第12章 交易管理與並行控制

- 12-1 交易的基礎
- 12-2 資料庫管理系統的交易管理
- 12-3 並行控制的三種問題
- 12-4 並行控制與排程
- 12-5 並行控制的處理方法

12-1 交易的基礎-說明

Logical 邏輯
Physical 實質上 (實体)
一個 Physical 会有很多個 Logical.

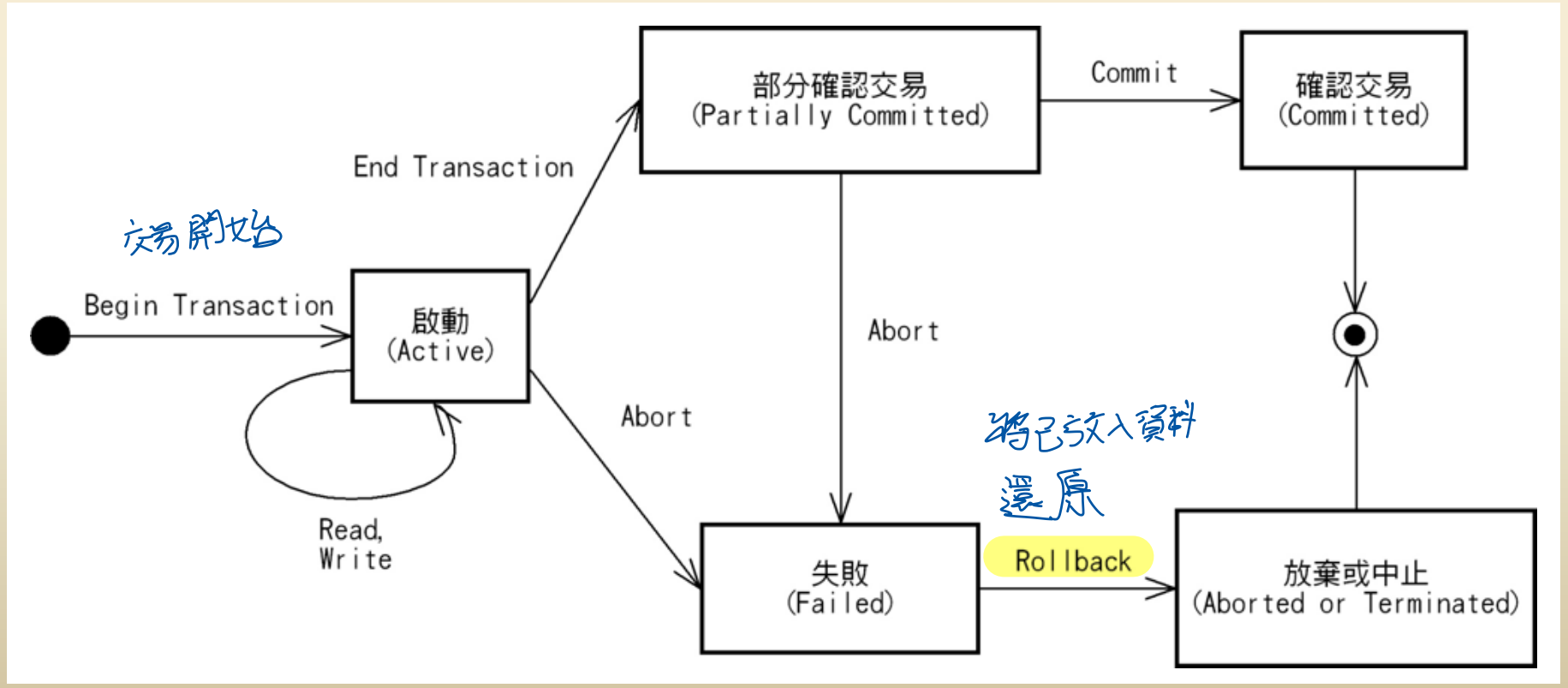
- 資料庫系統最主要的操作是存取資料庫的資料，如果有多個存取操作需要執行，而且這些操作是無法分割執行，則整個操作過程，對於資料庫系統來說是一個(**邏輯**)「**交易**」（**Transaction**），或譯成「**異動**」
- (**邏輯**)交易是將多個資料庫單元操作視為同一個不可分割的邏輯單元，這些資料庫單元的操作，只有兩種結果：一種是全部執行完成；另一種就是通通不執行，而且不可能有執行一半的情形發生
- **邏輯交易(Logical Transaction) vs 實際交易(Physical transaction)**

12-1 交易的基礎-單元操作

- 交易是將資料庫單元操作的集合視為一個不可分割的邏輯單位（Logical Unit），然後使用並行控制（Concurrency Control）和復原(回復)處理（Recovery）機制來保障交易執行成功
- 資料庫的單元操作只有兩種，如下所示：
  - 讀取（Read）：從資料庫讀取資料
  - 寫入（Write）：將資料寫入資料庫

12-1-1 交易狀態-圖例

- 資料庫管理系統執行整個交易的過程可以分成數種交易狀態（Transaction State），如下圖所示：



12-1-1 交易狀態-交易停止執行的原因有三種1

交易成功

- **交易成功就是正常結束交易的執行，以交易狀態來說，如果交易從啟動狀態開始，可以到達確認交易狀態，就表示交易成功**

交易失敗

- **交易失敗是送出放棄指令（Abort或Rollback）來結束交易的執行**
  - **放棄交易**：交易本身因為條件錯誤、輸入錯誤資料或使用者操作而送出放棄指令（Abort或Rollback）來放棄交易的執行
  - **中止交易**：因為系統負載問題或死結（Deadlock）情況，由資料庫管理系統送出放棄指令

12-1-1 交易狀態-交易停止執行的原因有三種3

交易未完成

- **交易有可能因為系統錯誤、硬體錯誤或當機而停止交易的執行，因為沒有送出放棄指令，此時交易是尚未完成的中斷狀態，即只執行到一半就被迫中斷執行。**
  - 因為資料庫管理系統並不允許此情況發生，所以在重新啟動後，其回復處理（Recovery）機制會從中斷點開始，重新執行交易至交易成功或失敗來結束交易的執行

12-1-2 交易管理的四大特性

- 單元性（Atomicity）
- 一致性（Consistency）
- 隔離性（Isolation）
- 永久性（Durability）

12-1-2 交易管理的四大特性1

單元性（Atomicity）

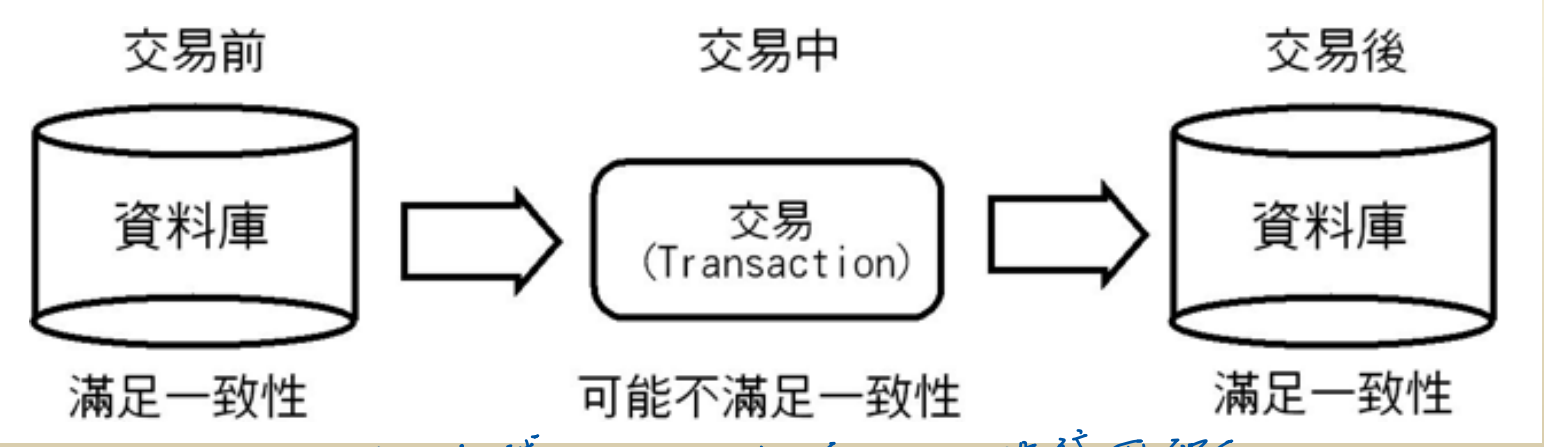
- **DBMS應保證系統中每一個交易(異動)都具備單元性(an atomic unit of processing)，即必須保證每一交易(異動)必須完全執行完畢或完全未執行**
- 單元性是將交易過程的所有資料庫單元操作視為同一項工作，不是全部執行完，就是通通不執行

12-1-2 交易管理的四大特性2

一致性（Consistency）

(正確性 Correctness)

- **DBMS應保證系統在執行交易(異動)前與交易(異動)後都在一致性的狀態下(不同的一致性的狀態)**
- 一致性是指交易可能會更改或更新資料庫的資料，不過在交易之前和之後，資料庫的資料仍然需要滿足完整性限制條件，維持資料的一致性



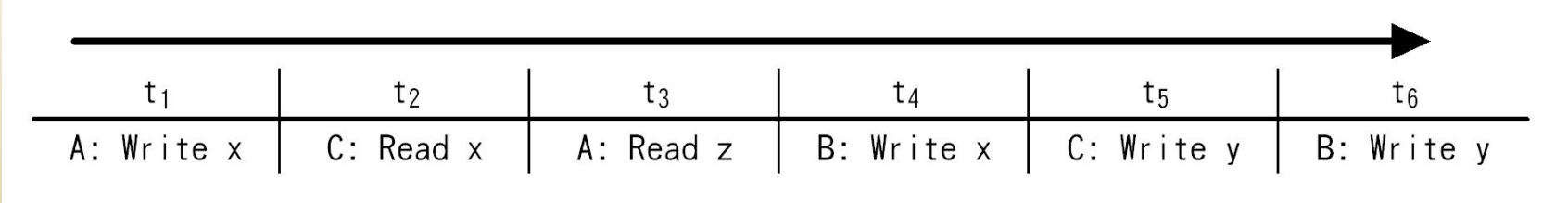
如：執行前借貸平衡、執行後仍借貸平衡。
⇒有一致性

12-1-2 交易管理的四大特性3

隔離性（Isolation）

- DBMS應保證系統所有交易(異動)在執行中，和其他交易(異動)是隔離的；即交易(異動)執行中應不被其他交易(異動)所干擾；即在執行過程中，變數值彼此獨立互不影響

解法：執行前，將資料 Lock 住。



12-1-2 交易管理的四大特性4

永久性（Durability）

透過 log file

- **DBMS應保證系統所有交易(異動)一旦確認(交付)(committed)更新後，此更新值一定會對資料庫更新，即不可因系統損壞等原因而未執行此確認(交付)更新**

系統会在損壞後再重新 Commit
(SQL Server 会將每次交易存入 ldf，避免此情況)

- 永久性（Durability）是指當交易完成執行確認交易（Commit）後，其執行操作所更動的資料已經永久改變，資料庫管理系統不只需要將資料從資料庫緩衝區實際寫入儲存裝置，而且不會因任何錯誤，而導致資料的流失

先寫入 ldf
⇒後 mdf

記憶体会暫存 ldf 檔

12-2-2 SQL語言的交易支援-ANSI-SQL(Commit)

- **ANSI-SQL的交易並沒有提供明確指令指出交易的開始，在交易執行的控制方面提供：COMMIT和ROLLBACK兩個指令，如下所示：**
  - 確認交易（Commit）：如果交易執行過程沒有錯誤，下達COMMIT指令，將交易更改的資料實際寫入資料庫，以便執行下一個交易，如下所示：

```
DELETE FROM Students WHERE sid = 'S001'
//
COMMIT
```

- **一般Commit是由程式設計師在程式內下的指令或DBMS自動下?** 一般交是DBMS下，有权限才可由程式設計師下。(資深的)

12-2-2 SQL語言的交易支援-ANSI-SQL(Rollback)

- 復原交易（Rollback）：如果交易執行過程有錯誤，就是下達ROLLBACK指令放棄交易，並將資料庫回復到交易前狀態，如下所示：

```
UPDATE Students SET birthday='1968-09-12', GPA=3.0
WHERE sid = 'S001'
//
ROLLBACK
```

X 不考

12-2-2 SQL語言的交易支援-Transact-SQL

(SQL Server)

- 在Transact-SQL的交易是使用BEGIN TRAN指令開始，如果交易成功，就使用確認交易COMMIT TRAN指令結束，如下所示：

```
BEGIN TRAN
……
COMMIT TRAN
```

- 如果交易失敗，回復交易是使用ROLLBACK TRAN指令結束，如下所示：

```
BEGIN TRAN
……
ROLLBACK TRAN
```

12-3 並行控制的三種問題

- 12-3-1 遺失更新問題
- 12-3-2 未確認交易相依問題
- 12-3-3 不一致分析問題

12-3-1 並行控制的三種問題

- 遺失更新（Lost Update）問題是指交易已經更新的資料被另一個交易覆寫，換句話說，整個交易等於白忙一場 (參見下圖)
- 未確認交易相依（Uncommitted Dependency or dirty read）問題是指存取已經被另一個交易更新，但尚未確認交易的中間結果資料(此交易後rollback)
- 不一致分析（Inconsistent Analysis）問題也稱為「不一致取回」（Inconsistent Retrievals）問題，這是因為並行執行多個交易，造成其中一個交易讀取到資料庫中不一致的資料。

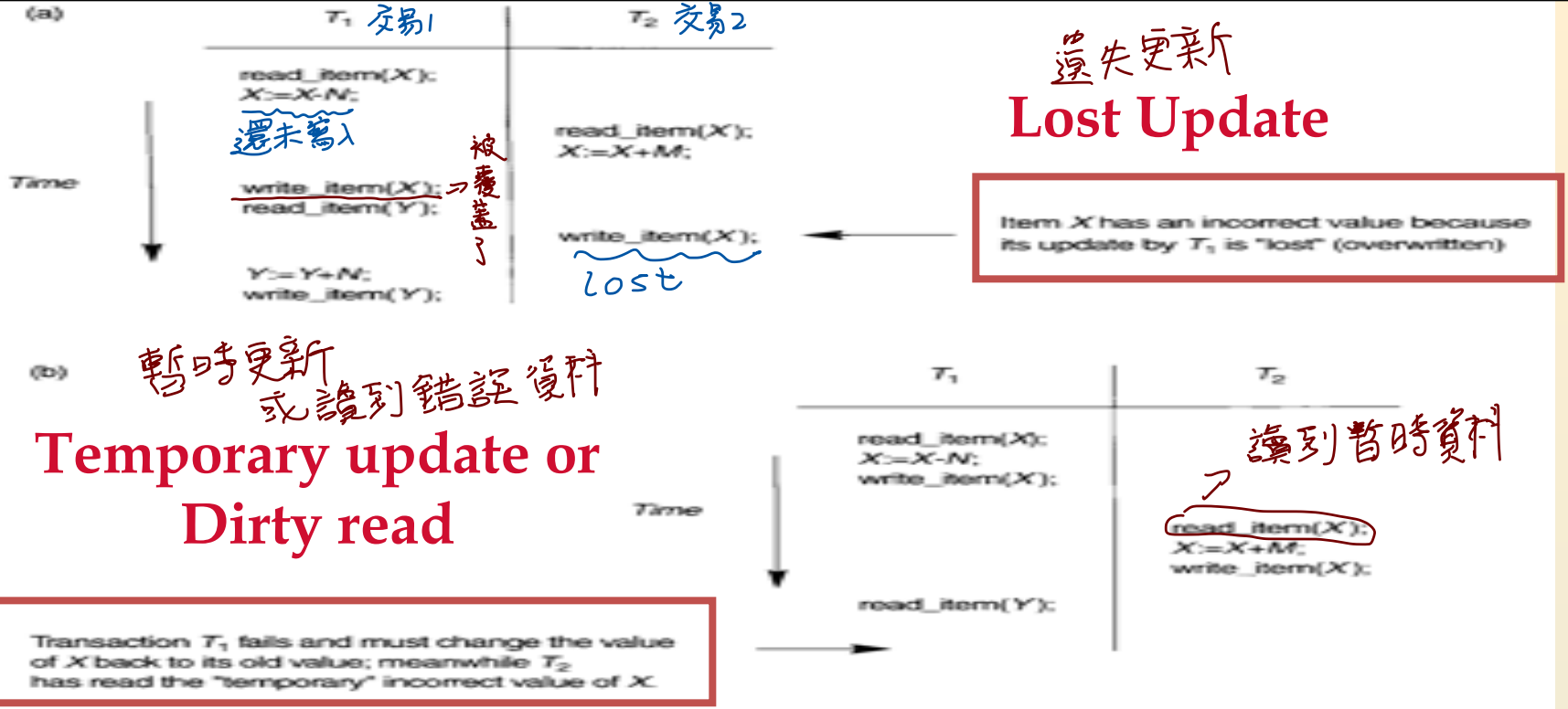


Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.

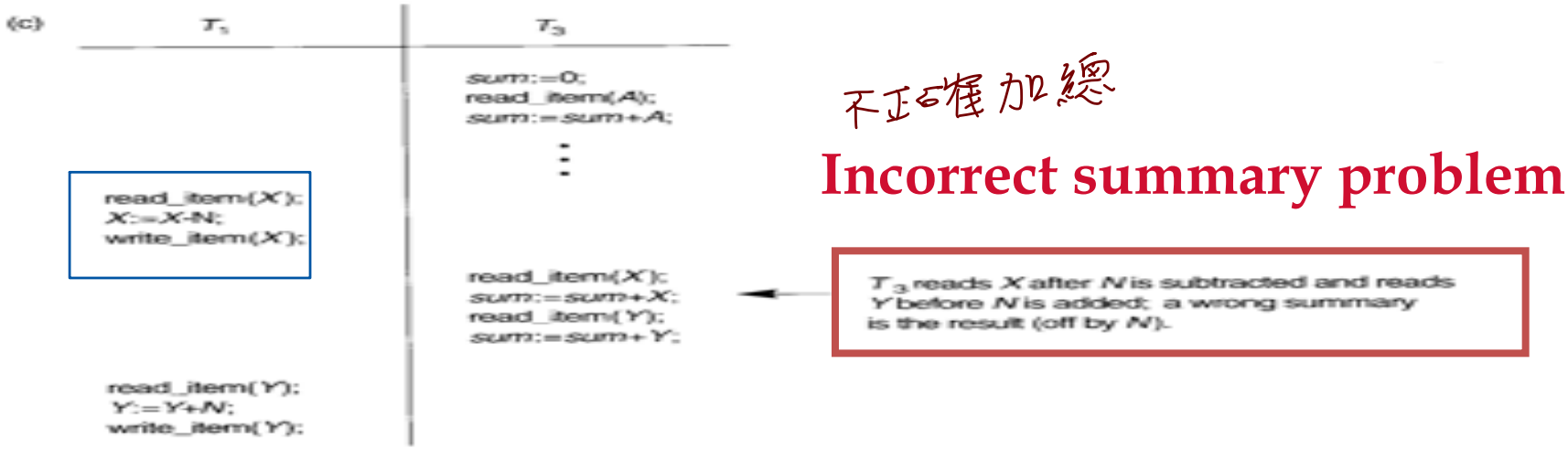


Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.

解決上述問題的想法—1/2

- **解決更新未執行的問題**
  - 當某筆資料被某交易(異動)更新但尚未確認(commit交付)前，此筆資料應限制無法讓其他交易(異動)讀取或更新
- **解決讀取暫時更新資料的問題**
  - 解決方法同上
- **解決不正確的資料加總的問題**
  - 假設交易(異動)T1 取存某些資料，除非T1已經確認(交付)或回復(committed or rollback)，否則其他交易(異動)無法對這些資料有任何的更新異動；若 T1 僅是讀取的動作，則其他交易(異動)亦可進行讀取

解決上述問題的想法—2/2

- **可以利用鎖定(LOCK)技術解決上述問題，但不當的使用鎖定技術，容易造成系統死結(deadlock)，所以，必須設法使系統執行較適當的交易(異動)行為**
  - 可接受的交易(異動)行為是可循序排程(serializable schedule)

DBMS、程式設計師都能下Lock，但通常是DBMS

12-4 並行控制與排程

- 12-4-1 並行控制的排程
- 12-4-2 排程是否是等價的
- 12-4-3 並行控制演算法
- 12-4-4 SQL語言的隔離性等級

12-4-1 並行控制的排程-排程

- 「排程」（Schedules）是在執行一系列並行交易時，各交易資料庫單元操作和運算的完整執行順序，主要是針對寫入（Write）和讀取（Read）資料庫單元操作的執行順序
- 會互相插入執行的一組交易(異動) ，但每一交易(異動)均按原次序進行

排程：當一組交易還未進來，另一組交易進來

12-4-1 並行控制的排程-循序性排程

循序性排程（Serial Schedules）

- 循序性排程是一種在各交易之間沒有並行性的排程，各交易的資料庫單元操作的執行順序並不會「交錯」（Interleaving）執行，如下圖所示：

Schedule A

時間點	交易A	交易B	交易C
t_1	Write x		
t_2	Read z		
t_3			Read x
t_4			Write y
t_5		Write x	
t_6		Write y	

12-4-1 並行控制的排程-可循序性排程

可循序性排程（Serializable Schedules）

- 非循序排程(交易間有插入的動作)，但結果和某個循序排程相同稱為可循序排程

只要跟某個循序性排程相同即可

Schedule B

時間點	交易A	交易B	交易C
t_1	Write x		
t_2			Read x
t_3	Read z		
t_4		Write x	
t_5			Write y
t_6		Write y	

12-4-2 排程是否是等價的-衝突定義

- 兩個交易的資料庫單元操作產生衝突（Conflicting）的定義，如下所示：

定義12.1：兩個交易的資料庫單元操作產生「衝突」（Conflicting），如果滿足：

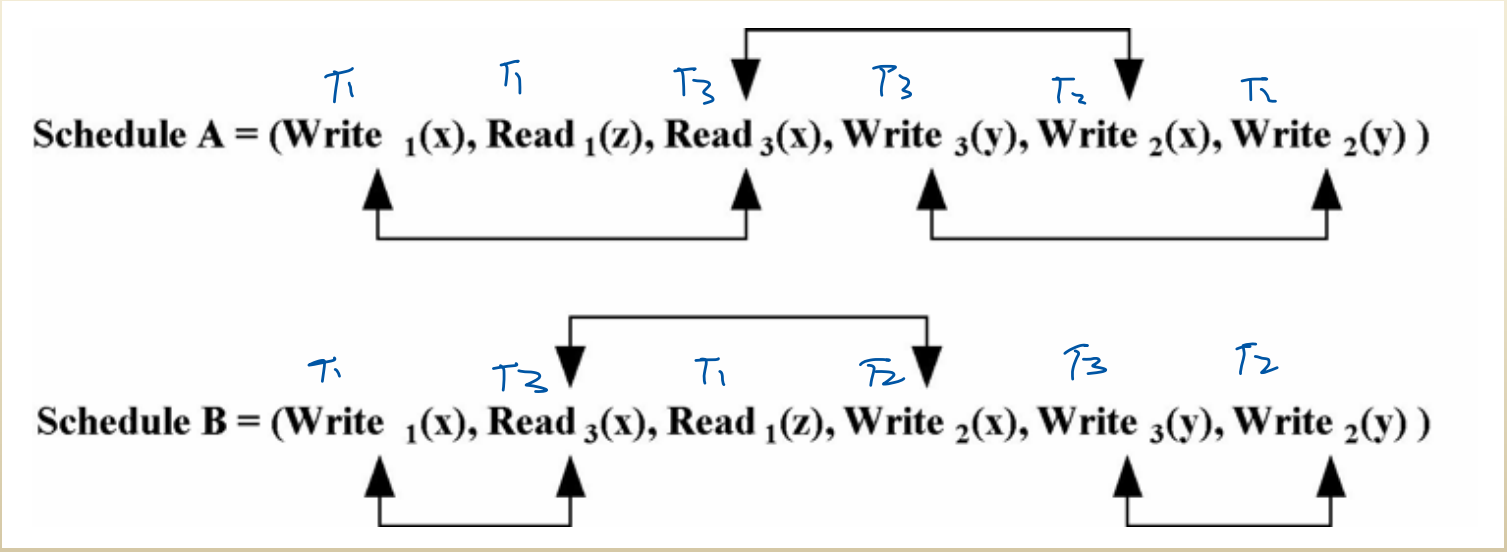
(1) 兩個交易的資料庫單元操作存取同一個資料。

(2) 兩個資料庫單元操作中，至少有一個是寫入操作。

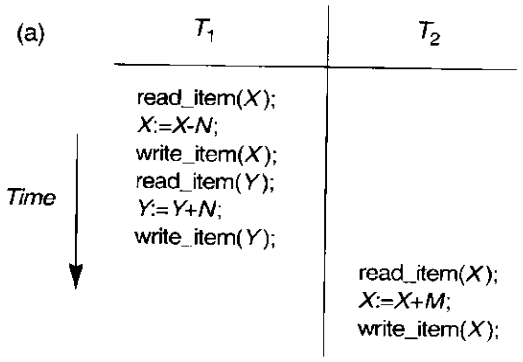
12-4-2 排程是否是等價的-排程等價

定義12.2：如果兩個排程所有衝突情況的順序是相同的，就表示這兩個排程是「等價的」（Equivalent）。

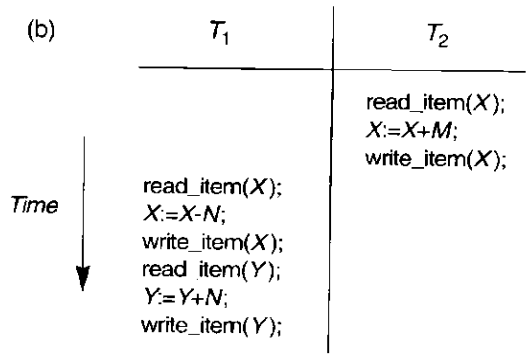
- 在Schedule A和Schedule B兩個排程可以找出3個衝突情況，依序如下所示：



Serial schedule

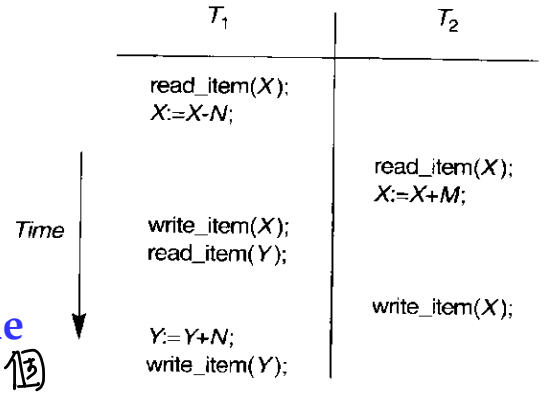


Schedule *A*

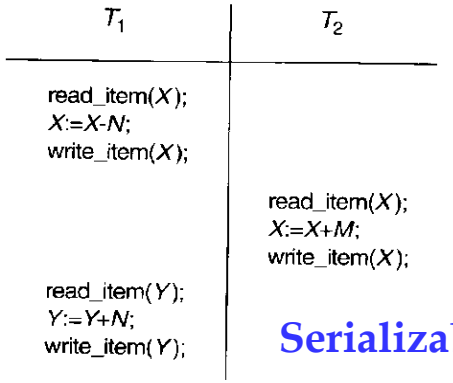


Schedule *B*

Nonserial schedule --interleaving



Schedule *C*



Schedule *D*

Non-Serializable Schedule

因為其結果不等同於任何一個 Serial Schedule.

Serializable Schedule

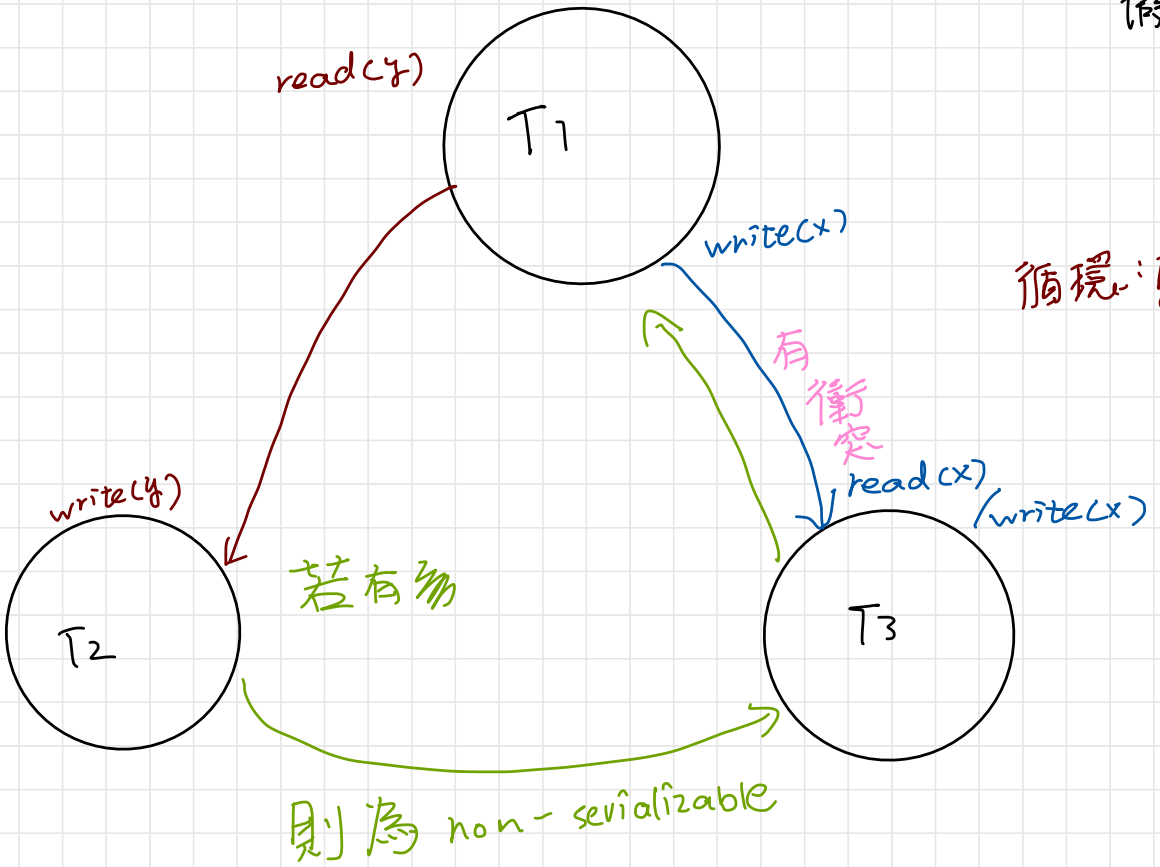
※若交易過多難以判斷，則可畫先行圖

使用先行圖(優先順序圖) (Precedence Graph) 測試是否滿足可循序性

- **演算法20.1：先行圖(優先順序圖)(*Precedence Graph*)**
  - 對排程S中每一個交易(異動)Ti ，均建一節點，註明Ti
  - 對排程S中，若有某交易(異動)Tj 在交易(異動)Ti 執行 write-item (X)後，執行 read_item(X) ，則劃一邊線 (Ti→ Tj)
  - 對排程S中，若有某交易(異動)Tj 在交易(異動)Ti 執行 write-item (X)後，執行 write_item(X) ，則劃一邊線 (Ti→ Tj)
  - 對排程S中，若有某交易(異動)Tj 在交易(異動)Ti 執行 read-item (X)後，執行 write_item(X) ，則劃一邊線 (Ti→ Tj)
  - 若最後的先行圖(優先順序圖)未形成循環，則此排程S為(衝突)可循序排程

先行圖

假設有3個交易



循環： 

Serializable Schedule

(a)

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	
	read_item(X);
	X:=X+M;
	write_item(X);

Time

Schedule A

Serializable Schedule

(b)

T_1	T_2
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(X);	
X:=X-N;	
write_item(X);	
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Time

Schedule B

non-serializable

(c)

T_1	T_2
read_item(X);	
X:=X-N;	
	read_item(X);
	X:=X+M;
write_item(X);	
read_item(Y);	
	write_item(X);
Y:=Y+N;	
write_item(Y);	

Time

Schedule C

Serializable

(d)

T_1	T_2
read_item(X);	
X:=X-N;	
write_item(X);	
	read_item(X);
	X:=X+M;
	write_item(X);
read_item(Y);	
Y:=Y+N;	
write_item(Y);	

Schedule D

Figure 19.5 Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

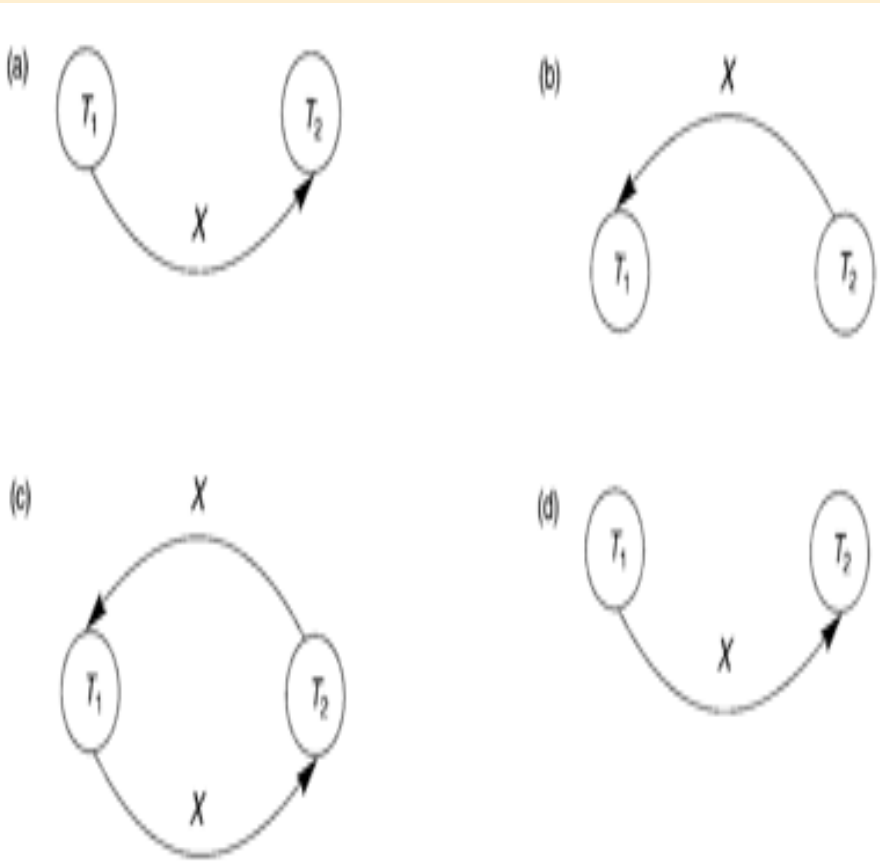


Figure 19.7 Constructing the precedence graphs for schedules A to D from Figure 19.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

(a)

transaction T_1	transaction T_2	transaction T_3
read_item (X); write_item (X); read_item (Y); write_item (Y);	read_item (Z); read_item (Y); write_item (Y); read_item (X); write_item (X);	read_item (Y); read_item (Z); write_item (Y); write_item (Z);

non-Serializable

(b)

transaction T_1	transaction T_2	transaction T_3
	read_item (Z); read_item (Y); write_item (Y);	$T_2 \rightarrow T_3$
		read_item (Y); read_item (Z);
Time ↓ read_item (X); write_item (X);	$T_1 \rightarrow T_2$	
		write_item (Y); write_item (Z);
	read_item (X);	
read_item (Y); write_item (Y);		
	write_item (X);	

$T_2 \rightarrow T_1$
$T_3 \rightarrow T_1$

Schedule E

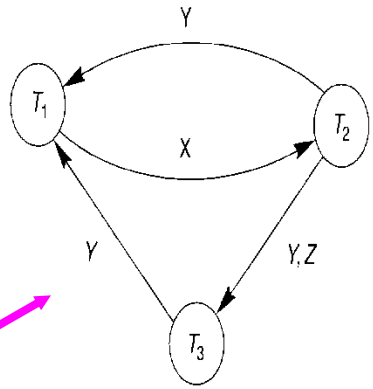
(c)

transaction T_1	transaction T_2	transaction T_3
		read_item (Y); read_item (Z);
Time ↓ read_item (X); write_item (X);		
		write_item (Y); write_item (Z);
	read_item (Z);	
read_item (Y); write_item (Y);		
	read_item (Y); write_item (Y); read_item (X); write_item (X);	

Schedule F

Figure 19.8 Another example of serializability testing. (a) The READ and WRITE operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

(d)



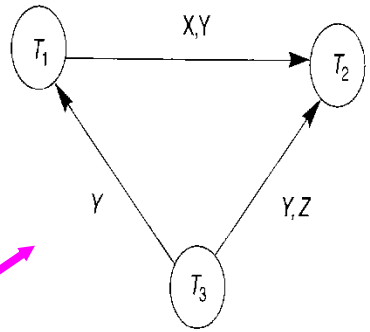
Equivalent serial schedules

None

Reason

cycle $X(T_1 \rightarrow T_2)$, $Y(T_2 \rightarrow T_1)$
cycle $X(T_1 \rightarrow T_2)$, $YZ(T_2 \rightarrow T_3)$, $Y(T_3 \rightarrow T_1)$

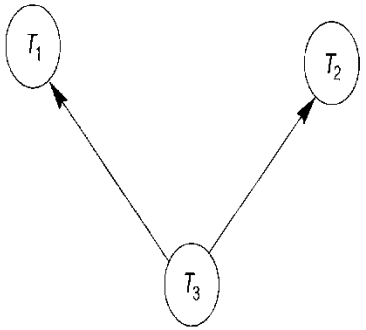
(e)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

(f)



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Figure 19.8 Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.

12-5 並行控制的處理方法

- 12-5-1 鎖定法
- 12-5-2 死結 → 等待圖
- 12-5-3 饑餓問題 → 每次交易都搶不到CPU
- 12-5-4 避免死結的並行控制處理方法

並行控制的鎖定技術

- 鎖定(*lock*)是描述允許對資料進行某種交易(異動)行為之變數

鎖定的型態

- **二元鎖定(Binary Locks)：二元鎖定有兩種狀態，鎖定與解除鎖定(locked and unlocked) ，當某資料X 被鎖定，則其他交易(異動)無法取存資料X，反之，當解除鎖定則所有交易(異動)均可取存此資料** 已無在使用
- **分享/排他鎖定(Shared/Exclusive(or read/write) Locks)：有3種狀態， read_lock(X), Write_lock(X), and Unlock(X)**
  - read-locked 又稱分享鎖定(share-locked)，允許其他交易(異動)同時讀取
  - write-locked 又稱排他鎖定(exclusive-locked)，不允許其他交易(異動)同時取存此資料

鎖定的轉變

- **當交易(異動)對某資料進行鎖定後，允許在某狀態下改變鎖定的型態稱為鎖定的轉變(*lock conversion*)**
  - 例如：某交易(異動) T 可以由 read_lock(X)升級(*upgrade*)為 write_lock(X)；同樣也允許由 write_lock(X)降級(*downgrade*)為 read_lock(X)

鎖定技術的取存協定(protocol)

- 若交易(異動)T 要讀取某資料，必須先獲得S(分享)鎖定
- 若交易(異動)T 要更新某資料，必須先獲得X(排他)鎖定，若 T已經獲得S(分享)鎖定，則DBMS系統會自動將它升級為X(排他)鎖定
- 當確認(交付)更新或放棄而回復時，X 鎖定會被解除

並行控制的鎖定解決方案

Share_lock: 讀
Exclusive_lock: 寫

- **Lost update —但會造成 dead lock**
  - **$S\_Lock_1(A)$, $R_1(A)$**
  - **$S\_Lock_2(A)$, $R_2(A)$**
  - **$X\_Lock_1(A)$—wait Xact 2 to release $S\_Lock_2(A)$**
  - **$X\_Lock_2(A)$—wait Xact 1 to release $S\_Lock_1(A)$**
- **Temporary update or uncommitted dependency**
  - **$X\_Lock_1(A)$**
  - **$S\_Lock_2(A)$—wait ($R_2(A)$)**
  - **Commit or $Rollback_1$—release $X\_Lock_1(A)$**
  - **$S\_Lock_2(A)$, $R_2(A)$**
- **Incorrect summary or inconsistent analysis—但會造成 dead lock**
  - **$S\_Lock_1(A)$, $R_1(A)$**
  - **$X\_Lock_2(B)$, $W_2(B)$**
  - **$S\_Lock_1(B)$—wait Xact 2 to release $X\_Lock_2(B)$**
  - **$X\_Lock_2(A)$—wait Xact 1 to release $S\_Lock_1(A)$**

交易(異動)排程(Schedules)

T1	*T2*
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- **Schedule：會互相插入執行的一組交易(異動)，但每一交易(異動)均按原次序進行**
  - 代表資料庫一連串實際的交易(異動)動作
  - 例如：

$R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C)$

- **初始狀態 + Schedule → 最後狀態**

允許的排程

- **循序排程(serial schedule)**
  - 排程中每一交易(異動)均一個接一個執行，沒有插入執行的動作，是可以允許的排程，稱為循序排程
  - 說明：循序排程中之交易(異動)執行次序不同，可能有不同的結果，但皆是可被接受的
- **可循序排程(Serializable schedules)**
  - 非循序排程(交易間有插入的動作)，但結果和某個循序排程相同稱為可循序排程

循序排程與可循序排程的定義

- 在排程中的每一交易(異動)其運算均一個接一個執行稱為循序排程，否則稱為非循序排程A
- 某排程其結果和某個循序排程相同稱為可循序排程

應保證可循序排程
而非測試是否可循序排程

- 若每次在交易(異動)執行中，均需測試是否為可循序排程，一旦發現非順序式(序列)排程則，必須消除此排程產生的影響，在實務上非常沒效率；所以，應該採取依照某種協定產生的排程一定是可循序排程的策略
- 有許多並行處理的協定，可以保證排程為可循序排程，最常用的技術為二階段鎖定技術 (***two-phase locking)***
  - 其他協定：時間戳記排序協定(timestamp ordering)、多版本協定(multiversion protocol)、及樂觀協定(optimistic protocol)

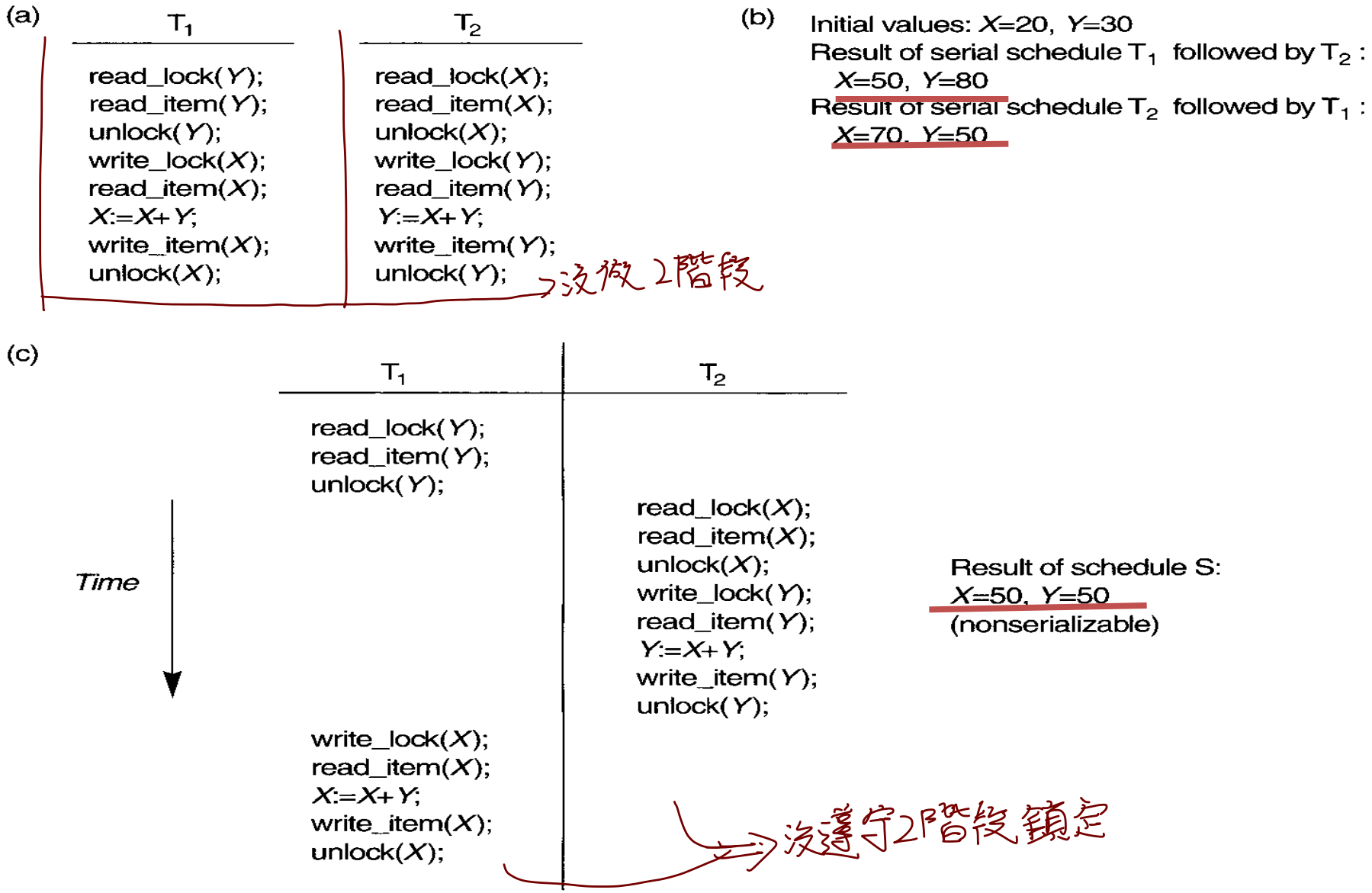


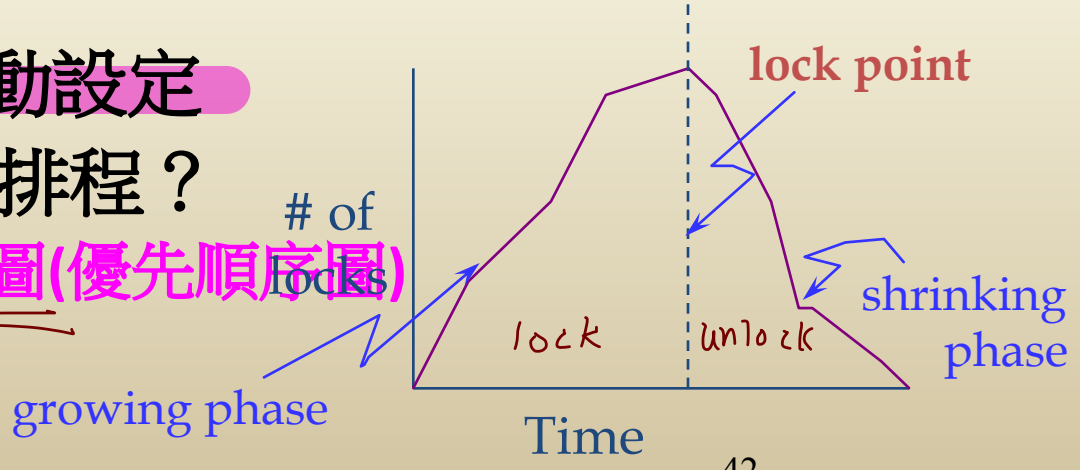
Figure 20.3 Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

使用二階段鎖定 (2PL) 保證可循序排程

- **二階段鎖定協定：交易(異動)中所有的鎖定運算(read-lock, write_lock)均在第一個解除鎖定運算(unlock)前出現，此交易(異動)分為：**
  - 擴充或成長階段(***expanding or growing (first) phase***)：此階段不斷增加鎖定，直到第一個解除鎖定為止 第一個 unlock 就只能不斷 unlock
  - 縮減階段(***shrinking (second) phase***)：此階段僅能釋放(解除)鎖定，不能再增加鎖定

基本型二階段鎖定
(Basic)Two-Phase Locking (2PL)

- **2PL**：
  - 某交易(異動)T要讀取資料前，必須先取得對此資料的分享鎖定(S lock)
  - 某交易(異動)T要維護(modify)資料前，必須先取得對此資料排他的鎖定(X lock)
  - 一旦交易(異動)T開始解除鎖定時，即不可以再增加其他鎖定
- **鎖定一般由 DBMS自動設定**
- **如何保證一定可循序排程？**
  - **不會產生循環的先行圖(優先順序圖)**

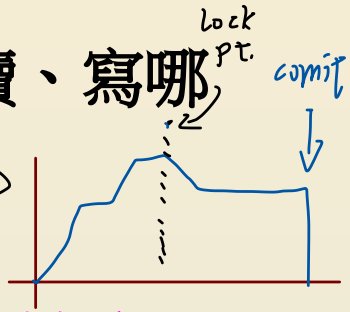


先行圖(優先順序圖)與可循序排程的關係

- 定理(Theorem)：一個排程為可循序排程若且唯若此排程不會形成循環的先行圖(優先順序圖)
- 定理： 2PL 可以保證排程的先行圖(優先順序圖)不會循環

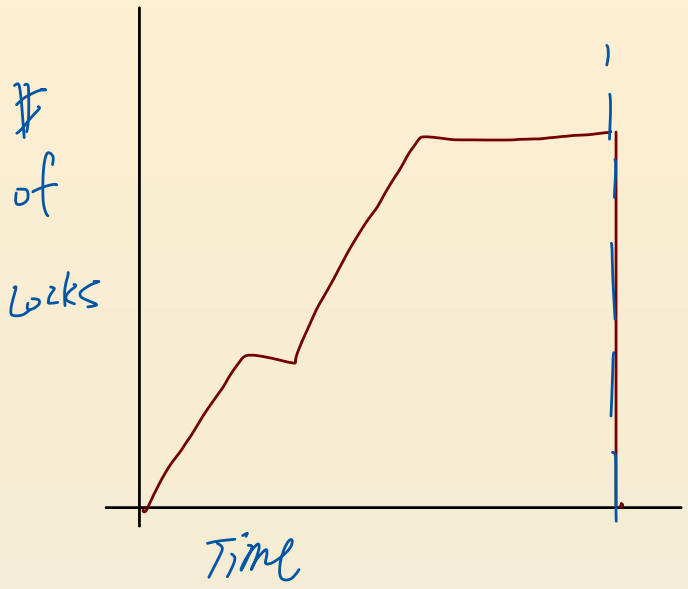
保守型又稱靜態型2PL (Conservative ， Static2PL) 、嚴格型2PL (Strict 2PL)

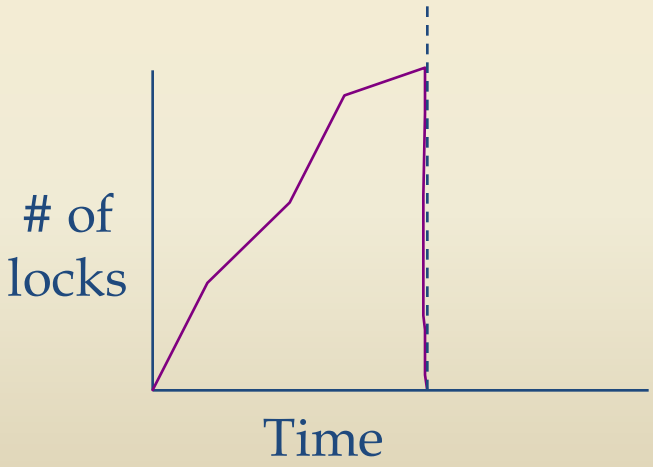
- 保守型2PL 是在執行交易(異動)前，先將所有欲取存的資料鎖定 實務上不實用
  - 可以避免產生死結的協定
  - 但一般交易(異動)可能不容易事先知道要讀、寫哪些資料，所以，實務上不實用
- 嚴格型2PL 是直到確認(交付)交易(異動)(commit)或被中斷(abort)才釋放排他性鎖定 ；所以，其他交易(異動)在此交易(異動)確認(交付)交易(異動)或被中斷前，無法取存(讀、寫)被此交易(異動)更新的資料，也同時可以確保此排程具可復原(回復)性 (recoverability)



嚴格型2PL 的擴充–
精確型2PL(Rigorous 2PL)

- **精確型2PL是必須直到確認(交付)交易(異動)(commit)或被中斷(abort)才釋放所有鎖定 (exclusive or shared)**
  - 某交易(異動)T要讀取資料前，必須先取得對此資料的分享鎖定(S lock)
  - 某交易(異動)T要維護(modify)資料前，必須先取得對此資料排他的鎖定(X lock)
  - 保有所有鎖定直到交易(異動)結束時才全部釋放
- **可以保證具可循序排程與具可復原(回復)性排程**





保守型與嚴謹型2PL的比較

- 保守型2PL在交易(異動)前即鎖定所有資料，所以，交易(異動)一開始即進入縮減期(只有縮減期)
- 嚴謹型2PL則直到確認(交付)交易(異動)或被中斷，才釋放所有鎖定，所以，整個交易(異動)在交易(異動)結束前都是擴充期(只有擴充期)

死結與飢渴
Deadlock and Starvation

- 雖然2PL可以保證所有排程均為可循序排程，但無法保證所有可循序排程均可執行(有些可循序排程被禁止執行)，且2PL使用鎖定會產生另外兩個問題：死結與飢渴(*deadlock and starvation*)
- 飢渴(Starvation)是一直被中斷無法執行的交易(異動)

死結及死結預防協定 (Prevention protocol)

- **死結發生在一組交易(異動)中，每個交易(異動)均在等待取存某個資料，但此資料卻被另外某交易(異動)鎖定住**
- **死結預防協定**
  - **時間戳記預防機制**(Timestamp prevention)
  - **非時間戳記預防機制**(Non-Timestamp prevention)
    - 使用保守型2PL機制，要求所有交易(異動)在執行前先鎖定所有欲使用之資料，若某些資料無法鎖定，則放棄所有鎖定，未來在嘗試鎖定所有欲存取的資料，但此方法會限制並行性
    - 使用不等待技術(No wait technique)，即若交易(異動)無法取得所定，則中斷倒回此交易(異動)(rollback)
  - **使用設定時間量技術**(Timeout technique)當某交易(異動)超過預設的等待時間，則將此交易(異動)中斷倒回(rollback)

不實用

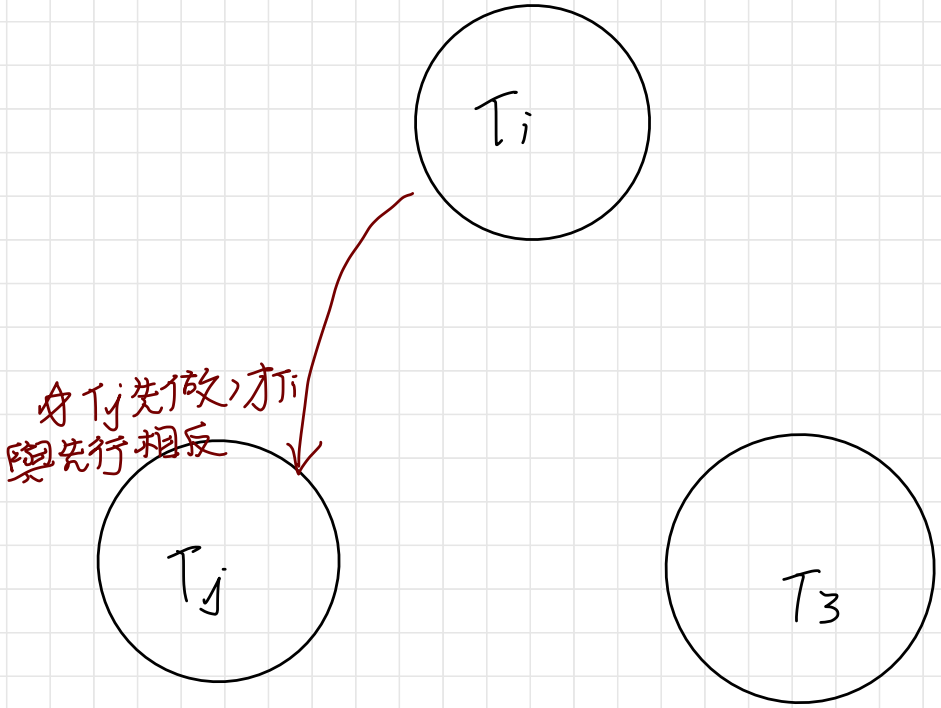
等待圖(Waits-for graph)

狗與先行圖相反

形成循環：DeadLock

- **鎖定管理程式(Lock Mgr)必須維護一等待圖**
  - 每一交易(異動)設定一節點
  - 若交易(異動)Ti等待交易(異動)Tj釋放的鎖定，則劃一由Ti到Tj的邊線
  - 最後若形成循環即為死結

等待圖



T_1'	T_2'
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

Figure 20.4 Transactions T_1' and T_2', which are the same as T_1 and T_2 of Figure 20.3 but which follow the two-phase locking protocol. Note that they can produce a deadlock.

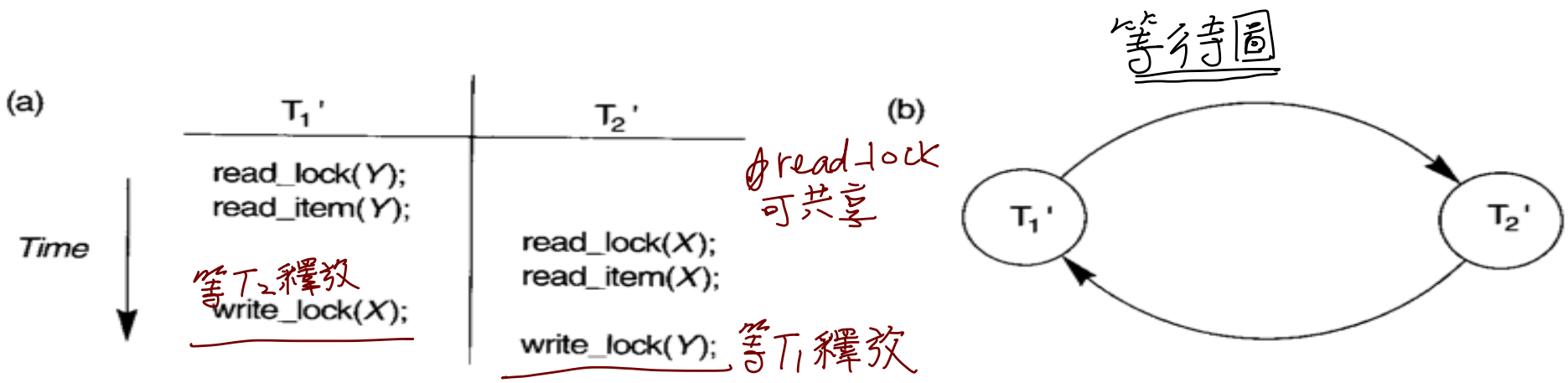


Figure 20.5 Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

死結預防機制—時間戳記協訂 timestamp protocol

- 時間戳記TS(T_i)是當交易(異動)一旦進入系統時，即由系統設定給每一交易(異動)的唯一值；若 T_1 在 T_2之前執行，則TS(T_1)<TS(T_2)
  - 較早(舊)執行的交易(異動)，其時間戳記值較小，但卻有較大的權重值(*smaller value and higher priority)*

死結預防機制—時間戳記協訂—續

X1(A), X2(B), S1(B), S2(A)

- **假設T_i要求某鎖定，但T_j已先取得衝突的鎖定(conflicting lock)(即無法和T_i共有的鎖定)**
  - **等待-中斷機制(Wait-Die)**：若$TS(T_i) < TS(T_j)$(即T_i為較舊、較先執行的交易(異動)，有較高權重值)，允許則T_i交易(異動)等待，否則中斷T_i，並在未來以同樣時間戳記重新啟動T_i
  - **中斷對方-等待機制(Wound-Wait)**：若$TS(T_i) < TS(T_j)$(即T_i為較舊、較先執行的交易(異動)，有較高權重值)，則中斷Tj並在未來以同樣時間戳記重新啟動T_j；否則T_i等待
  - **兩種機制均可避免死結**(deadlock-free) why and how!

    因為在**等待-中斷機制**只有較新的交易(異動)會等待，不會形成循環；在**中斷對方-等待機制**只有較舊的交易(異動)會等待，也不會形成循環

預防機制的另一種思維

- 理論上，死結可能會影響到一大群交易(異動)
  - T_1 等待 T_2 等待 T_3 …T_n等待 T_1
- 但實務上多數是由兩個交易(異動)造成死結
- 所以，另一種思維是不需要預防，而隨時偵測是否發生死結，發生時再修復他，此方法稱為死結偵測機制(*Deadlock detection)*

死結偵測機制
Deadlock Detection

- **鎖定管理程式(Lock Mgr)必須維護一等待圖**
  - 每一交易(異動)設定一節點
  - 若交易(異動)Ti等待交易(異動)Tj釋放的鎖定，則劃一由Ti到Tj的邊線
- **定期檢查等待圖是否形成循環，若形成循環，則中斷某交易(異動)以破壞此循環**
  - 選擇中斷的交易(異動)，最好避免已經執行很久、且已經進行多種更新運算的交易(異動)
- **另一種簡單機制，使用時間量機制(*time-outs*)**
  - 若T1 在某預設時間內無任何運算動作，即中斷之

應採取預防或偵測機制？

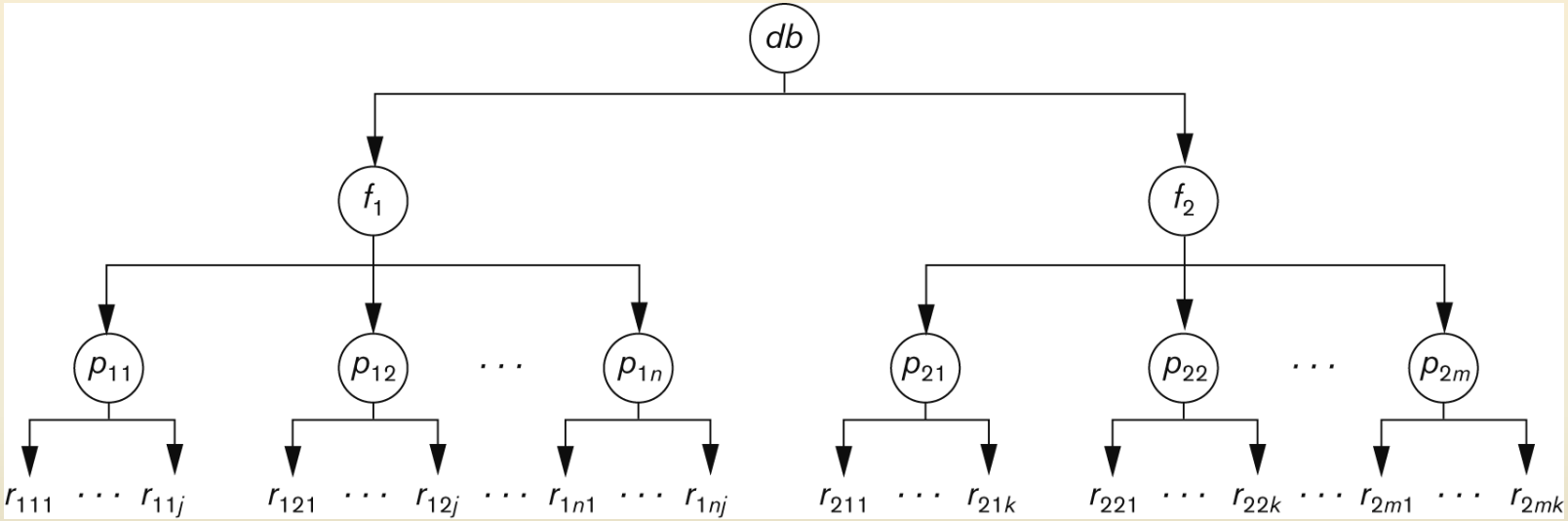
- **預防機制為防止發生死結可能必須中斷許多交易(異動)**
- **偵測機制可能會造成死結而浪費系統資源**
  - 可以偵測更頻繁，但會耗費CPU資源
- **一般採取的策略是：**
  - 採用偵測機制，因為死結發生情形並不多
  - 但若系統發生過多的死結，必須思考資料庫結構設計是否適當及系統負荷量是否太集中

資料項的鎖定範圍大小

- **資料的可鎖定單位即它的範圍大小**
  - 範圍大小可大可小
- **資料項目範圍大小對於並行控制的效能影響很大。假如範圍大則並行程度越低，範圍小則並行程度越高**
- **資料項目的鎖定範圍例如**
  - 記錄中的一個欄位值 (值組的一個屬性)
  - 一筆記錄(一個值組)
  - 一個磁碟區塊
  - 一整個檔案
  - 整個資料庫

多層範圍鎖定的階層

圖21.7 說明多層範圍鎖定機制的階層範例



多層範圍鎖定

- **所以為了有效率的實作多層範圍階層的鎖定方式，必須增加一種鎖定型態，稱為意圖鎖定 (intention lock)**
- **意圖鎖定的形態**
  - 意圖分享鎖定 (Intention-shared, IS)
  - 意圖互斥鎖定 (Intention-exclusive, IX)
  - 意圖分享與部分互斥鎖定 (Shared-intention-exclusive, SIX)

多層範圍鎖定機制的鎖定相容性矩陣

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

圖21.8 多層範圍鎖定機制的鎖定相容性矩陣

多層範圍鎖定協定

- **多層範圍鎖定協定包括下列幾項規則**
  1. 必須遵守鎖定的相容性 (參見前圖21.8)(前頁)
  2. 在任何模式下鎖定都必須從根節點開始鎖定。
  3. 只有在節點*N* 的上層節點已被異動*T* 用IS 或IX狀態鎖定時，節點*N* 才可被異動*T* 以*S* 或IS 狀態鎖定
  4. 只有在節點*N* 的上層節點已被異動*T* 用IX 或SIX 狀態下鎖定時，節點*N* 才可被異動*T* 以X、IX 或SIX 狀態鎖定
  5. 只有在異動*T* 尚未解除任何節點的鎖定時，才可對節點作鎖定動作 (為了符合2PL 協定)
  6. 只有在節點*N* 底下的子節點都沒有被異動*T* 鎖定時，異動*T* 才可對節點*N* 作解除鎖定

Lock 由上往下, unlock 由下往上

Example

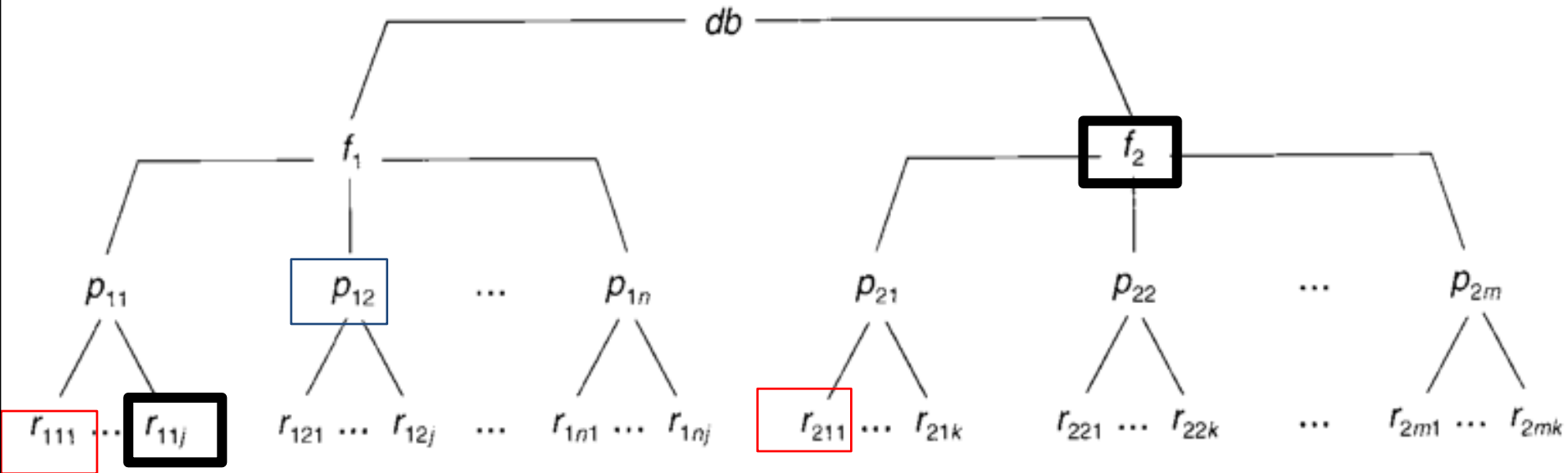


Figure 20.7 A granularity hierarchy for illustrating multiple granularity level locking.

- T1 要更新r111 及 r211資料. X-Lock, 上面I-X
- T2 要更新所有page P12的資料.
- **T3 讀取r11j 及整個f2 file的資料.**

$P_{11} \sim P_{21}$
f_1要I-S
f_2

T_1	T_2	T_3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
IX(p_{11})		IS(p_{11})
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(r_{211}) 成功，要先unlock		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(db)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(db)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(db)

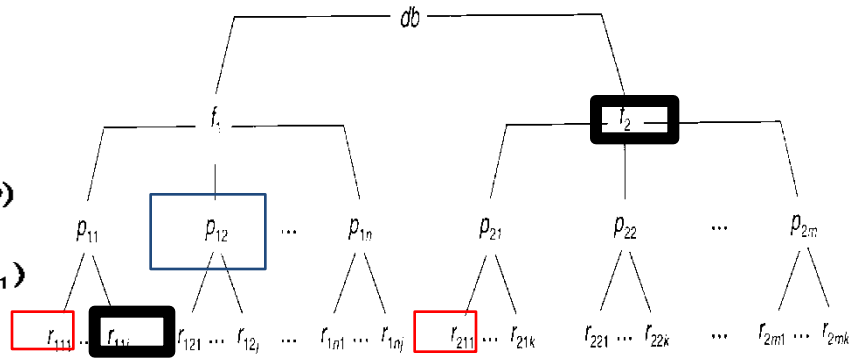


Figure 20.7 A granularity hierarchy for illustrating multiple granularity level locking.

Shows the possible serializable schedule for these 3 Xacts.

	--	IS	IX	S	SIX	X
--	√	√	√	√	√	√
IS	√	√	√	√	√	
IX	√	√	√			
S	√	√		√		
SIX	√	√				
X						

- T1 要更新r111 及 r211資料.
- T2 要更新所有page P12的資料.
- **T3 讀取r11j 及整個f2 file的資料.**

Figure 20.9 Lock operations to illustrate a serializable schedule.

圖21.9 可循序化排程的鎖定動作

- 所謂2PL是每一個交易均必須遵守一旦UNLOCK就不能再LOCK，而不同交易間必須遵守鎖定相容性：因為T1已經取得IX(F2)和T3的S(f2)不相容，所以，T1必須先解除對F2的鎖定，T3才能下S(f2)

T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)