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

- 死锁
- 多粒度锁

前测小问题

- 使用锁一定会保证事务顺利的并发执行吗？执行会成功吗？

一 基于图的协议*

1.1 概述

- 基本思路：

  分析自助餐的管理

  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‘□’，即

  $\mathbf{D} = \{d1, d2, ..., dh\}$ 对任何i，都有 $di \square di+1$

  基于图的协议的基本思路是什么?

- 这种偏序的本质含意是要求：

  如果 $di \square dj$，那么任何访问di 和dj的事务，都必须保证di 先于 dj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：

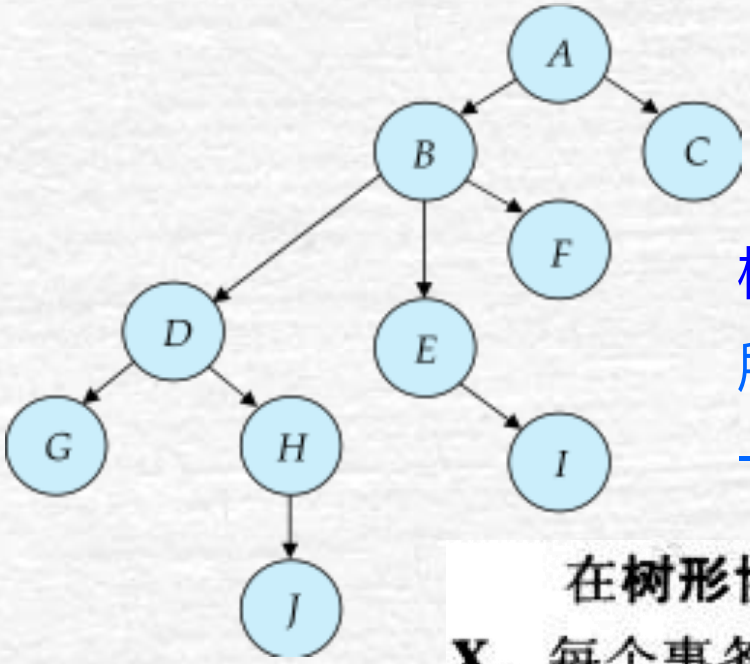
  想象一群人在一个餐厅享用自助餐

  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.

  比如，下面的树协议就是一种简单的基于图的协议.

1.2 树形协议



如何理解树协议?

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。

1.3 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} 持有两棵互相分的子树的锁。

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).

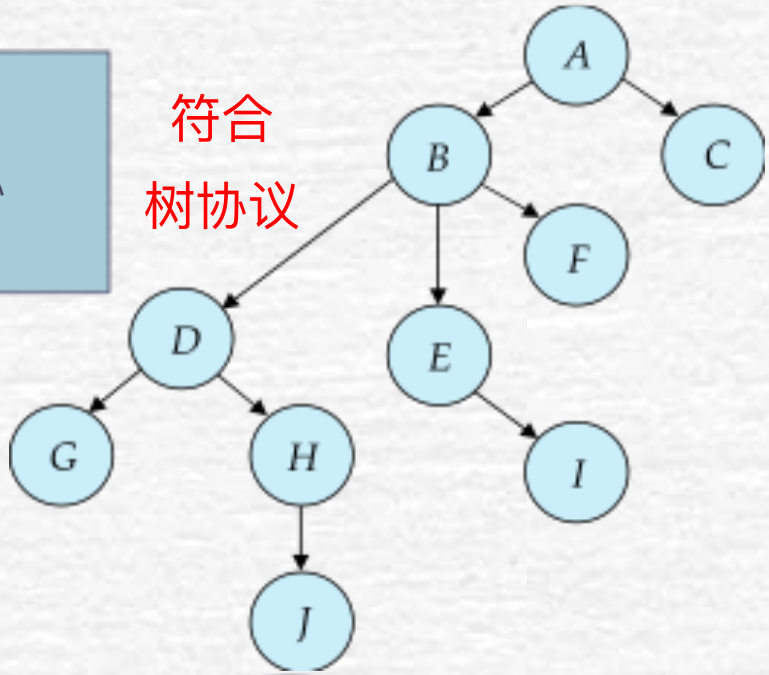
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).

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!

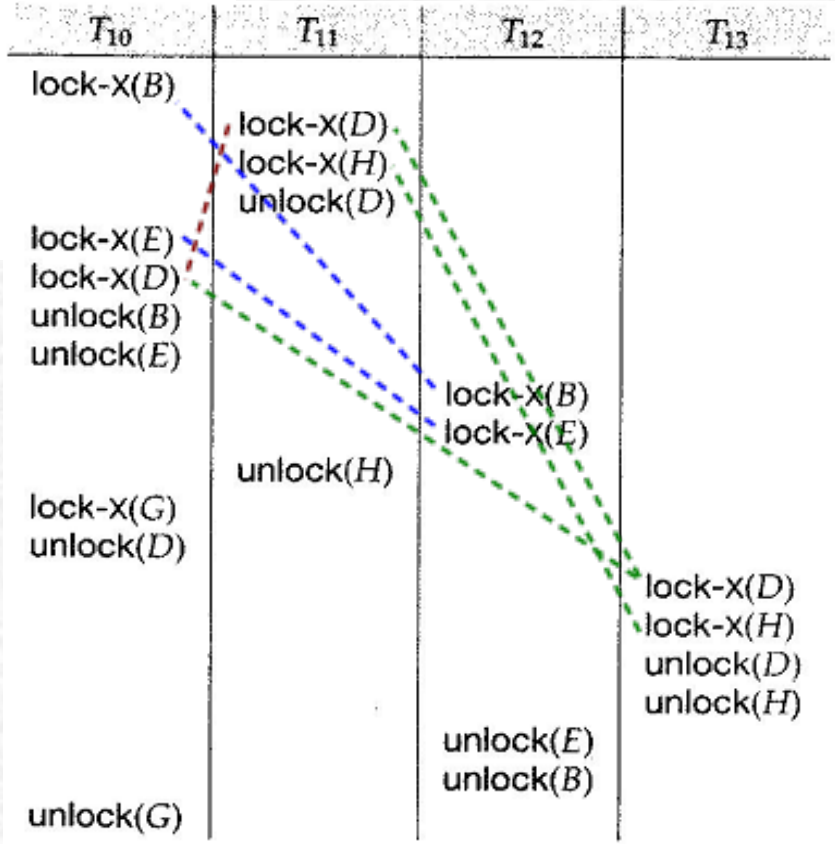
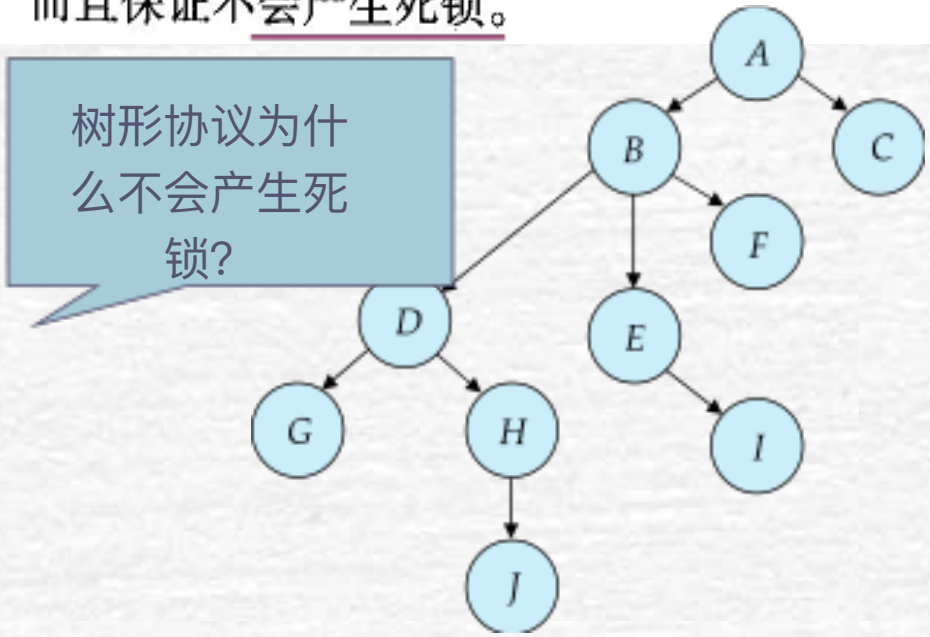


图 15-12 在树形协议下的可串行化调度

1.3 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} 持有两棵互相分的子树的锁。

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。



T3	T4
Lock-X(E)	
	Lock-X(D)
Lock-X(D)	
	Lock-X(E)
Unlock(E)	
Unlock(D)	
	Unlock(D)
	Unlock(E)

成功 成功等待 等待

调度P:存在死锁现象，它符合树形协议吗？

(出现死锁，但T3，T4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)

T1	T2
Lock-X(B)	
	Lock-X(B)
	Lock-X(D)
Lock-X(E)	
Lock-X(D)	
	Lock-X(E)
Unlock(B)	
Unlock(E)	
Unlock(D)	
	Unlock(B)
	Unlock(D)
	Unlock(E)

成功 等待 成功

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！

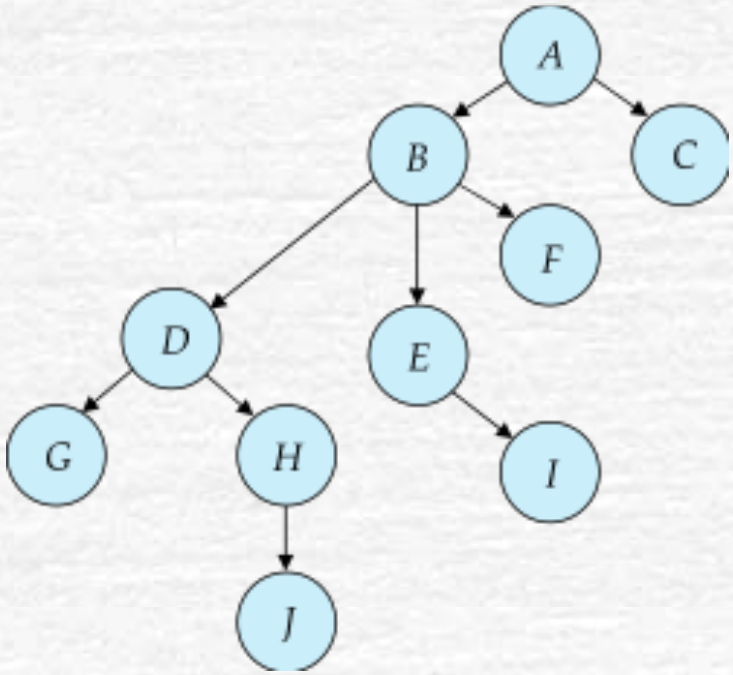
因为：T1锁住E前必先锁住父节点B，这时T2不能申请到D锁，因它之前申请不到B锁(等待T1释放)(树形协议只有X锁，如有S锁则会出现死锁现象)

1.4 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)

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)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
			commit
unlock(G)			
abort			

Read(D)

图 15-12 在树形协议下的可串行化调度

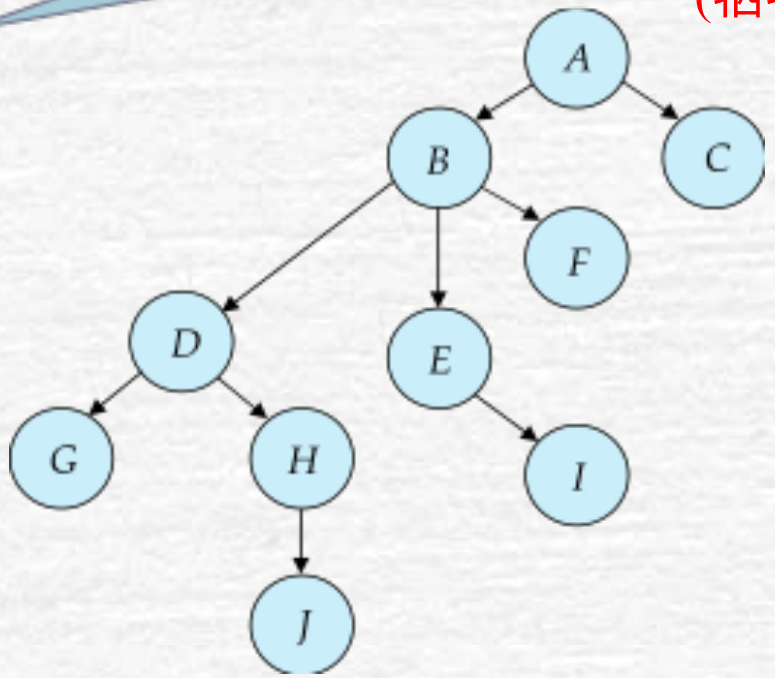
树形协议不能保证可恢复性!
(T13虽读了T10的数据D,
但T13已提交不能再回滚)

1.4 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?



Read(B)
Write(E)
Write(D)

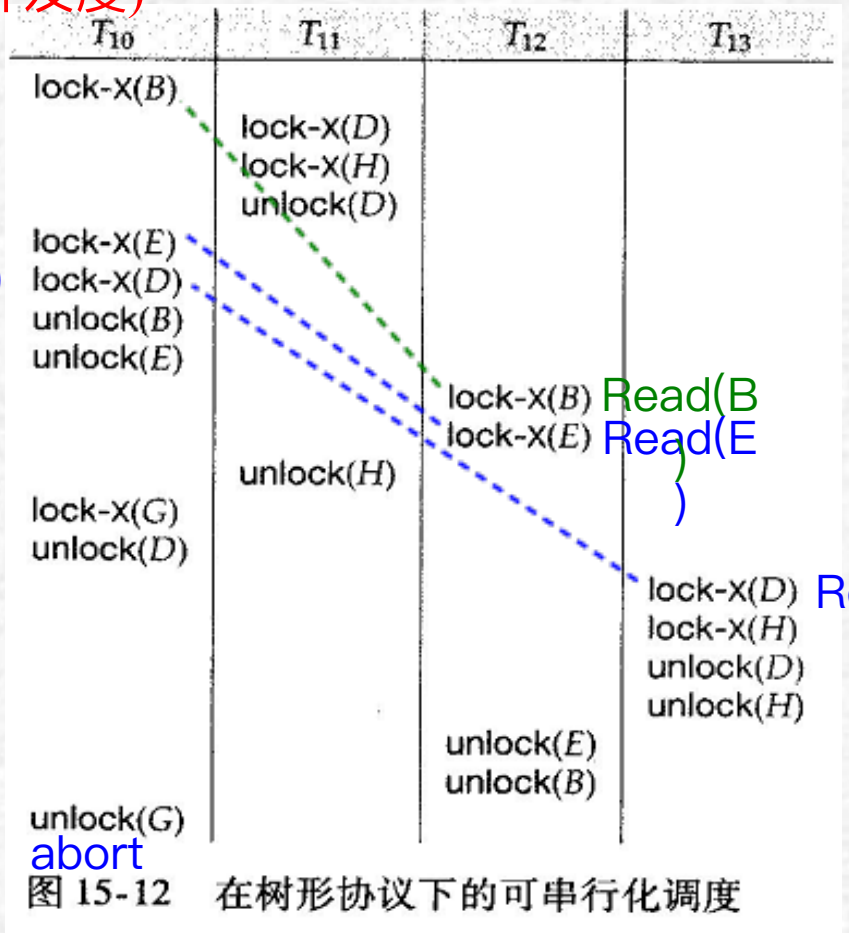


图 15-12 在树形协议下的可串行化调度

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)

1.5 树形协议小结

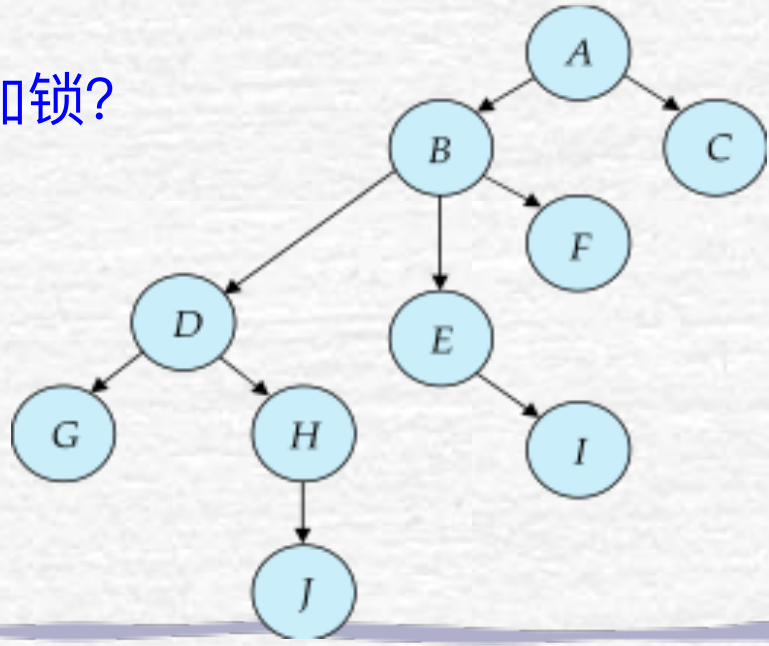
- 优点
  - 树形协议保证了冲突可串行化；
  - 同时不会产生死锁，不需要回滚；
  - 树形协议可较早地释放锁，以减少事务间的等待时间，从而可增强调度的并发性。
- 缺点
  - 树形协议不能保证事务的可恢复性；
  - 事务不能保证不发生级联回滚；
  - 事务有可能会给那些根本不访问的数据项加锁，从而增加了锁的开销和额外的等待时间，

    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!



二 死锁处理（25）

2.1 死锁检测方法

什么是等待图，
主要用途?

T_3	T_4
lock-x (*B*) ←	
read (*B*)	
B := *B* - 50	
write (*B*)	
	lock-s (*A*) ←
	read (*A*)
	lock-s (*B*) ←
lock-x (*A*)	←

grant-X*(B,T3)*-成功

grant-S*(A,T4)* -成功

grant-S*(B,T4)* –等待

grant-X*(A,T3)* -等待

死锁特点：多个事务相互等待某一方释放资源！

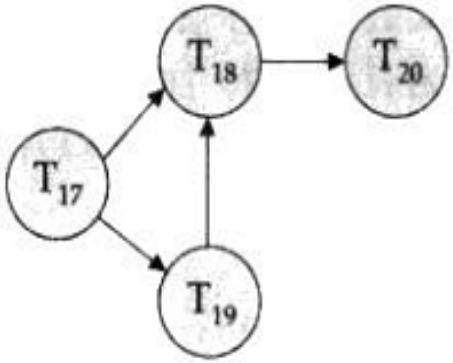


图 15-13　无环等待图

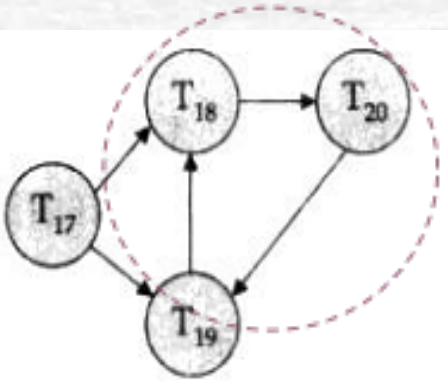


图 15-14　有一个环的等待图

发生死锁时，其等待图中一定带有向环！

死锁可以用称为**等待图**（wait-for graph）的有向图来精确描述。该图由 $G=(V, E)$ 对组成，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，边集 E 的每一元素是一个有序对 $T_i \rightarrow T_j$。如果 $T_i \rightarrow T_j$ 属于 E，则存在从事务 T_i 到 T_j 的一条有向边，表示事务 T_i 在等待 T_j 释放所需数据项。

当且仅当等待图包含环时，系统中存在死锁。在该环中的每个事务称为处于死锁状态。要检测死锁，系统需要维护等待图，并周期性地激活一个在等待图中搜索环的算法。

2.2 死锁的解除方法 p.381

出现死锁时如何解除?

法1 1. **选择牺牲者**：给定处于死锁状态的事务集，为解除死锁，我们必须决定回滚哪一个(或哪一些)事务以打破死锁。我们应使事务回滚(某一事务)带来的代价最小。可惜“最小代价”这个词并不准确。很多因素影响事务回滚代价，其中包括：

a. 事务已计算了多久，并在完成其指定任务之前该事务还将计算多长时间。
b. 该事务已使用了多少数据项。
c. 为完成事务还需使用多少数据项。
d. 回滚时将牵涉多少事务。

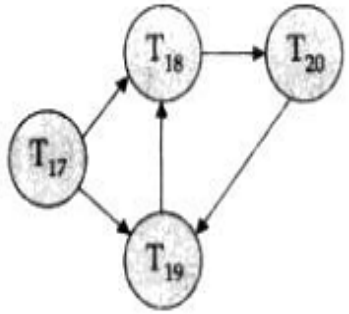


图 15-14 有一个环的等待图

对法1的改进

法2 2. **回滚**：一旦我们决定了要回滚哪个事务，就必须决定该事务回滚多远。最简单的方法是**彻底回滚**(total rollback)：中止该事务，然后重新开始。然而，事务只回滚到可以解除死锁处会更有效。这种**部分回滚**(partial rollback)要求系统维护所有正在运行事务的额外状态信息。确切地说，需要记录锁的申请/授予序列和事务执行的更新。死锁检测机制应当确定，为打破死锁，选定的事务需要释放哪些锁。选定的事务必须回滚到获得这些锁的第一个之前，并取消它在此之后的所有动作。恢复机制必须能够处理这种部分回滚。而且，事务必须能够在部分回滚之后恢复执行。有关参考文献见文献注解。

对法1-2可能引起不足的避免

法3 3. **饿死**：在系统中，如果选择牺牲者主要基于代价因素，有可能同一事务总是被选为牺牲者。这样一来，该事务总是不能完成其指定任务，这样就发生**饿死**(starvation)。我们必须保证一个事务被选为牺牲者的次数有限(且较少)。最常用的方案是在代价因素中包含回滚次数。

2.3 死锁的预防方法p.381

可采用预防策略：

避免死锁出现！

死锁只能是发现后才想法解除吗?

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，(1)在事务开始前通常很难预知哪些数据项需要封锁。(2)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。

防止死锁的另一种机制是对所有的数据项强加一个次序，同时要求事务只能按次序规定的顺序封锁数据项。我们曾经在树形协议中讲述过一个这样的机制，其中采用一个偏序的数据项。

防止死锁的第三种方法是使用抢占与事务回滚。我们给每个事务赋一个唯一的时间戳，系统仅用时间戳来决定事务应当等待还是回滚。并发控制仍使用封锁机制。

1. **wait-die** 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 T_i 的时间戳小于 T_j 的时间戳(即，T_i 比 T_j 老)时，允许 T_i 等待。否则，T_i 回滚(死亡)。老事务礼让: 新事务不等待,回滚!

例如，假设事务 T_{14}、T_{15}及 T_{16}的时间戳分别为 5、10 与 15。如果 T_{14}申请的数据项当前被 T_{15}持有，则 T_{14}将等待。如果 T_{16}申请的数据项当前被 T_{15}持有，则 T_{16}将回滚。

2. **wound-wait** 机制基于抢占技术，是与 wait-die 相反的机制。当事务 T_i 申请的数据项当前被 T_j 持有，仅当 T_i 的时间戳大于 T_j 的时间戳(即，T_i 比 T_j 年轻)时，允许 T_i 等待。否则，T_j 回滚(T_j 被 T_i伤害)。

年轻事务礼让:老事务不等待,强制使新事务回滚!

三 多粒度封锁协议（25）

3.1 多粒度锁

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

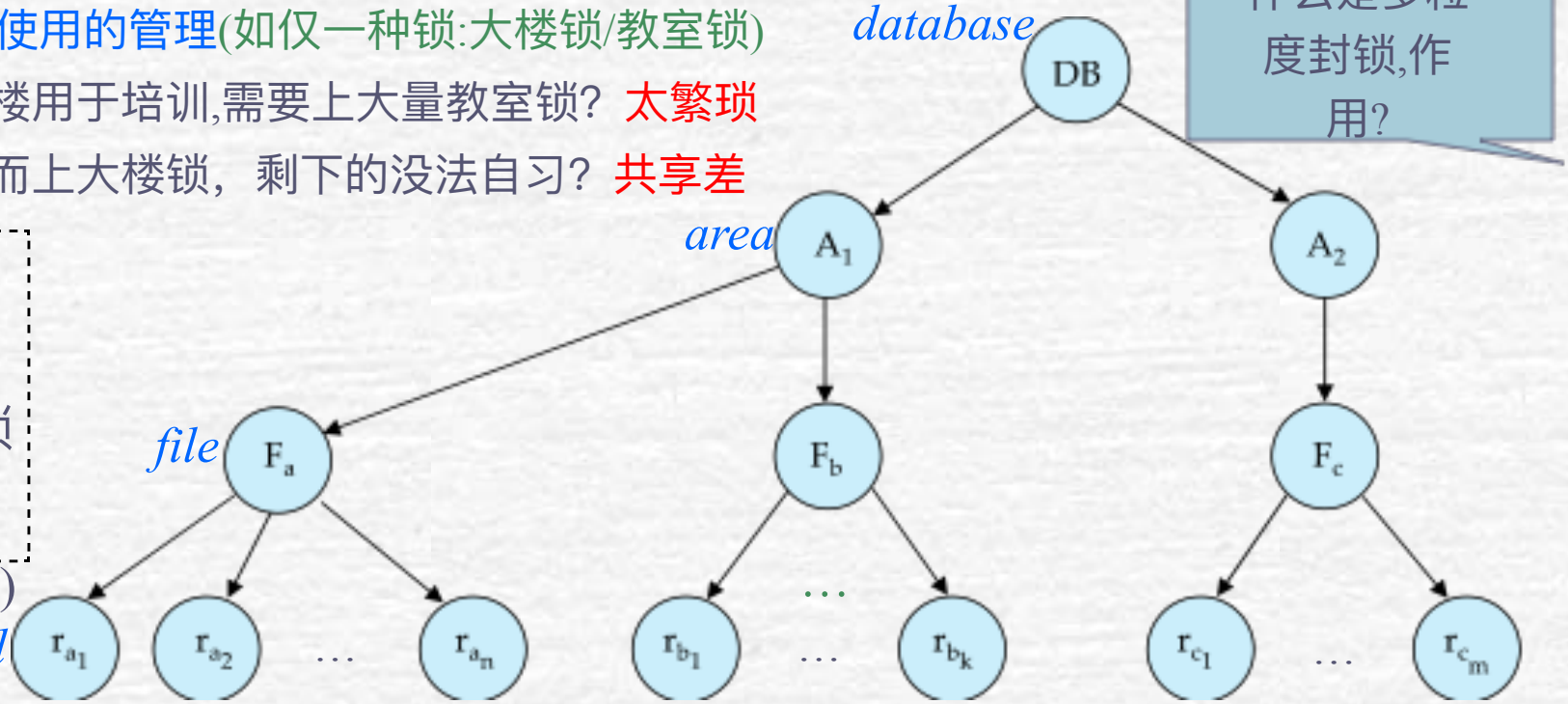
2)若仅租用部分教室而上大楼锁，剩下的没法自习? 共享差

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)

例示：

什么是多粒度封锁,作用?



到目前为止所讲的并发控制机制中，我们将一个个数据项作为进行同步执行的单元。

然而，某些情况下需要把多个数据项聚为一组，将它们作为一个同步单元，因为这样会更好。例如，如果事务 T_i 需要访问整个数据库，使用的是一种封锁协议，则事务 T_i 必须给数据库中每个数据项加锁。显然，执行这些加锁操作是很费时的。要是 T_i 能够只发出单个封锁整个数据库的加锁请求，那会更好。另一方面，如果事务 T_j 只需要存取少量数据项，就不应要求给整个数据库加锁，否则并发性就丧失了。

费事费力!使用率低!

3.2 多粒度锁的类型

什么是隐式封锁,显示封锁,意向锁?

显式锁与隐式锁的作用 意向锁的作用

树中每个结点都可以单独加锁。正如我们在两阶段封锁协议中所做的那样，我们将使用**共享**(shared)锁与**排他**(exclusive)锁。当事务对一个结点加锁，或为共享锁或为排他锁，该事务也以同样类型的锁隐式地封锁这个结点的全部后代结点。例如，若事务 T_i 给图 15-15 中的 F_b **显式**(explicit lock)地加排他锁，则事务 T_i 也给所有属于该文件的记录**隐式**(implicit lock)地加排他锁。没有必要显式地给 F_b 中的单条记录逐个加锁。

假设事务 T_j 希望封锁文件 F_b 的记录 r_{b_6}。由于 T_i 显式地给 F_b 加锁，因此意味着 r_{b_6} 也被加锁(隐式地)。但是，当 T_j 发出对 r_{b_6} 加锁的请求时，r_{b_6} 没有显式加锁！系统如何判定是否 T_j 可以封锁 r_{b_6} 呢？T_j 必须从树根到 r_{b_6} 进行遍历，如果发现此路径上某个结点的锁与要加的锁类型不相容，则 T_j 必须延迟。

现假设事务 T_k 希望封锁整个数据库。为此，它只需给层次结构图的根结点加锁。不过，请注意 T_k 给根结点加锁不会成功，因为目前 T_i 在树的某部分持有锁(具体地说，对文件 F_b 持有锁)。但是，系统是怎样判定根结点是否可以加锁呢？一种可能的方法是搜索整棵树。然而，这个方法破坏了多粒度封锁机制的初衷。获取这种知识的一个更有效的方法是引入一种新的锁类型，即**意向锁**类型(intention lock mode)。如果一个结点加上了意向锁，则意味着要在树的较低层进行显式加锁(也就是说，以更小的粒度加锁)。在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁。因此，事务不必搜索整棵树就能判定能否成功地给一个结点加锁。希望给某个结点(如 Q)加锁的事务必须遍历从根到 Q 的路径。在遍历树的过程中，该事务给各结点加上意向锁。

(多粒度锁加锁示例)

注：加S(或X)锁-需先在各父节点加IS(或IX)意向锁!

解决低层锁问题 解决高层锁问题

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

返回

多粒度锁加锁示例

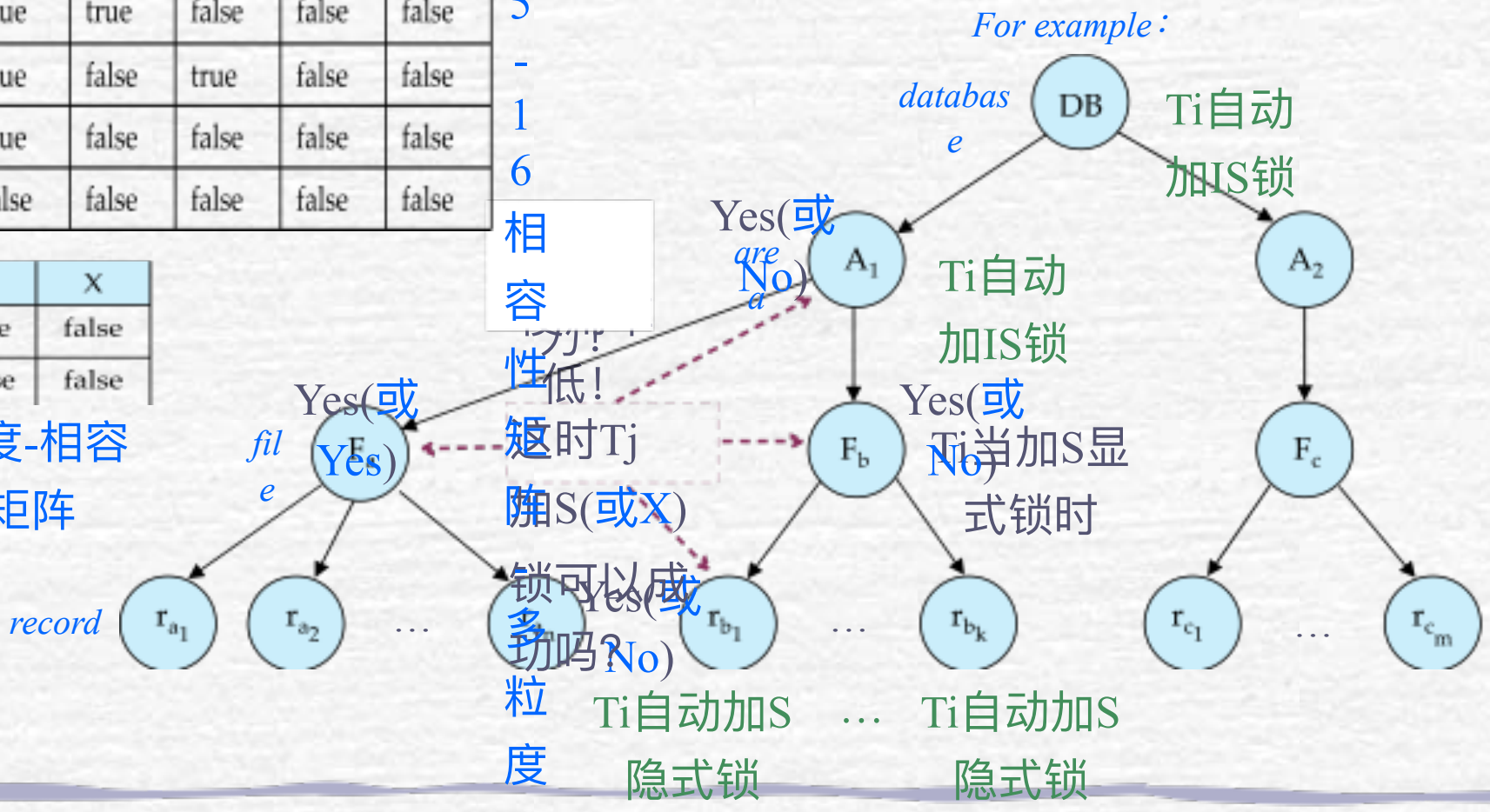
当Ti已加锁时, Tj 加锁还能成功吗?

	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

	S	X
S	true	false
X	false	false

单粒度-相容性矩阵



低!
这时Tj
加S(或X)
锁可以成
功吗?
多粒度

3.3 多粒度封锁协议p.383

多粒度封锁协议规则，及基本特点?

多粒度封锁协议(multiple-granularity locking protocol)：采用这些锁类型保证可串行性。每个事务 T_i 要求按如下规则对数据项 Q 加锁：

1. 事务 T_i 必须遵从图 15-16 所示的锁类型相容函数。
2. 事务 T_i 必须首先封锁树的根结点，并且可以加任意类型的锁。
3. 仅当 T_i 当前对 Q 的父结点具有 IX 或 IS 锁时，T_i 对结点 Q 可加 S 或 IS 锁。
4. 仅当 T_i 当前对 Q 的父结点具有 IX 或 SIX 锁时，T_i 对结点 Q 可加 X、SIX 或 IX 锁。
5. 仅当 T_i 未曾对任何结点解锁时，T_i 可对结点加锁（也就是说，T_i 是两阶段的）。
6. 仅当 T_i 当前不持有 Q 的子结点的锁时，T_i 可对结点 Q 解锁。

多粒度协议要求加锁按自顶向下的顺序(根到叶)，而锁的释放则按自底向上的顺序(叶到根)。

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