数据库系统原理

教程：数据库系统概论（第5版）

结合：CMU 15-445/645 INTRO TO DATABASE SYSTEMS

华中科技大学 计算机学院

左琼

第十一章　并发控制

Principles of Database Systems

第11章 并发控制概述

11.1 并发控制概述

11.2 封锁

11.3 活锁和死锁

11.4 并发调度的可串行性

11.5 两段锁协议

11.6 封锁的粒度

11.7 小结

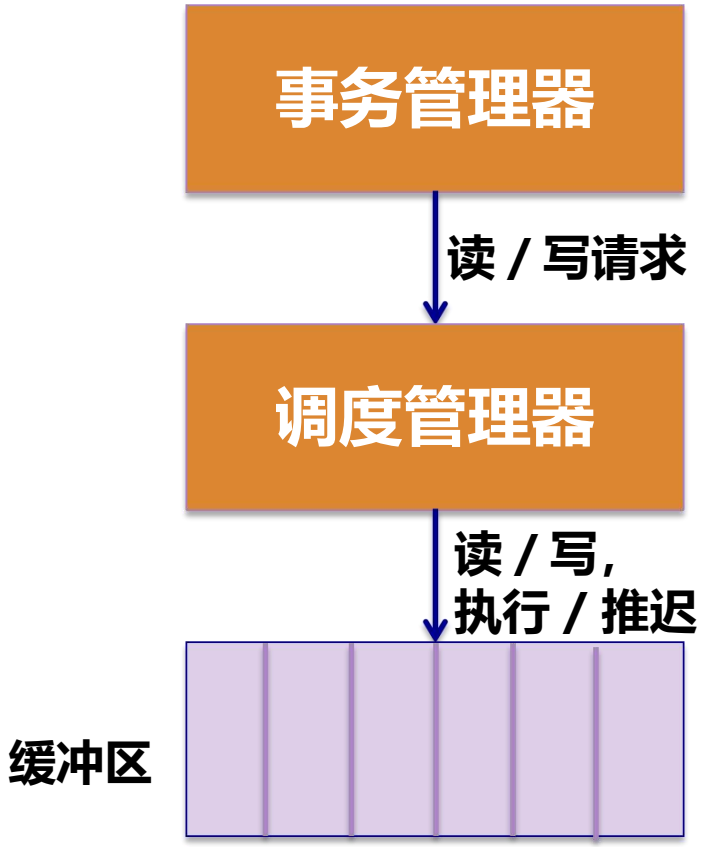
11.1 并发控制概述

- 并发控制是为保证多用户并发操作数据库中信息时的正确性、一致性所采取的措施。
- 11.1.1 事务调度

- 事务的执行顺序(按时间排序的一个序列）称为一个调度，表示事务的指令在系统中执行的时间顺序。
- 一组事务的调度必须保证：

  1）包含了所有事务的操作指令

  2）一个事务中指令的顺序必须保持不变



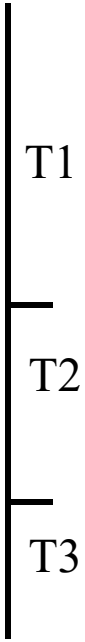
11.1 并发控制概述

调度方式：

(1) 串行调度

- 属于同一事务的指令紧挨在一起；
- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行；
- 对于有n个事务的事务组，可以有n!个有效调度。
- 缺点：

  不能充分利用系统资源，发挥数据库共享资源的特点。

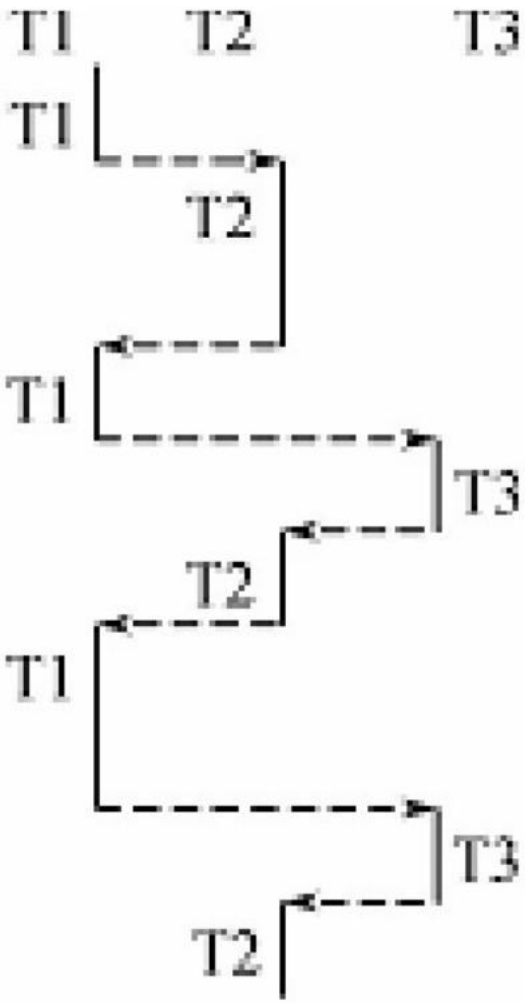


事务的串行执行方式

11.1 并发控制概述

(2) 交叉并发方式

- 在单处理机系统中，事务的并行执行是这些并行事务的并行操作轮流交叉运行；
- 单处理机系统中的并行事务并没有真正地并行运行，但能够减少处理机的空闲时间，提高系统的效率。



11.1 并发控制概述

(3) 同时并发方式（simultaneous concurrency）

多处理机系统中，每个处理机可以运行一个事务，多个处理机可以同时运行多个事务，实现多个事务真正的并行运行。

- 并发执行的优点：
  - 一个事务由不同的步骤组成，所涉及的系统资源也不同。各步骤并发执行，提高系统的吞吐量；
  - 如果各个事务涉及数据库的不同部分，采用并发会减少平均响应时间。
- 事务并发执行带来的问题：
  - 会产生多个事务同时存取同一数据的情况；
  - 可能会存取和存储不正确的数据，破坏事务一致性和数据库的一致性。
- 核心问题：在保证一致性的前提下最大限度地提高并发度。

事务执行示例

从A过户
50￥到B

开始状态：
A=1000￥
B=2000￥
A+B=3000￥

从A过户存款
的10%到B

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);
```

事务执行示例（续）

串行调度1

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);

开始状态：
A=1000￥
B=2000￥
A+B=3000￥

A=950￥
B=2050￥
A+B=3000￥

结束状态：
A=855￥
B=2145￥
A+B=3000￥

事务执行示例（续）

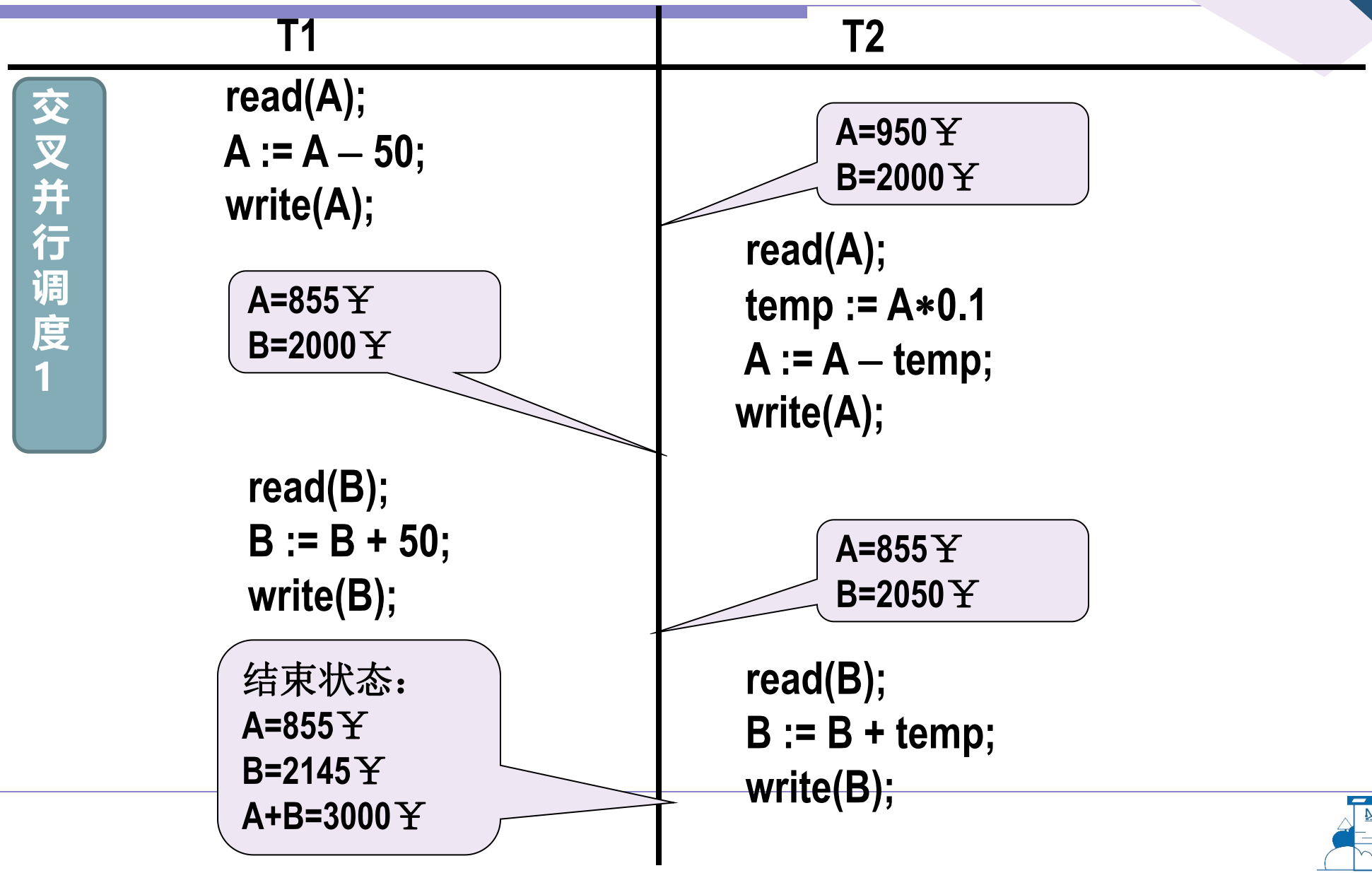
串行调度2

T1	T2
	read(A);
	temp := A*0.1
	A := A – temp;
	write(A);
read(A);	
A := A – 50;	
write(A);	
read(B);	
B := B + 50;	
write(B);	
	read(B);
	B := B + temp;
	write(B);

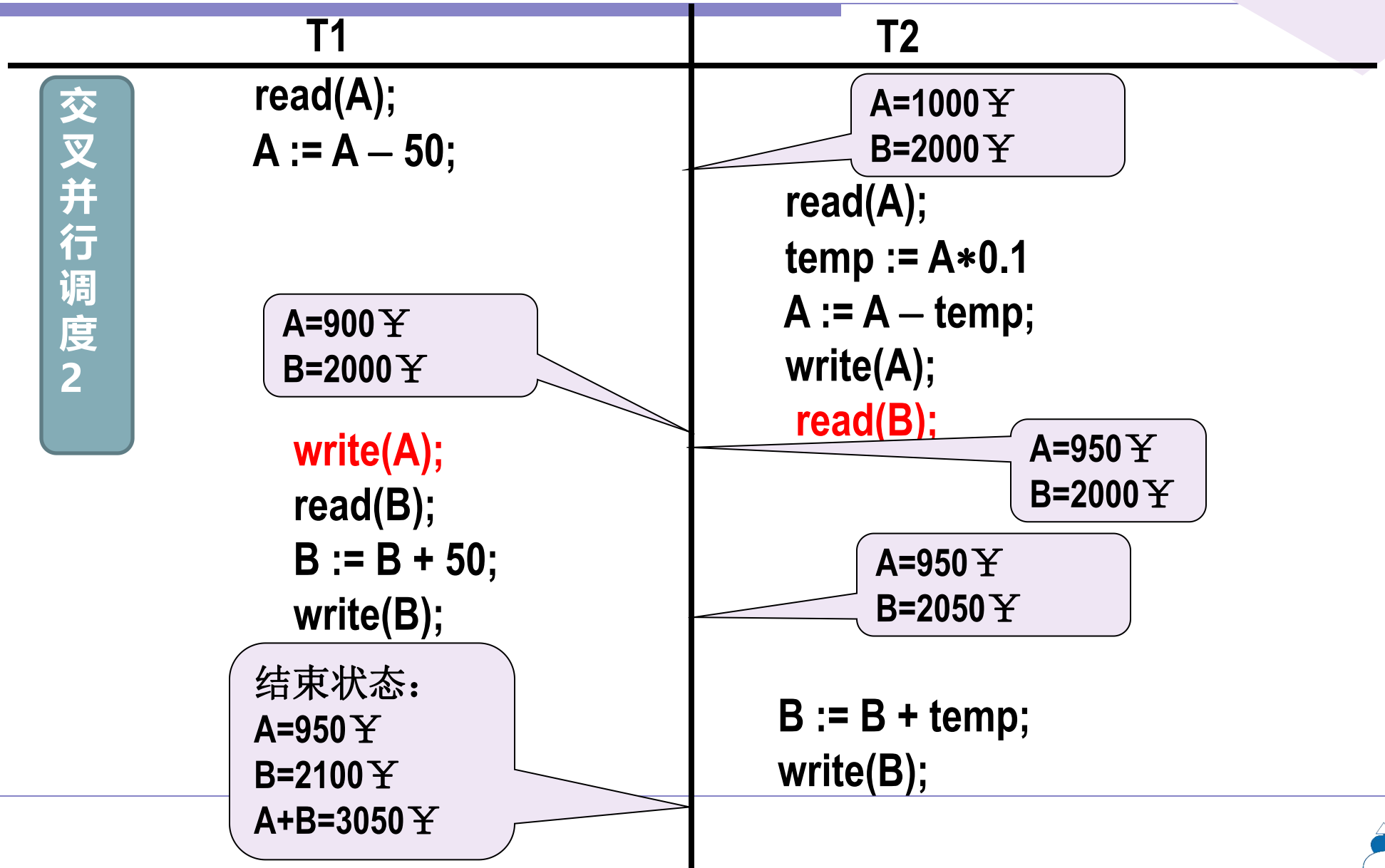
A=900￥
B=2100￥

结束状态：
A=850￥
B=2150￥
A+B=3000￥

事务执行示例（续）


T1
T2
交叉并行调度1
read(A);
A := A – 50;
write(A);
A=950￥
B=2000￥
read(A);
temp := A*0.1
A := A – temp;
write(A);
A=855￥
B=2000￥
read(B);
B := B + 50;
write(B);
A=855￥
B=2050￥
read(B);
B := B + temp;
write(B);
结束状态：
A=855￥
B=2145￥
A+B=3000￥

事务执行示例（续）


T1
T2
交叉并行调度2
read(A);
A := A – 50;
A=1000￥
B=2000￥
read(A);
temp := A*0.1
A := A – temp;
write(A);
read(B);
A=900￥
B=2000￥
A=950￥
B=2000￥
write(A);
read(B);
B := B + 50;
write(B);
A=950￥
B=2050￥
结束状态：
A=950￥
B=2100￥
A+B=3050￥
B := B + temp;
write(B);

并行调度的问题

并发操作带来的数据不一致性：

1) 丢失更新（lost update）

两个以上事务从DB中读入同一数据并修改，其中一事务（后提交的事务）的提交结果破坏了另一事务（先提交的事务）的提交结果，导致先前提交事务对DB的修改被丢失。这种现象使数据库操作出现严重错误。

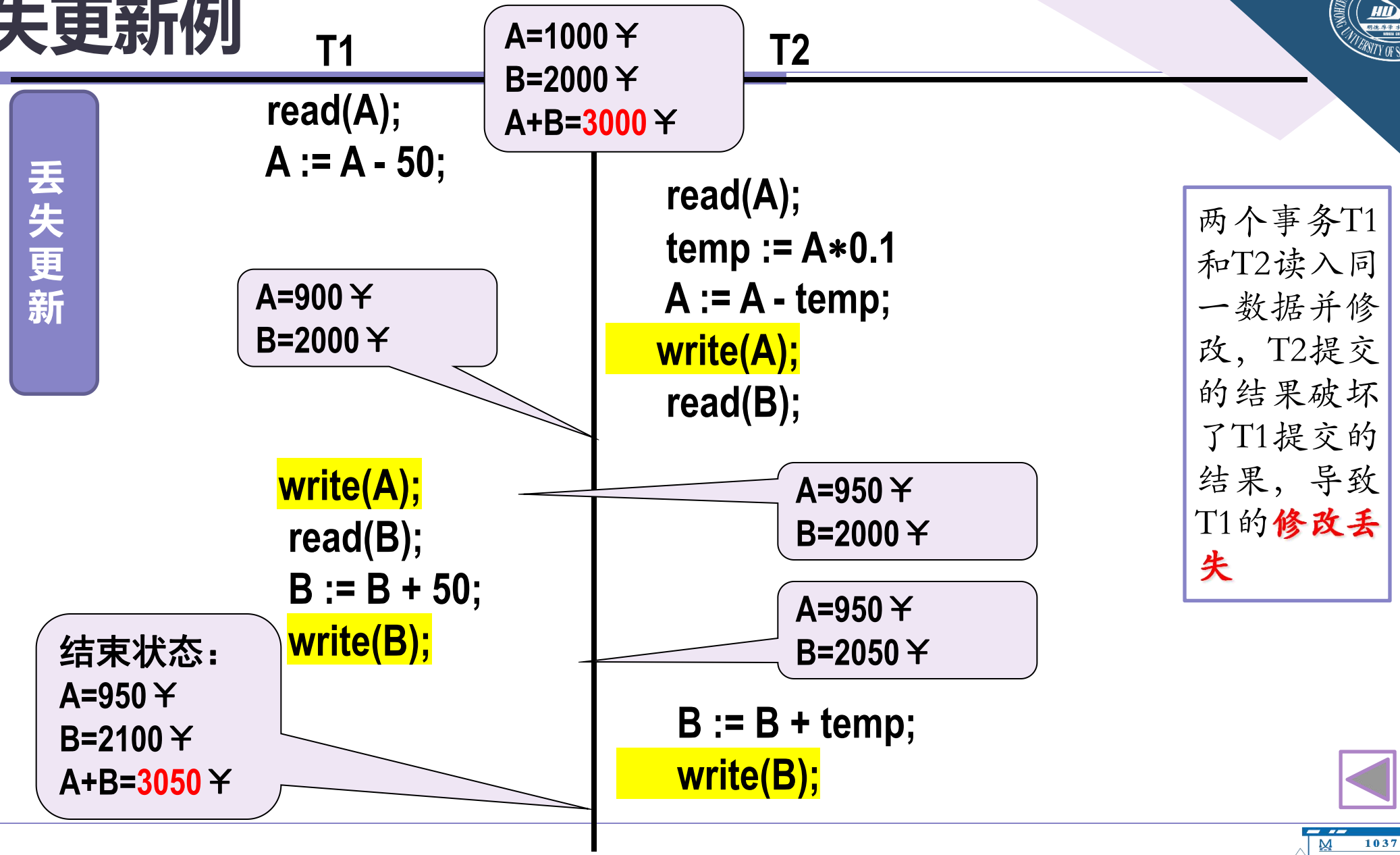
2) 不可重复读（read non-repeatable）

同一事务重复读同一数据，但获得结果不同。即从数据库中得到了不一致的数据。

3) 读“脏”数据 （read dirty）

读未提交随后又被撤消(Rollback)的数据。即从数据库中得到了临时性数据。

丢失更新例



两个事务T1和T2读入同一数据并修改，T2提交的结果破坏了T1提交的结果，导致T1的**修改丢失**

不可重复读例

不可重复读

时间	TA	数据库中内容	TB
1	统计全班平均成绩优秀的学生=5人	全班30人：5人优秀，20人良好，5人合格。	
2	统计全班平均成绩良好的学生=20人；	全班30人：5人优秀，20人良好，5人合格。	
3		全班30人：5人优秀，21人良好，4人合格。	某学生某门课的成绩从0分变为86分，从而平均成绩从合格变为良好。
4	统计全班平均成绩合格的学生=4人； 计算参与统计总人数=29人。	全班30人：5人优秀，21人良好，4人合格。	

应用系统是否接受此类数据不一致？

一事务读取后，另一事条对之进行了修改

不可重复读例

- 一事务读取数据后，另一并发事务删去了其中部分数据，某些记录意外消失。

不可重复读

时间	TA	数据库中团购人数	TB
1	团购发起人统计参加团购的人数是否达到100人	100人	
2		99人	某团员撤销订单
3	按照团购条件向商家发起订单，结果被驳回	99人	

不可重复读

- 不可重复读包括三种情况：

  (1) 事务T1读取某一数据后，事务T2对其做了修改，当事务T1再次读该数据时，得到与前一次不同的值；

  (2) 事务T1按一定条件从数据库中读取了某些数据记录后，事务T2删除了其中部分记录，当T1再次按相同条件读取数据时，发现某些记录消失了；

  (3) 事务T1按一定条件从数据库中读取某些数据记录后，事务T2插入了一些记录，当T1再次按相同条件读取数据时，发现多了一些记录。

  后两种不可重复读有时也称为幻影现象（Phantom Row）。

读“脏”数据例

读脏

时间	T_1	T2
t1	读X=100 X= X×2＝200	
t2		读X=200
t3	Rollback （X＝100）	
		T2读了无效的数据

产生上述三类数据不一致的原因：并发操作破坏了事务的隔离性。因此对事务的并发操作必须加以控制，才能避免此类现象的发生。

并发控制：用正确的方法调度并发操作，使一个事务的执行不受其他事务的干扰，从而避免数据的不一致现象。

课堂练习

- 判断下列并发操作的正确性。如有错误，则指出属于哪一类错误。

T1	T2
①读 A=10，B=5	
②	读 A=10 A=A*2 写回
③读 A=20，B=5 求和 25 验证错	

不可重复读

课堂练习

- 判断下列并发操作的正确性。如有错误，则指出属于哪一类错误。

T1	T2
①读 A＝100	
A＝A*2 写回	
②	读 A＝200
③ROLLBACK	
恢复 A＝100	

读“脏”

课堂练习

- 判断下列并发操作的正确性。如有错误，则指出属于哪一类错误。

T1	T2
①读 A＝10	
②	读 A＝10
③A=A-5 写回	
④	A=A-8 写回

丢失更新

11.1 并发控制概述

- 并发控制机制的任务：
  - 对并发操作进行正确调度；
  - 保证事务的隔离性；
  - 保证数据库的一致性。

数据的种种不一致性，
是由于并发操作破坏了事务隔离性。

- 并发控制：正确的方式调度并发事务，使得一个事务的执行不受其他事务的干扰，避免造成数据的不一致性。其主要技术包括：
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
  - 多版本并发控制（MVCC）

  商用的DBMS一般都采用封锁方法

11.2 封锁

1. 封锁的定义

事务T在对某个数据对象操作之前，先向系统发出请求，加锁(Lock)，于是事务T对这个数据对象就有一定的控制，直到T释放它的锁为止(Unlock)。

2. 锁的类型

- 排它锁(Exclusive Locks，X锁，写锁)：若事务T对数据R加上X锁，则其他事务不能对R进行任何封锁,保证了其他事务不能再读取和修改R。
- 共享锁(Share Locks，S锁，读锁)：若事务T对数据R加上S锁，则其他事务能对R加S封锁，保证了其他事务能读取但不能修改R。

3. 封锁的粒度

- 封锁对象的大小称为封锁的粒度。
- 关系数据库的封锁对象：属性值、元组、关系、索引

封锁

4. 锁的相容矩阵

T1已获得的锁 \ T2在同一对象上的锁请求	S	X
S	True（相容）	false
X	False（不相容）	false

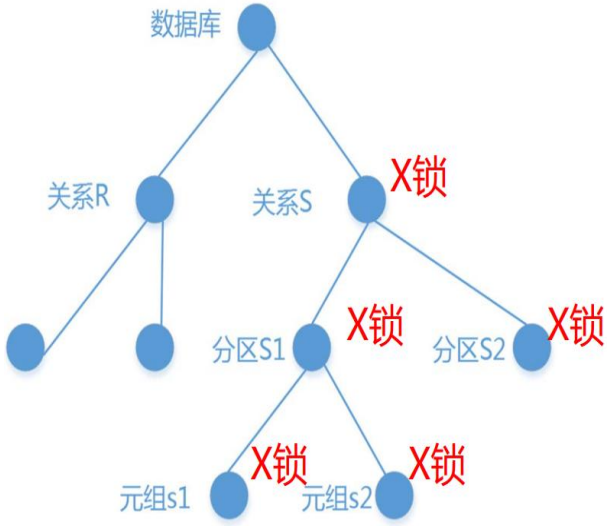
5. 封锁规则

① 对将要存取的数据须先申请加锁，加锁成功才能存取；

② 已被加锁的数据不能再加不相容锁；

③ 一旦退出使用应适时释放锁；

④ 未被加锁的数据不可对之解锁。



并发控制例—银行账户

到银行取钱

① 银行读取帐户
余额400元

④ 取款100元
更新帐户余额

序号	金额	帐户余额
……	……	……
5	-700.00	400.00
6	2000.00	2400.00
7	-100.00	300.00

余额错误

单位发工资

② 单位读取帐户
余额400元

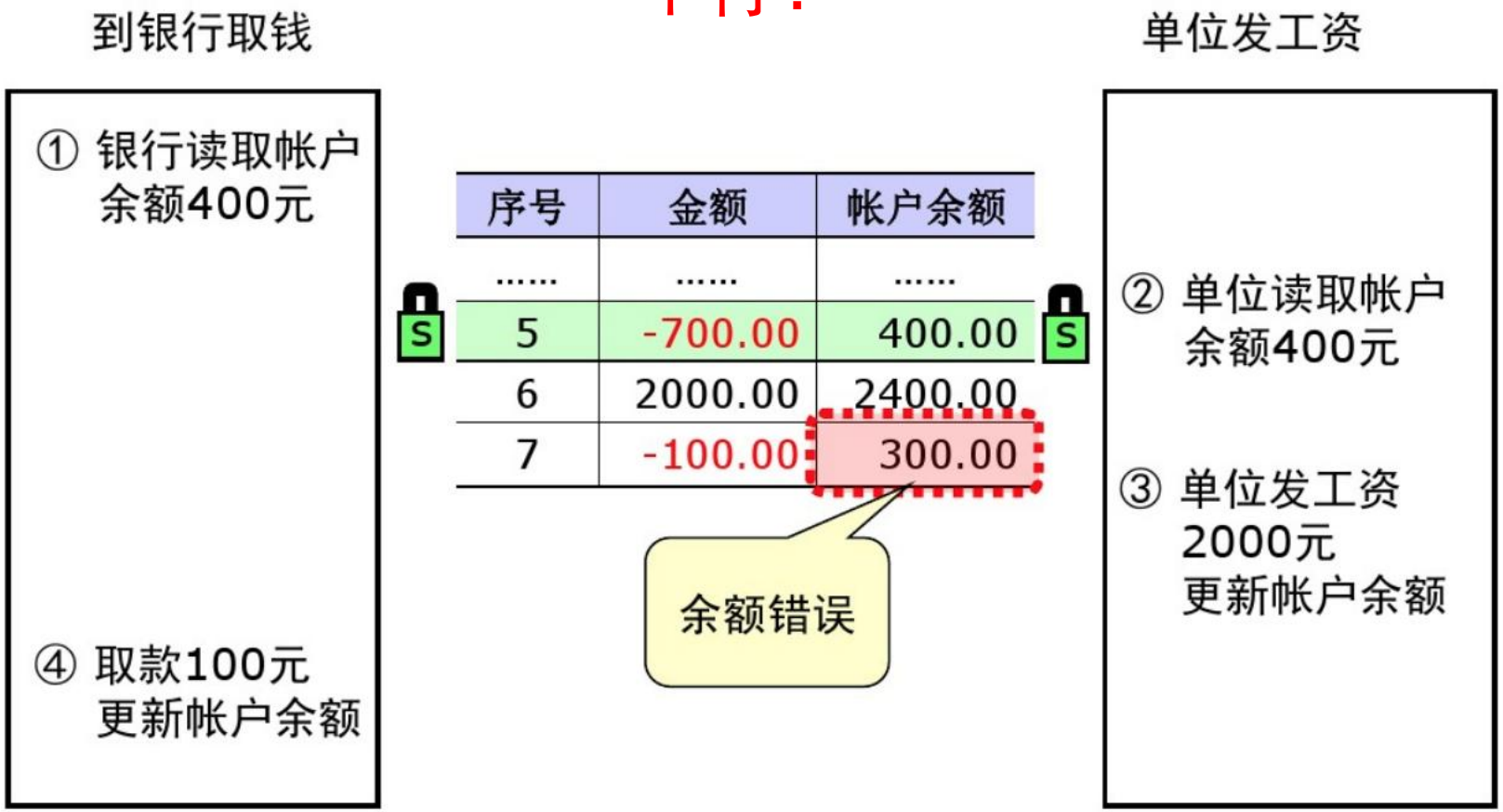
③ 单位发工资
2000元
更新帐户余额

- 存在什么问题?
- 解决方法：加锁——
在读最后一条记录时加
- 加什么锁？S or X ？

并发控制例—银行账户（续）

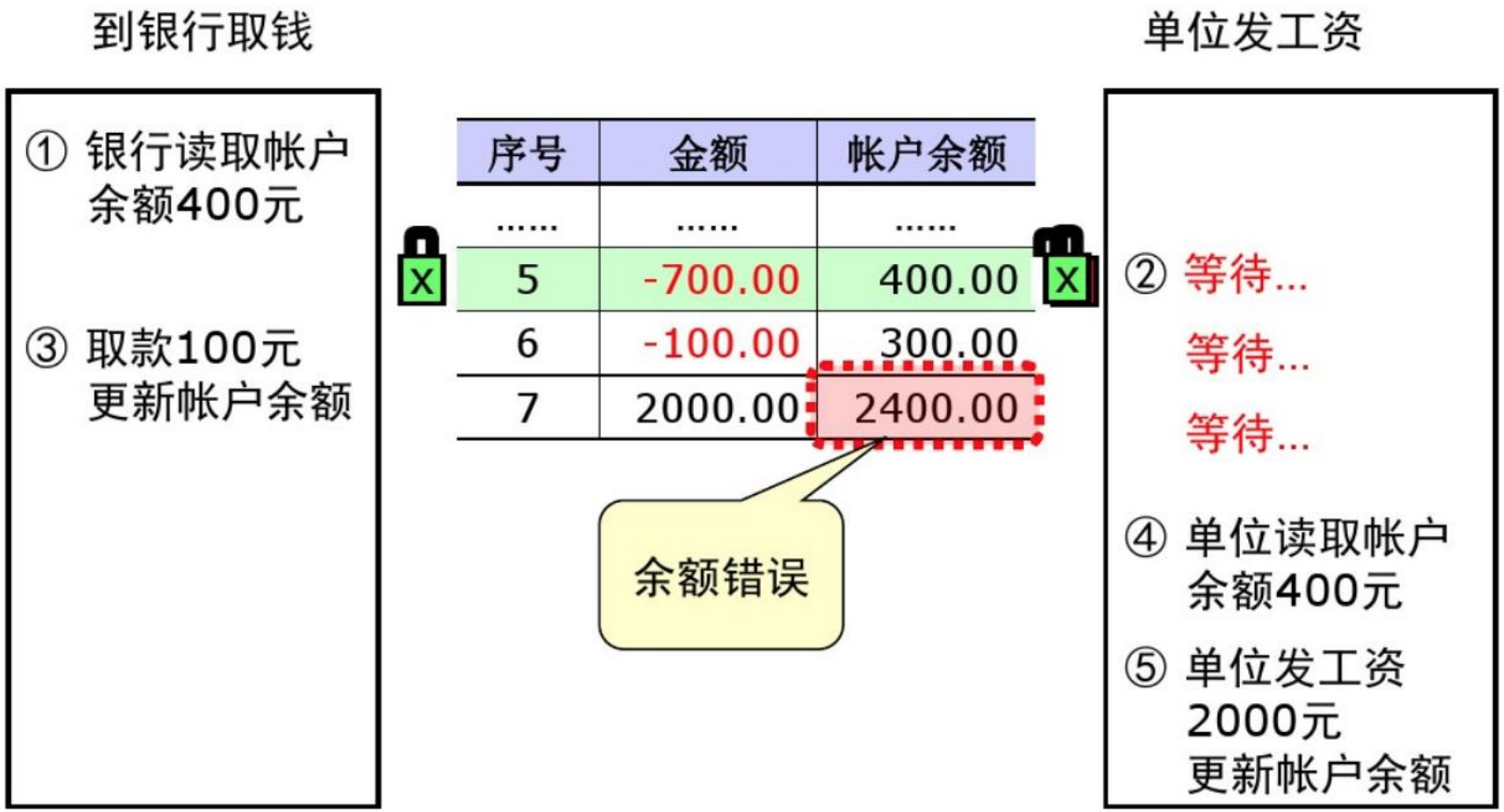
- 读前上共享锁S?

不行！



并发控制例—银行账户（续）

- 读前上排它锁X，读完立即释放？ 不行！



三级封锁协议

系统中的每一个事务都必须遵从的关于何时对数据项加何种锁，何时解锁的一组规则：封锁协议（Locking Protocol）。

- 何时申请X锁或S锁
- 持锁时间、何时释放

- 不同的封锁协议，在不同的程度上为并发操作的正确调度提供一定的保证。不同级别的封锁协议达到的系统一致性级别是不同的。
- 常用的封锁协议:
  - 支持一致性维护的三级封锁协议
  - 支持并行调度可串行化的两段锁协议
  - 避免死锁的协议

1．一级封锁协议

1）策略

事务Ti在修改数据Di之前须先对Di加X锁，直到事务Ti结束（commit / rollback）才释放。

2）功能

- 防止丢失更新；
- 保证Ti可恢复（若意外终止，则回滚后才可释放锁）。

3）问题

在1级封锁协议中，如果仅是读数据，是不需要加锁的，不能防止不可重复读和读“脏”数据（why?）

T1	T2
Xlock(A)= T 读 A=16	
	Xlock(A)=F
A:=A-1=15	
	Xlock(A)= F
COMMIT	
	Xlock(A)= F
Unlock（A）	
	Xlock(A)=T 读 A＝15
	A:=A-1=14
	COMMIT

写丢失避免了

T1	T2
A=50 B=100	
	Xlock(B)=T B:=B*2=200
Xlock(S)=T S:=A+B=150	
	COMMIT
	Unlock(B)
A=50 B=200	
Xlock(S1)=T S1:=A+B=250	
S!＝S1	

不可重复读

T1	T2
Xlock(C)=T C=100	
C:=2*C=200	
	C=200
ROLLBACK	
UNLOCK(C)	

读“脏” 数据

2．二级封锁协议

1）策略

1级封锁协议 + 事务Ti在读取Di之前必须对Di加S锁，读完后即可释放该S锁。

2）功能

- 防止丢失更新；
- 防止读脏。

3）问题

在2级封锁协议中，不能防止不可重复读（why?）

“由于读完数据后即可释放S锁”

	T_1	T_2	T_1	T_2
1)	SLOCK(A)=T A=50 SLOCK(B)=T B=100		Xlock(C)=T C=100 C:=2*C=200	
2)	Unlock(A) Unlock(B)			SLOCK(C)=F
3)	Xlock(S)=T S:=A+B=150		ROLLBACK	
4)		Xlock(B)=T B=B*2=200		SLOCK(C)=F
5)		COMMIT	UNLOCK(C)	
6)		Unlock(B)		SLOCK(C)=T
7)	SLOCK(A)=T SLOCK(B)=T A=50 B=200 Unlock(A) Unlock(B)			C=100
8) 9)	Xlock(S1)=T S1:=A+B=250, S!=S1			UNLOCK(C)

不可重复读不能避免

读“脏”数据避免了

3．三级封锁协议

1）策略

1级封锁协议 + 事务Ti在读取Di之前必须对Di加S锁，直至Ti结束后即可释放该S锁。

2）功能

- 防止丢失更新；
- 防止读“脏”；
- 防止读不可重复

对比：

一级封锁协议：

事务Ti 在修改数据Di 之前须先对Di 加X锁，直到事务Ti结束（commit / rollback）才释放。

二级封锁协议：

1级封锁协议 + 事务Ti 在读取Di 之前必须对Di 加S锁，读完后即可释放该S锁

	T_1	T_2	T_1	T_2
1)	SLOCK(A)=T A=50 SLOCK(B)=T B=100		Xlock(C)=T C=100 C:=2*C=200	
2)		Xlock(B)=F		SLOCK(C)=F
3)	Xlock(S)=T S:=A+B=150		ROLLBACK	
4)		Xlock(B)=F		SLOCK(C)=F
5)	Xlock(S1)=T A=50 B=100 S1:=A+B=150		UNLOCK(C)	
6)	Unlock(A) Unlock(B)			SLOCK(C)=T
7)		Xlock(B)=T B=100 B=B*2=200		C=100
8)	S＝S1 Unlock(S) Unlock(S1)			UNLOCK(C)

不可重复读避免了

读“脏”数据避免了

三级封锁协议

- 三级封锁协议的主要区别：在于何种操作需要申请封锁，以及获得封锁后何时释放锁（持锁时间）。分别三级封锁协议中不同级别的协议，得到的一致性保证是不同的。

	X锁		S锁		一致性保证		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不丢失修改	不读脏数据	可重复读
一级		√			√		
二级		√	√		√	√	
三级		√		√	√	√	√

三级协议的主要区别：

- 什么操作需要申请封锁
- 何时释放锁（即持锁时间）

11.3.1 活锁

1. 含义：事务因故永远处于等待状态。

T_1	T_2	T_3	T_4	T_n
LOCK(R)=T				
	LOCK(R)=F			
	等待	LOCK(R)=F		
UNLOCK(R)	等待			
	等待	LOCK(R)=T		
	等待	…	LOCK(R)=F	
	等待	UNLOCK(R)		
	等待		LOCK(R)=T	

2. 预防方法

FCFS（First Come First Server）：先来先服务

对于事务有优先级的系统，可设置一个最长等待时间，与优先级结合，调度事务的执行。

11.3.2 死锁

1. 含义：两个或两个以上事务均处于等待状态，每个事务都在等待其中另一个事务封锁的数据，导致任何事务都不能继续执行的现象称为死锁。

时间	T_A	T_B
t1	X Lock A	
t2		X Lock B
t3	X Lock B 等待	
t4		X Lock A 等待
t5	等待	等待

死锁实例1

死锁情况：两个事务的两条SQL产生

```
Table T1(id primary key, name)
```

```
Session 1:
Begin;

Select * from t1 where id=1 for update;

Update t1 set name='deadlock' where id=5;
```

```
Session 2:
Begin;

Delete from t1 where id=5;

Delete from t1 where id=1;
```

死锁发生

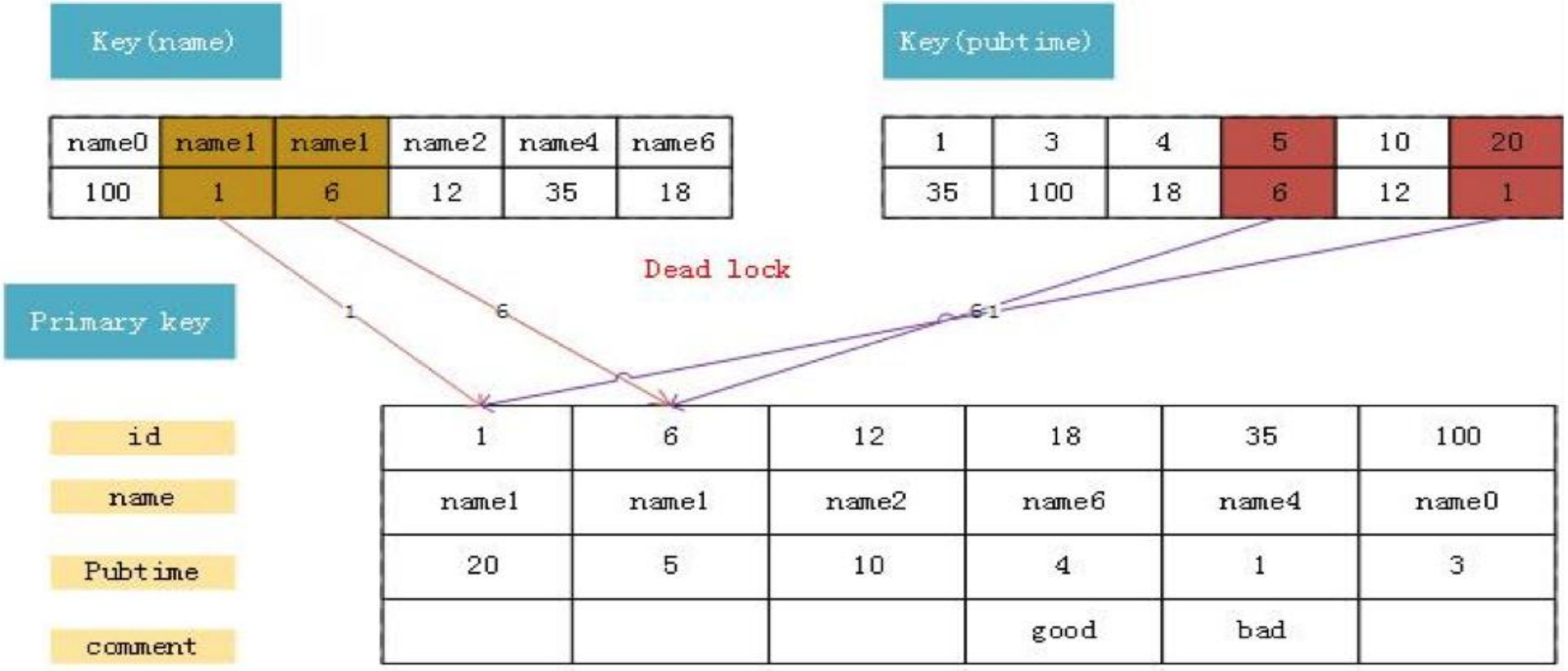
1	2	3	4	5	6
Aaa	bbb	ccc	ddd	eee	ff

死锁实例2

死锁情况：两个事务的一条SQL产生

```
Table T2(id primary key, name key, pubtime key, comment)
```

Session 1:	Session 2:
Begin:	Begin:
Update t2 set comment= 'avd' where name=' name1'	Select * from t2 where pubtime>=5 for update



11.3.2 死锁

2. 死锁产生条件

① 互斥（排它性控制）

② 不可剥夺（释放前，其它事务不能剥夺）

③ 部分分配（每次申请一部分）

④ 环路（每事务获得的数据同时又被另一事务请求）

3. 死锁的解决办法：破坏死锁产生的条件

1) 预防： 一次封锁法：每个事务事先一次获得其需数据的全部锁。

- 特征：
  - 简单，无死锁；粒度大，并发性低；
  - 难以事先精确确定封锁对象。

【例】

T_A 获得所有数据A、B锁，

T_A 连续执行，

T_B 等待；

T_A 执行完后释放A、B锁，

T_B 继续执行，不会发生死锁。

11.3.2 死锁

3. 死锁的解决办法（续）

2）预防：顺序封锁法

事务按预先确定的数据封锁顺序实行封锁。

- 缺点：

事务动态变化，很难按规定的顺序去施加封锁。

上例中：

设封锁顺序：A→B；

T1、T2均按此顺序申请锁；

若T1先获得A、B锁；

T2则先申请A锁，等待；

T1释放A、B锁；

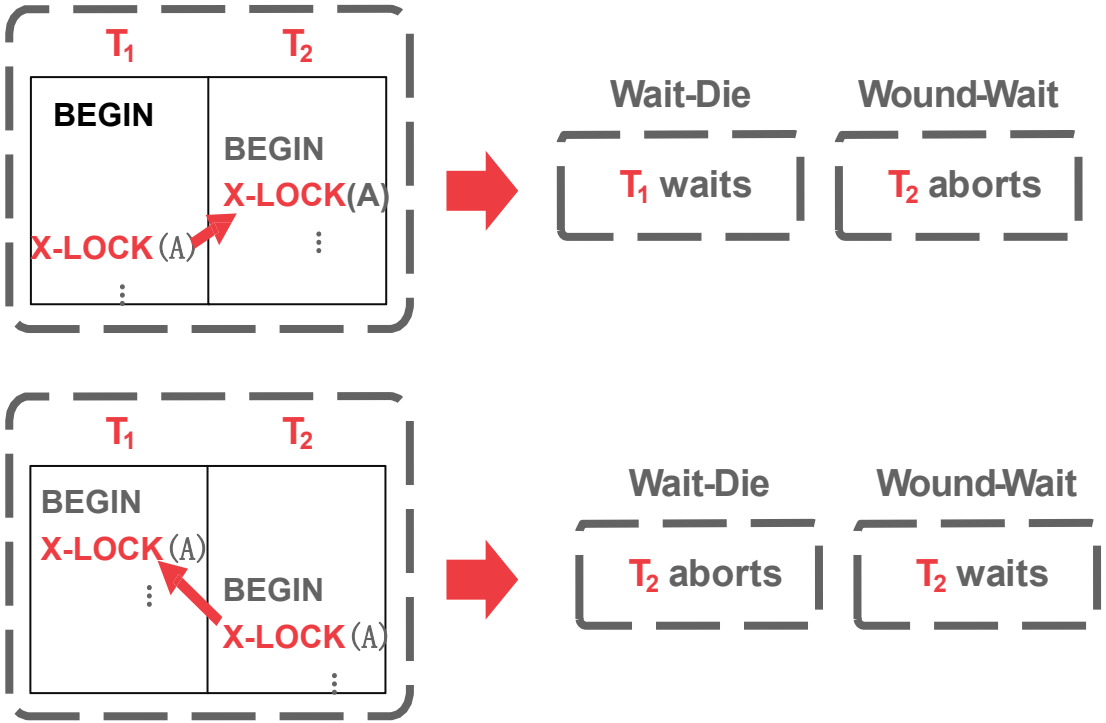
T2获得锁运行。

11.3.2 死锁

为什么能确保无死锁?
---只允许单向等待锁

3）预防：时间戳优先级法

- 根据时间戳分配优先级：较老时间戳 = 高优先级 (e.g., T1 > T2)
- Wait-Die (“Old Waits for Young”)：请求锁事务比持有锁的事务具有更高的优先级，则请求事务等待其完成；否则，请求事务撤销。
- Wound-Wait (“Young Waits for Old”)：请求的事务比持有锁的事务具有更高的优先级，则撤销持有锁的事务，释放锁；否则，请求事务等待。



11.3.2 死锁

3. 死锁的解决办法（续）

在操作系统中广为采用的预防死锁的策略并不很适合数据库的特点。

DBMS在解决死锁的问题上更普遍采用的是诊断并解除死锁的方法。

- 诊断：

  1）超时法

  若事务的等待时间超过了规定的时限，就认为发生了死锁。

  - 缺点：

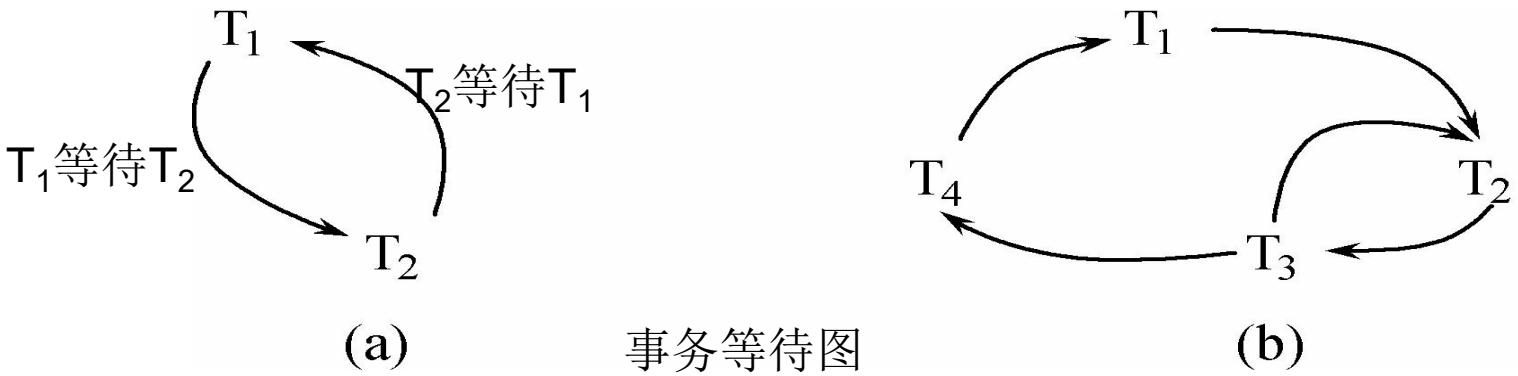
    1）若时限设置得太短，容易误判；

    2）若时限设置得太长，死锁不易及时发现。

11.3.2 死锁

2）等待图法：用事务等待图动态反映所有事务的等待情况。

① 构造一事务等待图——有向图$G=(T, U)$，图中每个节点表示正在运行的事务，每条边表示等待的情况；



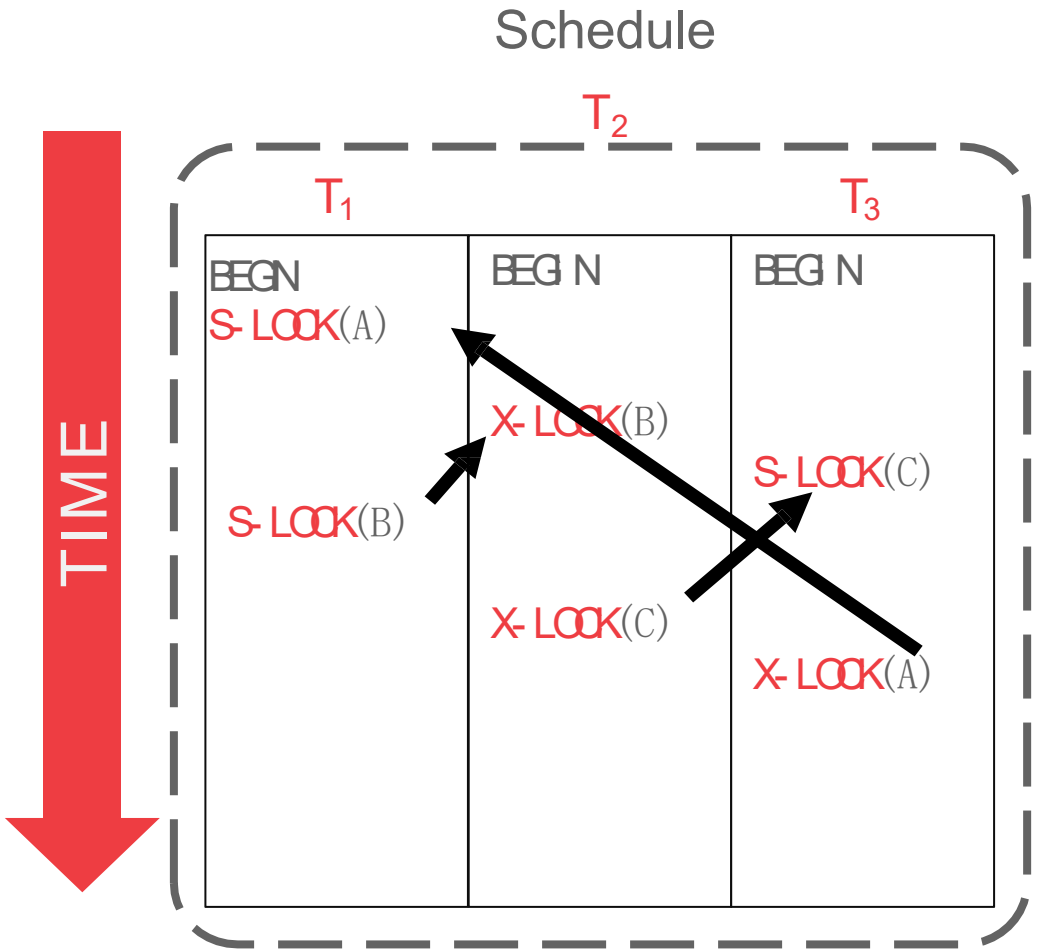
- T为结点集，每个结点表示正运行的事务；
- U为边集，每条边表示事务等待的情况；
- 若 T_1 等待 T_2，则 T_1，T_2 之间划一条有向边，从 T_1 指向 T_2。

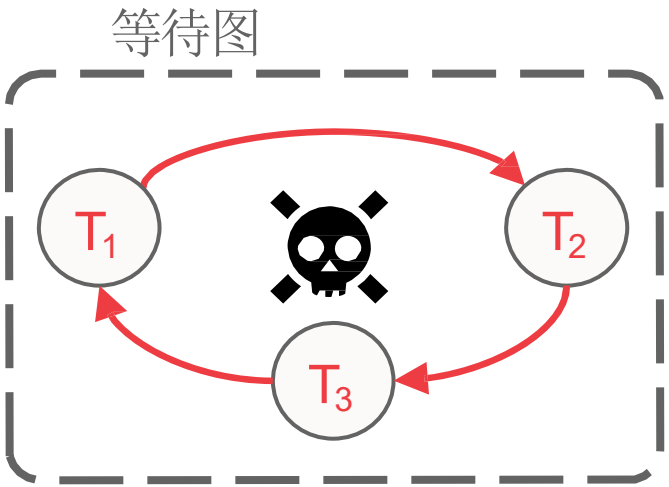
② 周期性检测（如每隔数秒）该等待图；

③ 判断存在回路否；

④ 存在，则撤消某一事务：选择一个处理死锁代价最小的事务（NP难度问题），释放该事务所有的锁，使其它事务得以继续运行。

等待图法





并发控制子系统周期性地（比如每隔数秒）生成事务等待图，检测事务。如果发现图中存在回路，则表示系统中出现了死锁。

死锁的诊断与解除（续）

- 解除死锁

  选择合适的事务撤销取决于多种因素….

  - 事务年龄 (最小的时间戳)
  - 事务进度 (执行最少/最多查询)
  - 所锁定的数据库对象数量
  - 需回滚事务的数量

- 还需考虑事务重启动的次数，以防事务饿死。

第11章 并发控制概述

11.1 并发控制概述

11.2 封锁

11.3 活锁和死锁

11.4 并发调度的可串行性

11.5 两段锁协议

11.6 封锁的粒度

11.7 小结

- 什么样的并发操作调度是正确的?
- 如何保证并发操作调度是正确的?

11.4 并发调度的可串行性

1. 可串行化(Serializable)调度

当且仅当多个事务并发执行的结果与该事务任一串行执行的结果相同时，则该并发执行是可串行化的。

【例】假设有两个事务T1、T2，数据库中A、B的初值均为2；

T1：读B；A = B + 1；写回A　　简写为：r1(B); w1(A)

T2：读A；B = A + 1；写回B　　简写为：r2(A); w2(B)

- 如果是串行调度，只有两种可能的调度序列：

  r1(B); w1(A); r2(A); w2(B)，　　结果为：A = 3，B = 4

  r2(A); w2(B); r1(B); w1(A)，　　结果为：A = 4，B = 3

- 两个事务并发执行，则可能的调度序列如下：

  r1(B); r2(A); w1(A); w2(B)

这个调度是不是可串行化的?

11.4 并发调度的可串行性

问题:

丢失修改

T_1	T_2
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

- r1(B); r2(A); w1(A); w2(B)

这是个不可串行化调度，错误的调度

可串行性(Serializability)

- 是并发事务正确调度的唯一准则!
- 一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度

11.4 并发调度的可串行性

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

- 执行结果与串行调度(a)的执行结果相同
- 是正确的调度

11.4 并发调度的可串行性

- 冲突可串行化——判断可串行化调度的充分条件。

  对于一个并发调度Sc，在保证冲突操作次序不变的情况下，如果通过交换不冲突操作可以得到一个串行调度，则Sc为冲突可串行化调度。

- 冲突操作——不同事务对同一个数据的读写操作或写写操作：
  - Ri (x)与Wj(x)　　/* 事务Ti读x，Tj写x*/
  - Wi(x)与Wj(x)　　/* 事务Ti写x，Tj写x*/
  - 其他操作是不冲突操作，如：

    Ri(x)与Rj(y)，Ri(x)与Rj(x)；Ri(x)与Wj(y)，或Wi(x)与Wj(y)，x<>y

- 交换（swap）原则——
  - 不同事务的冲突操作不能交换次序；
  - 同一事务的两个操作不可交换。

即：不同事务的任何2个动作在顺序上可以交换，除非：
他们涉及同一个数据库元素；
并且，至少有一个是写。

冲突可串行化调度

- 两个调度是冲突等价的，即：通过一系列相邻动作的非冲突交换能将它们中的一个转换为另一个。

【例1】 Sc1 = r1(A) w1(B) r2(B) w1(C) w2(B)

Sc2 = r1(A) w1(B) w1(C) r2(B) w2(B)

- 冲突可串行化调度是可串行化调度的充分条件，不是必要条件。还有不满足冲突可串行化条件的可串行化调度。

【例2】有3个事务： T1=W1(Y)W1(X)， T2=W2(Y)W2(X)， T3=W3(X)

调度L1=W1(Y)W1(X)W2(Y)W2(X)W3(X)是一个串行调度。

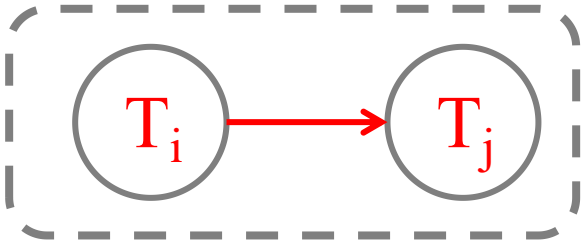
调度L2=W1(Y)W2(Y)W2(X)W1(X)W3(X)

  - 不满足冲突可串行化；
  - 但是调度L2是可串行化的，因为L2执行的结果与调度L1相同，Y的值都等于T2的值，X的值都等于T3的值

问题：

- 只有两个事务时，可以快速地通过交换无冲突操作来判断该调度是否是冲突可串行化。

  则：三个事务？多个事务？n个事务的可串行化调度结果n!种



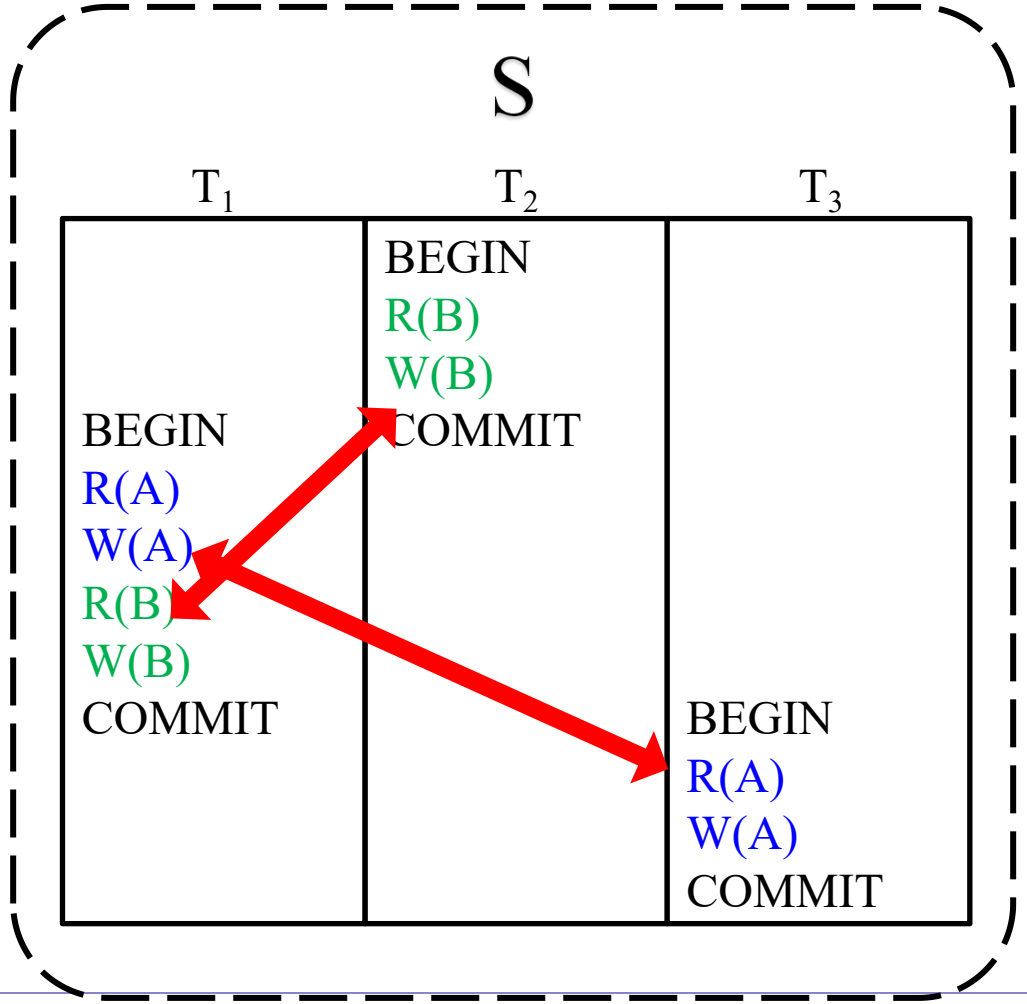
- 依赖图（Dependency）法：
  - 每个节点代表一个事务；
  - 找到所有的读写操作，从 T_i 节点到 T_j 节点的边表示：

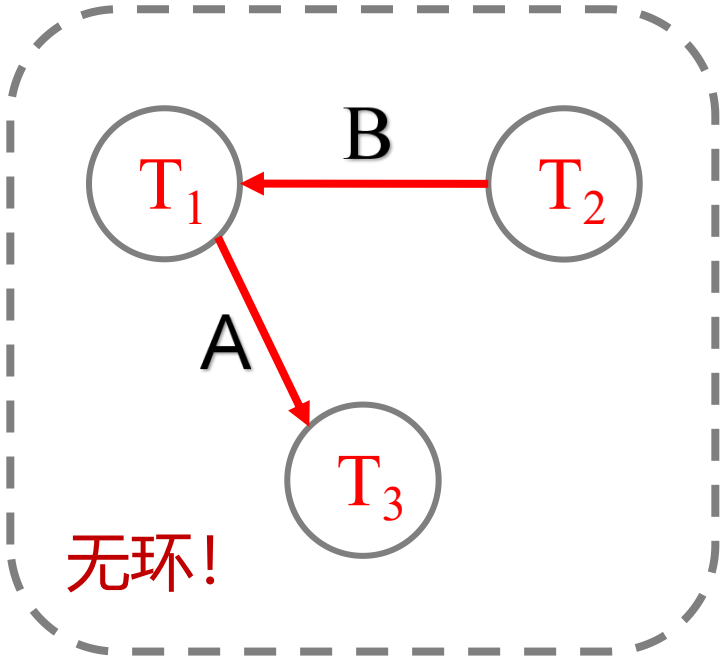
    T_i 中的操作 O_i 发生在 T_j 中的操作 O_j 之前，且 O_i 和 O_j 是冲突的（不同事务间的读写、写写冲突操作）。

  - 如果最后得到一个有向无环图，则结果是冲突可串行化的！

依赖图例1

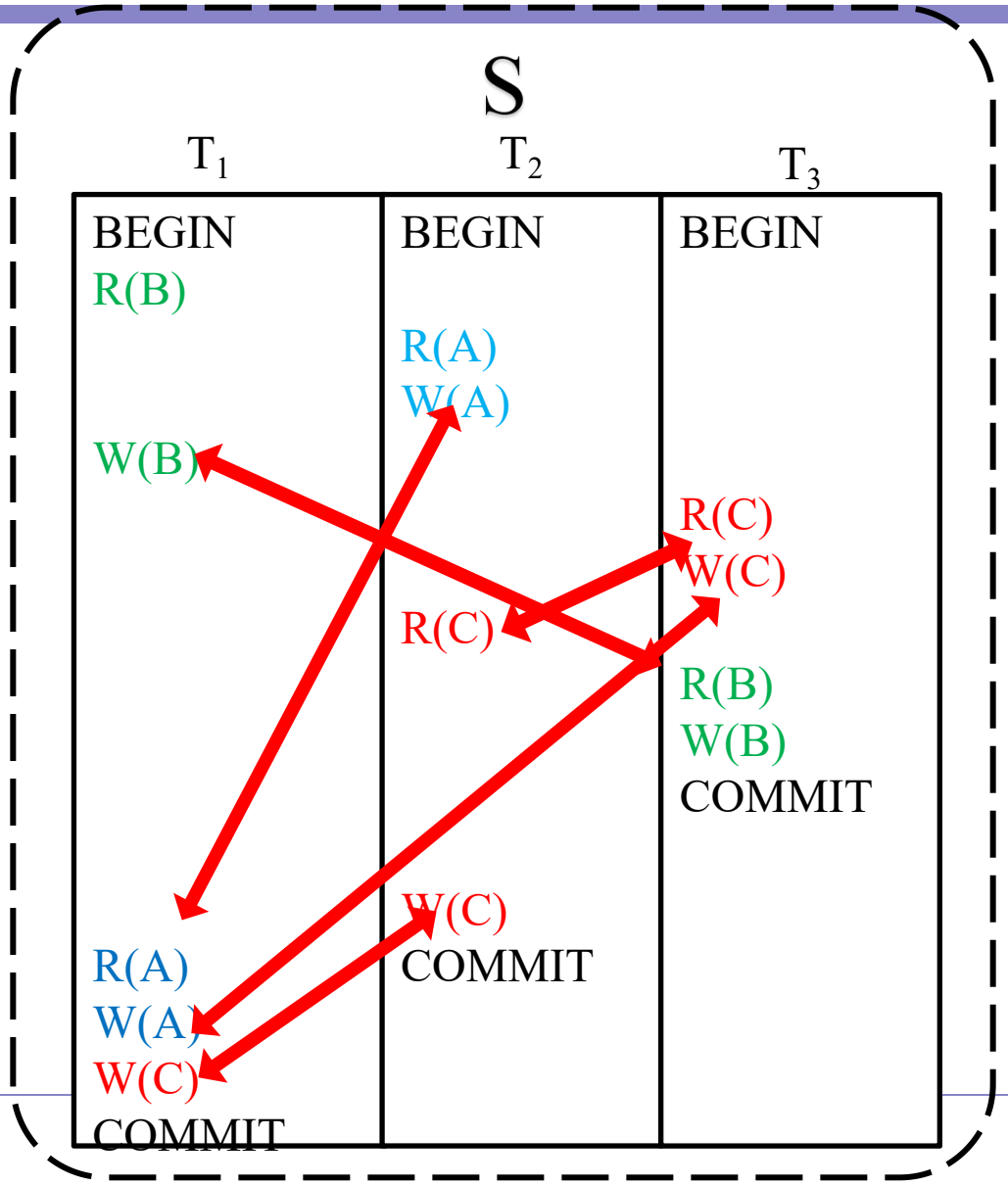
- 请判断以下调度是否是冲突可串行化的?

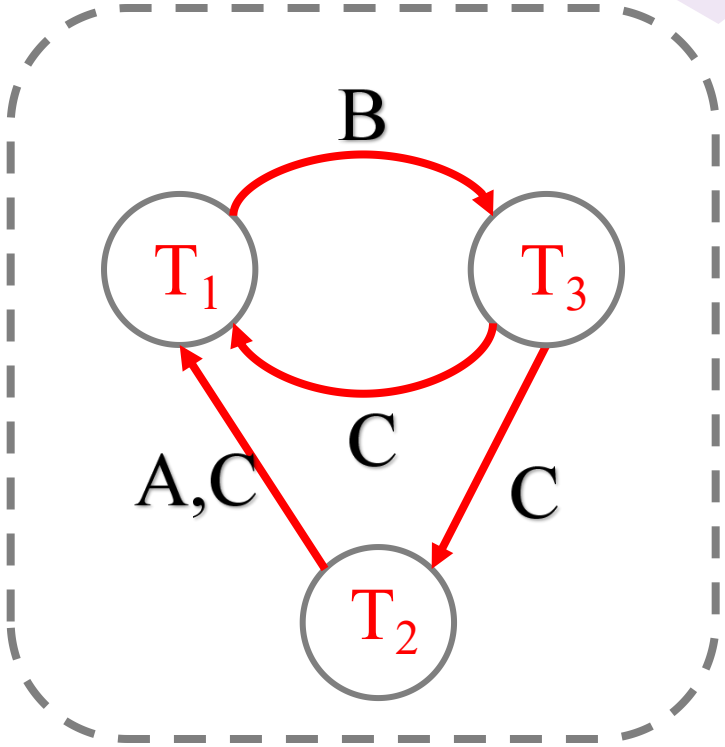




串行调度的依赖图一定是有向无环图。

依赖图例2



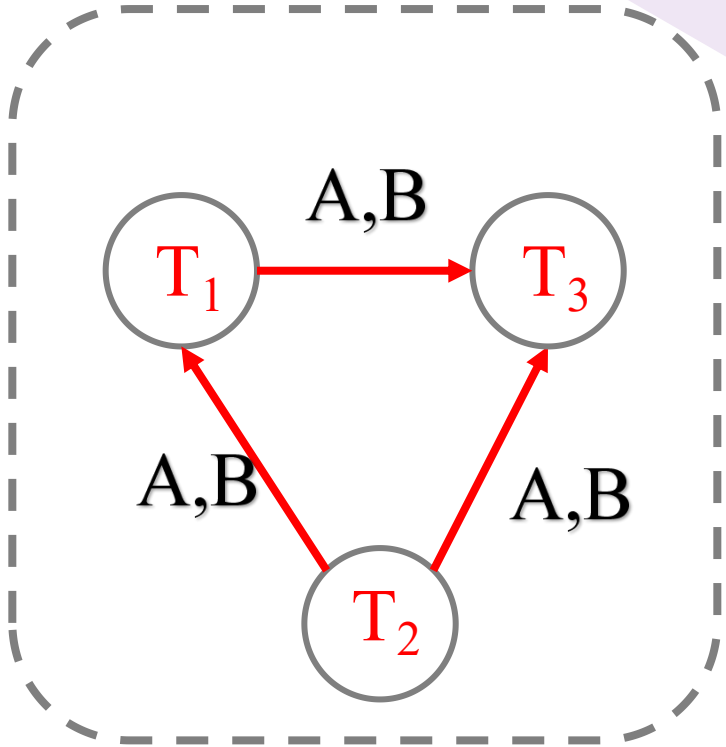


环：非冲突可串行化调度

依赖图例3

S

T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
R(A)		
A=A-10		
	R(A)	
	R(B)	
	R(C)	
		R(D)
		D=D+20
W(A)		
R(B)		
B=B+10		
		W(D)
		A=D-50
		W(A)
W(B)		
COMMIT		
		R(B)
		B=B-10
		W(B)
	C=C+A-B	
	W(C)	
	COMMIT	
		COMMIT



无环：冲突可串行化调度

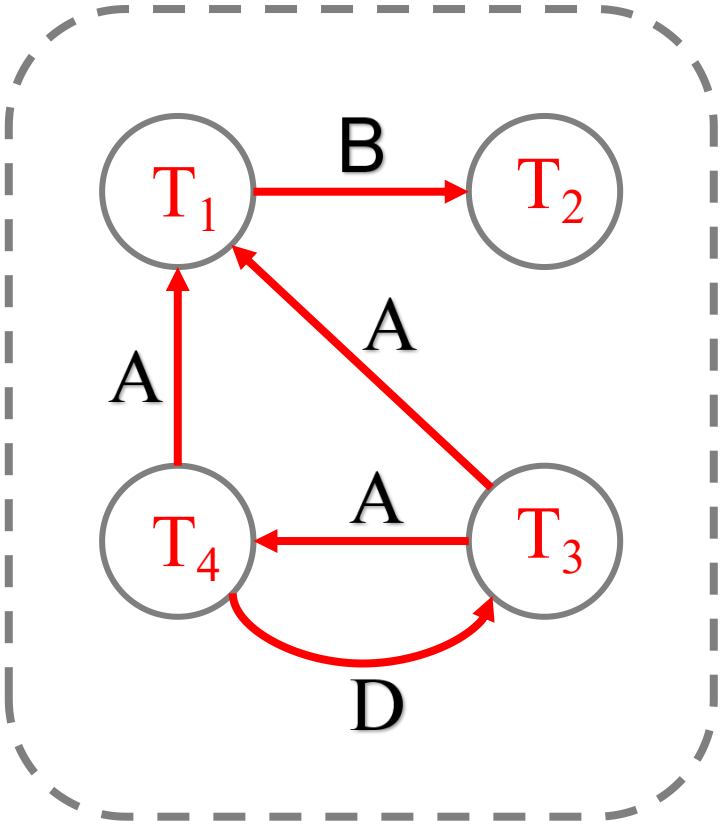
R1(A) R2(A) R2(B) R2(C) R3(D) **W1(A) R1(B)** W3(D)W3(A) **W1(B)**R3(B)W3(B)**W2(C)**

→ T2 T1 T3

依赖图例4

S

T_1	T_2	T_3	T_4
BEGIN	BEGIN	BEGIN	BEGIN
R(B)		R(A)	
		W(A)	
W(B)			
			R(D)
	R(C)		
		R(C)	
			R(A)
	R(B)		
	W(B)		
	COMMIT		
		W(D)	
		COMMIT	W(A)
			COMMIT
R(A)			
R(C)			
COMMIT			



T3 T4 T1 T2 或 T4 T3 T1 T2

11.5 两段锁协议（2PL: two-phase locking）

1. 两段锁协议目标

可串行性是并发调度正确性的唯一准则；

2PL是保证并发操作调度的可串行化而提供的封锁协议。

2. 两段锁含义

事务分为两个阶段对数据加锁和解锁：

第一阶段称为扩展阶段（获得锁）；

第二阶段称为收缩阶段（释放锁）。

3. 策略

① 在对任何数据读、写之前，须先获得该数据的锁。

② 在释放一个封锁之后，该事务不能再申请任何其它锁。

11.5 两段锁协议

【例】

- 事务1: 遵循两段锁协议的事务，其封锁序列为：

SLOCK(A) SLOCK(B) XLOCK(C) UNLOCK(B) UNLOCK(A) UNLOCK(C)

扩展阶段　　收缩阶段

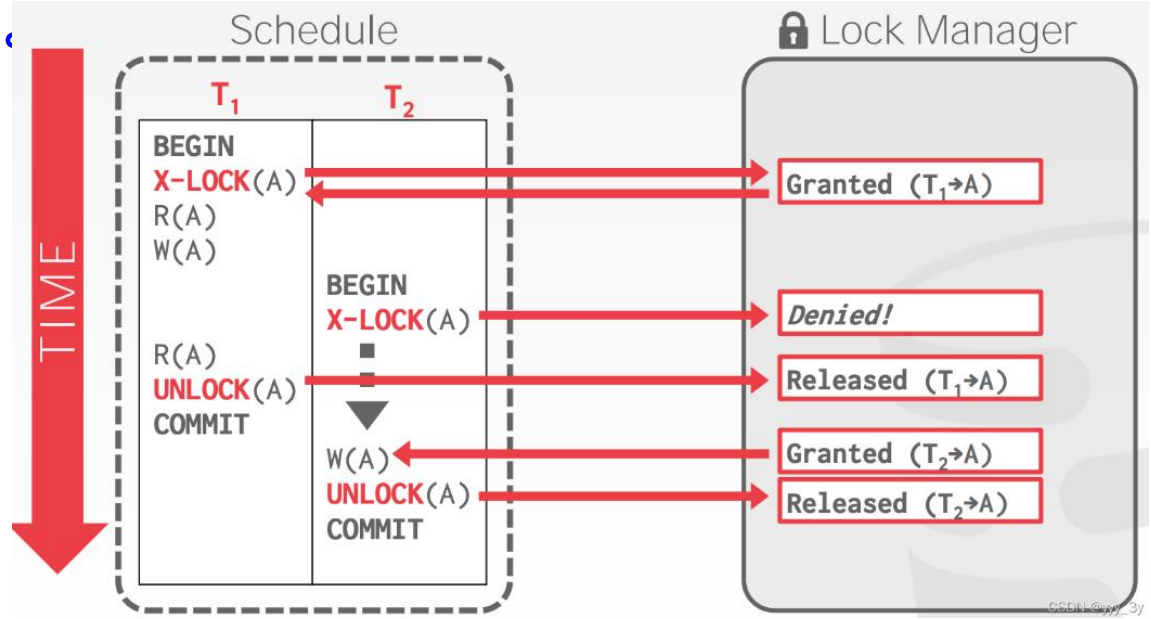
事务	加锁/解锁处理
begin;	
insert into test	加insert对应的锁
update test set...	加update对应的锁
delete from test	加delete对应的锁
commit;	事务提交时，同时释放insert、update、delete对应的锁

- 事务2: 不遵循两段锁协议的事务，其封锁序列为：

SLOCK(A) UNLOCK(A) SLOCK(B) XLOCK(C) UNLOCK(B) UNLOCK(C)

两段锁协议

- 定理：若所有并发事务都遵守两段锁协议，则对这些事务的所有并发调度都是可串行化的。



- 关于两段锁协议的两点说明：

1）两段锁协议是可串行化调度的充分条件，但不是必要条件。

2）所有遵守2PL的事务，其并行执行的结果一定是正确的。

但，2PL不能防止死锁（注意区别于防止死锁的一次封锁法）。

例

T_1	T_2
S Lock B B=2 Y=B X Lcok A	
	S Lock A 等待
A=Y+1 写回A=3 UNLock B UNLock A	等待
	S Lock A A=3 Y=A
A＝3 B＝4	X Lock B B=Y+1 写回B=4 UNLockB UNLockA

遵守2PL，可串行

T_1	T_2
S Lock B B=2 Y=B Unlock B X Lcok A	
	S Lock A 等待
A=Y+1 写回A=3 UNLock A	等待
	S Lock A A=3 Y=A
A＝3 B＝4	X Lock B B=Y+1 写回B=4 UNLockB UNLockA

不遵守2PL，可串行

T_1	T_2
	S Lock A A=2 X=A Unlock A
S Lock B B=2 Y=B UnLcok B	X Lock B 等待
	X LockB B=X+1 写回B=3 UNLockB
X Lock A A=Y+1 写回A=3 UNLock A	A＝3 B＝3

不遵守2PL，不可串行

两段锁协议

- 两段锁协议与防止死锁的一次封锁法:
  - 一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行，因此一次封锁法遵守两段锁协议。

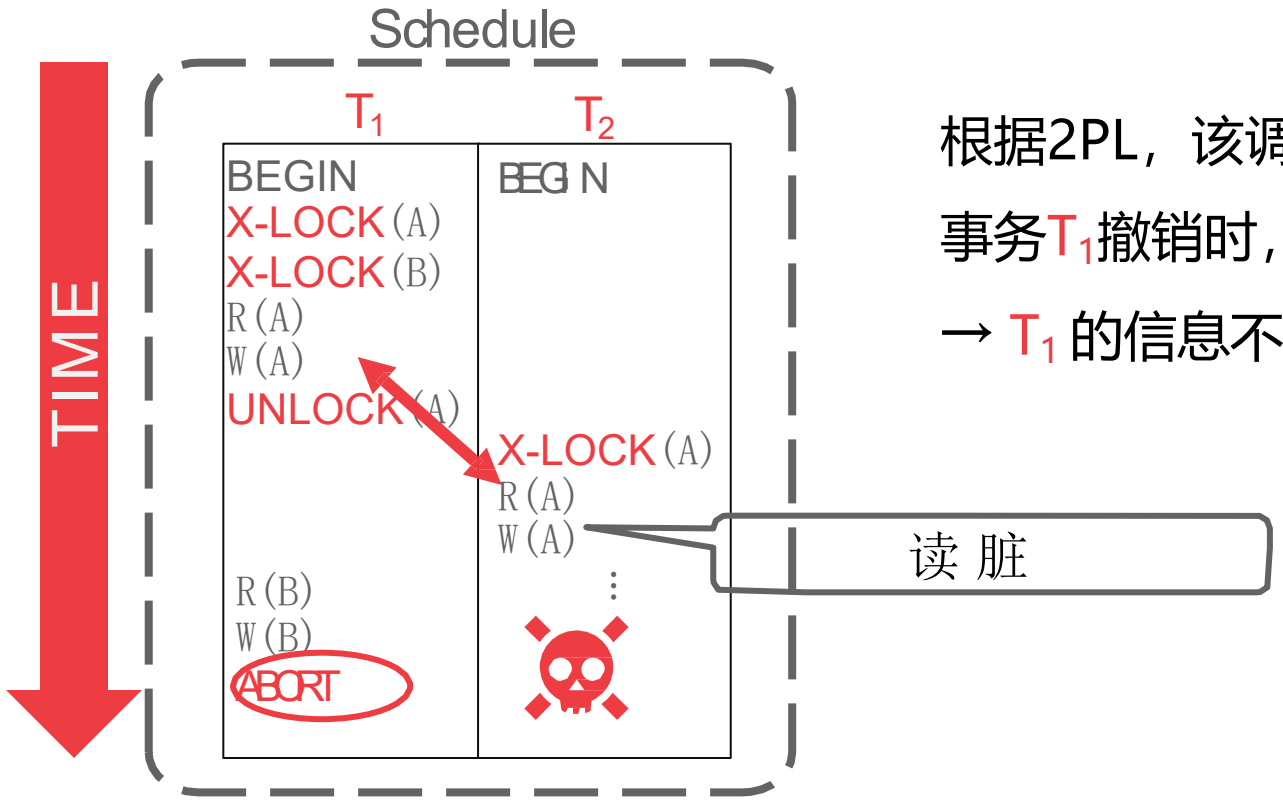
- **但是两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁**

T_1	T_2
Slock B	
R(B)=2	
	Slock A
	R(A)=2
Xlock A	
等待	Xlock B
等待	等待

遵守两段锁协议的事务可能发生死锁

两段锁协议

- 2PL足够保证冲突可串行化（所产生的调度图是无环），但存在级联撤销（cascading abort）。

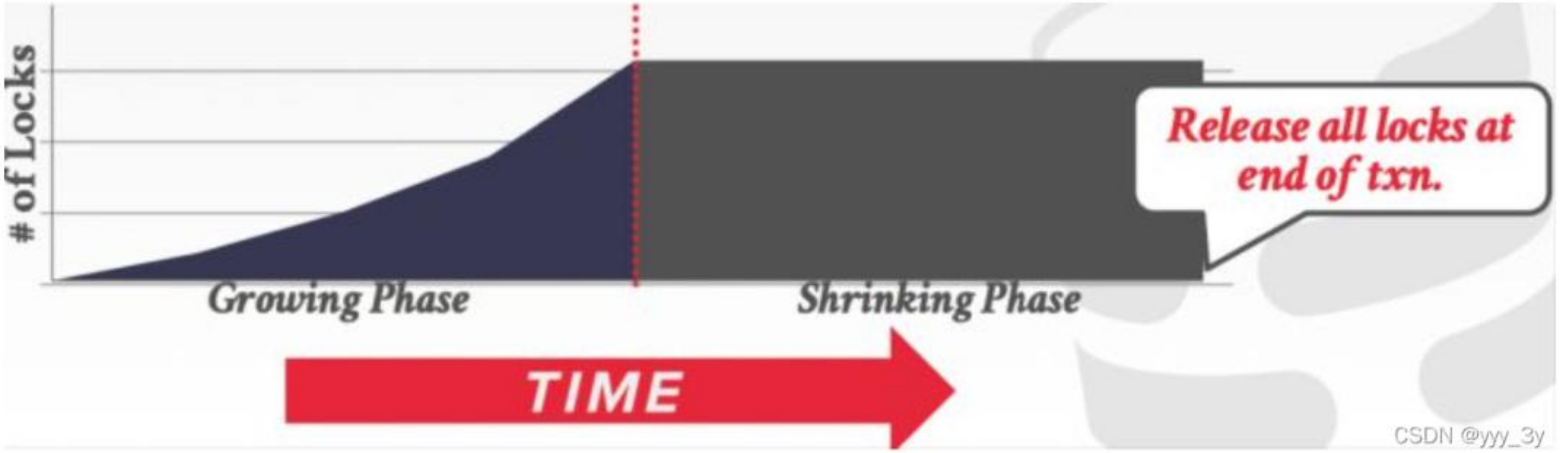


根据2PL，该调度是可行的，但当事务T_1撤销时，数据库须撤销T_2

→ T_1 的信息不能泄露给外界

两段锁协议

- STRONG STRICT 2PL：规定在 Shrinking Phase 阶段，只有到 commit 或 abort 时才会一次性释放全部锁（可以避免脏读的情况）。
  - 调度是严格的：如果在一事务结束前，该事务所写的值不会被其它事务读或写。
  - 优点：不会引发级联撤销，撤销事务只需恢复修改元组的原始值.



两段锁协议 vs. 三级封锁协议

- 两类不同目的的协议:
  - 两段锁协议：
    保证并发调度的正确性。
  - 三级封锁协议：
    在不同程度上保证数据一致性。

- 遵守第三级封锁协议必然遵守两段锁协议。

2PL课堂练习

考虑以下两个事务：

T1：

```
Read(B);
Read(A);
If A=0 then B=B+ 5 ;
Write(B);
```

T2：

```
Read(A);
Read(B);
If B=0 then A=A+1    else A=A+B;
Write(A);
```

(1) 请给T1和T2增加加锁和解锁指令，使他们遵从2PL协议；

(2) 使用上述锁机制后，可能发生死锁么？若有请给出其调度的一个例子。

2PL课堂练习-参考解答

（1）参考如右图。

（2）可能发生死锁。
上述调度T2的SLock(B)提前到T1的XLock(B)之前，就会锁住T1，进而T2的XLock(A)申请不得，锁住T2，成为死锁。

T1	T2
SLock(B)=T;	
READ(B);	
SLock(A)=T;	
READ(A);	
If A=0 then B=B+ 5 ;	SLock(A)=T;
XLock(B)=T;	READ(A);
WRITE(B);	SLock(B)=F;
UNLOCK(A);	…
UNLOCK(B);	…
UNLOCK(B);	…
	SLock(B)=T;
	READ(B);
	If B=0 then A=A+1 else A=A+B;
	XLock(A)=T;
	WRITE(A);
	UNLOCK(B);
	UNLOCK(A);
	UNLOCK(A);