Chapter 7

同步範例

章節目標

- 解釋有限緩衝區、讀寫器和哲學家進餐的同步問題
- 描述 Linux 和 Windows 用於解決行程同步問題的特定工具
- 說明如何使用 POSIX 和 Java 解決行程同步問題
- 設計和開發解決方案，以使用 POSIX 和 Java API 處理行程同步問題

7.1 典型的同步問題

- 這些問題被用來測試幾乎每一個新提出的同步技術
  - 有限緩衝區問題 (bounded-buffer problem)
  - 讀取者－寫入者問題 (readers-writers problem)
  - 哲學家進餐問題 (dining-philosophers problem)

7.1.1 有限緩衝區問題

- *n* 個緩衝區的池 (pool)，其中每個緩衝區可保存一項
- mutex 二進制號誌可提供存取此緩衝區池的互斥性且其初值為 1
- 號誌 empty 的初值為 *n*
- 號誌 full 的初值為 0

圖 7.1　生產者行程的結構

```
while (true) {
        . . .
    /* produce an item in next_produced */
        . . .
    wait(empty);
    wait(mutex);
        . . .
    /* add next_produced to the buffer */
        . . .
    signal(mutex);
    signal(full);
}
```


圖 7.2　消費者行程的結構

```
while (true) {
   wait(full);
   wait(mutex);
      . . .
   /* remove an item from buffer to next_consumed */
      . . .
   signal(mutex);
   signal(empty);
      . . .
   /* consume the item in next_consumed */
      . . .
}
```


7.1.2 讀取者–寫入者問題

- 假設一個資料庫可以被許多並行行程所共用
  - 讀取者(reader)：只是讀取資料庫
  - 寫入者 (writer)：更新 (讀或寫) 資料庫
- 若有兩個讀取者同時存取共用資料，並不會產生不良的結果
  - 如果一個寫入者和某些其它的行程 (讀取者或寫入者) 同時存取共用的資料時，紛亂可能因而產生
- 行程共用下述資料結構：

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```


圖 7.3　寫入者行程的結構

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```


圖 7.4　讀取者行程的結構

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```


7.1.2 讀取者–寫入者問題

- 讀取者－寫入者問題有許多變形都含有優先權
  - **第一種** (first) 讀取者－寫入者問題
    - 除非有一寫入者已獲得允許使用這共用資料，否則讀取者不需保持等候狀態
  - **第二種** (second) 讀取者－寫入者問題
    - 只要一個寫入者預備好之後，需盡快使其能撰寫共用資料
- 解決任何一個問題都可能導致飢餓
  - 在第一種狀況下，寫入者可能會飢餓
  - 在第二種狀況下，讀取者可能會飢餓

7.1.3 哲學家進餐問題

- 哲學家將他們的生活全部用於思考與吃飯
  - 一碗米飯，並不和同事交換意見，哲學家每次只能拿取一枝筷子
  - 一個飢餓的哲學家同時拿取兩枝筷子時，再放下她的兩枝筷子，且重新開始思考
- 共用資料為 ：

  semaphore chopstick[5];

  - chopstick 的每一元素之初值為 1

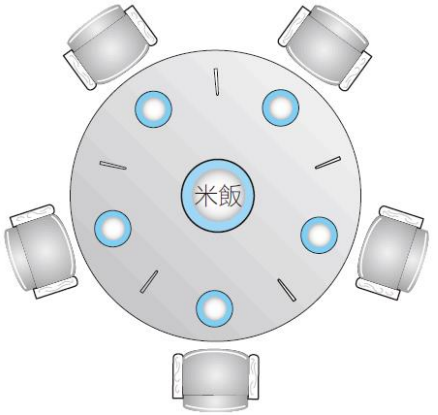

圖7.6　哲學家 *i* 的結構

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```


圖 7.7　哲學家進餐問題的監控解決方案

```
monitor DiningPhilosophers
{
   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;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }
```


圖 7.7　哲學家進餐問題的監控解決方案

```
void test(int i) {
   if ((state[(i + 4) % 5] != EATING) &&
    (state[i] == HUNGRY) &&
    (state[(i + 1) % 5] != EATING)) {
      state[i] = EATING;
      self[i].signal();
   }
}

initialization_code() {
   for (int i = 0; i < 5; i++)
      state[i] = THINKING;
 }
}
```


7.1.3 哲學家進餐問題

- 哲學家 *i* 必須在下列的執行順序中呼叫 pickup() 與 putdown() 這兩個動作：

```
DiningPhilosopher.pickup(i);
...
eat
...
DiningPhilosopher.putdown(i);
```

- 因此也不會有死結的情況發生。但是我們注意到，有些哲學家可能會「飢餓」至死

死結問題的可能解法

- 至多允許四個哲學家可同時坐在此桌旁。
- 允許一個哲學家只有在他左右兩枝筷子均為可用時，才可拾取。
- 使用一不對稱的解決方法—就是座次為奇數的哲學家先拾取他左邊的筷子，然後在拾取他右邊的筷子；而座次為偶數的哲學家先拾取他右邊的筷子，在拾取他左邊的筷子。