作業系統

Synchronization and Deadlock

B10641020 馬晨恩

目錄

內容

一、同步應用

歌劇表演在中世紀常以舞團為單位，在提出邀請的城鎮進行展演。在過去，一個舞團不一定能將所有演出服裝和道具帶到演出現場，進而出現當地服裝道具租借商來提供相關資源。租借商租借的對象不限於單一舞團，也不會同時只有一個舞團提出租借需求。然而，庫存量有限，商人需要針對當前提出租借需求的各舞團進行資源提供排程，避免資源死結。商人一開始會分配些許資源給各舞團，而資源不夠的團隊會再提出需求。若遇到無法提供完整需求給一舞團的情況，商人會等到已被租借的資源還回時，再評估該舞團的需求，拿到資源的舞團即可進行演出。

情境假設: 今晚有 5 個舞團進城展演，他們都提出了道具租借要求，也剛好都是同樣的 3 類道具: 第一類道具總共有 10 件，第二類道具總共有 5 件，第三類道具總共有 7 件。前述情境應用到本次作業，用字轉換為: **5 個執行緒競爭 3 類資源，第一類資源數為 10，第二類資源數為 5，第三類資源數為 7。**執行緒在本文中常以「緒」表示。

二、Deadlock Detection

1.程式流程:

I. 死結程式

```
/*initialize total resources */
Lock_Resource L = new Lock_Resource();

Thread T1 = new Thread(new A(L, a: 7, b: 5, c: 3, name: "T1"));
Thread T2 = new Thread(new A(L, a: 3, b: 2, c: 2, name: "T2"));
Thread T3 = new Thread(new A(L, a: 9, b: 0, c: 2, name: "T3"));
Thread T4 = new Thread(new A(L, a: 6, b: 8, c: 2, name: "T4"));
Thread T5 = new Thread(new A(L, a: 4, b: 3, c: 3, name: "T5"));

T1.start();
T2.start();
T3.start();
T4.start();
T5.start();
```

圖一: 造成死結的主程式，完整副程式請參閱附錄: DeadlockExample2.java

```
T1 is now executing.
T1 got its all first locks.
T1 got its all second locks.
T1 got its all third locks.
T5 is now executing.
Out of resources, Deadlock will occur.
```

圖二: 造成死結後的畫面

II. 死結偵測: 銀行家演算法

經考量後，個人認為程式碼資訊量太多，不容易看出重點，故在此以演算法呈現。詳細程式碼請參閱: Banker_Deadlock_Detection.java。

參數定義:

m:=執行緒數量; n:=資源種類數目; Thread i:=執行緒 i, for 0 <= i <n;

Allocation[m][n]:=各執行緒已取得的資源分配; Request[m][n]:=各執行緒對於各資源的需求;

Available[n]:=各資源剩餘可用數量; order[m]:=可分配順序;

count:=已通過安全檢驗的執行緒數目;

Finish[m]:=Boolean 陣列，得到完整需求的執行緒; Work[n]:=歸還後當前各資源可分配數量;

Method banker_detection()

Input: m, n, Available, Allocation, Request

Output:

<Begin>

Step 1. 設定所有參數配置

Step 2. 顯示所有參數配置

Step 3. Safe()

Step 4. 結束 banker_detection()

<End>

Method Safe()

Input: None

Output:

初始化 Work 為 Available

若一執行緒起初沒有得到任何資源，其對應的 Finish[i]為 false，否則為 true

<Begin>

Step 1. 找到符合以下兩個條件的執行緒:

(1) 其對應的 Finish[i]為 false (2) 其各項需求都小於 Work

若都不符合，則前往 Step 3。

Step 2. (1) Work = Work + Allocation[i] (2) Finish[i] = true

(3) order[count] = i (4) count + 1 (5)若 count=n，前往 Step 4

(6) 回到 Step 1

Step 3. 若 Finish[i] = false, i 不在 order[m]裡面, 則系統暴露在死結的風險

先透過 order[m]印出安全序列，再透過 Finish[m]印出不安全序列

結束演算法。

Step 4. 系統在安全狀態，透過 order[m]印出安全序列，結束演算法。

<End>

執行畫面:

```
System updated as below:
Thread =>Allocation => Request => Available
Thread0     0 1 0   |   7 4 3   |   3 3 2
Thread1     2 0 0   |   1 2 2   |
Thread2     3 0 2   |   6 0 0   |
Thread3     2 1 1   |   4 7 1   |
Thread4     0 0 2   |   4 3 1   |
Proceed in 3 secs...

Work updated as below:
5  3  2
Work updated as below:
5  3  4
The system is now at risk of deadlock.
Unsafe sequence: <Thread_1, Thread_4, Thread_0, Thread_2, Thread_3>
```

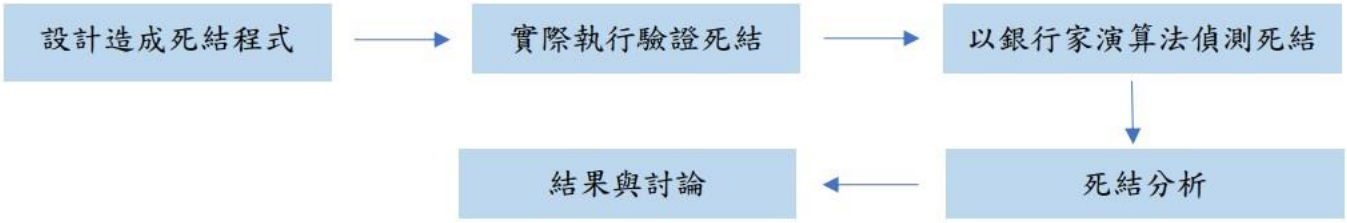
圖三: 印出各項參數配置後，演算法會逐步讓使用者了解可供額度的變化，最後找出會造成系統死結的執行緒。由上圖可看出，額度更新到第二個被分配的緒後進入不安全狀態。第三個緒開始，也就是紅色框的部分，這些緒會造成系統風險。

2.Deadlock 分析

最上方的程式為一資源有限，但各執行續總需求大過資源總數的死結程式，執行結果必然會達到死結。得到死結後，實驗使用銀行家演算法的死結偵測程式，可以觀察到初始配置和需求的總和與前述的死結程式的參數配置相同。接下來，我們可以透過幾點觀察實驗結果。首先，可以藉由零星算數，確定初始分配是正確的: 所有資源總和，Allocation + Available = (10, 5, 7)，與前述相符。再者，可以透過 Work 的更新觀察到系統有確保安全的緒得到資源。最後，當系統執行完所有安全的緒之後，面對到會造成風險的緒需求，也就是{(7,4,3), (6, 0, 0), (4, 7, 1)}。從最後一次 Work 更新可以得知: 當前可用配額為 (5, 3, 4)。而序列後三者都超過了額度，因此系統選擇不給予資源，並結束執行。這樣一來，不僅確保該風險有被偵測到，也不會為自己帶來死結。

3.實驗步驟

根據題意實驗步驟:



銀行家演算法偵測實驗步驟:

4.實驗結果與討論

在多資源多執行緒的情況下，死結的發生會來自於執行緒於自身資源之外的需求。這一點是確定的，因為一開始的資源分配若超過額度，那這個系統就不成立。不過，一個需求也不完全在超過可供額度就會造成死結。銀行家演算法可以透過非序列(non-sequential)方式，先找出可以給予資源的執行緒，讓該緒滿足需求的同時，將其手中原有的資源加回可供給額度中。倘若在這樣彈性的情況下，還有執行緒拿不到資源，就表示該緒肯定會造成系統死結，而演算法的任務就是確保系統頂多進入非安全狀態，因而不對其釋出資源。

上述的演算法做了一個假設: 如果確認執行緒 i 的需求不超過當前各資源可用數量，就將該執行緒手中的所有資源加到當前可用資源。這代表演算法信任該執行緒: 既然當下已經不會產生死結，後續也不會再提出造成死結的更大需求量，自然地就會將其手中的資源歸還。但若這個假設是錯的，則死結就有可能發生，也會讓系統進入不安全的狀態。該死結會在下一輪偵測中被檢驗出，也保證系統會被警示任何死結的風險。

最後考慮到演算法時間複雜度的部分: 在最差的情況下，需要執行 n 輪偵測才能確認安全性。每一輪偵測中，所有執行緒都會被檢驗一次 (m)，其握有的資源會跟 Work 比較一次 (n)，綜合起來我們 得到 $(m * n^2)$的複雜度。

三、Deadlock 解決方法

1.原理說明

當一個新執行緒進入系統時，它必須聲明它可能需要的每種資源類型的最大請求數。這個號碼可能超過系統中的資源總數。因此當用戶請求資源時，系統必須確定這樣的資源分配能否使系統維持於安全狀態。 如果可以，則分配資源；否則，該緒必須等到其他緒釋放足夠多資源。因應演算法而設計的資料結構需要維持得當來支持後續運行；事實上，這些資料結構也代表了系統資源分布狀況。

2.程式流程

演算法呈現: 經考量後，個人認為程式碼資訊量太多，不容易看出重點，故在此以演算法呈現。此演算法有兩大重點: 1. Safety 方法判斷系統安全性; 2. Resource-Request 方法決定是否接受緒的資源請求。Safety 方法會在 Resource-Request 方法中被重複使用。詳細程式碼請參閱附錄: Banker_Deadlock_Avoidance.java。

參數定義:

m:=執行緒數量; n:=資源種類數目; Thread i:=舞團 i, for 0 <= i <n;
Allocation[m][n]:=各執行緒已取得的各資源分配; Max[m][n]:=各執行緒對於各資源的最大需求;
Need[m][n]:=各執行緒既有資源之外的需求，以{Need[i][j] = Max[i][j] - Allocation[i][j]}初始化
Available[n]:=各資源剩餘可用數量; order[m]:=可分配順序; count:=已通過安全檢驗的執行緒數目; k:=執行緒序號; Request[n]:=需求內容;
Finish[m]:=Boolean 陣列，得到完整需求的執行緒; Work[n]:=歸還後的當前各資源可分配數量;

Method banker_detection()

Input: m, n, Available, Allocation, Max
Output:
<Begin>
Step 1. 設定所有參數配置
Step 2. 顯示所有參數配置
Step 3. Safety()
Step 4. set-request()
Step 5. 離開 banker_detection 方法
<End>

Method Safety()

Input: m, n, Available, Allocation, Max
Output:
顯示所有參數設定
初始化 Work 為 Available
初始化所有 Finish[i]為 false
<Begin>
Step 1. 找到符合以下兩個條件的執行緒:
(1) 其對應的 Finish[i]為 false (2) 其對應到的 Need[i]都小於 Work
若都不符合，則前往 Step 3。
Step 2. (1) Work = Work + Allocation[i] (2) Finish[i] = true
(3) order[count] = i (4) count + 1 (5)若 count=n，跳到 Step 4 (6) 回到 Step 1

Step 3. 若 Finish[i] = false, i 不在 order[m]裡面, 則系統暴露在死結的風險
先透過 order[m]印出安全序列，再透過 Finish[m]印出不安全序列
結束演算法。

Step 4. 系統在安全狀態，透過 order[m]印出安全序列，結束演算法。

<End>

Method Resource-Request(int k, Request[n])

Input: k, Request[n]

Output:

<Begin>

Step 1. 確認 Request <= Need[k]，若不，警示當前需求大於最大請求，並返回呼叫方法

Step 2. 確認 Request <= Available，若不，警示當前需求必須等待，並返回呼叫方法

Step 3. (1) Available[i] = Available[i] - Request[i] (2) Allocation[k][i] = Allocation[k][i] + Request[i];
(3) Need[k][i] = Need[k][i] - Request[i];

Step 4. 透過 Safety()確認系統安全性，若不安全，則復原 Step 3 的更新，並給出警示

Step 5. 結束演算法

<End>

Method set-request()

Input: k, Request[n]

Output:

<Begin>

Step 1. 讓使用者決定是否繼續請求資源，若否，則前往 Step 5

Step 2. 設定 k, Request[n]

Step 3. Resource-Request(k, Request)

Step 4. 讓使用者決定是否繼續請求資源，若是，則前往 Step 1

Step 5. 離開 set-request 方法

<End>

執行畫面

```
System updated as below:
Allocation => Max => Need => Available
Thread0         0 1 0   |   7 5 3   |   7 4 3   |   3 3 2
Thread1         2 0 0   |   3 2 2   |   1 2 2   |
Thread2         3 0 2   |   9 0 2   |   6 0 0   |
Thread3         2 1 1   |   2 2 2   |   0 1 1   |
Thread4         0 0 2   |   4 3 3   |   4 3 1   |
Proceed in 3 secs...

Work updated as below:
5  3  2
Work updated as below:
7  4  3
Work updated as below:
7  4  5
Work updated as below:
7  5  5
Work updated as below:
10  5  7
No risk of deadlock
Safe sequence: <Thread_1, Thread_3, Thread_4, Thread_0, Thread_2>
```

圖四: 確認初始狀態正確以及 Max 是否造成不安全狀態

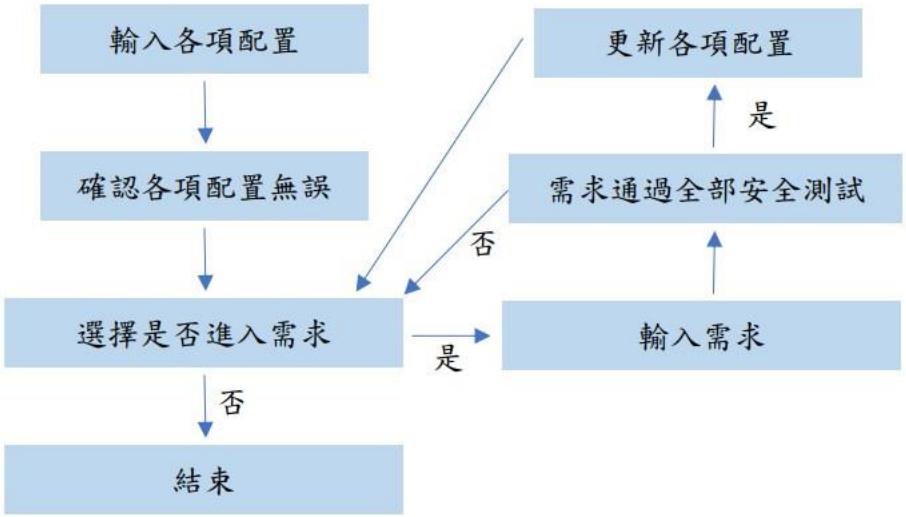
```
Proceed to Request test (Y/N)?:
Y
Thread's serial number (0,1,2,...,n-1): 1
Input of Thread_1 request (array like, separated by space):
1 0 2
Work updated as below:
5  3  2
Work updated as below:
7  4  3
Work updated as below:
7  4  5
Work updated as below:
7  5  5
Work updated as below:
10  5  7
No risk of deadlock
Safe sequence: <Thread_1, Thread_3, Thread_4, Thread_0, Thread_2>
Request test passed.

System updated as below:
Allocation => Max => Need => Available
Thread0         0 1 0   |   7 5 3   |   7 4 3   |   2 3 0
Thread1         3 0 2   |   3 2 2   |   0 2 0   |
Thread2         3 0 2   |   9 0 2   |   6 0 0   |
Thread3         2 1 1   |   2 2 2   |   0 1 1   |
Thread4         0 0 2   |   4 3 3   |   4 3 1   |
Proceed in 3 secs...
```

圖五: 使用者自行決定 Request，並通過測試，系統更新各配置

```
Continue Request test (Y/N)?:
Y
Thread's serial number (0,1,2,...,n-1): 0
Input of Thread_0 request (array like, separated by space):
0 2 0
The system is now at risk of deadlock.
Unsafe sequence: <Thread_0, Thread_1, Thread_2, Thread_3, Thread_4>
Even the Request passed, it is still unsafe to the system and thus cancelled.
Continue Request test (Y/N)?:
Y
Thread's serial number (0,1,2,...,n-1): 4
Input of Thread_4 request (array like, separated by space):
3 3 2
Request_4 exceeded the maximum claim.
Continue Request test (Y/N)?:
N
Leaving Request test...
The algorithm has come to its end.
```

圖六: 使用者自行決定 Request，但皆無法通過測試，最後結束演算法

3.實驗步驟



4.實驗結果與討論

這一大題實驗為避免數字測試不精確以及花太多時間構思變數關係，所以採用了課本的數據。演算法是一樣的，使用者帶任何數字都可以得到預期的結果。銀行家演算法在避免死結的方法上，運用了 Max 和 Need 的結構，系統一開始必須先確認 Max 和 Need 的安全性。銀行家演算法同樣地運用非序列方式，尋找可能給予資源的排序。若無法找到，則代表系統已處於不安全狀態。

進入 Request-Resource 後，使用者可以自行決定提出任一執行緒的 Request。一開始會有兩個條件篩除 Request: Request <= Need, Request <=Allocation，前者是為避免系統風險，後者是受限於系統當前可用資源，而必須等待。過了篩選後，也不代表安全，系統需測試接受該 Request 後的安全性。若是不安全，一樣會予以拒絕。綜合本段落所述並對照圖五與圖六執行結果，第一個 Request 通過所有測試後，系統更新所有參數。第二個 Request 通過前兩個測試，但依然帶有風險，因此被拒絕。第三個 Request 已超過 Need，無法被系統接受。

在避免死結的問題上，銀行家演算法同樣做了假設: Need 就是除了自身以配置的資源外，可能會有的最大 Request。因此，如果 Need 通過安全性測試的話，系統在一開始會認定安全。不過，執行緒也有可能在後續提出更大的請求。此時，Request-Resource 就能站出來當第二道關卡，為系統安全性把關。這樣一來，我們就能確認此演算法確實能有效地避免死結。最後考慮到演算法時間複雜度，我們假設使用者針對每一執行緒都會且只會提出一次 Request (m)，且每一個 Request 也都會進行一次 Safety 檢測 $(m * n^2)$，綜合起來總複雜度為 $(m^2 * n^2)$。

四、心得感想

做完了這份作業，我認為自己在三個面向有所精進: 明確了解 Detection 和 Avoidance 的差別、Java 實作作業系統程式、整合過去學科。

實作之前，我對於 Detection 和 Avoidance 兩者的區分不過是一個有 Max 結構，一個沒有，僅此而已。而實作的過程中，我發現到有程式邏輯接不上的盲點，因而再回到課本詳細的複習一次。複習完後，我明確的瞭解到: 兩者的 Finish 設定不同；Request 安全性判別方式不一樣；Avoidance 實作上可以根據 Request 更新各項參數，而 Detection 無法.....等。基於觀念的導正，我才有辦法解開盲點。

一整個學期下來，我對於作業系統的瞭解僅止於理論層面。雖然課本裡有許多實作範例，但總是因為個人的惰性而沒有去真正實作練習。我總是想著有指派作業的話再練習就好，這就是一個讓自己停止進步的因素。我在這裡特別提到 Java 是因為，以我過往的了解，Java 的應用層面大多為使用者導向，例如: Android 軟體，不清楚有到系統層面的應用。也特別是因為以往 Java 個人都是用來操作物件導向和系統分析設計的練習，不像這次作業的程式有使用到 System call，往後我就能有一份額外的應用了。

學科的整合，我想也是作業系統的一大特色。重點整合到的有兩者，一是資料結構，二是演算法。前者，我想無庸置疑的，基於程式複雜程度，沒有使用結構化設計會顯著地提升執行難度。雖然我使用到的都是以陣列為基礎的結構，不過構思如何讓幾個結構彼此間能溝通，也是資料結構學習的一大成果。演算法的整合聚焦在我以文字敘述程式流程的段落，我會思考如何讓文字盡量簡單卻又能表達出清楚的步驟。運用到學習過的演算法表達方式，將上百行的程式碼濃縮成一小段文字。此外，每一段實作結果與討論文末，我都提到演算法的複雜度分析。因為有考慮到複雜度的程式，才能算是演算法。

在學期末能有一份實作專題，也算是消除了心裡的一些疑慮。我會想說頂大名校的學生都做了一些難度不低的系統作業，我們好像差了些什麼。考試成績只能算一個數字，不能真正代表我與他們的距離有縮小了。幸好，這份作業讓我有機會稍微弭平一些疑慮。實作的同時，我常認知到自己的不足，也盡力用手邊資源補上缺口。不想草草了事，我才看見自己在這個學科上的成長。

五、參考文獻

Using Java to realize banker algorithm
https://developpaper.com/using-java-to-realize-banker-algorithm/
Java 實現銀行家演算法
https://tw.java366.com/blog/detail/cca810e4dad23f0e459c869a6f60f55b
Operating System Concept, 10e
Operating System Concepts - 10th edition (os-book.com)

六、附錄

死結程式: DeadlockExample2.java
死結偵測程式: Banker_Deadlock_Detection.java
死結解決方法程式: Banker_Deadlock_Avoidance.java