以不能配置給它

143

範例

資源 行程	最大需求量 Request	目前需求量 Allocation	還需數目 Need
P1	10	5	5
P2	4	2	2
P3	9	2	7

<p2、p1、p3>是安全序列，所以目前系統正處於安全狀態而不會發生死結 (假設資源共有12項)

144

資源配置圖演算法

- 資源配置圖演算法在系統配置某項資源給行程之前
  - 先將配置圖中的箭號反向。表示由行程要求某項資源轉變為行程持有某項資源
  - 然後使用偵測迴圈的演算法檢查新的配置圖中是否會出現迴圈。

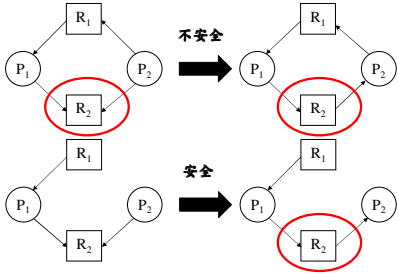
145

資源配置圖演算法

- 如果不會，則代表配置該資源後系統仍處於安全狀態，所以不會發生死結。
- 反之，則代表配置該資源後，系統會進入不安全狀態而可能發生死結，因此系統不應該配置該項資源給該行程。
- 不適用於有多項同種資源的系統之中。即使偵測出資源配置圖有迴圈，系統也不一定是處在不安全的狀態。

146

(不)安全狀態的資源配置(R2配給P2)



147

範例

資源 / 行程	需要數目 Request	已持有數目 Allocation	還需數目	未配置數目
P1	4	1	3	2
P2	6	4	2	
P3	8	3	5	

<p2、p3、p1>or<p2、p1、p3>是安全序列，所以目前系統正處於安全狀態而不會發生死結(假設資源共有10項)

148

銀行家演算法

- 當銀行有許多資金要投資到不同的地方，銀行家也必須避免銀行資金調度進入不安全的狀態。
- 每一個新進入到系統的行程必須要先註冊所需要的各種資源的最大數量。所需要的數量不可以超過系統所擁有的數量。

149

銀行家演算法(94tpu 2(d))

- 當行程要求某些資源時，系統便判斷這樣的配置是否會導致系統進入不安全狀態。如果不會才允許配置該資源。
- 使用安全演算法來測試系統是否處在安全狀態。
- 使用資源要求演算法來決定是否允許資源的要求。

150

銀行家演算法資料結構

- n個行程及m種資源，銀行家演算法要以下資料結構
  - Available陣列：一長度為m的陣列，存放目前未配置的各種資源數目。Available[i]=3表示目前R_i資源仍有3項未配置給任何行程。Available[i]=系統資源總量-$\sum_j$ Allocation[j][i];
  - Max陣列：一長度為n*m的陣列，記錄每個行程對每種資源所需要的數目， Max[i,j]=5表示目前P_i需要5項R_j資源以完成工作。
  - Allocation陣列：一長度為n*m的陣列，記錄每個行程目前所持有的各種資源的數量。Allocation[i,j]=2表示目前P_i正持有2項R_j資源。
  - Need陣列：一長度為n*m的陣列，記錄目前每個行程仍需要每種資源的數量。Need[i,j]=2表示P_i仍需要2項R_j資源，以完成工作。 Need[i][j]=Max[i][j]-Allocation[i][j];

151

安全演算法

1. 宣告兩個長度分別為 m(資源種類)與n(行程數)陣列Work[m] 與 Finish[n]，並將 Work 初始化為 Available，Finish 陣列中所有元素初始為 False。
2. 尋找 i 使得 Finish[i] = False 而且 Need[j][i] ≤ Work[i]，如果找不到這樣的 i，執行步驟 4。
3. Allocation[i]= Allocation[i]+Need[i];
   Work[i]=Work[i]-Need[j][i]; Finish[j]=True;
   Work[i]=Work[i]+Allocation[j][i]; 執行步驟 2。
4. 如果 Finish 陣列中所有元素都為 True，則系統目前處於安全狀態中。

152

安全演算法

對每一行程Pi，使得

$$\forall i, Need(i) \leq \sum_{i \neq j, for\ some\ j} Allocation(j) + Available(j)$$

則表示系統處於安全狀態

153

資源要求演算法(Resource Request Algorithm)

1. 宣告 n × m 的 Request 陣列存放行程所要求各項資源的數量，如 Request[i, j] = 3 表示行程 P_i 要求 3 項資源 R_j。
2. 如果 Request[i] ≤ Need[i]，執行步驟 3；否則因為行程要求過多的資源而發生錯誤。
3. 如果 Request[j][i] ≤ Available[i]，則執行步驟 4；否則因為目前系統中尚未配置的資源不足，行程 P_i 必須等待。
4. 作以下的運算：

   Available[i] = Available[i] – Request[j][i];
   Allocation[i] = Allocation[i] + Requset[j][i];
   Need[j][i] = Need[j][i] – Request[j][i];

   使用安全演算法檢驗運算後的結果，如果處於安全狀態則允許配置該資源給 P_i；否則P_i 必須等待，並且回存步驟 4 執行前的結果。

154

行程與資源的配置(90tku(4),94tku(2), 97nttu(5), 92tpu 3,97tpu 2)

	Max[5][3]			Allocation[5][3]			Available[3]		
行程	需要資源數目			持有資源數目			系統未配置資源數目		
	A	B	C	A	B	C	A	B	C
P_1	8	0	2	5	0	0 [3 0 2]	3	1	2
P_2	5	2	1	3	1	0 [2 1 1]			
P_3	1	2	2	0	1	2 [1 1 0]			
P_4	7	6	4	2	5	2 [5 1 2]			
P_5	3	3	5	0	2	3 [3 1 2]	[3 0 1]→ [0 1 1]		

<p5、p3、p2、p1、p4 >是安全序列，所以目前系統正處於安全狀態而不會發生死結(假設3種資源分別有13、10、9項)

155

行程與資源的配置

- 若行程P5要求資源(3，0，1)，經銀行家演算法後，發現配置給P5後仍可使系統處於安全的狀態，所以允許配置資源給P5
- 若配置資源給後P5 ，行程P4要求資源(0，1，2)，經銀行家演算法後，發現配置給P4後會使系統處於不安全的狀態，所以不允許配置資源給P4

156

行程與資源的配置

	Max[3][4]				Allocation[3][4]				Available[4]			
行程	需要資源數目				持有資源數目				系統未配置資源數目			
	A	B	C	D	A	B	C	D	A	B	C	D
P_1	1	2	3	4	1	1	1	1	5	4	5	4
P_2	2	1	2	1	1	0	1	0				
P_3	4	3	2	1	2	2	0	0				

若P3要求(4 6 2 4)系統可以配置給它嗎？(假設4種資源分別有9、7、7、5項) Need[][] Available[]?

157

行程與資源的配置

	Max[4][3]			Allocation[4][3]			Available[3]		
行程	需要資源數目			持有資源數目			系統未配置資源數目		
	A	B	C	A	B	C	A	B	C
P_1	6	3	4	0	1	0	4	4	5
P_2	4	1	1	2	0	0			
P_3	7	3	2	2	1	2			
P_4	2	3	5	1	0	0			

若P2要求(2 1 1)系統可以配置給它嗎？(假設3種資源分別有9、6、7項) Need[][] Available[]?

158

行程與資源的配置

	Max[5][3]			Allocation[5][3]			Available[3]		
行程	需要資源數目			持有資源數目			系統未配置資源數目		
	A	B	C	A	B	C	A	B	C
P_1	7	5	3	0	1	0	3	3	2
P_2	3	2	2	2	0	0			
P_3	9	0	2	3	0	2			
P_4	2	2	2	2	1	1			
P_5	4	3	3	0	0	2			

若P2要求(1 0 2)系統可以配置給它嗎？(假設3種資源分別有10、5、7項)若P5要求(3 3 0)系統可以配置給它嗎？Need[][]?

159

行程與資源的配置

2. Consider the following snapshot of a system.

	Allocation				Max				Available			
	A	*B*	*C*	*D*	*A*	*B*	*C*	*D*	*A*	*B*	*C*	*D*
P_0	0	0	1	2	0	0	1	2	1	5	2	0
P_1	1	0	0	0	1	7	5	0				
P_2	1	3	5	4	2	3	5	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				

(a) What is the content of the matrix *Need*? (5%)

(b) List the safe sequence if it exists. (10%)

(c) If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately? Explain the reason of your answer. (5%)

死結偵測

- 一系統有2部印表機，2部終端機，1部磁帶機，假設系統有3個行程，其狀態如下：

  P1正使用1部終端機，還需要1部印表機

  P2正使用1部終端機及印表機，還需要1部磁帶機

  P3正使用1部磁帶機，還需要1部終端機
- 請畫出系統資源分配圖？
- 本系統是否發生死結？若沒有，請列出其安全序列？

161

死結的偵測

- 具單一資源的型式

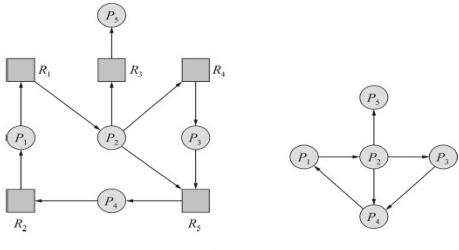


圖 7.8 (a) 資源配置圖 (b) 與其等候圖

i2

死結的偵測

- **Available**：是一長度爲 m 的向量，可表示每種資源型式的可用數量。
- **Allocation**：爲一 $n \times m$ 的矩陣，可定義出每一行程所佔用之各種資源型式的數量。
- **Request**：爲一 $n \times m$ 的矩陣，可表示出每一個行程的現在需求。若 $Request[i][j] = k$，則表示出行程 P_i 還要資源型式 R_j 中之 k 個 instances。

1. 令 *Work* 與 *Finish* 分別是長度爲 m 與 n 的向量。開始時，令 $Work = Available$。對 $i = 0, 1, \cdots, n-1$，若 $Allocation_i \neq 0$，則 $Finish[i] = false$；否則，$Finish[i] = true$。
2. 搜尋滿足下列條件的 i：
   a. $Finish[i] == false$，且
   b. $Request_i \leq Work$
   若無滿足這兩條的 i 存在，則跳越至步驟 4。
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   回到步驟 2。
4. 對某些 $0 \leq i < n$ 的 i，若 $Finish[i] == false$，則系統必處於死結狀態。此外，若 $Finish[i] == false$，則行程 P_i 便陷入死結狀態。

行程與資源的配置

	Need[5][3]			Allocation[5][3]			Available[3]		
行程	需要資源數目			持有資源數目			系統未配置資源數目		
	A	B	C	A	B	C	A	B	C
P_1	0	0	0	0	1	0	0	0	0
P_2	2	0	2	2	0	0			
P_3	0	0	1	3	0	3			
P_4	1	0	0	2	1	1			
P_5	0	0	2	0	0	2			

系統是在死結狀態嗎？(假設3種資源分別有7、2、6項)
Max=Allocation+Request

164

摘要

- 要讓數個行程共享一些資料變數，必須要避免資料的不一致。
  - 有一些不同的臨界區演算法可以滿足互斥的要求，不過這些解決方式因為使用了忙碌等待而降低了系統的效能。
  - 使用號誌可以較有效率地解決大部分同步的問題。
  - 臨界區域與監督程式能較高階與方便地解決較複雜的同步問題。

165

摘要

- 同步的經典問題
  - 包括有限緩衝區、讀取者與寫入者、哲學家晚餐問題。
  - 牽涉到了大型並行控制的問題。
  - 被經常拿來測試新設計的同步機制。
- 數個行程共享一些系統資源，就可能發生死結。
  - 只有在互斥、佔有與等待、禁止搶先、循環等待等 4 個條件同時成立時才會發生。
  - 破除其中的任一個條件，可以預防死結的發生。

166

摘要

- 可以利用不同的演算法來避免死結的發生。
  - 避免死結比預防死結的演算法較不會降低系統資源的使用率，但是系統需要記錄較多有關行程的資訊。
  - 系統也可以提供偵測及解除死結的方法來處理死結，當系統偵測出死結發生的狀況
    - 可以利用終止行程。
    - 或是回收行程的資源。

167