数据管理

- 数据库事务表示实际任务。
- 事务可以定义为具有四个核心属性的程序单元。
  - 原子性：逻辑工作单元。是不可分割的单位，要么完全成功或者失败，不会部分成功或失败。
  - 一致性：完整性逻辑单元。将 Database 从一种一致状态移动到另一种一致状态。
  - 独立性：并发逻辑单元。事务独立于数据库系统内可能同时发生的其他事务执行。
  - 持久性：恢复逻辑单元。数据库系统中的故障不会导致已完成事务的影响被撤消。 一旦 “提交”，效果在失败后就可以恢复。
- 提交：将整个事务提交，事务的作用对整个数据库可见。
- 回滚：撤销当前事务对数据库的影响。
- 多用户模式下可能出现的问题：丢失更新；缺乏依赖项；数据不一致。
  - 丢失更新

The Lost Update Problem

Transaction A	TIME	Transaction B
Fetch V (= 10)	T1	
	T2	Fetch V (=10)
Update V V=V+10 (=20)	T3	
	T4	Update V V=V+20 (=30)
Commit	T5	
	T6	Commit

At time after T4 the update performed by transaction A is lost; the stored value of V is 30.

  - 缺乏依赖项

The Uncommitted Dependency

Transaction A	TIME	Transaction B
	T0	Commit
	T1	Fetch V (=10)
	T2	Update V V=V+10(=20)
Fetch V (=20)	T3	
	T4	Rollback
Update V V=V+20 (V=40)	T5	

The effect of the rollback in Transaction B is to undo the update of V at time T2. However, Transaction A has fetched the updated value of V at time T3. If another transaction executed the same code as A after T4 the result would be different.

- 数据不一致

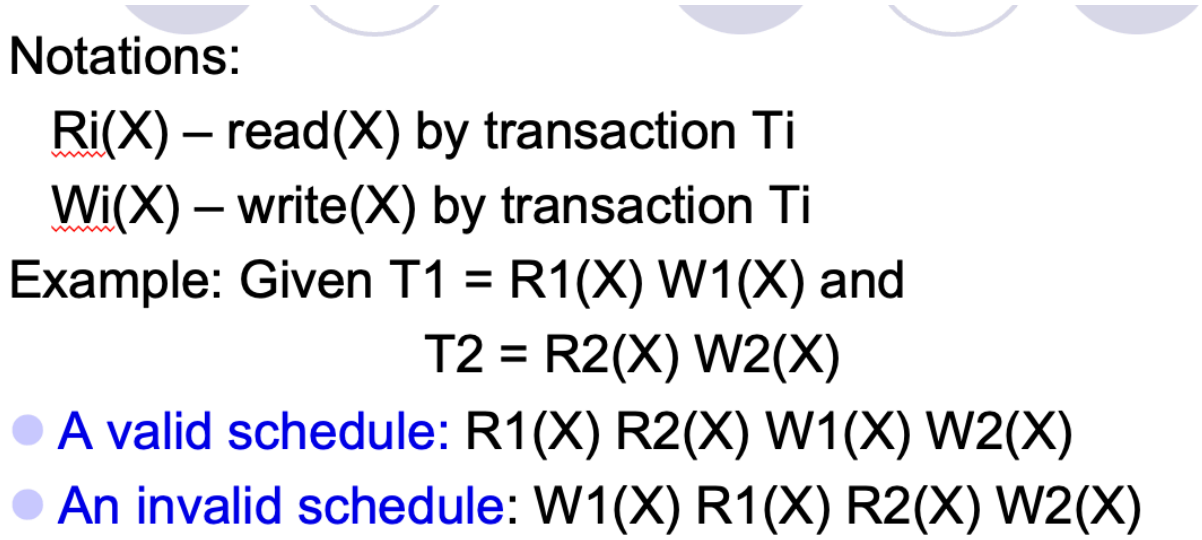
Inconsistent Analysis

Transaction A	TIME	Transaction B 8
Fetch Acc1 (= 40)	T1	
Sum=Sum+Acc1		
Fetch Acc2 (= 50) Sum = Sum+Acc2	T2	
	T3	Fetch Acc3 (= 30)
	T4	Update Acc3 Acc3 = Acc3 -10(= 20)
	T5	Fetch Acc1 (= 40)
	T6	Update Acc1 Acc1 = Acc1 + 10 (= 50)
	T7	Commit
Fetch Acc3 (= 20) Sum=Sum+Acc3	T8	

What is the value of the sum total? Is it “right”?

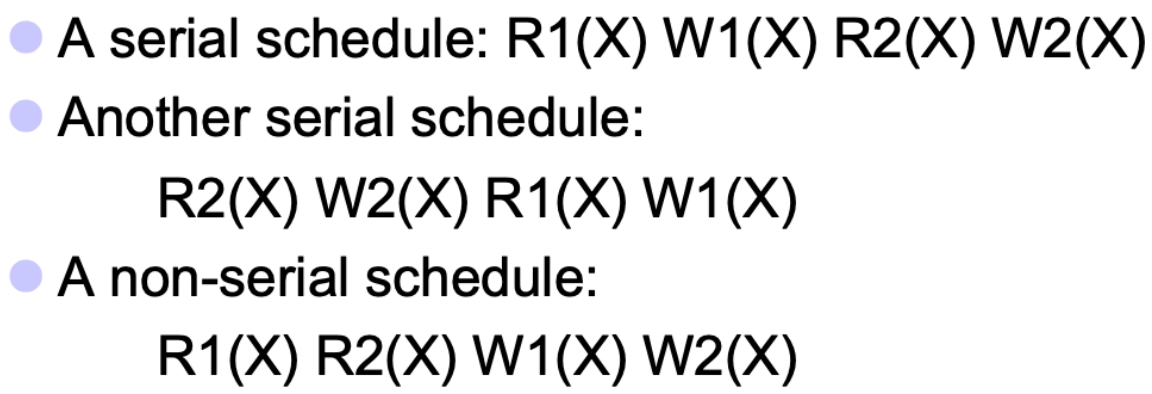
调度

- 可以通过仔细调度并发事务的操作来实现并发控制。
- 定义：n 个事务 T1， T2， ...， Tn 的有效计划 S 是这些事务中所有操作的顺序，但受约束的约束，对于每个事务 Ti，S 中 Ti 的操作必须以与它们在 Ti 中相同的顺序出现。
- 仅考虑有效的时间表。
- 例子



串行化调度

- 定义：如果来自不同事务的操作不是交错的，则调度是串行化调度。
- 例子



- 性质：
  - 1）串行化调度可以保证事务的一致性；
  - 2）只允许串行化调度会导致系统性能羸弱。
  - 3）串行化调度不是事务一致性的必要条件。

    例子：

Example: If X and Y are independent, then the following two schedules always produce the same result:

non-serial schedule: R1(X) W1(X) R2(X) W2(X) R1(Y) W1(Y)

serial schedule: R1(X) W1(X) R1(Y) W1(Y) R2(X) W2(X)

冲突的操作

- 定义：如果满足以下条件，则称调度中的两个操作发生冲突。**条件1**）它们属于不同的事务。**条件2**）他们访问相同的数据项。**条件3**）并且其中之一是写入操作。
- 例子

read-write conflict: (R1(X), W2(X)) or (W1(X), R2(X))

write-write conflict: (W1(X), W2(X))

等价调度

- 定义：对于同一组事务的两个调度，如果任意两个冲突操作在这两个调度中的顺序是相同的，那么这两个调度是（冲突）等价的。
- 例子

Example: Consider two transactions:

T1 = R1(X)W1(X)R1(Y)W1(Y)

T2 = R2(X)W2(X).

The following two schedules are equivalent:

S1: R1(X) W1(X) R2(X) W2(X) R1(Y) W1(Y)

S2: R1(X) W1(X) R1(Y) W1(Y) R2(X) W2(X)

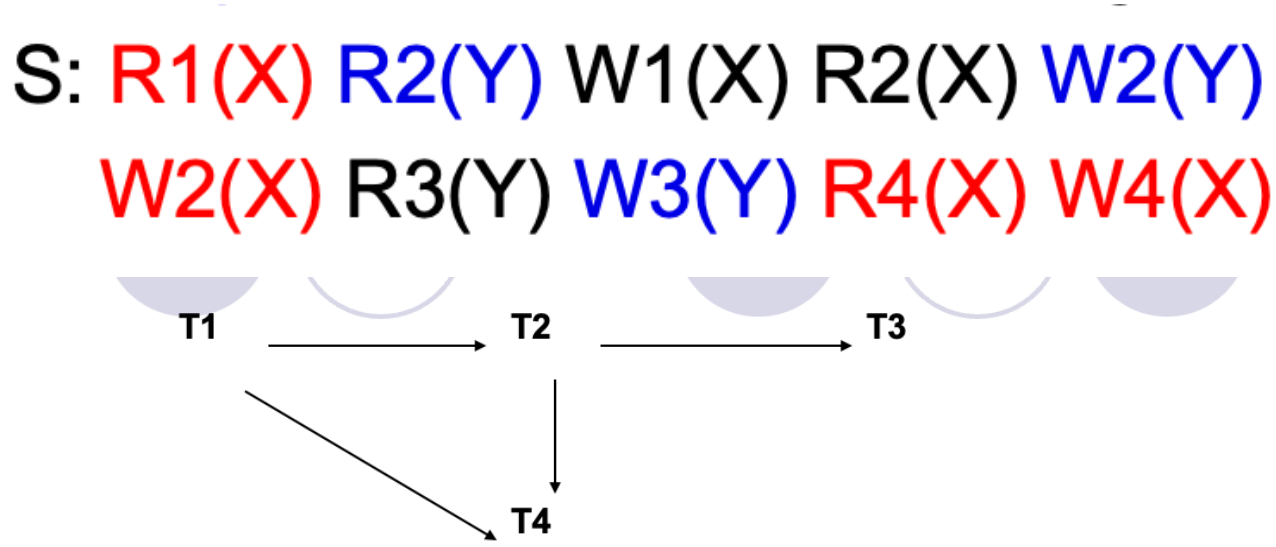
（冲突）可串行化调度

- 定义：如果一个包含n个事务的调度S与这n个事务的至少一个串行调度等价，那么该调度S是可串行化的。
- 定义：在上述定义的基础上，如果通过不断交换两个不冲突的操作，最终得到一个串行化调度，则称为冲突可串行化。
- 上述的S1是可串行化的。

- 性质
  - 可串行化调度保证事务的一致性。
  - 一个非串行但可串行化的调度通常比串行调度允许更高程度的并发。

验证调度是否可串行化（TSS)

- **输入**：一个包含 n 个事务 T1,...,TnT1,...,Tn 的调度 SS
- **输出**：判断 SS 是否可串行化的决策。
- **步骤**：
  1. 创建调度S的前导图（precedence graph）
     - 每个事务Ti都成为图中的一个顶点。
     - 对于每一对冲突操作op1和op2，如果op1(属于事务Ti)出现在op2(属于事务Tj)之前，则在图中从Ti到Tj创建一个有向边（如果该边尚未创建的话）。
  2. 判断前导图是否有环
     - 如果前导图中没有环，则调度S是可串行化的；
     - 如果前导图中有环，则调度S不是可串行化的。
- **结论**：如果调度S的前导图没有环，则S等价于通过对前导图进行拓扑排序所生成的任何串行调度。
- 例子



Two possible orders of topological sorting:
(1) T1 T2 T3 T4
(2) T1 T2 T4 T3
==> S is equivalent to both of the above two serial schedules.

基于锁的并发控制协议

- 锁的类型：read-lock(rli(X))读锁、write-lock(wli(X))写锁和uli(X)释放锁。还有Slock（shared lock）共享锁：允许多个事务同时读取同一数据项，但阻止任何事务对数据项进行写操作。还有Xlock（Exclusive Lock）排他锁：防止其他事务同时对同一数据项进行任何类型的操作（读或写）。
- 事务需要正确的使用锁
  - 只读：使用读锁
  - 只写：使用写锁
  - 读写：使用写锁
  - 例子

    Example: update(X): R(X), X = X+1, W(X)

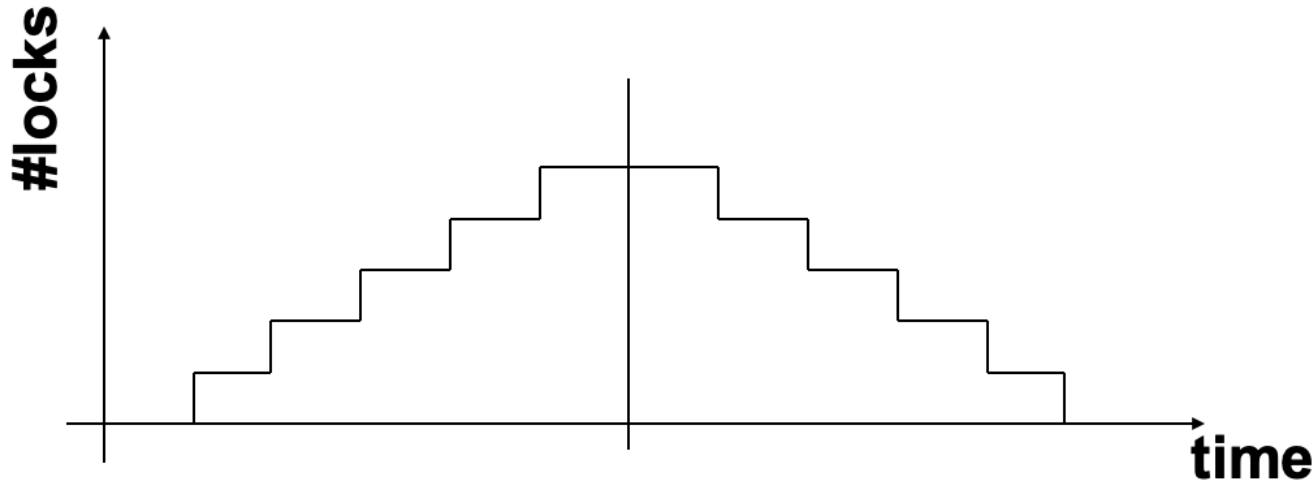
    - Use a write lock rather than a read lock for R(X)
  - 可以对同一数据项设置多个兼容的锁。
  - 不能同时对同一数据项放置不兼容的锁。

    Lock Compatibility table

	rli(X)	wli(X)
rli(X)	Yes	No
wli(X)	No	No

两段锁协议

- 2PL定义：事务释放锁后，它无法请求新的锁。
- 成长期：需要时请求锁定
- 收缩阶段：不需要释放锁



Typical behavior of a single transaction

- 例子

Example:

T1: R(X) X=X+1 W(X) R(Y) Y=Y-1 W(Y)

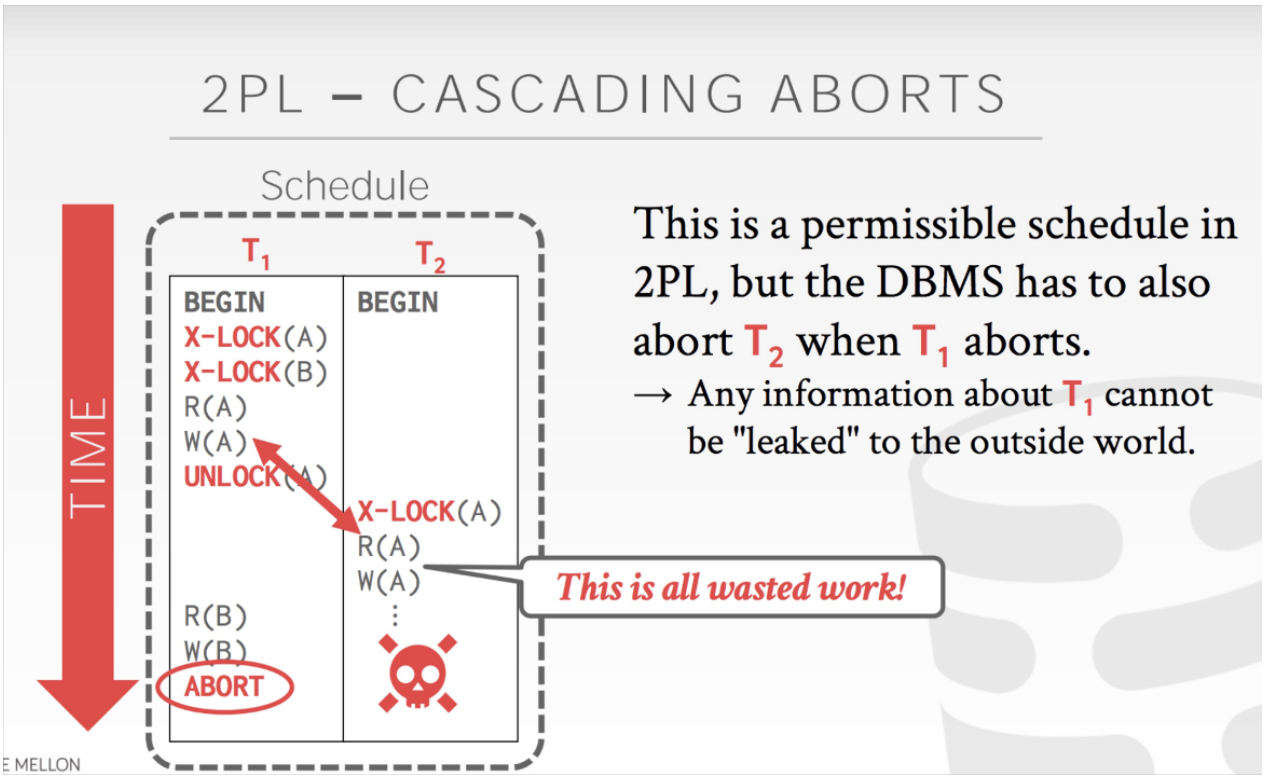
T2: R(X) X=2*X W(X) R(Y) Y=2*Y W(Y)

Schedule S wl1(X)直接获得, R1(X), W1(X), wl1(Y), ul1(X), t2到这个时候才获得x的锁wl2(X), R2(X), R1(Y), W2(X), W1(Y), ul1(Y), wl2(Y), ul2(X)获得最后一个lock后，t2马上释放不需要的锁, R2(Y), W2(Y), ul2(Y)

- S satisfies the 2PL protocol.

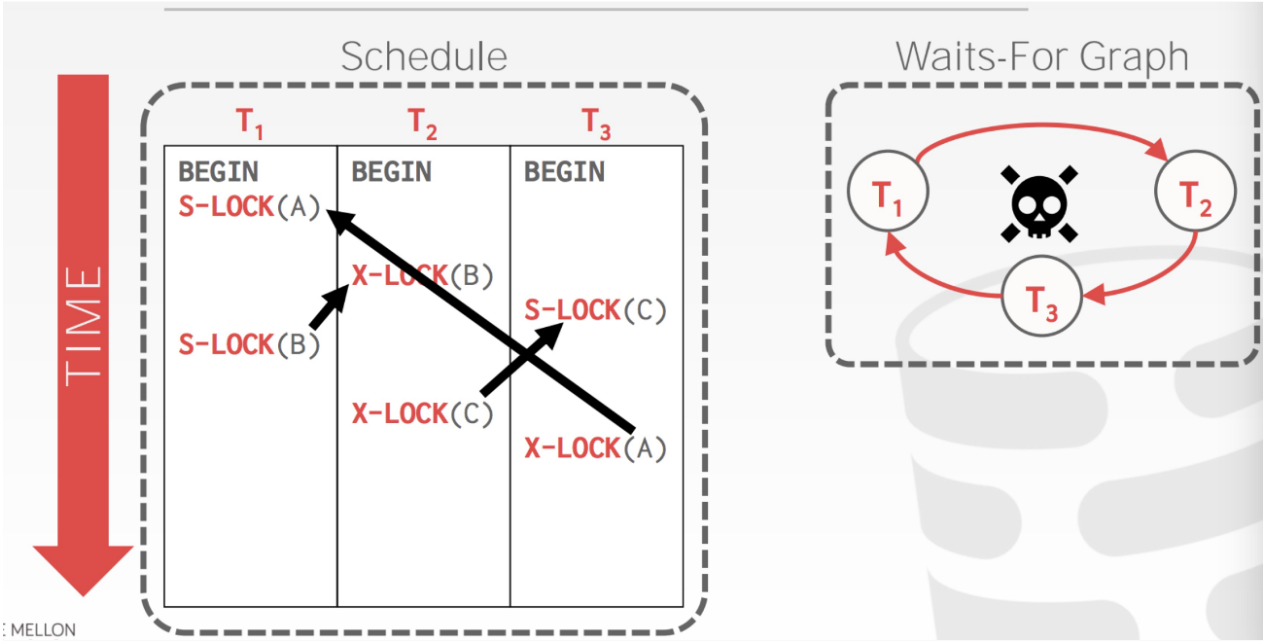
事务T_1	事务T_2
Slock A	
R(A)=260	
	Slock C
	R(C)=300
Xlock A	
W(A)=160	
	Xlock C
	W(C)=250
	Slock A
Slock B	等待
R(B)=1000	等待
Xlock B	等待
W(B)=1100	等待
Unlock A	等待
	R(A)=160
	Xlock A
Unlock B	
	W(A)=210
	Unlock C

- 对于事务的集合，不只有一种调度可以满足两段锁协议。因此，我们可以添加如下要求保证调度唯一。
  - **尽可能采用先到先服务策略**：在满足两阶段锁协议（2PL）规则的前提下，优先调度先到的请求。
  - **尽早释放锁**：当事务不再需要某个锁时，应在两阶段锁协议允许的情况下尽快释放该锁。
- 定理：任何遵循两段锁协议的调度都是可串行的。
- 两段锁协议存在的问题：
  - **可能导致级联回滚**：如果一个事务回滚，其释放的锁可能导致其他依赖该事务的事务也必须回滚，从而引发一连串的回滚操作。



- **不能防止死锁**：多个事务之间可能因为互相等待彼此持有的锁而产生死锁。2PL 本身没有内置的死锁检测或预防机制。

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



死锁

- **系统进入死锁状态的条件**：如果存在一组事务，其中每个事务都在等待该组中另一个事务释放资源，则系统处于死锁状态。
- **死锁预防协议**：死锁预防协议确保系统永远不会进入死锁状态。
- **一些预防策略**：
  1. **预声明策略**：要求每个事务在执行之前锁定其所有需要的数据项（预声明所有资源）。
  2. **基于图的协议**：对所有数据项施加一个部分顺序（partial ordering），并要求事务只能按照该顺序锁定数据项。
- 以下方案仅使用事务时间戳来实现死锁预防：
  1. **等待-死亡（wait-die）方案** — 非抢占式
     - 年长事务（时间戳较小）可以等待年轻事务（时间戳较大）释放数据项。
     - 年轻事务绝不会等待年长事务，而是被回滚（死亡）。
     - 一个事务在获取所需数据项之前，可能会多次被回滚。

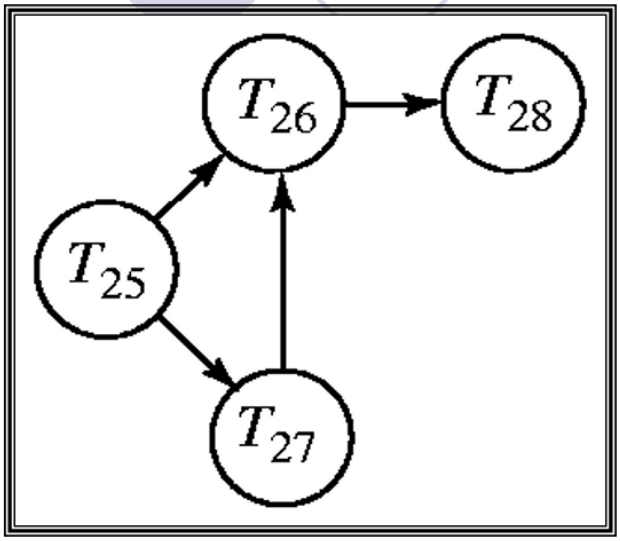
1. **伤害-等待（wound-wait）方案** — 抢占式
   - 年长事务（时间戳较小）会“伤害”年轻事务（时间戳较大），强制其回滚，而不是等待。
     - 年轻事务可以等待年长事务释放资源。
     - 相比等待-死亡方案，回滚次数可能较少。

- 在**等待-死亡（wait-die）**和**伤害-等待（wound-wait）**方案中，被回滚的事务会以其原始时间戳重新启动。
- 因此，较老的事务相对于较新的事务具有优先权，从而避免了**饥饿问题**。
- **基于超时的方案**：
  - 事务只会在指定的时间内等待锁。如果超过该时间，等待超时，事务将被回滚。
  - 因此，不可能发生死锁。
  - 实现简单，但可能导致**饥饿问题**（某些事务可能因反复超时而无法完成）。

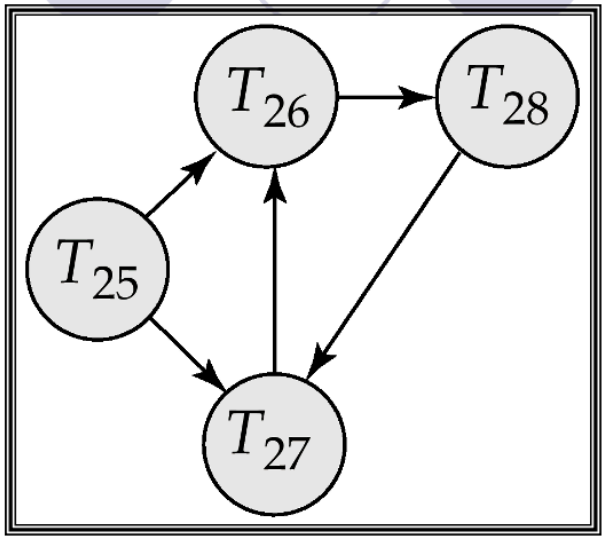
- 同时，确定合理的超时时间间隔值也具有一定难度。

死锁探测

- 死锁可以通过一个等待图（wait-for graph）来描述，该图由一对$(G = (V, E))$组成：
  - V是顶点的集合，表示系统中的所有事务。
  - E是边的集合，每个元素是一个有序对 ($T_i \rightarrow T_j$)。
  - 如果 ($T_i \rightarrow T_j$) 在E中，则表示存在一条从T_i到T_j的有向边，意味着T_i正在等待T_j释放某个数据项。
  - 当事务 T_i请求一个当前由事务T_j持有的数据项时，边$T_i \rightarrow T_j$会被插入到等待图中。只有当T_j不再持有T_i需要的数据项时，这条边才会被移除。
- 系统处于死锁状态当且仅当等待图中存在一个环。因此，必须定期调用死锁检测算法来查找图中的环。
- 例子



Wait-for graph without a cycle



Wait-for graph with a cycle

死锁恢复

- 当检测到死锁时：
  - 必须回滚某些事务（将其作为受害者）以打破死锁。
  - 选择作为受害者的事务应为将产生最小成本的事务。
- **回滚** — 确定需要回滚的事务的范围：
  - **完全回滚**：终止事务并重新启动它。
  - 更有效的做法是仅回滚事务到打破死锁所必需的程度。
- **饥饿问题**发生在同一个事务总是被选择作为受害者时，为避免饥饿问题，应将**回滚次数**包含在成本因素中。