并发与死锁

哲学家就餐问题

哲学家就餐问题是在计算机科学中的一个经典问题，用来演示在并行计算中多线程同步时产生的问题。

哲学家就餐问题

- 假设有五位哲学家围坐在一张圆形餐桌旁，做以下两件事情之一：吃饭，或者思考。
- 吃东西的时候，他们就停止思考，思考的时候也停止吃东西。
- 餐桌中间有一大碗意大利面，每两个哲学家之间有一只餐叉。
- 因为用一只餐叉很难吃到意大利面，所以假设哲学家必须用两只餐叉吃东西。他们只能使用自己左右手边的那两只餐叉。

哲学家就餐问题

- 哲学家从来不交谈，这就很危险，可能产生死锁
- 每个哲学家都拿着左手的餐叉，永远都在等右边的餐叉
- 或者每个哲学家都拿着右手的餐叉，永远都在等左边的餐叉

哲学家就餐问题

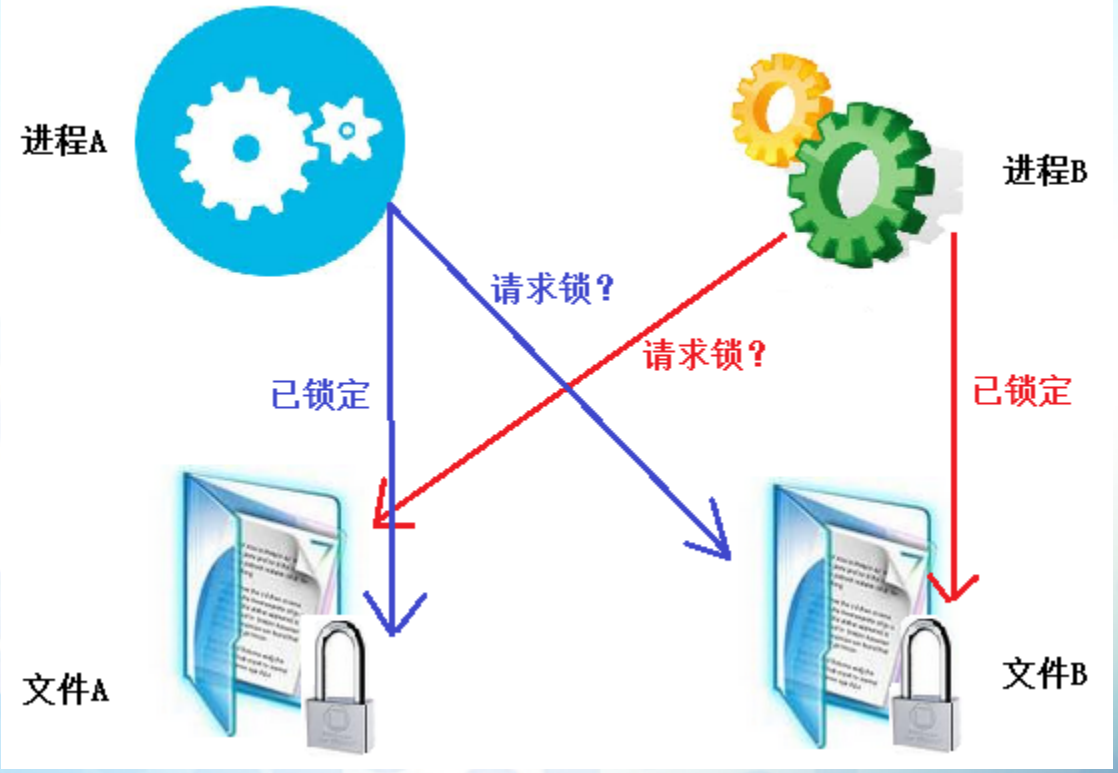
- 即使没有死锁，也有可能发生资源耗尽。
  - 假设规定当哲学家等待另一只餐叉超过五分钟后就放下自己手里的那一只餐叉，并且再等五分钟后进行下一次尝试。
  - 此策略消除了死锁，但仍然有可能发生“活锁”
  - 如果五位哲学家在完全相同的时刻进入餐厅，并同时拿起左边的餐叉，那么这些哲学家就会等待五分钟，同时放下手中的餐叉，再等五分钟，又同时拿起这些餐叉。

哲学家就餐问题

- 在实际的计算机问题中，缺乏餐叉可以类比为缺乏共享资源。
- 常用的计算机技术是资源加锁，用来保证在某个时刻，资源只能被一个程序或一段代码访问
- 当一个程序想要使用的资源已经被另一个程序锁定，它就等待资源解锁
- 当多个程序涉及到加锁的资源时，在某些情况下就有可能发生死锁

哲学家就餐问题

- 某个程序需要访问两个文件，当两个这样的程序各锁了一个文件，那它们都在等待对方解锁另一个文件，这就发生了死锁



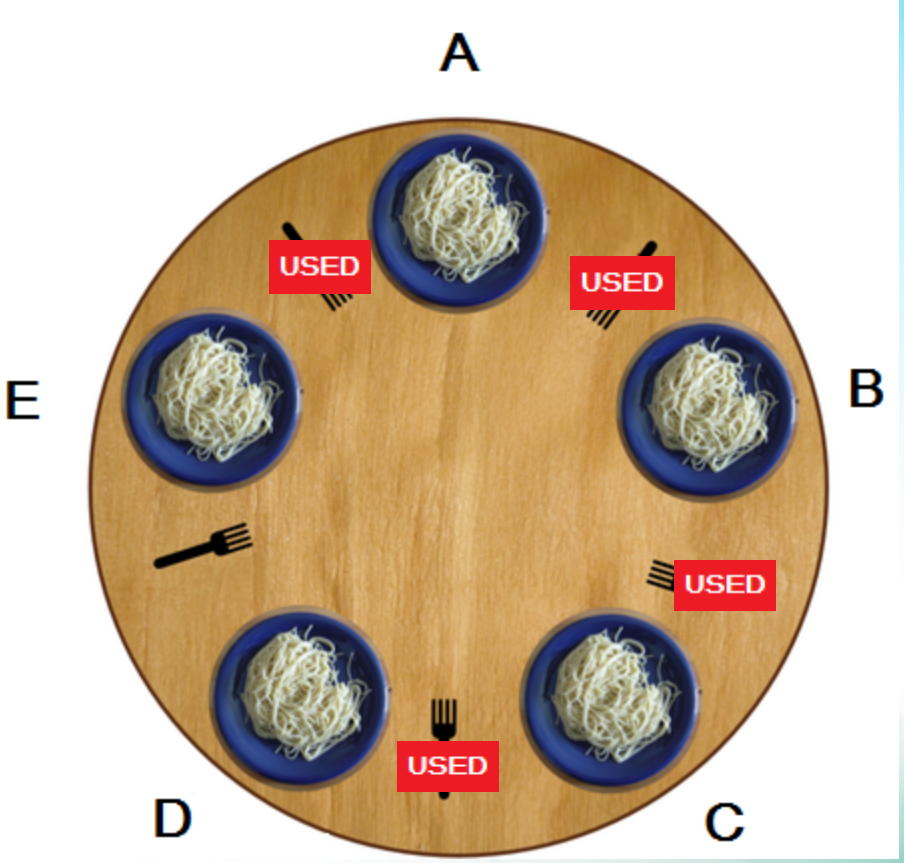
服务生解法

- 一个简单的解法是引入一个餐厅服务生，哲学家必须经过他的允许才能拿起餐叉。

- 因为服务生知道哪只餐叉正在使用，所以他能够作出判断避免死锁。

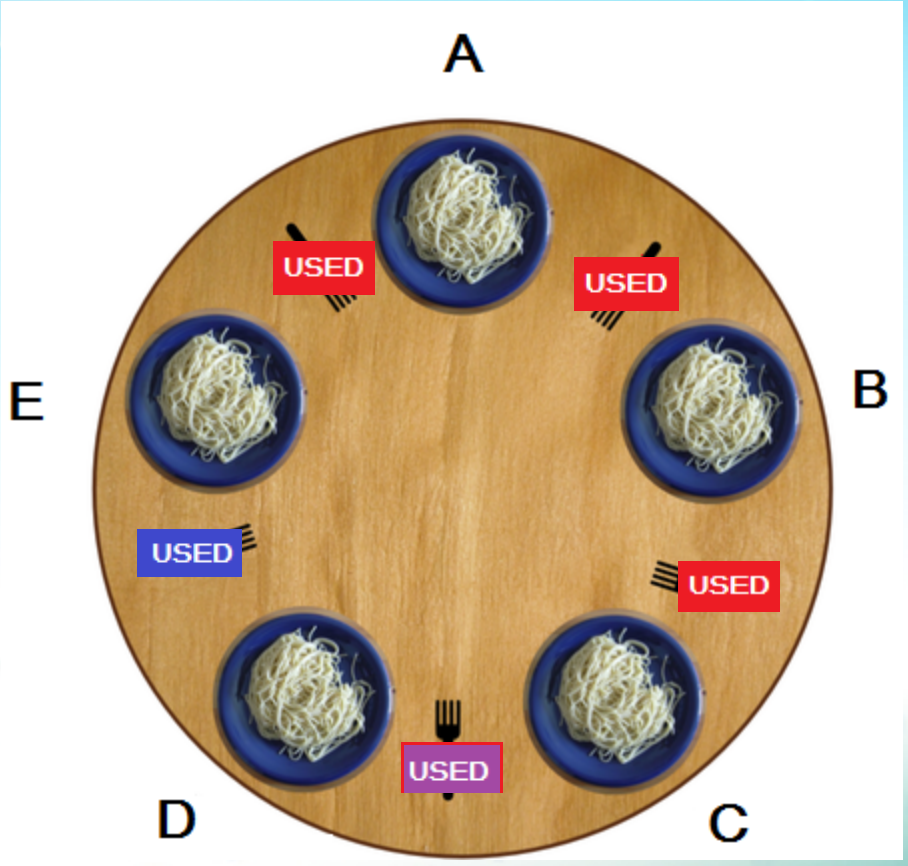
服务生解法

- 如果A和C在吃东西，则有四只餐叉在使用中
- B坐在A和C之间，所以两只餐叉都无法使用，而D和E之间有一只空余的餐叉
- 假设这时D, E想要吃东西。如果拿起了第五只餐叉，就有可能发生死锁。



服务生解法

- 如果E拿起了剩余的那个叉子
- 而C哲学家开始思考，放下了叉子
- 而D哲学家开始吃面，于是拿起了C左手的叉子
- 如果A一直吃面，就陷入了死锁，否则如果E放下叉子，D就可以吃到面



服务生解法

- 相反，如果他征求服务生同意，服务生会让他等待。
- 这样，我们就能保证下次当两把餐叉空余出来时，一定有一位哲学家可以成功的得到一对餐叉，从而避免了死锁。

资源分级解法

- 另一个简单的解法是为资源（这里是餐叉）分配一个偏序或者分级的关系
- 并约定所有资源都按照这种顺序获取，按相反顺序释放
- 而且保证不会有两个无关资源同时被同一项工作所需要。

资源分级解法

- 在哲学家就餐问题中，资源（餐叉）按照某种规则编号为1至5
- 每一个工作单元（哲学家）总是先拿起左右两边编号较低的餐叉，再拿编号较高的
- 用完餐叉后，他总是先放下编号较高的餐叉，再放下编号较低的

资源分级解法

- 在这种情况下，当四位哲学家同时拿起他们手边编号较低的餐叉时，只有编号最高的餐叉留在桌上，从而第五位哲学家就不能使用任何一只餐叉了
- 而且，只有一位哲学家能使用最高编号的餐叉，所以他能使用两只餐叉用餐。当他吃完后，他会先放下编号最高的餐叉，再放下编号较低的餐叉，从而让另一位哲学家拿起后边的这只开始吃东西。

资源分级解法

- 这个解法是由Dijkstra最早提出的
- 尽管资源分级能避免死锁，但这种策略并不总是实用的，特别是当所需资源的列表并不是事先知道的时候。
- 例如，假设一个工作单元拿着资源3和5，并决定需要资源2，则必须先要释放5，之后释放3，才能得到2，之后必须重新按顺序获取3和5。

资源分级解法

- 对需要访问大量数据库记录的计算机程序来说，如果需要先释放高编号的记录才能访问新的记录，那么运行效率就不会高，因此这种方法在这里并不实用。

Chandy/Misra解法

- 在1984年， Chandy和Misra 提出了一个不同的解法
- 这个解法允许任意的用户(任意编号$P_1, \ldots, P_n$)争用任意数量的资源
- 这个解法是完全意义上的分布式，也不需要初始化优先级
- 但是它违反了“哲学家从来不交谈”的要求

Chandy/Misra解法

1. 对每一对竞争一个资源的哲学家，新拿一个餐叉，给编号较低的哲学家。每只餐叉都是“干净的”或者“脏的”。最初，所有的餐叉都是脏的。
2. 当一位哲学家要使用资源（也就是要吃东西）时，他必须从与他竞争的邻居那里得到。对每只他当前没有的餐叉，他都发送一个请求。
3. 当拥有餐叉的哲学家收到请求时，如果餐叉是干净的，那么他继续留着，否则就擦干净并交出餐叉。
4. 当某个哲学家吃东西后，他的餐叉就变脏了。如果另一个哲学家之前请求过其中的餐叉，那他就擦干净并交出餐叉。

数据库并发

- 数据库是一个共享资源，可以供多个用户使用。
- 允许多个用户同时使用的数据库系统称为多用户数据库系统。
- 为了充分利用系统资源，发挥数据库共享资源的特点，应该允许多个事务并行的执行。
  - 航空订票系统数据库
  - 银行储户数据库

数据库并发

- 当多个用户并发地存取数据库时就会产生多个事务同时存取同一数据的情况。
- 若对并发操作不加控制就可能会存取和存储不正确的数据，破坏数据库的一致性。
- 所以数据库管理系统必须提供并发控制机制。
- 并发控制机制是衡量一个数据库管理系统性能的重要标志之一。

并发控制

- 例如考虑航空订票系统中的一系列操作
  1. 甲售票点读出某航班的机票余额A，设A=16
  2. 乙售票点读出同一航班的机票余额A，也为16
  3. 甲售票点卖出一张机票,修改余额$A \leftarrow A$-1，所以A为15，把A写回数据库
  4. 乙售票点也卖出一张机票，修改余额$A \leftarrow A$-1，所以A为15，把A写回数据库

  ■结果明明卖出两张机票，数据库中机票余额只减少1。

并发控制

- 这种情况称为数据库的不一致性，是由并发操作引起的。
- 在并发操作情况下，对甲、乙两个事务的操作序列的调度是随机的。
- 若按上面的调度序列执行，甲事务的修改就被丢失。
  - 原因：第4步中乙事务修改A并写回后覆盖了甲事务的修改

并发控制

- 并发操作带来的数据不一致性
  - 丢失修改（Lost Update）
  - 不可重复读（Non-repeatable Read）
  - 读“脏”数据（Dirty Read）

丢失修改

- 两个事务T1和T2读入同一数据并修改，T2的提交结果破坏了T1提交的结果，导致T1的修改被丢失。

- 上面飞机订票例子就属此类

丢失修改

T1	T2
(1) 读A=16	
(2)	读A=16
(3) A ←A-1	
写回A=15	
(4)	A ←A-1
	写回A=15

不可重复读

- 不可重复读是指事务T1读取数据后，事务T2执行更新操作，使T1无法再现前一次读取结果。
- 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

不可重复读

T_1	T_2
① R(A)=50	
R(B)=100	
求和=150	
②	R(B)=100
	B←B*2
	(B)=200
③ R(A)=50	
R(B)=200	
和=250	
(验算不对)	

- T1读取B=100进行运算
- T2读取同一数据B，对其进行修改后将B=200写回数据库。
- T1为了对读取值校对重读B，B已为200，与第一次读取值不一致

不可重复读

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

- 读“脏”数据是指：
  - 事务T1修改某一数据，并将其写回磁盘
  - 事务T2读取同一数据后，T1由于某种原因被撤销
  - 这时T1已修改过的数据恢复原值，T2读到的数据就与数据库中的数据不一致
  - T2读到的数据就为“脏”数据，即不正确的数据

读“脏”数据

T_1	T_2
① R(C)=100	
C←C*2	
W(C)=200	
②	R(C)=200
③ROLLBACK	
C恢复为100	

- T1将C值修改为200，T2读到C为200
- T1由于某种原因撤销，其修改作废，C恢复原值100
- 这时T2读到的C为200，与数据库内容不一致，就是“脏”数据

并发控制

- 数据不一致性：由于并发操作破坏了事务的隔离性
- 并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性

并发控制

- 并发控制的主要技术
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
- 商用的数据库管理系统一般都采用封锁方法

什么是封锁

- 封锁就是事务T在对某个数据对象（例如表、记录等）操作之前，先向系统发出请求，对其加锁
- 加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其它的事务不能更新此数据对象。

封锁类型

- 一个事务对某个数据对象加锁后究竟拥有什么样的控制由封锁的类型决定。
- 基本封锁类型
  - 排它锁（Exclusive Locks，简记为X锁）
  - 共享锁（Share Locks，简记为S锁）

排它锁

- 排它锁又称为写锁
- 若事务T对数据对象A加上X锁，则只允许T读取和修改A，其它任何事务都不能再对A加任何类型的锁，直到T释放A上的锁
- 保证其他事务在T释放A上的锁之前不能再读取和修改A

共享锁

- 共享锁又称为读锁

- 若事务T对数据对象A加上S锁，则其它事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁
- 保证其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改

锁的相容矩阵

T_2 \ T_1	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Y=Yes，相容的请求
N=No，不相容的请求

锁的相容矩阵

- 最左边一列表示事务T1已经获得的数据对象上的锁的类型，其中横线表示没有加锁。
- 最上面一行表示另一事务T2对同一数据对象发出的封锁请求。
- T2的封锁请求能否被满足用矩阵中的Y和N表示
  - Y表示事务T2的封锁要求与T1已持有的锁相容，封锁请求可以满足
  - N表示T2的封锁请求与T1已持有的锁冲突，T2的请求被拒绝

解决丢失修改问题

T_1	T_2
① Xlock A	
② R(A)=16	
	Xlock A
③ A←A-1	等待
W(A)=15	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	R(A)=15
	A←A-1
⑤	W(A)=14
	Commit
	Unlock A

没有丢失修改

- 事务T1在读A进行修改之前先对A加X锁
- 当T2再请求对A加X锁时被拒绝
- T2只能等待T1释放A上的锁后T2获得对A的X锁
- 这时T2读到的A已经是T1更新过的值15
- T2按此新的A值进行运算，并将结果值A=14送回到磁盘。避免了丢失T1的更新。

解决不可重复读问题

T_1	T_2
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
	等待
③ R(A)=50	等待
R(B)=100	等待
求和=150	等待
Commit	等待
Unlock A	等待
Unlock B	等待
④	获得XlockB
	R(B)=100
	B←B*2
⑤	W(B)=200
	Commit
	Unlock B

可重复读

- 事务T1在读A，B之前，先对A，B加S锁
- 其他事务只能再对A，B加S锁，而不能加X锁，即其他事务只能读A，B，而不能修改
- 当T2为修改B而申请对B的X锁时被拒绝只能等待T1释放B上的锁
- T1为验算再读A，B，这时读出的B仍是100，求和结果仍为150，即可重复读
- T1结束才释放A，B上的S锁。T2才获得对B的X锁

解决读“脏”数据问题

T_1	T_2
① Xlock C	
R(C)=100	
C←C*2	
W(C)=200	
②	Slock C
	等待
③ ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	R(C)=100
⑤	Commit C
	Unlock C

不读“脏”数据

- 事务T1在对C进行修改之前，先对C加X锁，修改其值后写回磁盘
- T2请求在C上加S锁，因T1已在C上加了X锁，T2只能等待
- T1因某种原因被撤销，C恢复为原值100
- T1释放C上的X锁后T2获得C上的S锁，读C=100。避免了T2读“脏”数据

活锁与死锁

- 封锁技术可以有效地解决并行操作的一致性问题，但也带来一些新的问题
  - 死锁
  - 活锁

活锁

- 事务T1封锁了数据R
- 事务T2又请求封锁R，于是T2等待。
- T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，T2仍然等待。
- T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求……
- T2有可能永远等待，这就是活锁的情形

活锁

T_1	T_2	T_3	T_4
lock R	.	.	.
.	lock R	.	.
.	等待	Lock R	.
Unlock	等待	.	Lock R
.	等待	Lock R	等待
.	等待	.	等待
.	等待	Unlock	等待
.	等待	.	Lock R
.	等待	.	.

活锁

采用什么策略能够避免活锁呢？

- 采用先来先服务的策略
  - 当多个事务请求封锁同一数据对象时
  - 按请求封锁的先后次序对这些事务排队
  - 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁

死锁

- 事务T1封锁了数据R1
- T2封锁了数据R2
- T1又请求封锁R2，因T2已封锁了R2，于是T1等待T2释放R2上的锁
- 接着T2又申请封锁R1，因T1已封锁了R1，T2也只能等待T1释放R1上的锁
- 这样T1在等待T2，而T2又在等待T1，T1和T2两个事务永远不能结束，形成死锁

死锁

T_1	T_2
lock R_1	•
•	Lock R_2
•	•
Lock R_2.	•
等待	•
等待	Lock R_1
等待	等待
等待	等待
	•

解决死锁的方法

- 两类方法
  1. 预防死锁
  2. 死锁的诊断与解除

死锁的预防

- 产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已为其他事务封锁的数据对象加锁，从而出现死等待。

- 预防死锁的发生就是要破坏产生死锁的条件

产生死锁的四个必要条件

- 互斥条件：一个资源每次只能被一个进程使用
  请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放
  不剥夺条件:进程已获得的资源，在未使用完之前，不能强行剥夺
  循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系

死锁的预防

- 预防死锁的方法
  - 一次封锁法
  - 顺序封锁法

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

- 存在的问题
  - 降低系统并发度
  - 难于事先精确确定封锁对象

顺序封锁法

- 顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁
- 顺序封锁法存在的问题
  - 维护成本
    - 数据库系统中封锁的数据对象极多，并且在不断地变化。
  - 难以实现
    - 很难事先确定每一个事务要封锁哪些对象

死锁的诊断与解除

- 死锁的诊断
  - 超时法
  - 事务等待图法

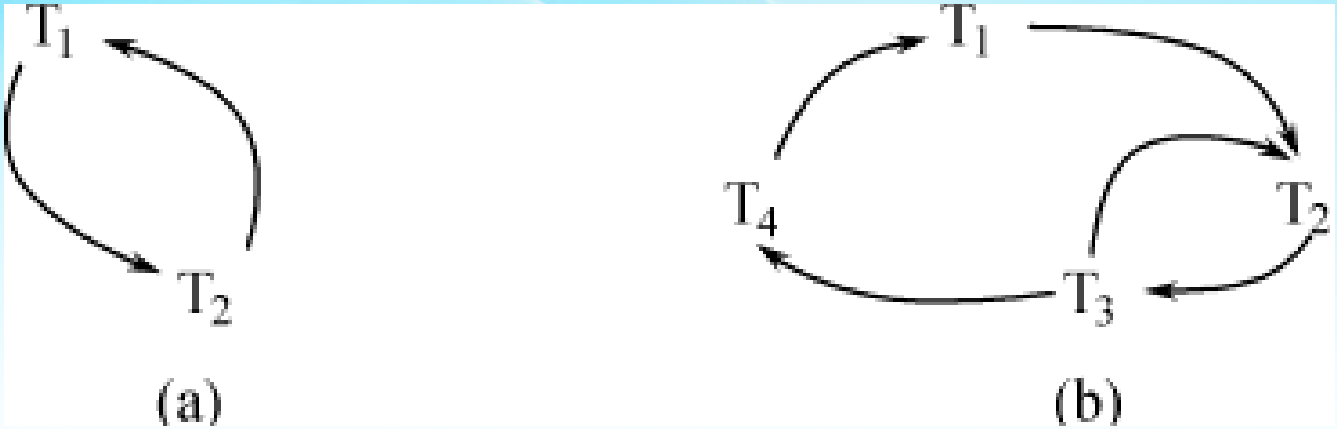
超时法

- 如果一个事务的等待时间超过了规定的时限，就认为发生了死锁
- 优点
  - 实现简单
- 缺点
  - 有可能误判死锁
  - 时限若设置得太长，死锁发生后不能及时发现

等待图法

- 用事务等待图动态反映所有事务的等待情况
  - 事务等待图是一个有向图G=(T，U)
  - T为结点的集合，每个结点表示正运行的事务
  - U为边的集合，每条边表示事务等待的情况
  - 若T1等待T2，则T1，T2之间划一条有向边，从T1指向T2

等待图法



事务等待图

- 图(a)中，事务T1等待T2，T2等待T1，产生了死锁
- 图(b)中，事务T1等待T2，T2等待T3，T3等待T4，T4又等待T1，产生了死锁
- 图(b)中，事务T3可能还等待T2，在大回路中又有小的回路

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

- 解除死锁
  - 选择一个处理死锁代价最小的事务，将其撤消
  - 释放此事务持有的所有的锁，使其它事务能继续运行下去

操作系统的并发

- 在操作系统中也存在和数据库管理系统中的一些问题
- 操作系统中的多进程相当于数据库管理系统中多用户
- 操作系统中的对硬件资源的占用相当于数据库中对数据的操作
- 因此操作系统中也有并发和死锁的现象