



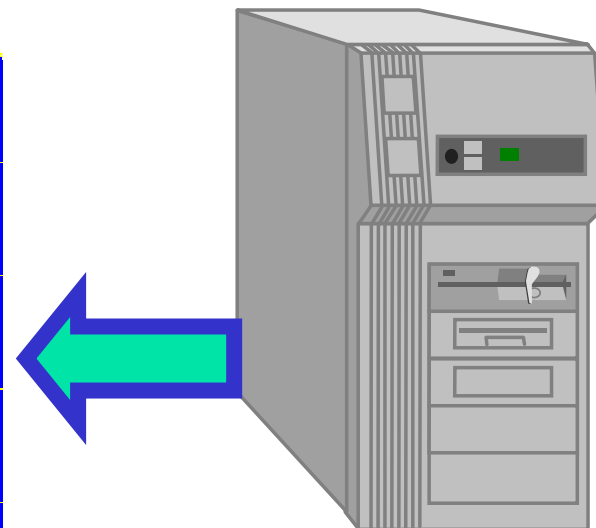
本讲（第八章）简要说明

- **授课目的与要求：** 掌握事务管理的有关概念以及并发控制的必要性及控制方法。
- **授课重点：** 事务的特性、串行化调度、两段封锁、基于时标的并发控制。
- **授课难点：** 串行化调度、两段封锁。
- **作业安排：** p263 3,4,6,15,16

第八章 事务管理

- 
-
- 两个以上查询试图修改同一数据项

帐号	借款	贷款	余额
1394567			1000	



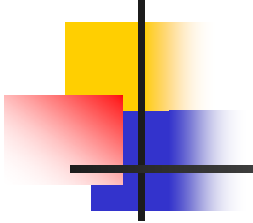
甲：
读余额1000；
取款800；
写余额200。



乙：
读余额1000；
取款500；
写余额500。



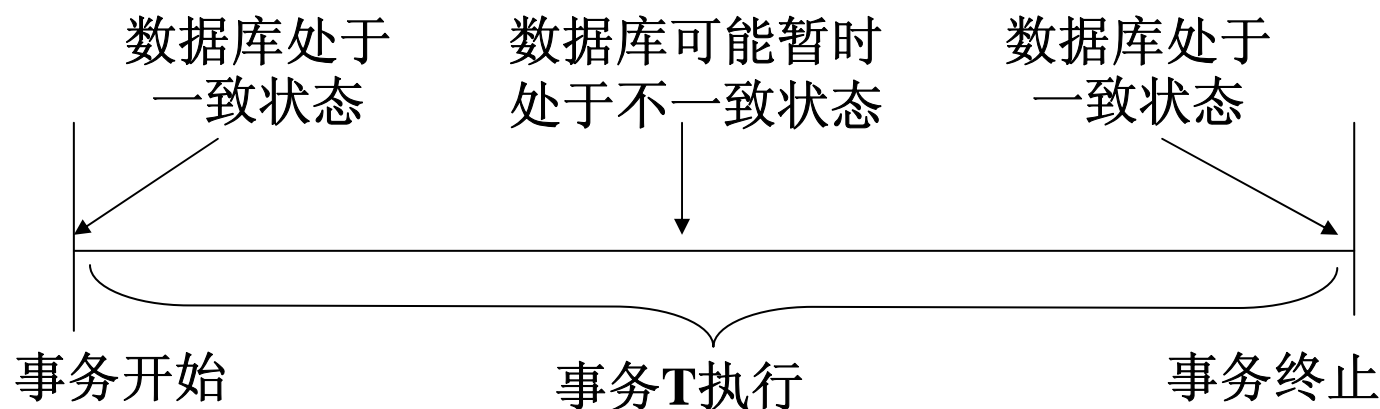
第八章 事务管理

- 
- 两个以上查询企图试图修改同一数据项
 - 系统在执行查询期间出现故障
 - 只读查询与修改查询

为此，引入“**事务**”这一术语，它在数据库领域中被作为**一致性**和**可靠性**计算的基本单位。

1.数据库的一致性

- 假如一个数据库满足其上规定的所有一致性（完整性）约束，那么我们称此数据库处于一致性状态。
- 注意：数据库可以在事务执行过程中出现暂时的一致现象。重要的是数据库应该在事务终止后，处于一致性状态。



2.可靠性

一个系统承受各种类型故障的弹性和恢复系统的能力。

一个有弹性的系统能够容忍系统故障，甚至在发生故障时能继续提供服务。

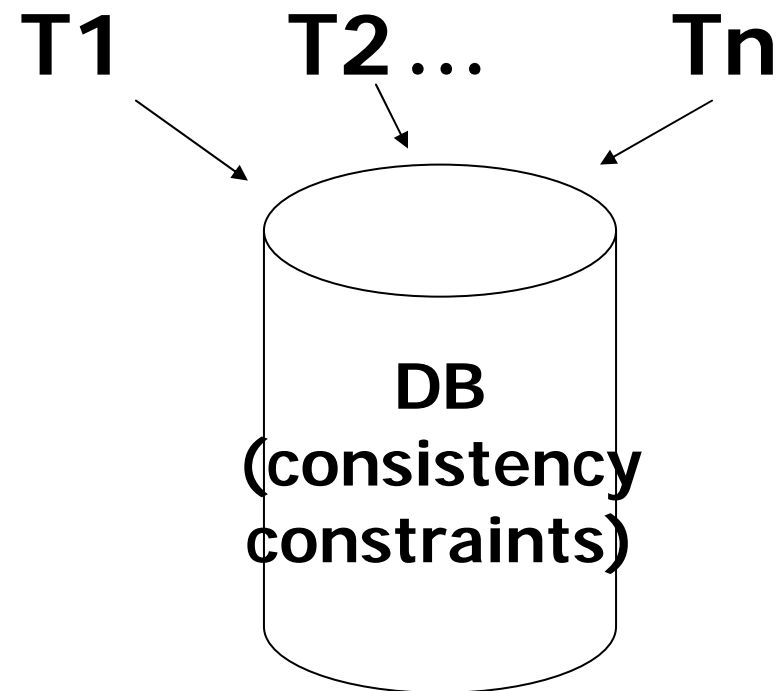
一个可恢复的**DBMS**能够在系统发生各种失败后，回到一致性的状态。

3.事务管理

负责使数据库始终保持一致性状态，甚至在有并发访问和错误发生时。



Concurrency Control



8.1 事务管理的基本概念

1. 什么是事务

直观来说，事务要在数据库上完成动作，引起状态转移，生成数据库的一个新版本。

一般来说，一个事务可以看作是由对数据库的读写操作与计算步骤一起构成的序列。

- 嵌入了数据库访问的程序
- 显示说明的一个事务
- 单个或多个**SQL**语句的集合

例1 DATABASE (C3)课程的成绩普遍增加10%.

UPDATE SC

SET GRADE= GRADE*1.1

WHERE C-NO = "C3"

这一查询交户执行就是一个事务，也可以由嵌入式SQL表示，并通过指定名字和作出如下说明使其成为一个事务。

Begin_transaction GRADE_UPDATE

Begin

EXEC SQL UPDATE SC

SET GRADE = GRADE*1.1

WHERE C-NO = "C3"

End

例 2 假设有一个航空订票数据库系统，关系**FLIGHT**记录了每次航班的有关数据，关系**CUST**记录了预定航班的所有顾客的信息，关系**FC**记录了哪个顾客预定了哪一次航班的信息。这些关系的定义如下：（有下划线的属性构成关键字）

FLIGHT (FNO,DATE, SRC, DEST, STSOLD, CAP)

CUST (CNAME, ADDR, BAL)

FC (FNO, DATE, CNAME, SPECIAL)

FNO,DATE----航班号，日期； SRC,DEST----起点，终点。

STSOLD----已售座位个数； CAP----旅客容量。

CNAME----旅客姓名； ADDR----地址； BAL----资金差额。

SPECIAL----旅客订票的特殊要求。

我们考虑一种简化的典型的订票情况：订票代办处输入航班号，起飞日期以及订票的顾客姓名，向航空公司订票。

完成此功能的事务实现如下(嵌入式SQL语言):

Begin_transaction Reservation

Begin

input(flight_no, date, customer_name); (1)

EXEC SQL UPDATE FLIGHT (2)

SET STSOLD = STSOLD+1

WHERE FNO = flight_no

AND DATE = date;

EXEC SQL INSERT (3)

INTO FC(FNO,DATE,CNAME,SPECIAL)

VALUES(flight_no, date, customer_name, null);

output("reservation completed") (4)

End.

考虑可能没有座位的情况，航空订票事务修改如下：

Begin_transaction Reservation

begin

input (flight_no, date, customer_name);

EXEC SQL SELECT STSOLD, CAP FOR UPDATE

INTO temp1, temp2

FROM FLIGHT

WHERE FNO = flight_no

AND DATE=date;

if temp1 = temp2 then

begin

output (“no free seats”);

abort

end

else **begin**

EXEC SQL UPDATE FLIGHT

SET STSOLD = STSOLD + 1

WHERE FNO = flight_no

AND DATE = date;

EXEC SQL INSERT

INTO FC(FNO,DATE,CNAME,SPECIAL)

VALUES(flight_no,date,customer_name,null);

commit;

output ("reservation completed")

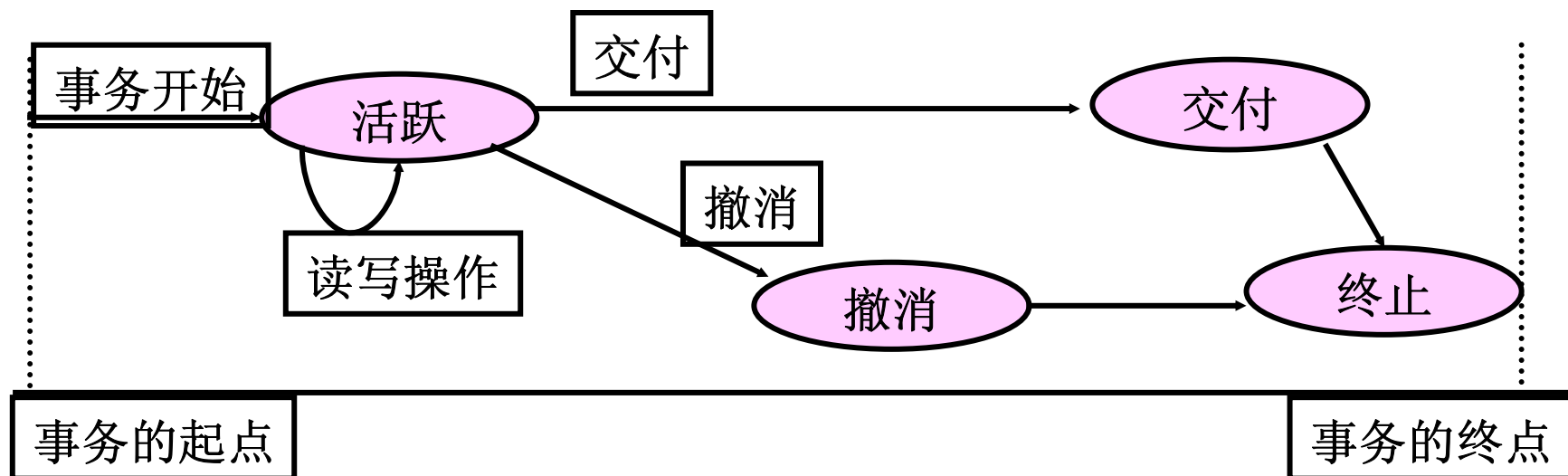
end

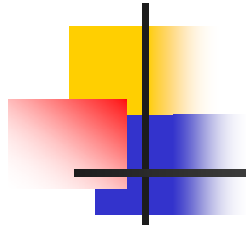
end-if

End.

2. 基本操作与状态

- 1) 事务开始：开始执行。
- 2) 事务读写：进行数据库操作。
- 3) 事务结束：完成所有操作。
- 4) 事务交付：完成所有操作，并保存所有结果。
- 5) 事务撤消：执行途中出现异常，系统或用户撤消事务。





UPDATE	}	T1
DELETE		
COMMIT		
INSERT	—	T2
CREATE	—	T3
INSERT	}	T4
...		
Log off		

3. 系统赋予事务的特性

- 1) **原子性 (Atomicity)**: 事务的所有操作或全部完成, 或全部不作。原子性在于保证正确性。如银行转帐, 取款机。
- 2) **一致性 (Consistency)**, 可串性(**Serializability**): 多个事务并发执行与它们的某一串行执行的结果等价。
- 3) **隔离性 (Isolation)**: 任何事务不能访问到其他未交付事务的中间结果, 防止多米诺效应。
- 4) **持久性 (Durability)**: 保证已交付事务的结果不丢失, 且与以后的故障无关。

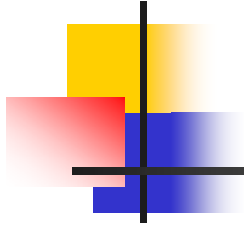
DBMS用并发控制机制维持可串性、隔离性, 用恢

1) Atomicity

- 事务的原子性要求：假如事务的执行被任何故障打断，DBMS要决定如何处理事务的故障恢复。
- 一般有两类故障。一类是**事务本身可能的故障**，由于输入数据的错误，死锁或其它因素产生失败。此时保持事务的原子性称为**事务恢复**。另一类是**系统崩溃**，如：介质故障，处理机故障，通信链路断开，断电等。这种情况下保持事务的原子性称为**崩溃恢复**。
- 这两种故障的区别是，在发生系统崩溃期间，易失性存储器（内存）中的信息可能会丢失或不能访问。

2) Consistency

- 事务的一致性就是它的正确性。
- 换一句话说，事务是将数据库的一个一致性状态转化到另一个一致性状态的正确程序。
- 事务的一致性不仅是语义数据控制（完整性约束）的重点，也是并发控制机制的目标。



例3 考虑以下两个并发事务 (T_1 和 T_2), 它们都要访问数据项 x 。假设 x 的值在它们执行之前为**50**。

T_1 :

Read(x)

$x \leftarrow x + 1$

Write(x)

Commit

T_2 :

Read(x)

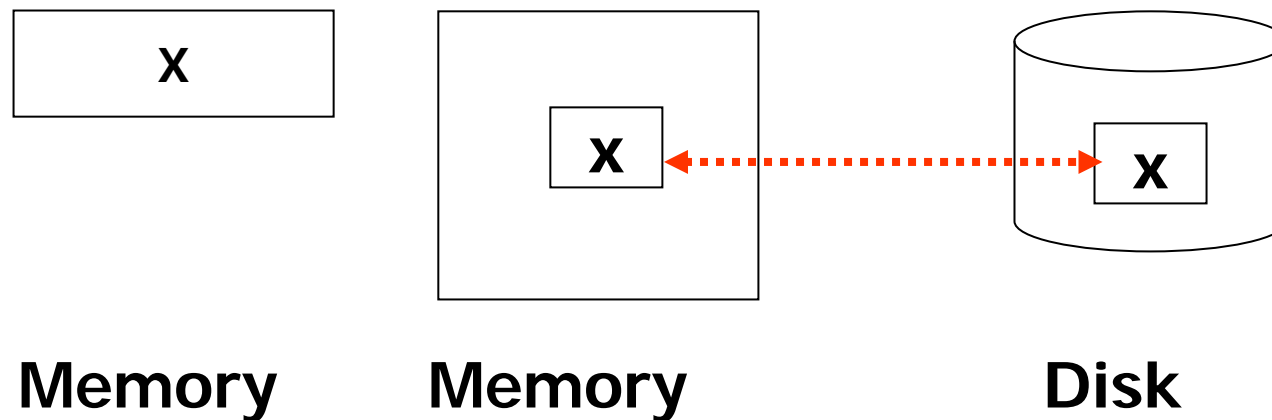
$x \leftarrow x + 1$

Write(x)

Commit

8.1 事务管理的基本概念

Storage hierarchy



- The space of disk blocks
- The virtual or main memory address space
- The local address space of the transaction

Operations:

- Input (x): block with $x \rightarrow$ memory
- Output (x): block with $x \rightarrow$ disk
- Read (x,t): do **input(x)** if necessary
 $t \leftarrow$ value of x in block
- Write (x,t): do **input(x)** if necessary
value of x in block $\leftarrow t$

例3

两个事务执行动作的一种可能的顺序：

T₁: Read(x)
T₁: x ← x + 1
T₁: Write(x)
T₁: Commit
T₂: Read(x)
T₂: x ← x + 1
T₂: Write(x)
T₂: Commit

X = 52

由于两个事务是并发的，因此事务执行的动作又有可能是以下顺序：

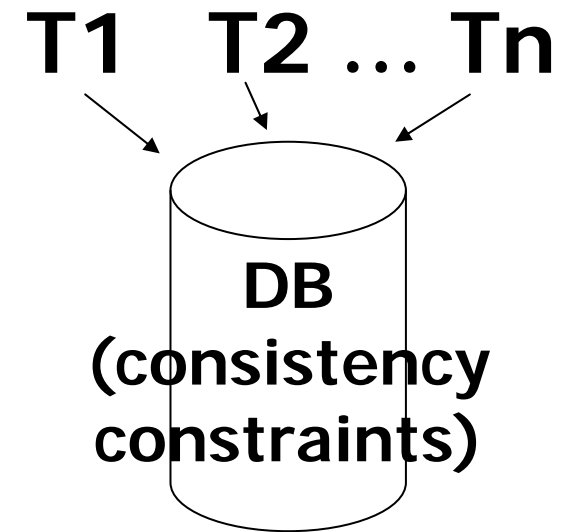
T₁: Read(x)
T₁: x ← x + 1
T₂: Read(x)
T₁: Write(x)
T₂: x ← x + 1
T₂: Write(x)
T₁: Commit
T₂: Commit

X = 51

REVIEW:

8.1 事务管理的基本概念

- 什么是事务
- 基本操作与状态
- 事务的**ACID**性质
 - 原子性 (Atomicity)
 - 一致性 (Consistency)
 - 隔离性 (Isolation)
 - 持久性 (Durability)



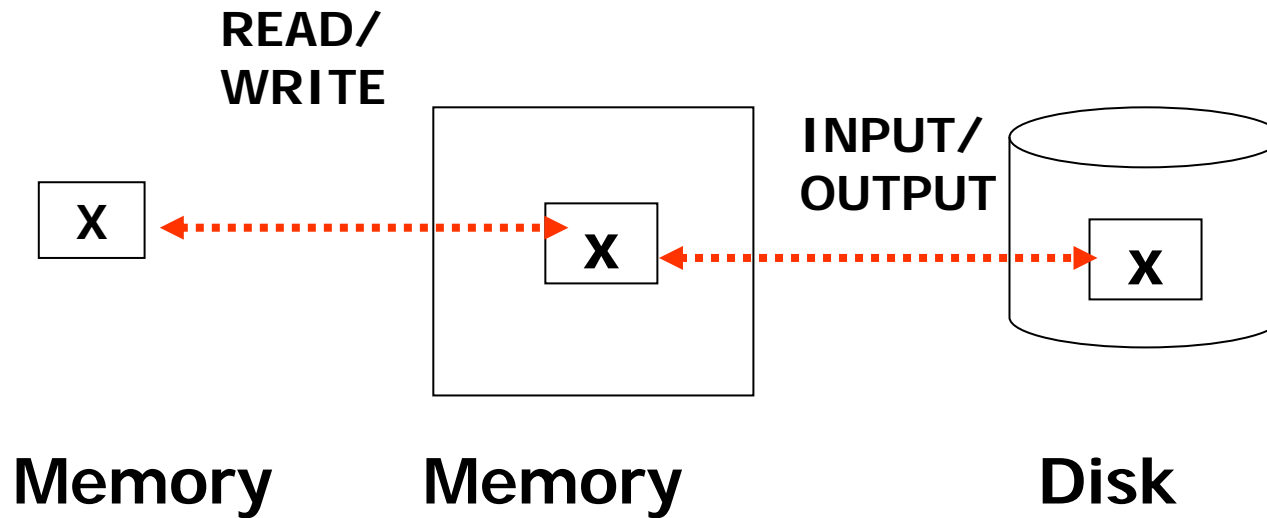
事务的ACID特性

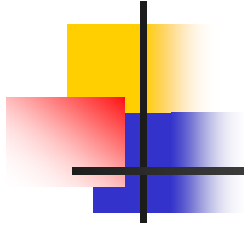
- 1) **原子性 (Atomicity)**: 事务的所有操作或全部完成, 或全部不作。原子性在于保证正确性。如银行转帐, 取款机。
- 2) **一致性 (Consistency)**, 可串性(Serializability): 多个事务并发执行与它们的某一串行执行的结果等价。
- 3) **隔离性 (Isolation)**: 任何事务不能访问到其他未交付事务的中间结果, 防止多米诺效应。
- 4) **持久性 (Durability)**: 保证已交付事务的结果不丢失, 且与以后的故障无关。

DBMS用并发控制机制维持可串性、隔离性, 用恢

8.1 事务管理的基本概念

Storage hierarchy





污数据(dirty data):

是指事务在提交前修改过的那些数据。

四种并发控制不当可能出现的现象：

Dirty Read（污读）：事务 T_1 修改了一数据项的值，事务 T_2 在 T_1 结束或撤销之前读取了此数据。如果 T_1 被撤消， T_2 就读取了数据库中不存在的值。

$\dots, W_1(x), \dots, R_2(x), \dots, A_1, \dots, C_2(\text{or } A_2)$

或者

$\dots, W_1(x), \dots, R_2(x), \dots, C_2(\text{or } A_2), \dots, A_1$

Non-repeatable (不可重读) :

事务 T_1 读了一个数据项的值，然后事务 T_2 修改或删除了此数据项后提交了。若 T_1 需再读一遍此数据项。它会读到一个不同的值或者读不到此数据。于是在同一事务中的两次读操作返回不同的结果。

$\dots, R_1(x), \dots, W_2(x), \dots, C_2 \dots, R_1(x)$

Phantom (幻影) : 当 T_1 用谓词进行搜索, 且 T_2 插入了满足此谓词的新元组, 用该谓词再次搜索结果不同。

$..., R_1(P), ..., W_2(y \text{ in } P), ..., C_2 ..., R_1(P)$

Lost updates (丢失修改) :

$..., W_1(x), ..., W_2(x), ..., C_2 ..., R_1(x)$

3) Isolation

事务的隔离性要求每个事务看到的都是一个一致性的数据库。一个正在执行的事务不能在它提交前让别的并发事务看到它的结果。解决污读、不可重读、幻影和丢失修改的问题。

保证隔离性的第二个理由是**cascading abort**(多米诺效应)。假如一个事务在它提交之前允许其它事务看到其不完全的结果，再决定撤消，那么读过它的不完全结果的事务（污读）也要被撤消。

ANSI SQL 标准定义隔离性的级别如下：

Read Uncommitted（读未提交的数据）：

对于运行在这一级别的事务，前述所有四种现象都是可能的。

Read Committed（读已提交的数据）：

Non-repeatable和**phantom**可能发生，但**dirty read**不可能发生。（**Oracle**省缺**SQLPLUS**）

Repeatable Read（可重复读）：

只有**phantom**可能发生（锁粒度为元组时）。

Serializable：

前述现象都不会发生。



SQL PLUS中修改事务隔离性:

- SET TRANSACTION
[READ ONLY | READ WRITE] |
[ISOLATION LEVEL READ
UNCOMMITTED | READ COMMITTED |
REPEATABLE READ | SERIALIZABLE]



4) Durability

持久性是指事务这样的特性，它保证事务一旦提交，事务的结果在数据库中就永久存在，不会被抹掉。所以**DBMS**要保证一个事务的结果免于以后的系统故障。持久性这一特性带来了数据库恢复的问题。也就是说，如何将数据库恢复到一致性状态。

8.2 并发控制

1.并发不当的问题

- 1) 污读
- 2) 不可重读
- 3) 幻影
- 4) 丢失修改

Serializable:

前述现象都不会发生

8.2 并发控制

2. 可串行化调度

1) 调度：对多个事务的操作的执行次序的一种安排。

设有事务， **T1: R11 (x) W12 (y)**

T2: R21 (x) W22 (x)

T3: R31 (y) W32 (y)

a. 串行调度： **T1T2T3**

S1: R11(x)W12(y) R21(x)W21(x) R31(y)W32(y)

b. 并发调度：

S2: R21(x)R11(x)W12(y)R31(y)W22(x)W32(y)

T1:read(A);

A:=A-50;

write(A);

read(B);

B:=B+50;

write(B).

T2:read(A);

temp:=A*0.1;

A:=A-temp;

write(A);

read(B);

B:=B+temp;

write(B).

调度S1

T1

read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);
read(B);
B:=B+temp;
write(B).

调度S2

T1

T2

read(A);
A:=A-50;
write(A);
read(B);
B:=B+50;
write(B).

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);
read(B);
B:=B+temp;
write(B).

调度S3

T1

read(A);
A:=A-50;
write(A);

read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);

read(B);
B:=B+temp;
write(B).

调度S4

T1

read(A);
A:=A-50;
write(A);

read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);

read(B);
B:=B+temp;
write(B).

调度S5

T1

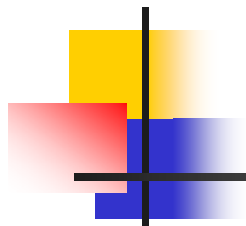
read(A);
A:=A-50;

write(A);
read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);
read(B);

B:=B+temp;
write(B).



- 串行调度的数目
- 调度的数目

8.2 并发控制

2) 两个调度的等价性

当两个调度 S_i 、 S_j 满足下面两个条件时，称为视图等价（计算等价）：

a. 在 S_i 、 S_j 中,每个读操作读取的数据项的值是由同一写操作产生的，因而读取的值相同。

b. 在 S_i 、 S_j 中,对每个数据项的最后写操作是相同的，因而写入数据库中的结果相同。

调度S3

T1

read(A);
A:=A-50;
write(A);

read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);

read(B);
B:=B+temp;
write(B).

=T1;T2

调度S4

T1

T2

read(A);
A:=A-50;
write(A);

read(B);
B:=B+50;
write(B).

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);

read(B);
B:=B+temp;
write(B).

=T2;T1

调度S5

T1

read(A);
A:=A-50;

write(A);
read(B);
B:=B+50;
write(B).

T2

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);
read(B);

B:=B+temp;
write(B).

不一致

3) 冲突的操作对与冲突等价

如果两个操作交换执行次序致使调度在视图上不等价，则称它们是**冲突**的，或者说，两个操作针对同一数据项且其中有一个写操作，则它们是冲突的。

如： $\langle R_i(x), W_j(x) \rangle, \langle W_i(x), R_j(x) \rangle,$

$\langle W_i(x), W_j(x) \rangle$ 是冲突的。

而 $\langle R_i(x), R_j(x) \rangle, \langle W_i(x), R_j(y) \rangle$ 且 $x \cap y = \emptyset$ 时不冲突。

S1冲突等价S2：如果不断交换S1中的非冲突操作对 $\langle O_i, O_j \rangle$ ，S1将变换为S2，则称**S1冲突等价S2**。

S: $R_{11}(z)R_{21}(x)W_{22}(y)W_{12}(x)$

等价于调度 T_2T_1 。

4) 可串行化调度

视图等价于某一串行调度的并发调度称为视图可串行化调度。

冲突等价于某一串行调度的并发调度称为冲突可串行化调度。

冲突可串行一定视图可串行，反之不真。

T3	T4	T6
Read(Q)	write(Q)	=T3;T4;T6
write(Q)		write(Q)

我们认为事务的串行执行方式是正确的，因而与它等价的并发执行方式（可串行化调度）也是正确的。

判断一个调度是否可串行化调度。

- 已知 S ,构造前趋图 $G=<V,E>$:
- $V=\{T_1,T_2,T_3,...T_N\}$,
- $E\subseteq V\times V$,
- $\langle T_i,T_j \rangle \in E$ iff ($\langle R_i(x),W_j(x) \rangle \in S_v$
 $\langle W_i(x),R_j(x) \rangle \in S_v$
 $\langle W_i(x),W_j(x) \rangle \in S$)

调度 S 是冲突可串行当且仅当 G 中无环

调度S5

T1

T2

read(A);
A:=A-50;

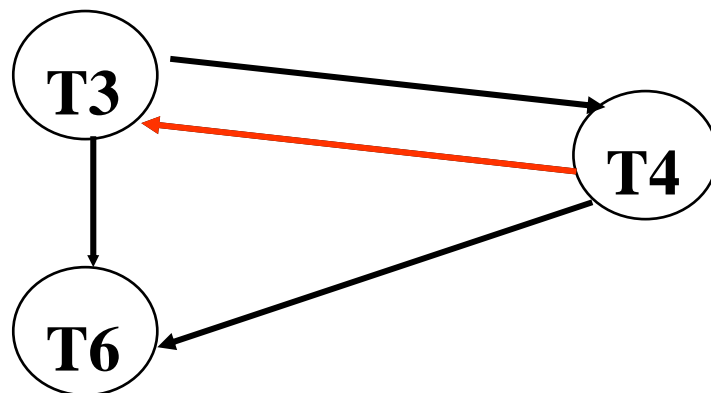
write(A);
read(B);
B:=B+50;
write(B).

read(A);
temp:=A*0.1;
A:=A-temp;
write(A);
read(B);

B:=B+temp;
write(B).

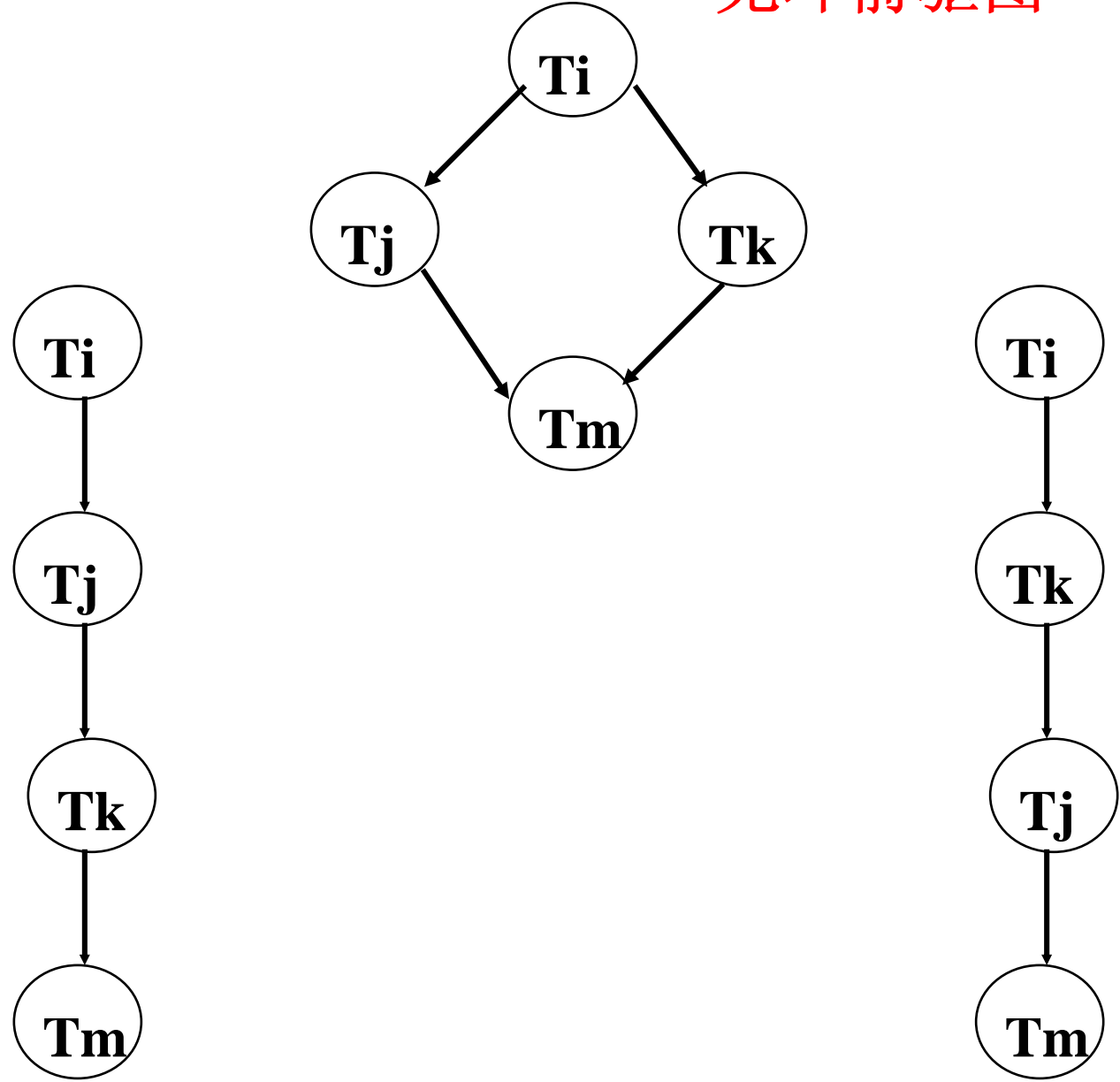
不一致

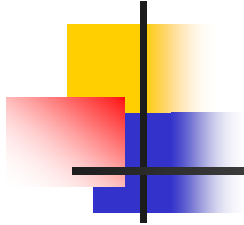
视图等价但不冲突等价的例：



T3	T4	T6
Read(Q)	write(Q)	=T3;T4;T6
write(Q)		write(Q)

无环前驱图





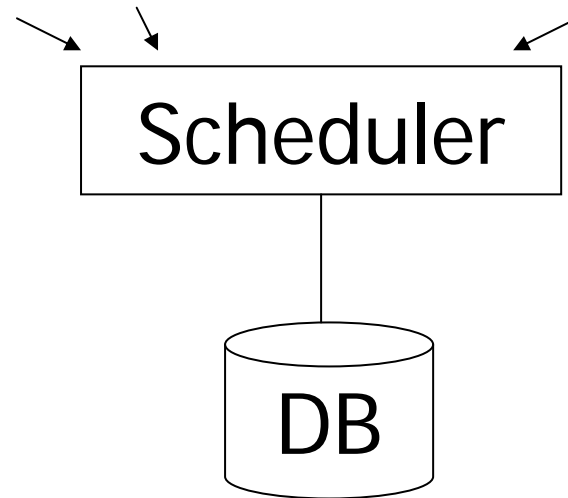
用户请求执行事务的时刻是随机的，系统如何在一种动态的环境下保证调度的可串行性？

并发控制机制的任务 是维持数据库的一致性，或者说维持事务调度的可串性的前提下，如何提高并行度。

3. 并发控制机制

阻止 P(S) 中出现环

T_1 T_2 T_n





3. 并发控制机制

- 1) 封锁机制
- 2) 基于时标的并发控制机制
- 3) 乐观的并发控制

3. 并发控制机制

1) 封锁机制

a. 二值锁（两态锁），

每一数据项具有锁值**LOCK(x)**：**0**（解锁）
或**1**（加锁）。

系统为事务访问数据**x**提供两个操作：

lock_item(x)和

unlock_item(x)。

1) 封锁机制

a. 二值锁（两态锁），

每个事务都必须遵循如下规则：

- ① 事务在进行读写数据项 x 之前，必须对其加锁 **lock_item(x)**。
- ② 事务在完成了对数据项 x 的所有读写操作之后，必须对其解锁 **unlock_item(x)**。
- ③ 事务在已经发出了一个 **lock_item(x)** 后，不再对 x 加锁。
- ④ 事务没有对数据项 x 加锁，就不能对 x 解锁 **unlock_item(x)**。

一个事务要访问被另一个事务加锁的数据项 x 时，必须等待它解锁，因而封锁机制就强制了事务操作的一种执行次序。

3. 并发控制机制

b. 共享锁、互斥锁(读锁、写锁)

共享锁：如果数据项被某事务用共享锁加锁，其它事务仍可用共享锁读它。

互斥锁：如果数据项被某事务用互斥锁加锁，其他事务欲操作这一数据项，必须等待该事务释放此互斥锁。

锁**LOCK(x)**的三种状态：读锁、写锁和解锁。

锁的操作：**read_lock(x)**、**write_lock(x)**和**unlock(x)**。

3. 并发控制机制

read_lock(x)

```
B: if LOCK(x) = 解锁  
  then LOCK(x) := 读锁;  
    no_of_reads(x) := 1;  
  else if LOCK(x) = 读锁  
    then no_of_reads(x) := no_of_reads(x) + 1;  
    else 等待直到LOCK(x) = 解锁被唤醒执行;  
  goto B;  
  endif;  
endif.
```

3. 并发控制机制

write_lock(x)

```
B:  if LOCK(x) = 解锁
    then LOCK(x) := 写锁;
    else 等待直到LOCK(x) = 解锁被唤醒执行;
        goto B;
    endif.
```

unlock(x)

3. 并发控制机制

```
if    LOCK(x) = 写锁
then  LOCK(x) := 解锁;
      唤醒等待解锁的事务执行;
else  if LOCK(x) = 读锁
      then no_of_reads(x) := no_of_reads(x) - 1;
          if      no_of_reads(x) = 0
          then    LOCK(X) := 解锁;
                  唤醒等待解锁的事务执行;
          endif;
      endif;
endif.
endif.
```

3. 并发控制机制

事务必须遵循的共享/互斥锁操作规则:

- ① 事务在对数据项 x 进行读操作之前, 必须发出 **read_lock(x)**或**write_lock(x)**请求。
- ② 事务在对数据项 x 进行写操作之前, 必须发出 **write_lock(x)**请求。
- ③ 事务在完成了对数据项 x 的所有读写操作之后, 必须发出**unlock(x)**请求。
- ④ 事务在对 x 加锁 (共享锁或互斥锁) 后, 不再对 x 进行任何形式的加锁。
- ⑤ 事务没有对 x 加锁, 就不能进行解锁**unlock(x)**。

T1:

Read_item(y);

Read_item(x);

$X := x + y;$

Write_item(x);

T2:

Read_item(x);

Read_item(y);

$Y := x + y;$

Write_item(y);

$x=20, y=30$

T1;T2:

$x=50, y=80$

$x=20, y=30$

T2;T1:

$x=70, y=50$

T1:

read_lock(y);

Read_item(y);

Unlock(y);

Write_lock(x);

Read_item(x);

$X := x + y$;

Write_item(x);

Unlock(x);

T2:

Read_lock(x);

Read_item(x);

Unlock(x);

Write_lock(y);

Read_item(y);

$Y := x + y$;

Write_item(y);

Unlock(y);

T1:

read_lock(y);
Read_item(y);
Unlock(y);

Write_lock(x);
Read_item(x);
X:=x+y;
Write_item(x);
Unlock(x);

T2:

Read_lock(x);
Read_item(x);
Unlock(x);
Write_lock(y);
Read_item(y);
Y:=x+y;
Write_item(y);
Unlock(y);

x=20,y=30

x=50,y=50

3. 并发控制机制

c. 两段封锁协议

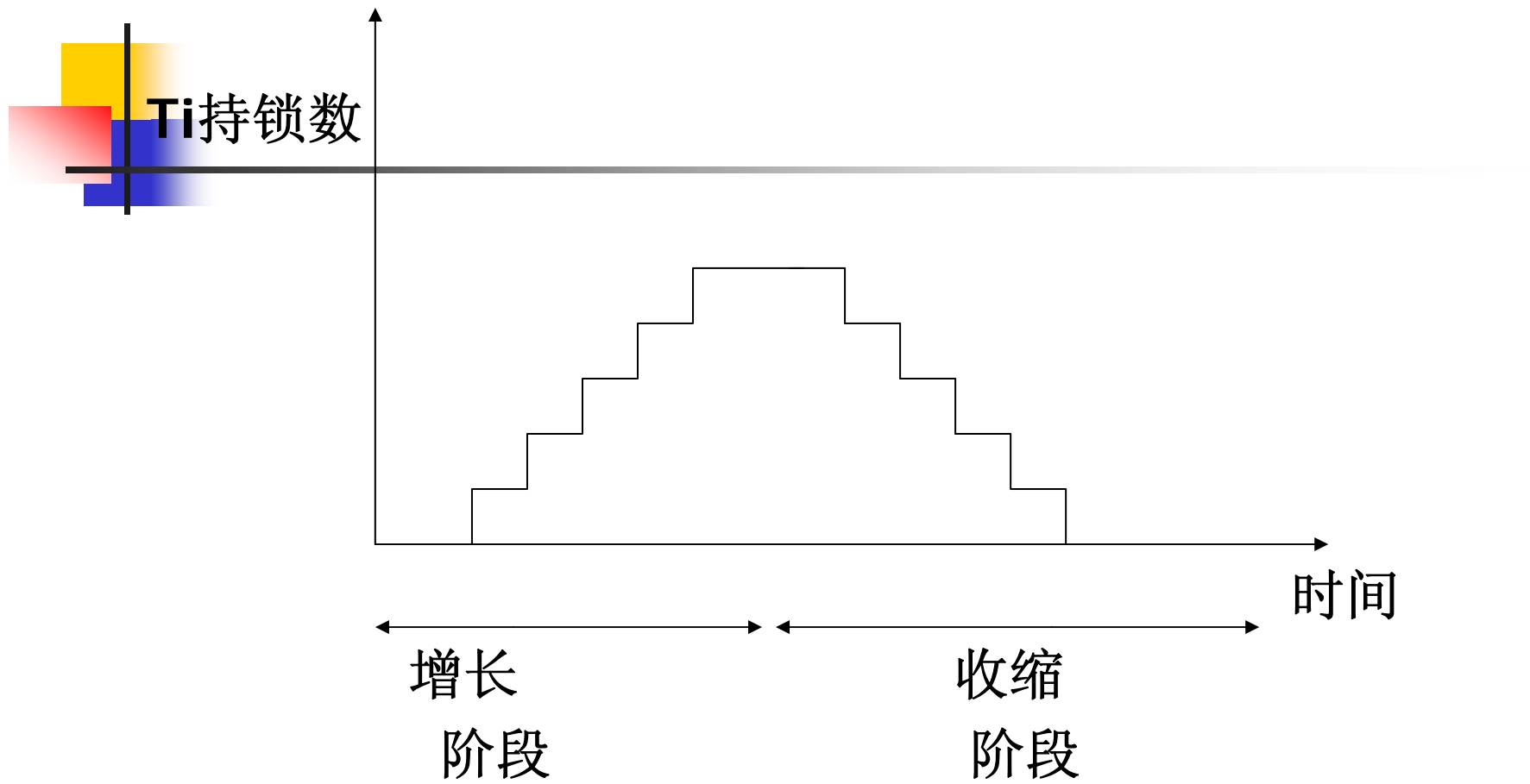
段锁协议规定在一个事务中所有的加锁操作必须出现在第一个解锁操作之前。即将一个事务分为两个阶段：

扩展阶段：可以新加锁，但是不能释放锁；

收缩阶段：可以释放锁，但是不能加新锁。

$T_i = \dots\dots l_i(A) \dots\dots u_i(A) \dots\dots$







定理

2PL \Rightarrow 冲突可串行调度

为易于证明:

定义 **Shrink(Ti) = SH(Ti) =**
Ti 的第一个放锁动作

引理

$$Ti \rightarrow Tj \text{ in } S \Rightarrow SH(Ti) <_S SH(Tj)$$

证明:

$Ti \rightarrow Tj$ 意味着

$S = \dots p_i(A) \dots q_j(A) \dots$; p, q 冲突

$S = \dots p_i(A) \dots u_i(A) \dots l_j(A) \dots q_j(A) \dots$



So, $SH(Ti) <_S SH(Tj)$

Theorem

2PL \Rightarrow conflict
serializable
schedule

证明:

(1) 假设 $P(S)$ 有环

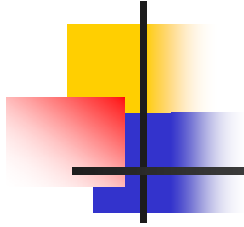
$$T_1 \rightarrow T_2 \rightarrow \dots T_n \rightarrow T_1$$

(2) 由引理: $SH(T_1) < SH(T_2) < \dots < SH(T_1)$

(3) 矛盾, 故 $P(S)$ 无环

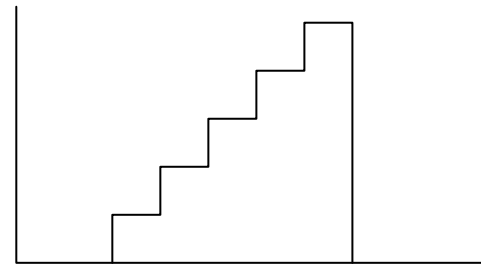
(4) $\Rightarrow S$ 是冲突可串行

放锁的顺序即串行调度的顺序



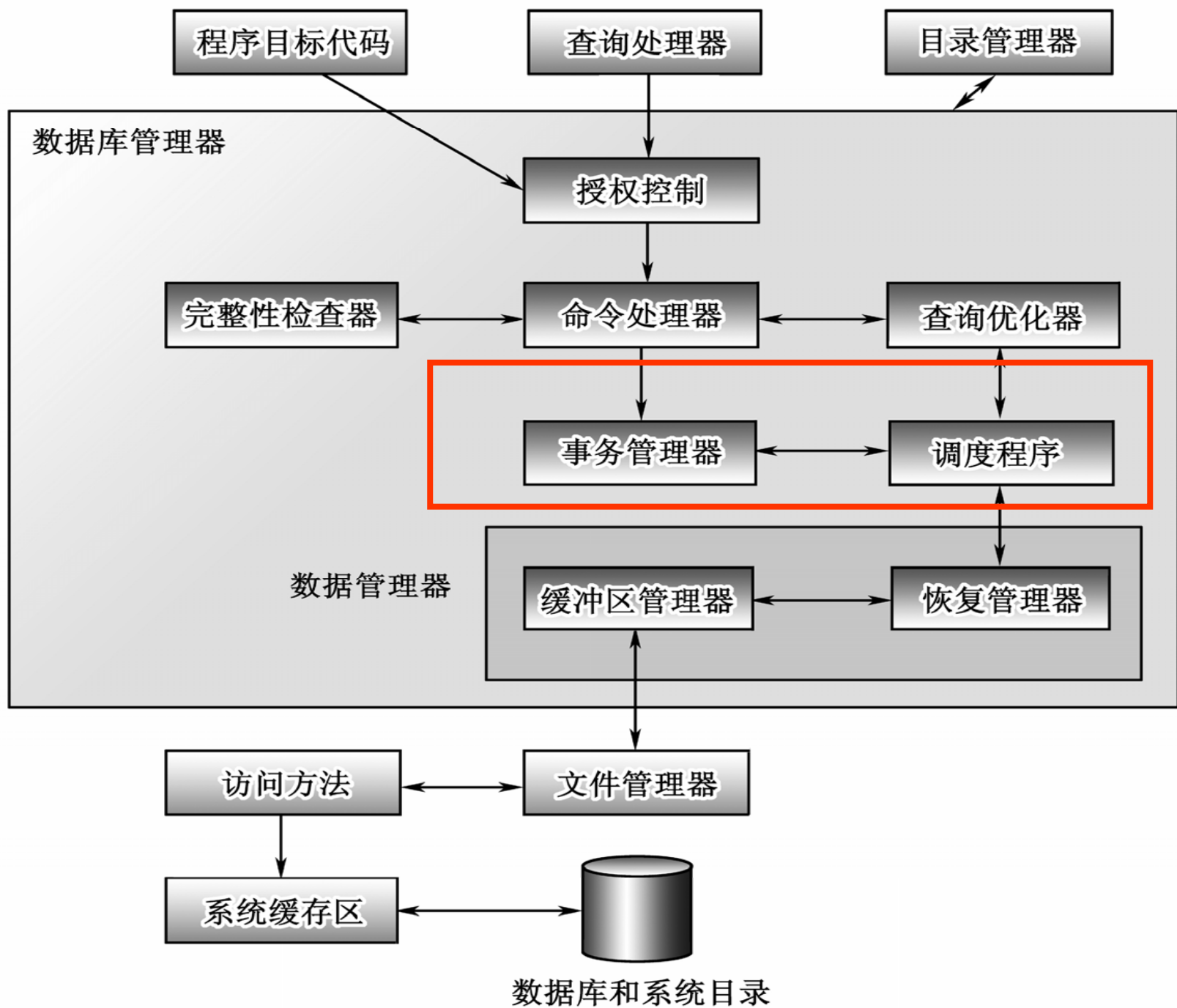
持所有所至事务交付

locks



time

严格两段锁，隔离性好，可防污读。



Ti

↓ **Read(A), Write(B), commit**

Scheduler, part I

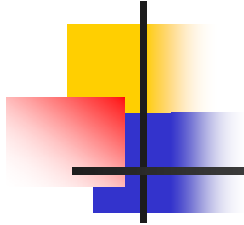
**lock
table**

I(A), Read(A), I(B), Write(B)...

Scheduler, part II

↓ **Read(A), Write(B)**

DB



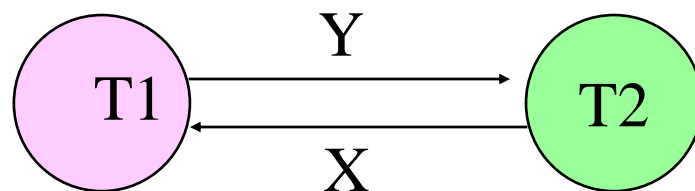
d.死锁和饿死

3. 并发控制机制

- 死锁：几个事务相互等待其他事务解锁而均不能正常结束的状态。

T1: $w1(x), w1(y)$; T2: $w2(x), w2(y)$

S: $w1(x), w2(y), w2(x), w1(y)$



等待图

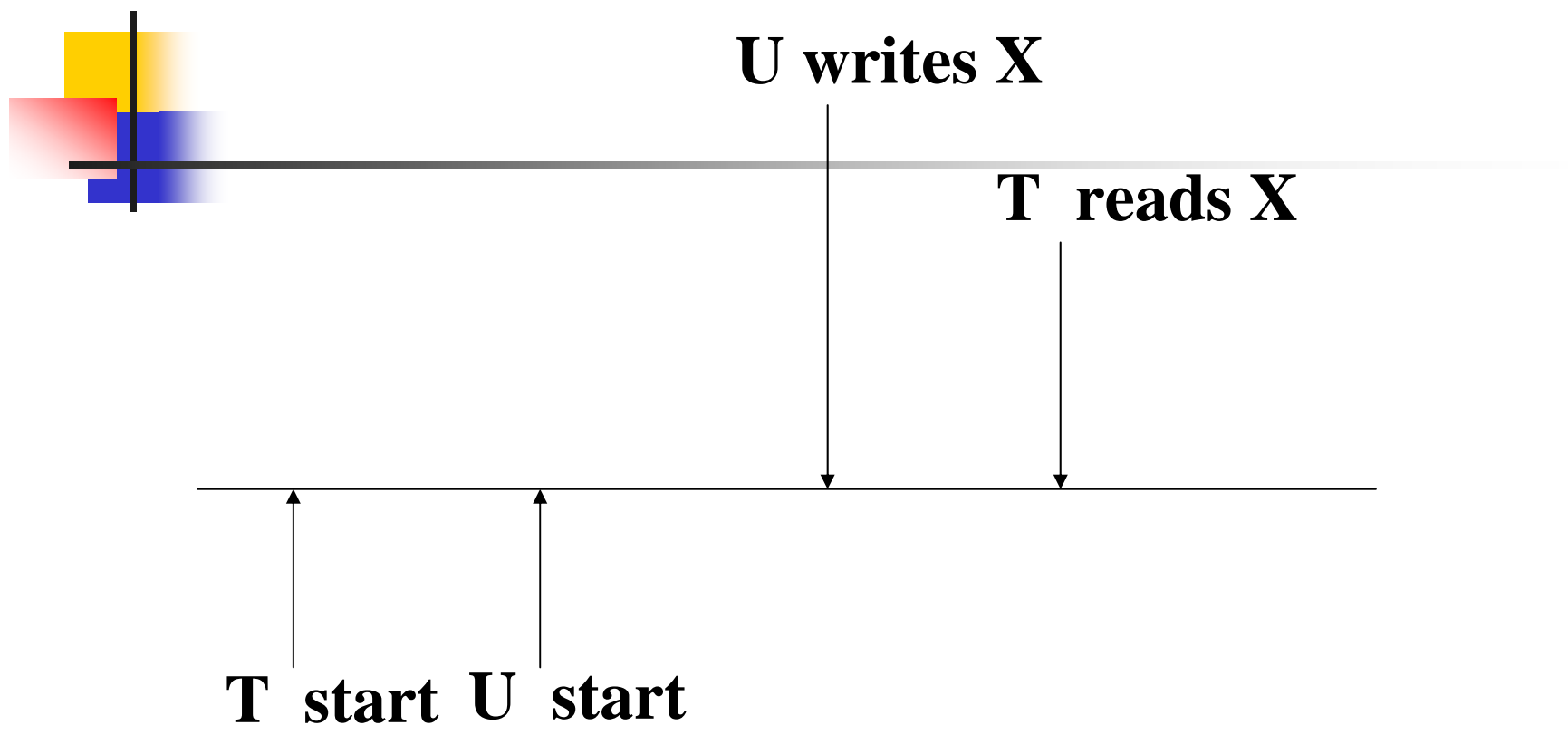
- 饿死：一个事务长时间等待其他事务解锁而不能正常执行的状态。

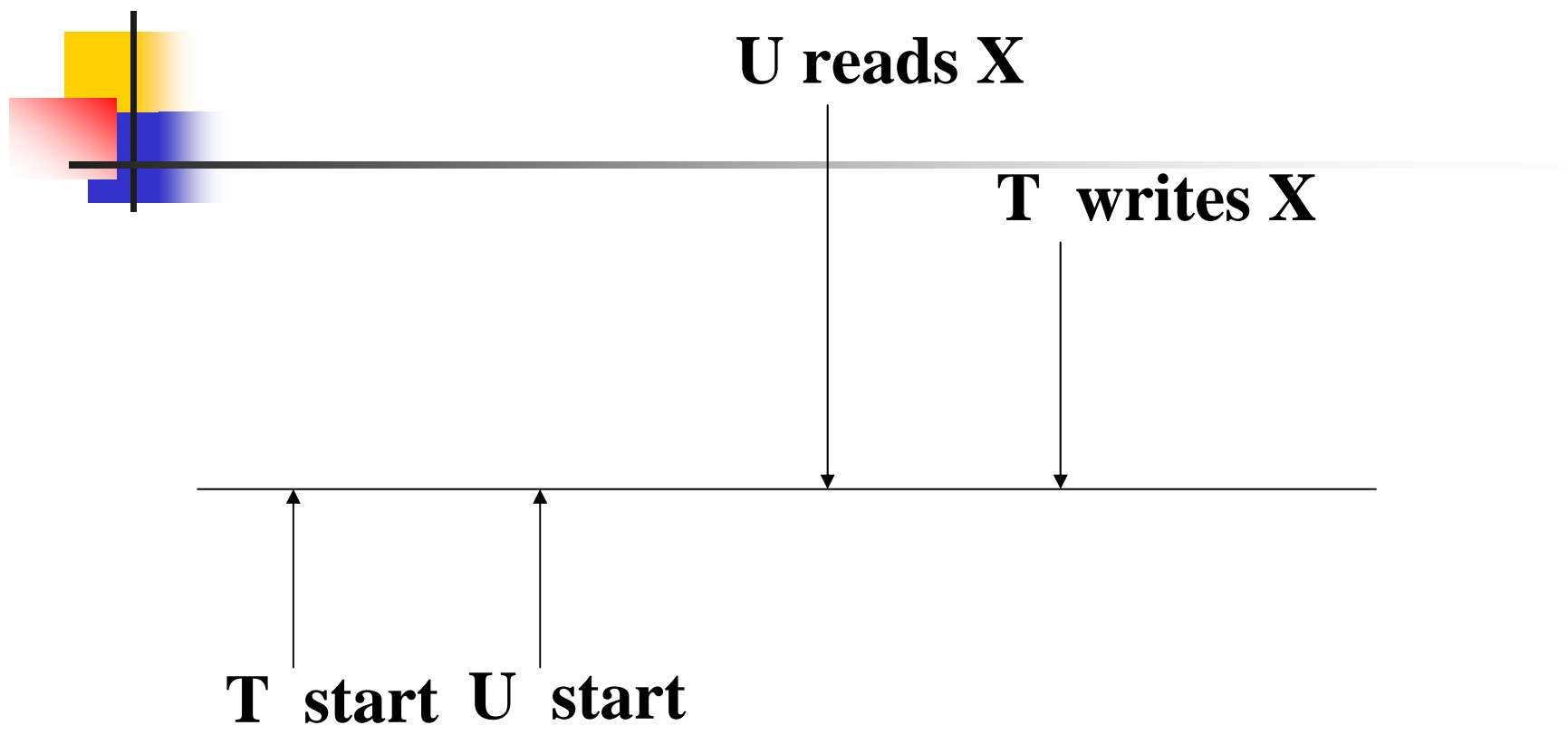
3. 并发控制机制

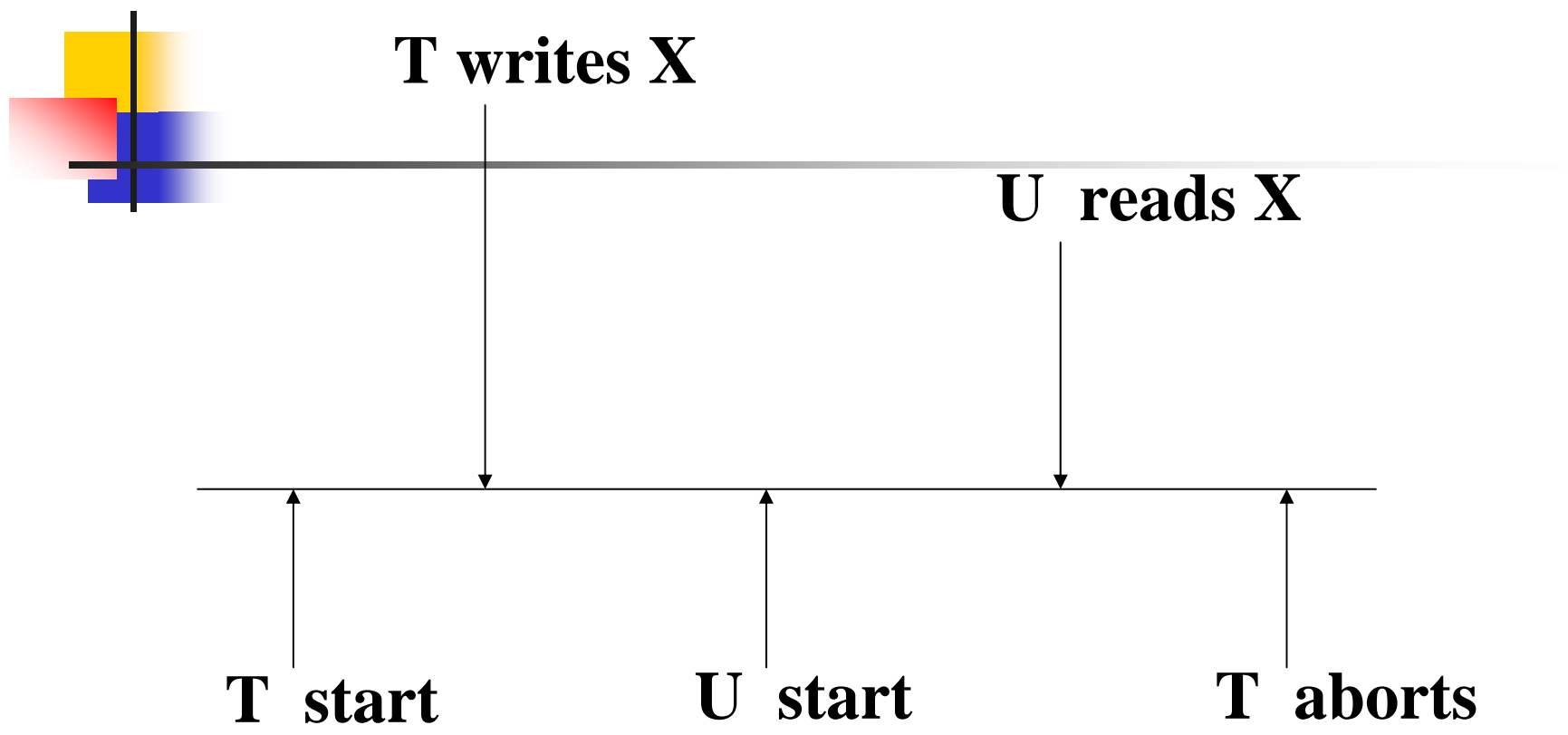


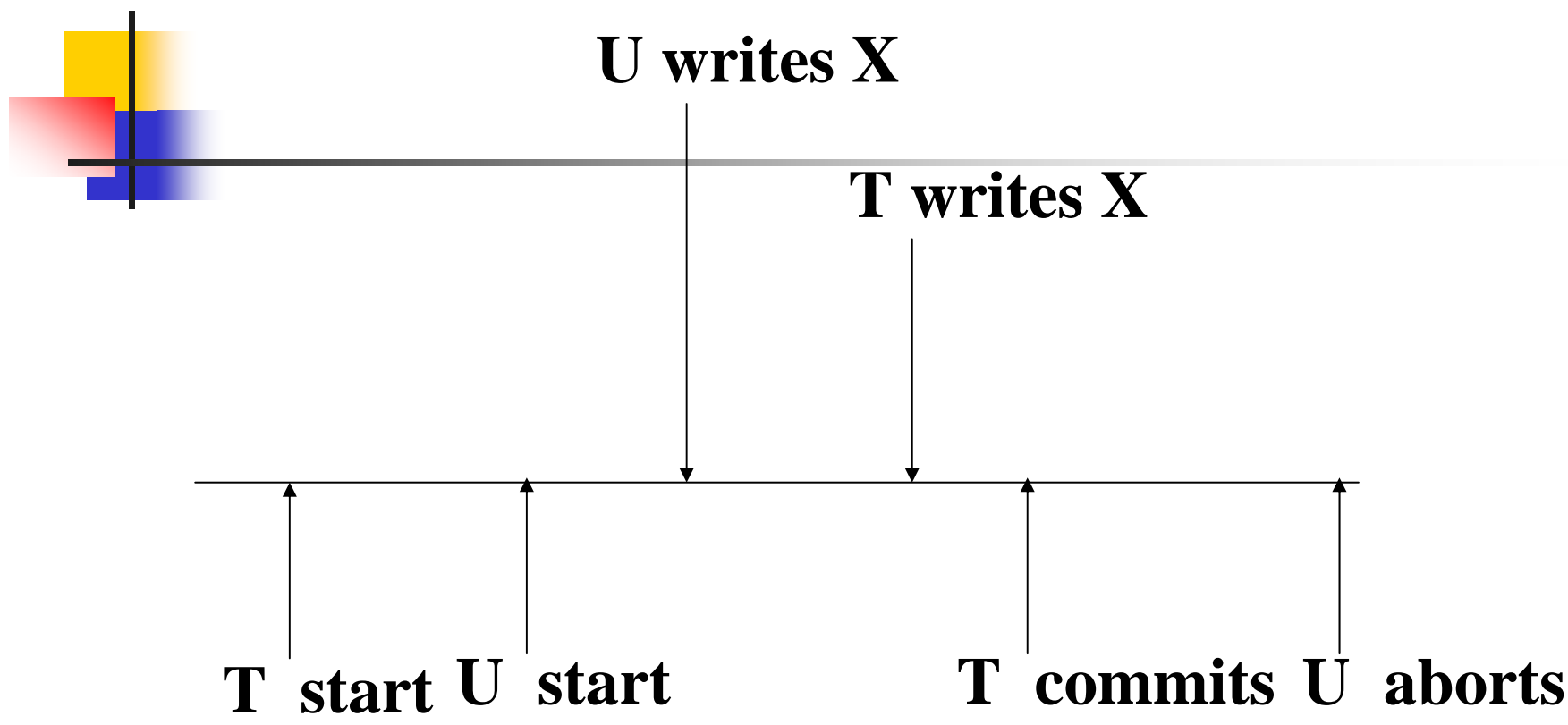
2) 基于时标的并发控制机制

基本思想:基于时标大小解决操作冲突, 保证按时间顺序执行冲突操作。









2) 基于时标的并发控制机制

3. 并发控制机制

基本思想:基于时标大小解决操作冲突, 保证按时间顺序执行冲突操作。

- a. 每个事务**T**开始执行时, 系统给它打上时标**t** (一般为当前时间)。事务越老, 时标越小。
- b. 系统中维持每个数据项**x**的最近读时标**RTM(x)**
和最近写时标**WTM(x)**。
- c. 当某事务**T**执行到某操作**O**时, 比较事务**T**的时标**t**与操作对象**x**的最近读写时标, 并按如下规则运作:

3. 并发控制机制

2) 基于时标的并发控制机制

① 当O是读操作时,

如果 $t < \text{WTM}(\mathbf{x})$, 不能读, 拒绝O,

并将T的时标赋一新值, 重新启动T。

如果 $t > \text{WTM}(\mathbf{x})$, 执行O, 修改x的读时标, 使得:
 $\text{RTM}(\mathbf{x}) = \max\{t, \text{RTM}(\mathbf{x})\}$

② 当O是写操作时,

如果 $t < \text{RTM}(\mathbf{x})$ 或 $t < \text{WTM}(\mathbf{x})$, 不能写, 拒绝O, 并将T的时标赋一新值, 重新启动T。

否则, 执行O, 修改x的写时标, 使得 $\text{WTM}(\mathbf{x}) = t$

3. 并发控制机制

例：

设有两个事务T1: R(A)W(A)

T2: R(A)W(A),

它们具有时标 $t_1 < t_2$ 。

RTM (A)	WTM (A)	操作序列
0	0	
t_1	0	T1: R(A)
t_2	0	T2: R(A)
t_2	0	T1: W(A), T1回退
t_2	t_2	T2: W(A)
t_3	t_2	T1: R(A)
t_3	t_3	T1: W(A)

时标的顺序即串行调度的顺序

3. 并发控制机制

2) 基于时标的并发控制机制

- ① 当O是读操作时，
如果 $t < \text{WTM}(x)$, 不能读，拒绝O，
并将T的时标赋一新值，重新启动T。
如果 $t > \text{WTM}(x)$, 执行O, 修改x的读时标，使得：
 $\text{RTM}(x) = \max\{t, \text{RTM}(x)\}$
- ② 当O是写操作时，
如果 $t < \text{RTM}(x)$ 或 $t < \text{WTM}(x)$, 不能写，拒绝O，
并将T的时标赋一新值，重新启动T。
否则，执行O, 修改x的写时标，使得 $\text{WTM}(x) = t$

4) 多版本 (Multiversion) 并发控制技术

系统对数据项 x 维持 k 个版本 x_1, x_2, \dots, x_k , 对每个版本保存

read_TS(x_i): x_i 的最近读时标

write_TS(x_i): x_i 的最近写时标

使用两条规则控制事务 T (时标为 t) 对数据项 x 的读写:

(1) 写: 设 x 的所有版本中具有最大写时标的版本为 x_i ,

if write_TS(x_i) \leq t < read_TS(x_i)

then 终止并回退 T

else 创建一个新的版本 x_j ,并置其写时标。

(2) 读: 设 x 的所有版本中写时标小于等于 t 的最后版本为 x_i , 返回该版本的值,并修改其读时标。 (读总可行)

3) 乐观的并发控制

3. 并发控制机制

不同于前面，在乐观并发控制中，对事务的执行过程不作任何检查，对数据库的修改也不立即进行，而是到事务结束时进行有效性检查。当事务的执行不破坏可串性时，交付事务；否则将事务撤消，并回退，重新启动。

此种方法在事务间冲突操作较少时，效率较高，当冲突操作很多时，引起大量重启，降低效率。

It is useful in some cases:

- Conflicts rare**
- System resources plentiful**
- Have real time constraints**

Transactions have 3 phases:

(1) Read

- all DB values read
- writes to temporary storage
- no locking


(2) Validate

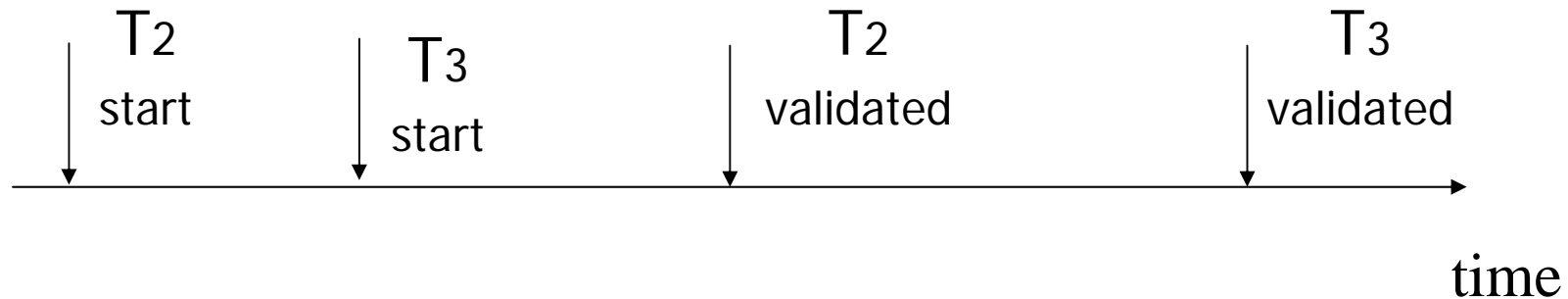
- check if schedule so far is serializable

(3) Write

- if validate ok, write to DB

Example of what validation must prevent:

$$\begin{array}{ll} RS(T_2) = \{B\} & RS(T_3) = \{A, B\} \neq \phi \\ WS(T_2) = \{B, D\} & WS(T_3) = \{C\} \end{array}$$




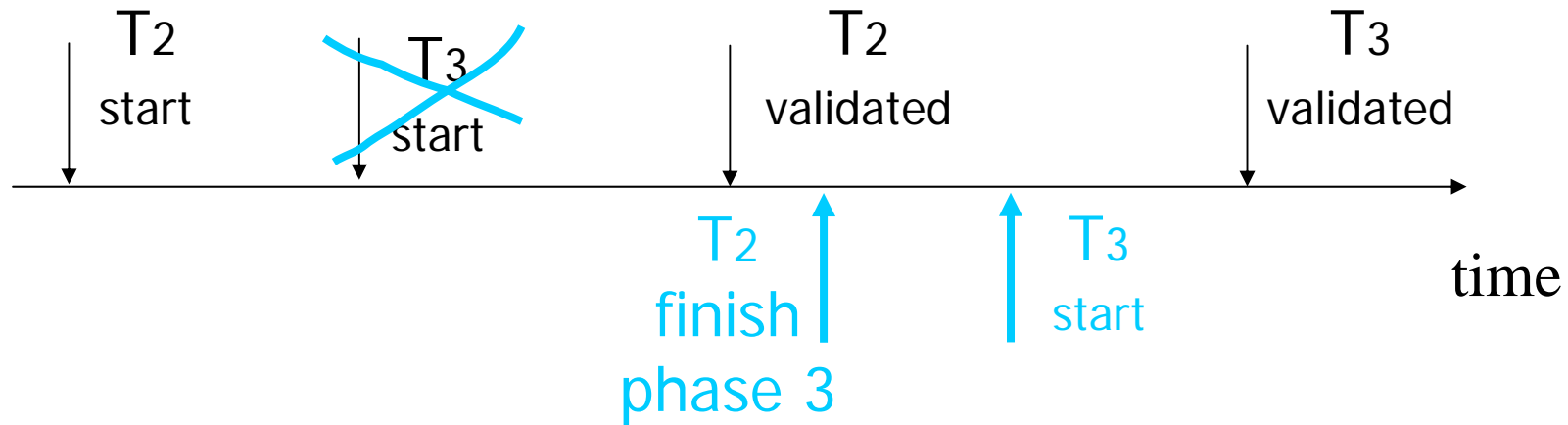
Example of what validation must ^{allow} prevent:

$RS(T_2) = \{B\}$

$WS(T_2) = \{B, D\}$

$RS(T_3) = \{A, B\} \neq \phi$

$WS(T_3) = \{C\}$



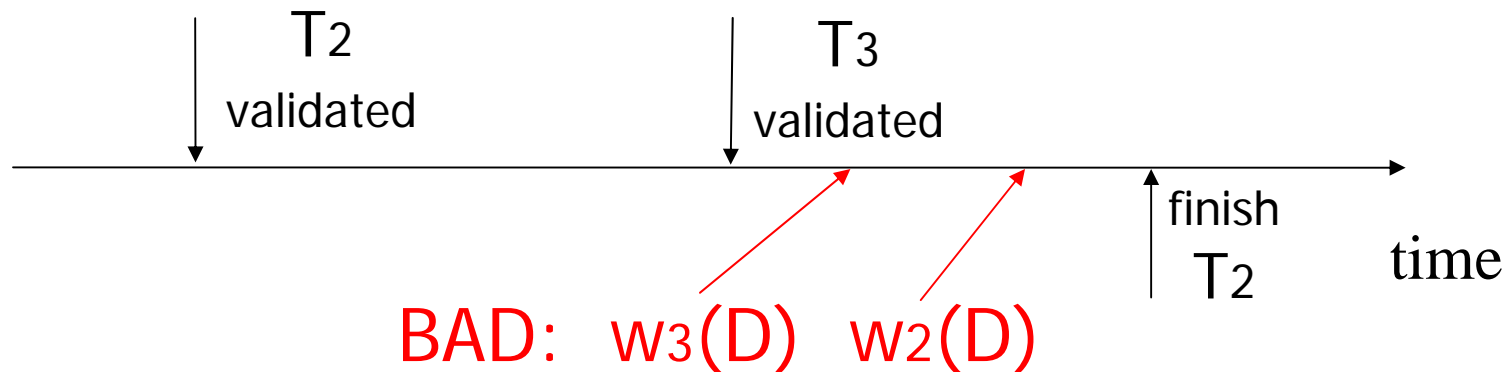
Another thing validation must prevent:

$$RS(T_2) = \{A\}$$

$$RS(T_3) = \{A, B\}$$

$$WS(T_2) = \{D, E\}$$

$$WS(T_3) = \{C, D\}$$



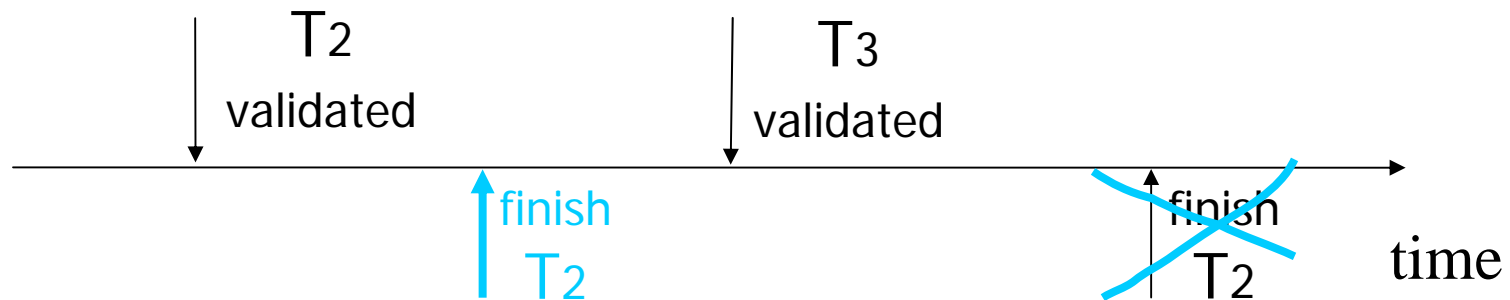
Another thing validation must ~~prevent~~ allow:

$RS(T_2) = \{A\}$

$RS(T_3) = \{A, B\}$

$WS(T_2) = \{D, E\}$

$WS(T_3) = \{C, D\}$



To implement validation, system keeps two sets:

- FIN = transactions that have finished phase 3 (and are all done)
- VAL = transactions that have successfully finished phase 2 (validation)

Validation rules for T_j :

(1) When T_j starts phase 1:

$\text{ignore}(T_j) \leftarrow \text{FIN}$ {此时的FIN无需再考虑}

(2) at T_j Validation:

if check (T_j) then

[$\text{VAL} \leftarrow \text{VAL} \cup \{T_j\}$;

do write phase;

$\text{FIN} \leftarrow \text{FIN} \cup \{T_j\}$]

Check (T_j):

检查跟T_j的读并发了的写，这一刻它可能完了也可能没完

For T_i ∈ VAL - IGNORE (T_j) DO

IF [WS(T_i) ∩ RS(T_j) ≠ ∅ OR

(T_i ∉ FIN AND WS(T_i) ∩ WS(T_j) ≠ ∅)]

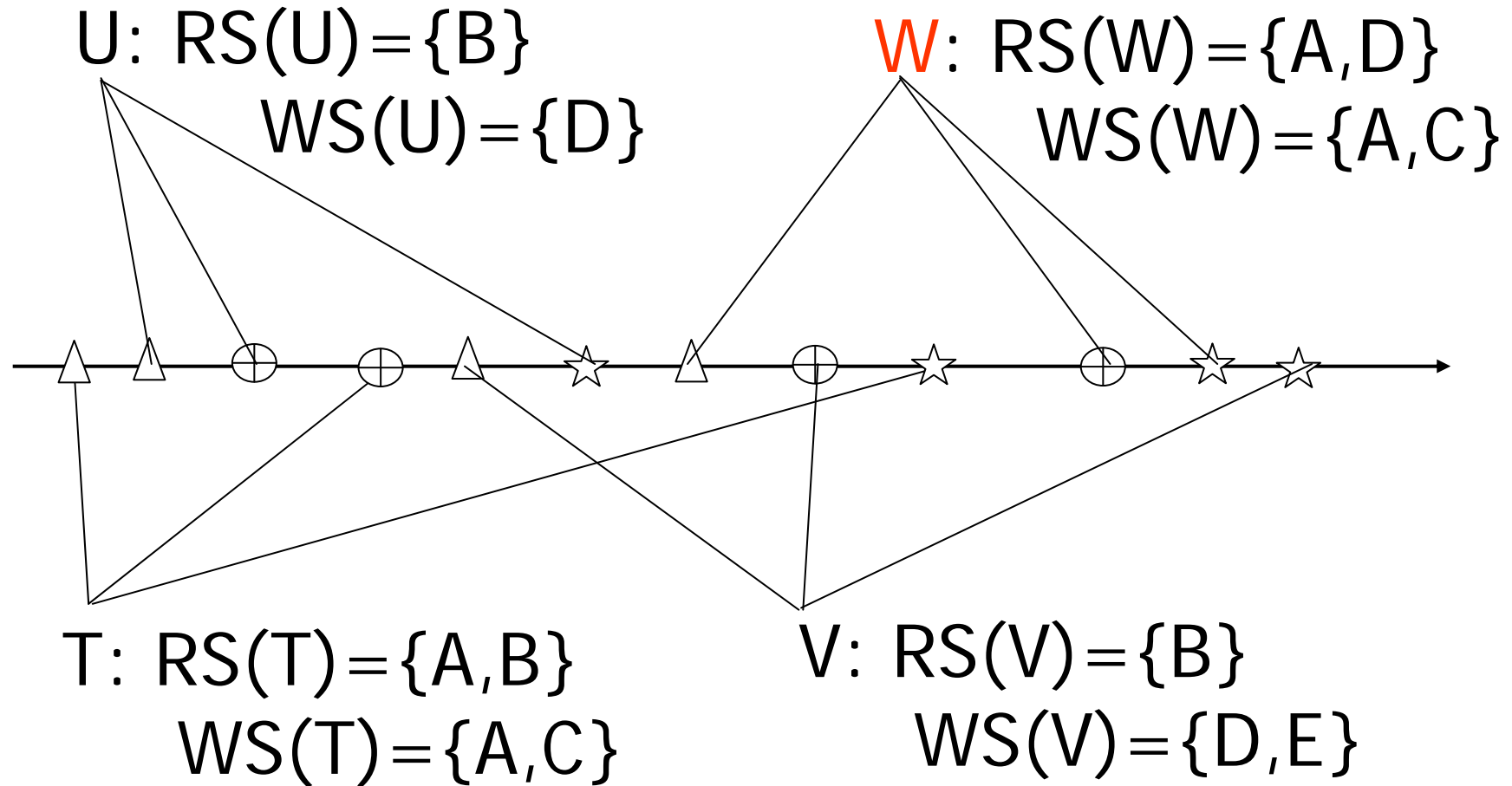
THEN RETURN false;

RETURN true;

检查将跟T_j的写并发的写，只有这一刻尚未完的事务才可能

Exercise:

△ start
⊕ validate
☆ finish





Key idea

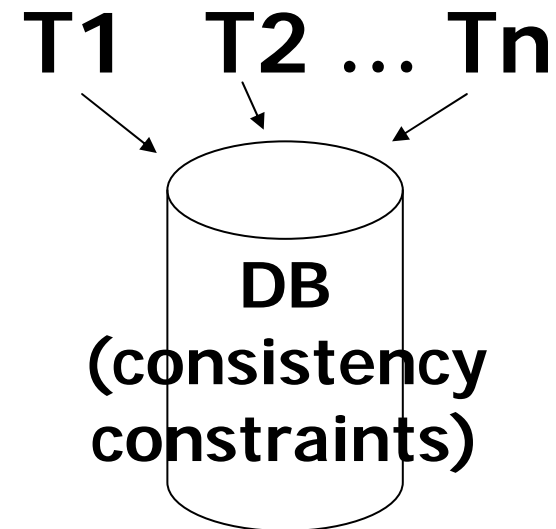
- Make validation atomic
- If T_1, T_2, T_3, \dots is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1 T_2 T_3 \dots$

8.2 并发控制

调度、串行调度、并行调度

可串行调度

- 视图（计算）可串行
- 冲突可串行（冲突操作对）
- 调度**S**是冲突可串行当且仅当前驱图**G**中无环





已知**S**,构造前趋图**G**=**<V,E>**:

1) $V = \{T_1, T_2, T_3, \dots, T_N\}$,

2) $E \subseteq V \times V$,

$\langle T_i, T_j \rangle \in E$ iff ($\langle R_i(x), W_j(x) \rangle \in S_V$

$\langle W_i(x), R_j(x) \rangle \in S_V$

$\langle W_i(x), W_j(x) \rangle \in S$)

调度**S**是冲突可串行当且仅当**G**中无环

引理: S_1, S_2 冲突等价 $\Rightarrow P(S_1) = P(S_2)$


证明:

假设 $P(S_1) \neq P(S_2)$

$\Rightarrow \exists T_i: T_i \rightarrow T_j$ 在 S_1 且不在 S_2

$\Rightarrow S_1 = \dots p_i(A) \dots q_j(A) \dots$	$\left\{ \begin{array}{l} p_i, q_j \\ \text{冲突} \end{array} \right.$
$S_2 = \dots q_j(A) \dots p_i(A) \dots$	

$\Rightarrow S_1, S_2$ 不冲突等价



注意: $P(S_1) = P(S_2) \Rightarrow S_1, S_2$ 冲突等价

反例:

$$S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)$$

$$S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)$$

定理

S_1 冲突可串行 $\iff P(S_1)$ 无环

(\Rightarrow) 假设 S_1 is 冲突可串行

$\Rightarrow \exists S_s: S_s, S_1$ 冲突等价

$\Rightarrow P(S_s) = P(S_1)$

$\Rightarrow P(S_1)$ 无环因为 $P(S_s)$ 无环

Theorem

S_1 冲突可串行 $\Leftrightarrow P(S_1)$ 无环

(\Leftarrow) 假设 $P(S_1)$ 无环

如下转换 S_1 :

(1) 取入度为零的事务为 T_1

(2) 将 T_1 的所有动作前移

$S_1 = \dots \underbrace{q_j(A) \dots p_1(A)} \dots$

(3) 现有 $S_1 = \langle T_1 \text{ 的动作} \rangle \langle \dots \text{ 剩余 } \dots \rangle$

(4) 重复上述动作直至全部串行

