



事务管理(并发控制技术2)

单 位：重庆大学计算机学院

主要学习目标

- 基于图的协议
- 死锁
- 多粒度封锁协议

基于图的协议*

- 基本思路:

若要开发“非两阶段封锁的、但要求保证冲突可串行化的”协议，
则一般需要每个事务如何存取数据库的附加信息。

- 可以开发各种不同模型，一类最简单的模型是偏序:

要求所有数据项集合D满足一种偏序 ‘ \rightarrow ’，即

$$\mathbf{D} = \{d_1, d_2, \dots, d_h\} \quad \text{对任何 } i, \text{ 都有 } d_i \rightarrow d_{i+1}$$

- 这种偏序的本质含意是要求:

如果 $d_i \rightarrow d_j$ ，那么任何访问 d_i 和 d_j 的事务，都必须保证 d_i 先于 d_j 被访问。

(即: 所有事务对D中数据项的访问，都必须遵从偏序约束)

- 直观示例:

想象一群人在一个餐厅享用自助餐

如要求统一从左至右方向有序取食，则不会发生争抢混乱。

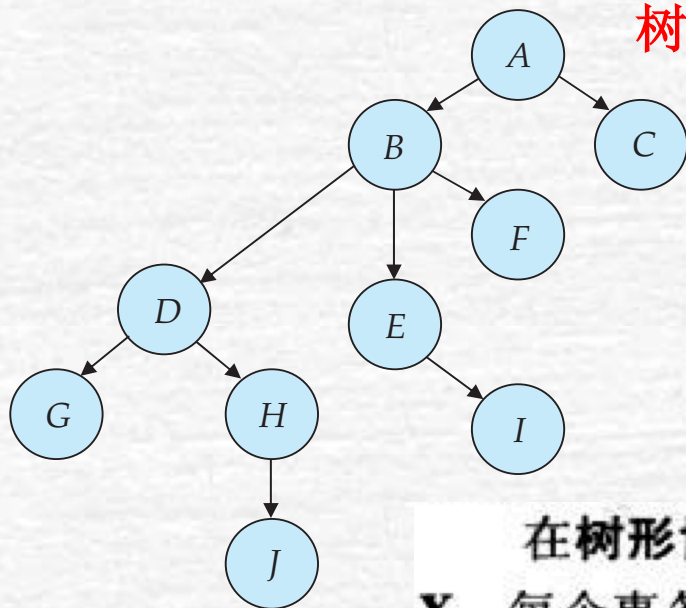
- 偏序意味着集合D可以视为是一个有向无环图.

树形协议

树形协议就是一种简单的基于图的协议。

树形协议是一种特殊的偏序关系：

所有数据项按照从上到下的父子关系，排出一种偏序(所有事务需按此偏序访问数据项)



自助餐的勺子

在树形协议 (tree protocol) 中，可用的加锁指令只有 **lock-X**。每个事务 T_i 对一数据项最多能加一次锁，并且必须遵从以下规则：

1. T_i 首次加锁可以对任何数据项进行。
2. 此后， T_i 对数据项 Q 加锁的前提是 T_i 当前持有 Q 的父项上的锁。
3. 对数据项解锁可以随时进行。
4. 数据项被 T_i 加锁并解锁后， T_i 不能再对该数据项加锁。

树形协议的优点

这4个事务参与的一个可能的调度如图 15-12 所示。注意，在执行时，事务 T_{10} 持有两棵互相含的子树的锁。

问：该调度是冲突可串行化调度吗？

T_{10} : lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G);
unlock(D); unlock(G).

T_{11} : lock-X(D); lock-X(H); unlock(D); unlock(H).

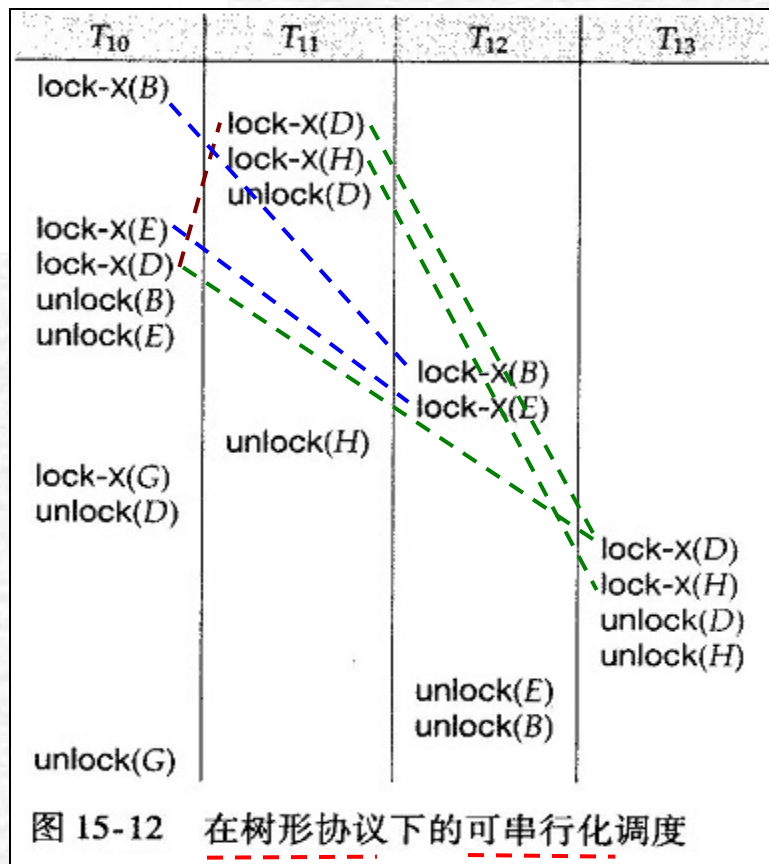
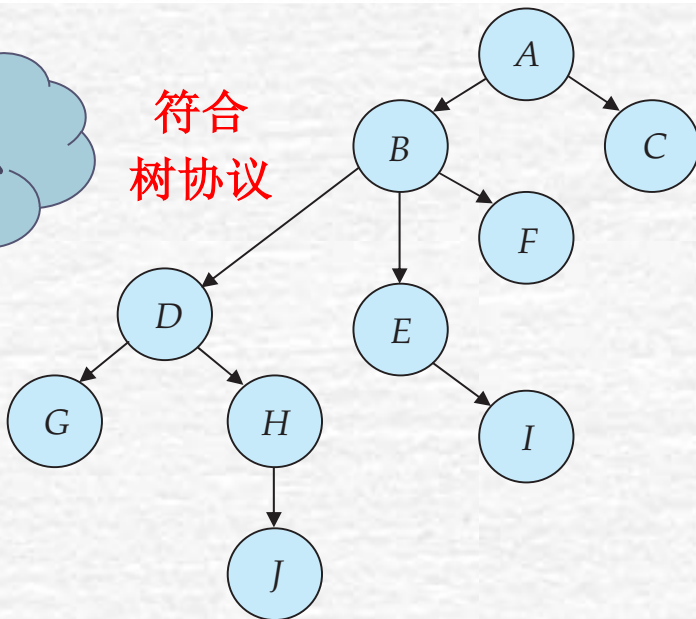
T_{12} : lock-X(B); lock-X(E); unlock(E); unlock(B).

T_{13} : lock-X(D); lock-X(H); unlock(D); unlock(H).

图15-12中的
调度符合树协
议吗？

符合
树协议

(一般地)树
形协议调度
都是可串行
化的调度！

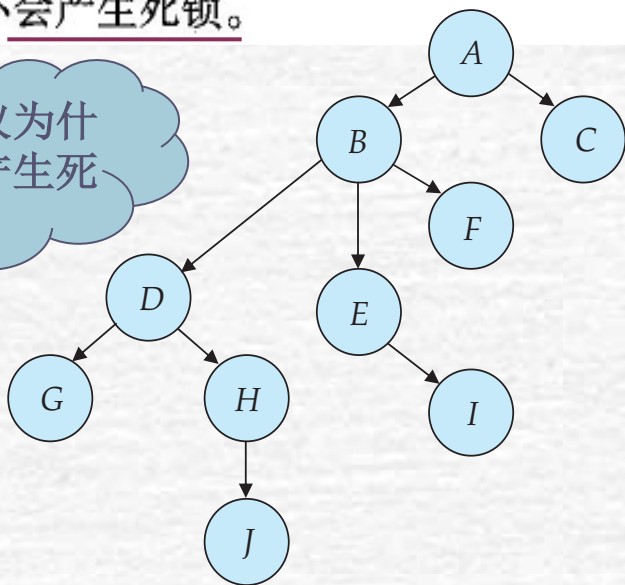


树形协议的优点（续）

这4个事务参与的一个可能的调度如图 15-12 所示。注意，在执行时，事务 T_{10} 持有两棵互相包含的子树的锁。

不难发现如图 15-12 所示的调度是冲突可串行化的。可以证明树形协议不仅保证冲突可串行化而且保证不会产生死锁。

树形协议为什么不会产生死锁？



T3	T4	T1	T2	
Lock-X (E)	成功	Lock-X (B)	成功	
	Lock-X (D) 成功		Lock-X (B)	等待
Lock-X (D)	等待		Lock-X (D)	
	Lock-X (E) 等待	Lock-X (E)		成功
Unlock (E)		Lock-X (D)		...
Unlock (D)			Lock-X (E)	
	Unlock (D)	Unlock (B)		
	Unlock (E)	Unlock (E)		
		Unlock (D)		
			Unlock (B)	
			Unlock (D)	
			Unlock (E)	

调度P: 存在死锁现象, 它符合树形协议吗?

(出现死锁, 但T3不符合第2条, 因E的父节点B未加锁) (这正是协议要先锁父节点的原因)

调度Q: 符合树形协议, 它存在死锁现象吗?

符合, 且不存在死锁!

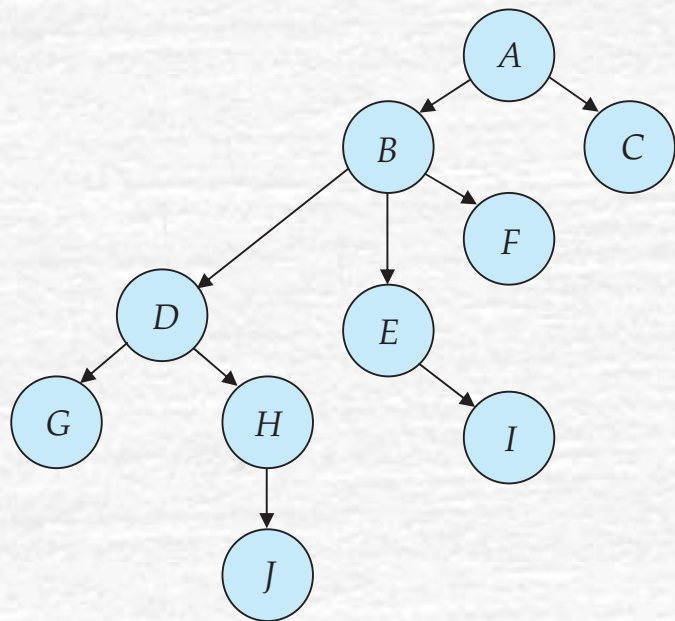
Q不可能出现“T1锁住E后又请求锁D, 而T2锁住D后又请求锁E”的死锁情形! 因为: T1锁住E前必先锁住父节点B, 这时T2不能申请到D锁, 因它之前申请不到B锁(等待T1释放)(树形协议只有X锁, 如有S锁则会出现死锁现象)

树形协议的缺点

树形协议能保证可恢复性吗？

只需将树形协议修改为：在事务结束前不允许释放任何X锁！
(牺牲一定的并发度)

要保证可恢复性应如何修改协议？



Write (D)

树形协议不能保证可恢复性！
(T13虽读了T10的数据D，
但T13已提交不能再回滚)

T ₁₀	T ₁₁	T ₁₂	T ₁₃
lock-X(B)	lock-X(D) lock-X(H) unlock(D)		
lock-X(E) lock-X(D) unlock(B) unlock(E)			
	unlock(H)	lock-X(B) lock-X(E)	
lock-X(G) unlock(D)			lock-X(D) lock-X(H) unlock(D) unlock(H) commit
unlock(G) abort		unlock(E) unlock(B)	

Read (D)

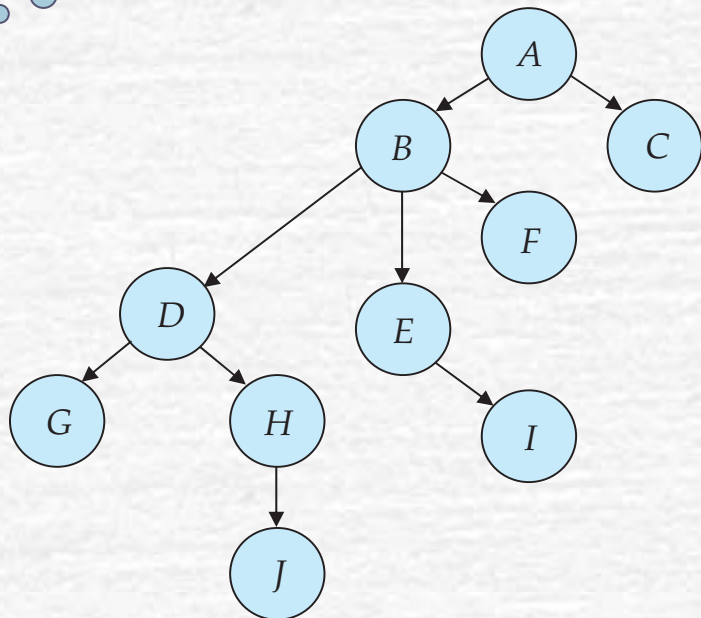
图 15-12 在树形协议下的可串行化调度

树形协议的缺点（续）

树形协议能保证无级联卷回吗？

仍需将树形协议修改为：在事务结束前不允许释放任何X锁！
(牺牲一定的并发度)

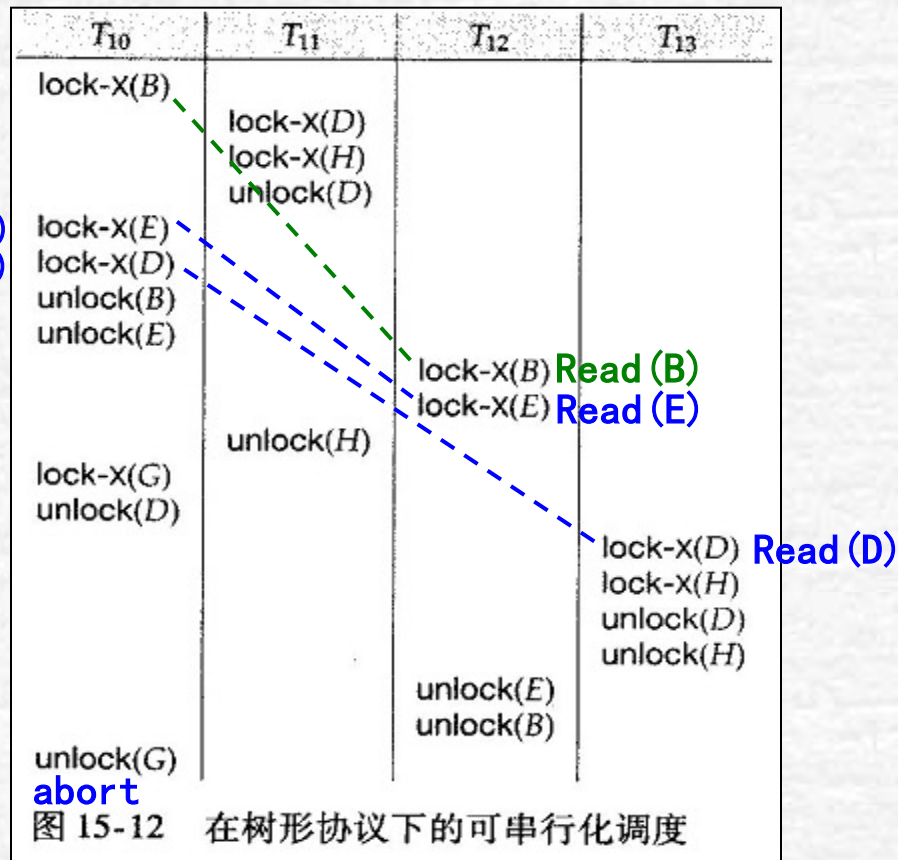
要保证无级联回卷应如何修改协议？



树形协议不能保证无级联卷回！
(T12和T13都读了T10的数据，
T10卷回时T12和T13也卷回)

Read (B)

Write (E)
Write (D)



树形协议小结

- 优点

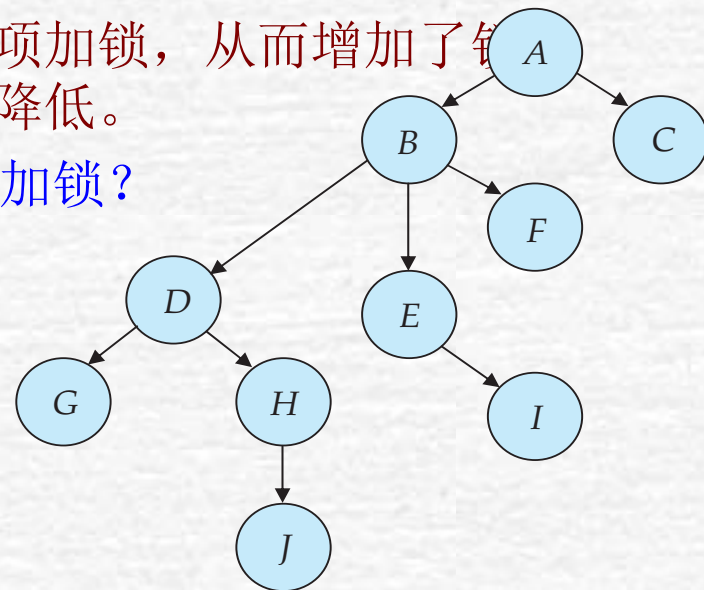
- 树形协议保证了冲突可串行化；
- 同时不会产生死锁，不需要回滚；
- 树形协议可较早地释放锁，以减少事务间的等待时间，从而可增强调度的并发性。

- 缺点

- 树形协议不能保证事务的可恢复性；
- 事务不能保证不发生级联回滚；
- 事务有可能会给那些根本不访问的数据项加锁，从而增加了锁的开销和额外的等待时间，引起并发性降低。

比如：一事务要处理A和J数据项，如何加锁？

不仅要求给数据项A和J加锁，
还必须给数据项B、D和H加锁！



多粒度封锁协议

通俗示例：教学大楼使用的管理(如仅一种锁:大楼锁/教室锁)

- 1)某单位租用整栋大楼用于培训,需要上大量教室锁? **太繁琐**
- 2)若仅租用部分教室而上大楼锁,剩下的没法自习? **共享差**

- 封锁的对象的大小称为**封锁粒度** (Granularity) 。
- 封锁的对象可以是逻辑单元,也可以是物理单元。
- 以关系数据库为例,封锁对象可以是这样一些逻辑单元:
属性值、属性值的集合、元组、关系、索引项、整个索引
直至整个数据库;也可以是这样一些物理单元:页(数据页
或索引页)、块等。

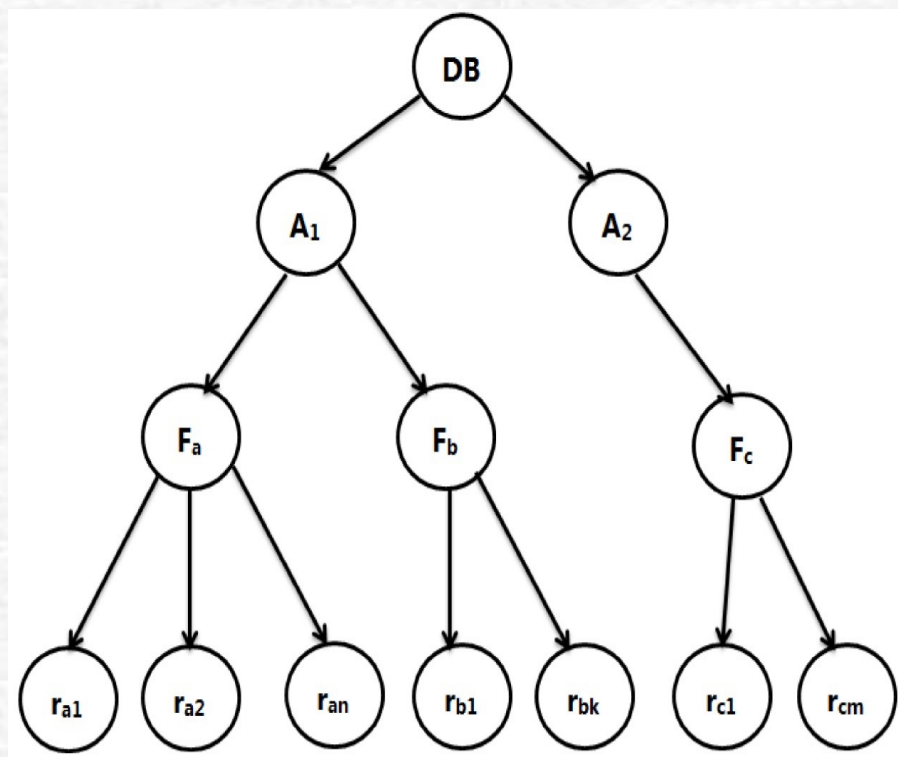
- 封锁粒度与系统的并发度和并发控制的开销密切相关。
- 直观地看，封锁的粒度越大，数据库所能够封锁的数据单元就越少，并发度就越小，系统开销也越小；
- 反之，封锁的粒度越小，并发度较高，但系统开销也就越大。

- 在一个系统中同时支持多种封锁粒度供不同的事务选择是比较理想的，这种封锁方法称为**多粒度封锁**（multiple granularity locking）。
- 选择封锁粒度时应该同时考虑封锁开销和并发度两个因素，适当选择封锁粒度以求得最优的效果。

多粒度锁：允许使用多种粒度/大小不同的锁
(不同粒度：楼，层，教室，座位)

一、基本概念

- 多粒度封锁协议的主要内容包括：
 - 允许多粒度层次图中的每一个结点被独立地加锁。
 - 对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁。



- 一个数据对象可以以两种方式被封锁：
 - 显示封锁：直接加到数据对象上。
 - 隐式封锁：该数据对象没有显示地被封锁，是由于其上级结点加锁而使得该数据对象加了锁。
- 但无论一个数据对象是以哪一种方式被加锁，其效果是一样的。

- 若事务 T_i 要封锁文件 F_a 中的某记录 r_a ，由于 T_i 显式的给 F_a 加锁，意味着 r_a 被隐式加锁。那么，当事务 T_j 申请对 r_a 加锁时，由于 r_a 上没有显式锁，所以系统必须从根结点到 r_a 进行搜索，如果发现路径上有某个结点持有不相容的锁，则 T_j 等待。
- 若事务 T_i 希望封锁整个数据库，只需要对根结点加锁。但是由于需要判断树中其他结点是否持有不相容锁，因此需要搜索整个树，效率就非常低！
- 因此，引进了一种新型锁——意向锁
 - 如果一个结点加上意向锁，意味着要在树的较低层进行显式加锁。在一个结点显式加锁前，该结点的全部祖先结点均加上意向锁。
 - 因此，事务不必搜索整棵树就能判定能否成功对一个结点加锁。

意向锁

IS锁

如果对一个数据对象加IS锁，表示它的后裔结点拟（意向）加 S锁。例如，要对某个元组加 S锁，则要首先对关系和数据库加 IS锁。

IX锁

如果对一个数据对象加 IX锁，表示它的后裔结点拟（意向）加 X锁。例如，要对某个元组加 X锁，则要首先对关系和数据库加 IX锁。

SIX锁

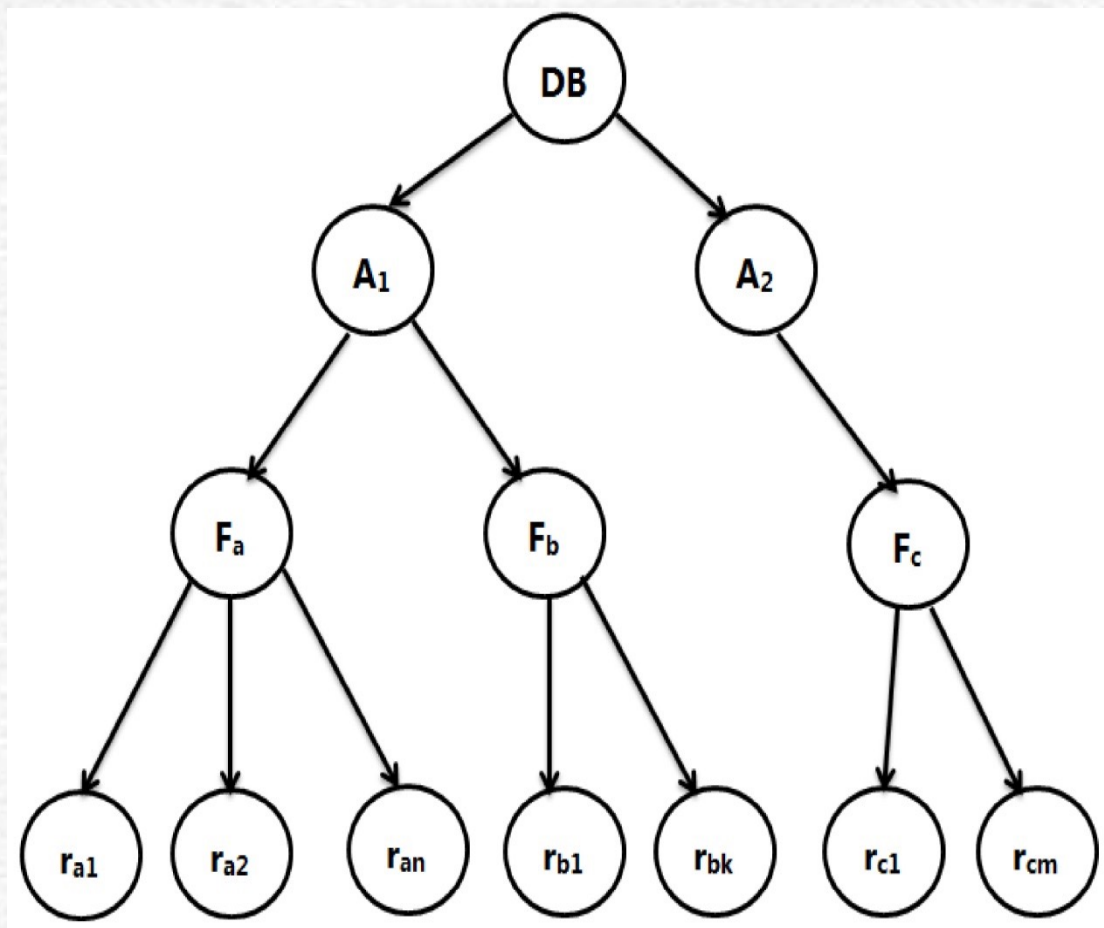
如果对一个数据对象加 SIX锁，表示对它加 S锁，再加IX锁，即 $SIX=S+IX$ 。例如对某个表加 SIX锁，则表示该事务要读整个表（所以要对该表加 S锁），同时会更新个别元组（所以要对该表加 IX锁）。

多粒度封锁协议要求的规则

- 事务 T_i 必须遵守锁类型相容矩阵。
- 事务 T_i 必须首先封锁树的根节点，并且可以加任意类型的锁。
- 仅当 T_i 当前对 Q 的父结点具有IX或IS锁时， T_i 对节点 Q 可加S或IS锁。
- 仅当 T_i 当前对 Q 的父结点具有IX或SIX锁时， T_i 对节点 Q 可以加X、SIX或IX锁。
- 仅当 T_i 未曾对任何结点解锁时， T_i 可对节点加锁（也就是说， T_i 是两阶段的）。
- 仅当 T_i 当前不持有 Q 的子结点的锁时， T_i 可对节点 Q 解锁。

- 多粒度协议要求加锁按自顶向下的顺序，而锁的释放则按自底向上的顺序。

- 粒度层次图

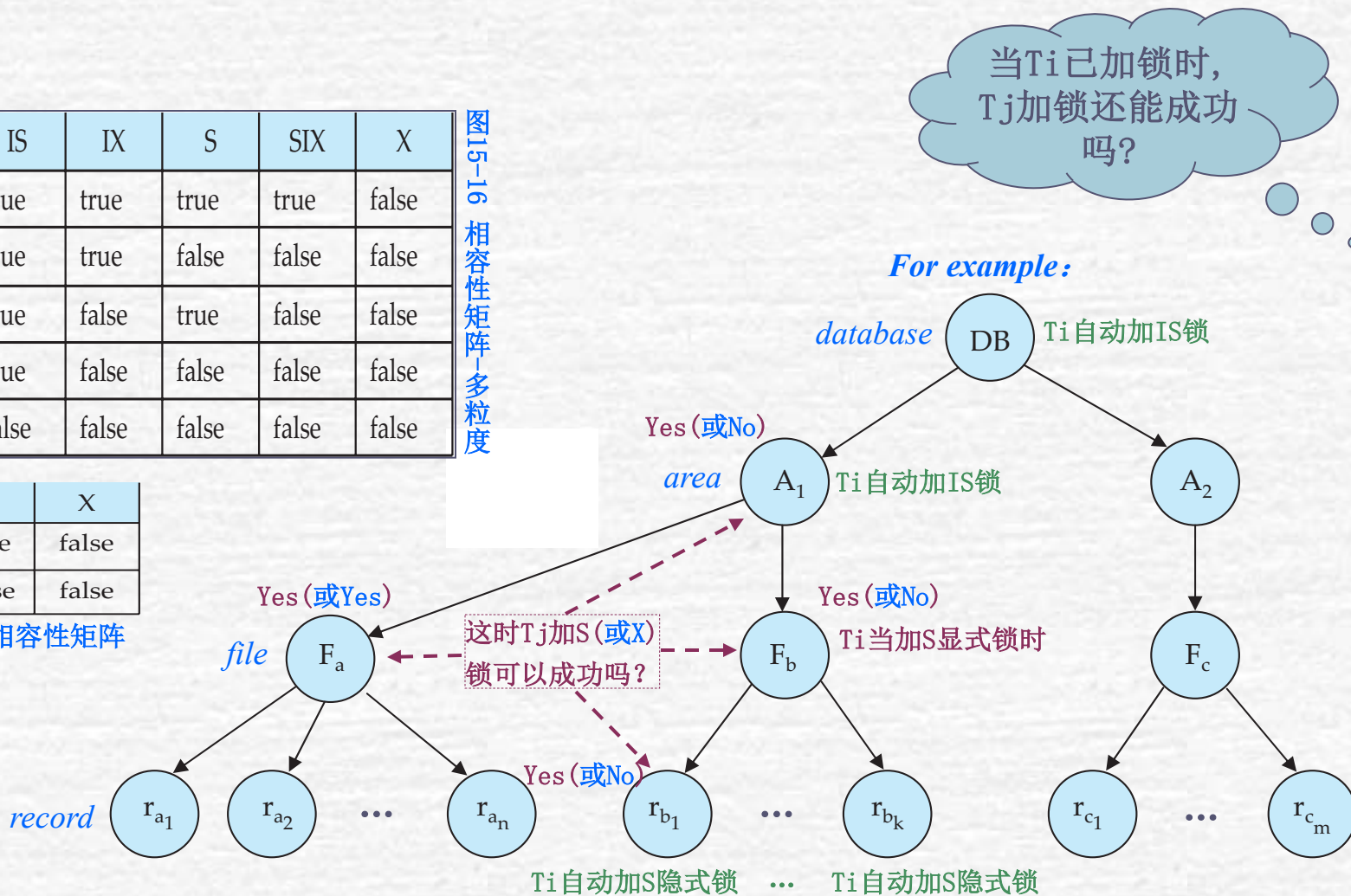


	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

图15-16 相容性矩阵—多粒度

	S	X
S	true	false
X	false	false

单粒度-相容性矩阵



- 假设**事务T1**读文件Fa的记录ra1，那么T1需要给数据库、区域A1，以及Fa加IS锁，最后给ra1加S锁。
- 假设**事务T2**要修改文件Fa的记录ra2，那么，T2需要给数据库、区域A1，以及Fa加IX锁，最后给ra2加X锁。
- 假设**事务T3**要读取文件Fa的所有记录，那么，T3需要给数据库、区域A1加IS锁，最后给Fa加S锁。
- 假设**事务T4**要读取整个数据库。它在给数据库加S锁后就可读取。
- T1、T3与T4可以并发的存取数据库。
- 事务T2可以与T1并发执行，但不能与T3或T4并发执行。
- 多粒度封锁协议可以保证可串行性，增强了并发性，减少了锁开销，可能存在死锁。

死锁的处理


➤ 如何解决死锁问题？

- 死锁预防：预先防止死锁发生，保证系统永不进入死锁状态；
- 死锁检测与恢复：允许系统进入死锁状态，但要周期性地检测系统有无死锁。如果有，则把系统从死锁中恢复过来。

两种策略都会引起事务回滚。如果系统进入死锁状态的概率相对较高，则通常采用死锁预防策略；否则使用死锁检测与恢复更有效。



➤ 预防死锁有两种方法：

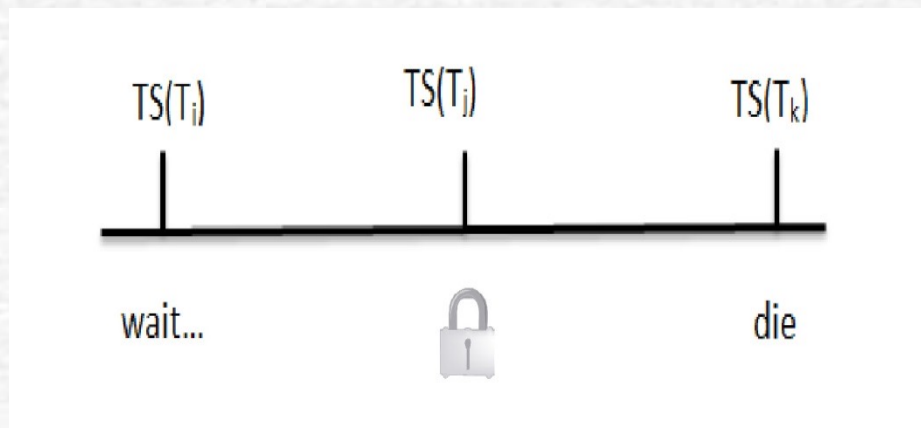
- 一种方法是通过对加锁请求进行排序或者要求同时获得所有的锁来保证不会发生循环等待。
 - 另一种方法比较接近死锁恢复，每当等待有可能导致死锁时，进行事务回滚而不是等待加锁。
- 

- 第一种方法下最简单的机制要求每个事务在开始之前封锁它的所有数据项。此外，要么一次全部封锁，要么全不封锁。
- 这个协议有两个主要的缺点：
 - 在事务开始前通常很难预知哪些数据项需要封锁。
 - 数据项使用率可能很低，因为许多数据项可能封锁很长时间却用不到。

- 预防死锁的第二种方法，也是最有效的方法就是采用抢占与事务回滚技术。
- 在这种方法里，赋予每个事务一个唯一的时间戳，系统利用时间戳来决定事务应当等待还是回滚。但并发控制仍使用封锁机制。
- 若一个事务回滚，则该事务重启时保持原有的时间戳。
- 利用时间戳的两种不同的死锁预防机制如下：
 - 等待-死亡(wait-die)机制：基于非抢占技术
 - 受伤-等待(wound-wait)机制：基于抢占技术。

等待-死亡机制

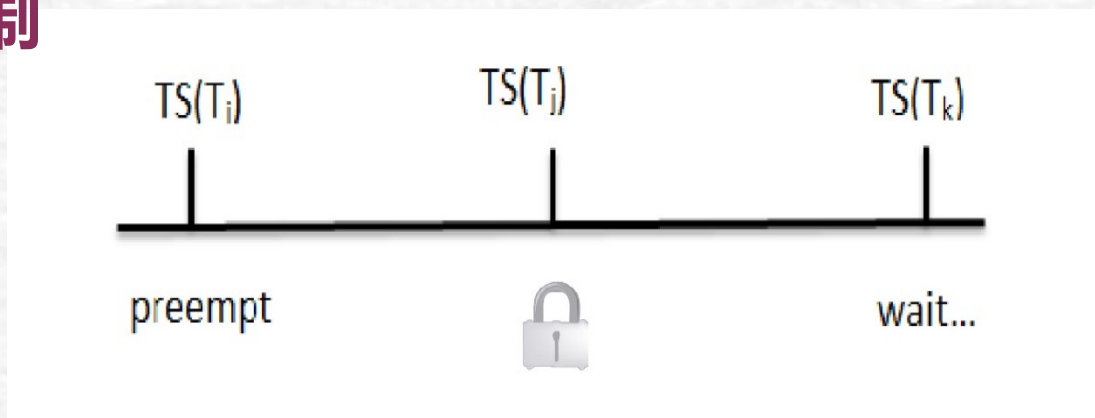
- 这种机制基于非抢占技术。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 $TS(T_i) < TS(T_j)$ 时，允许 T_i 等待。否则， T_i 回滚。
- 等待-死亡机制



受伤-等待机制

- 这种机制基于抢占技术，是wait-die的相反机制。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 $TS(T_i) > TS(T_j)$ 时，允许 T_i 等待。否则， T_i 抢占 T_j 持有的数据项，而 T_j 回滚。

- 受伤-等待机制



等待-死亡与受伤-等待的区别


- **等待区别：**在等待-死亡机制中，较老的事务必须等待较新的事务释放它所持有的数据项。因此，事务变得越老，它越要等待。与此相反，在受伤-等待机制中，较老的事务从不等待较新的事务；

等待-死亡与受伤-等待的区别

- **回滚区别：**在等待-死亡机制中，如果事务Tk由于申请的数据项当前被Tj持有而死亡并回滚，则当事务Tk重启时它可能重新发出相同的申请序列。如果该数据项仍被Tj持有，则Tk将再度死亡。因此，Tk在获得所需数据项之前可能死亡多次！而在受伤-等待机制中，如果Ti申请的数据项当前被Tj持有而引起Tj受伤并回滚，则当Tj重启并发出相同的申请序列时，Tj会等待而不是回滚！

- 如果系统没有采用能保证不产生死锁的协议，如何处理死锁？
- 系统必须采用检测与恢复机制。检测系统状态的算法周期性的激活，判断有无死锁发生。如果发生死锁，则系统必须试着从死锁中恢复。

- **死锁检测**（deadlock detection）即探查和识别死锁的方法。这种策略并不采取任何动作来使死锁不出现，而是系统事件触发执行一个检测算法。
- 也即在系统运行过程中，及时地探查和识别死锁的存在，并识别出处于死锁之中的进程和资源等。



➤ **死锁恢复** (deadlock recovery) 是指当检测并识别出系统中出现处于死锁之中的一组进程时，如何使系统恢复到正常状态并继续执行下去。

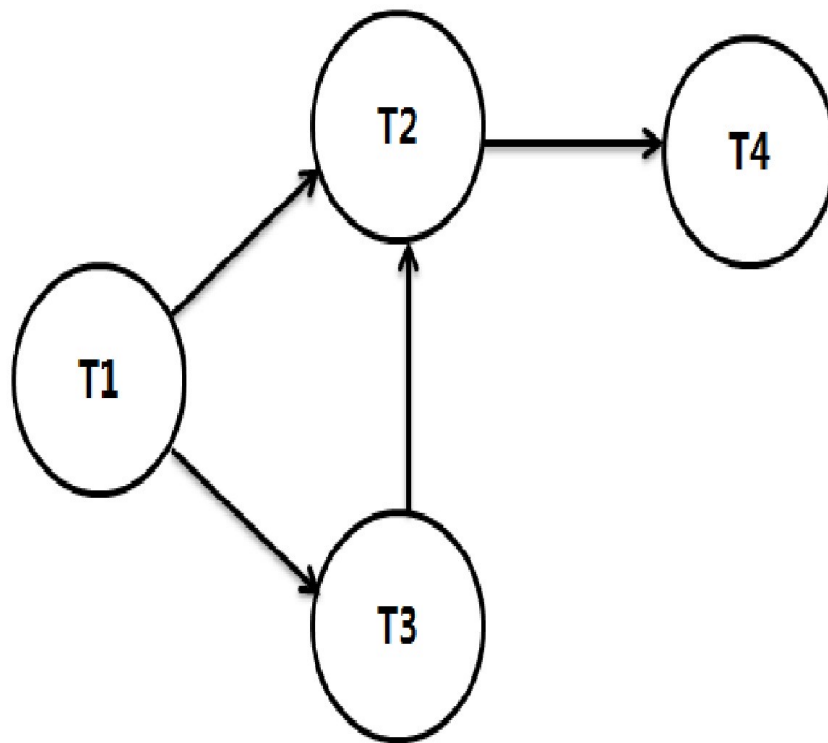
死锁检测算法

- 死锁可以用称为等待图的有向图来描述。
- 该图由两部分 $G=(V,E)$ 组成，其中 V 是顶点集， E 是边集。顶点集由系统中的所有事务组成；如果事务 T_i 在等待 T_j 释放所需数据项，则存在从 T_i 到 T_j 的一条有向边 $T_i \rightarrow T_j$ ，表示事务 T_i 在等待 T_j 释放所需数据项。
- 死锁检测算法就是要检查等待图中是否存在有向环，当且仅当等待图包含环时，系统中存在死锁。

事务T1在等待事务T2与T3。

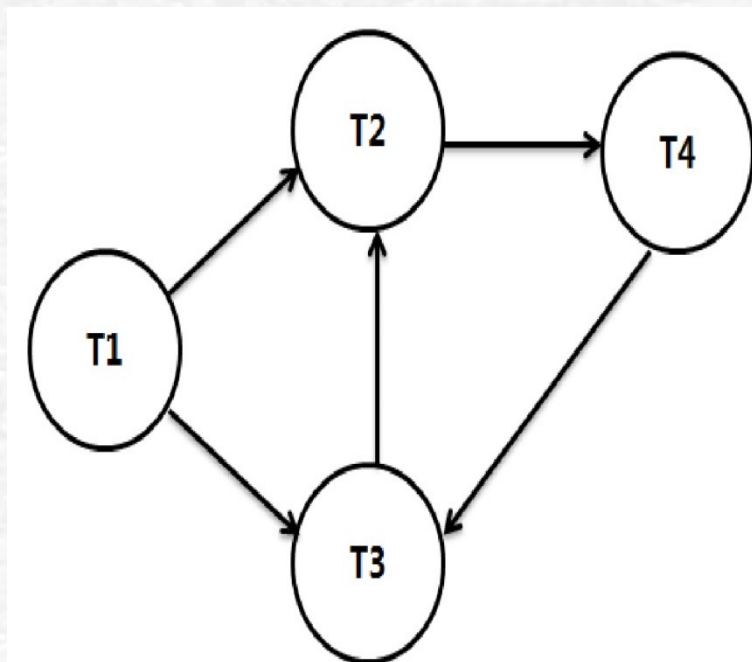
事务T3在等待事务T2。

事务T2在等待事务T4。



➤ 由于该等待图无环，因此系统没有处于死锁状态。

- 现在假设事务T4申请事务T3持有的数据项，则需要在等待图中加入 $T4 \rightarrow T3$ ，得到如图所示的新状态。



- 该等待图包含环： $T2 \rightarrow T4 \rightarrow T3 \rightarrow T2$ ，意味着事务T2、T3、T4都处于死锁状态。

死锁的恢复

- 当一个检测算法判定存在死锁时，系统必须从死锁中恢复。
- 解除死锁最通常的做法是回滚一个或多个事务。

- 主要考虑一下三个方面：
- 选择牺牲者：给定处于死锁状态的事务集，为了解除死锁，我们必须决定回滚哪一个事务来打破死锁。我们应该选择使回滚代价最小的事务作为牺牲者，例如：
 - 该事务已计算了多久？
 - 该事务已使用了多少数据项？
 - 完成该事务还需要多少数据项？
 - 回滚该事务将牵涉多少事务？

- **回滚：决定回滚多远：是彻底回滚，即中止该事务而后重启；还是部分回滚，即只回滚到可以解除死锁为止；**
- **避免饿死：避免同一事务总是作为回滚代价最小的事务而被选中。最常用的方法就是在代价因素中包含回滚次数。**



课堂小测试

- 死锁是怎么产生的？应该如何预防？

课程总结与作业安排

- 基本知识：
 - 树形协议
 - 死锁的检测、解除与预防
- 扩展学习：
 - 如何设计一个死锁检测算法？
- 作业
无