Concurrency Control

▲Concurrency Control [并发控制]

A database must provide a mechanism that will ensure that all possible schedules are either conflict or view serializable, and are recoverable and preferably cascade less.

A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency

Testing a schedule for serializability after it has executed is too late!

Goal – to develop concurrency control protocols [并发控制协议] that will assure serializability.

▲Lock-Based Protocols [基于锁定的协议]

A lock is a mechanism to control concurrent access to a data item [锁是一种控制对数据项的并发访问的机制]

Data items can be locked in two modes:

(1) exclusive (X) mode - Data item can be both read as well as written.

(X-lock is requested using lock-X instruction)

(2) shared (S) mode - Data item can only be read.

(S-lock is requested using lock-S instruction)

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix [锁相容性矩阵]:

	S	X
S	true	false
X	false	false

A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions. (兼容性)

Any number of transactions can hold shared locks on an item.

But if any transaction holds an exclusive lock [独占锁] on the item no other transactions may hold any lock on the item.

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted. (等待所有兼容性锁释放后再接受锁)

Example: T2:

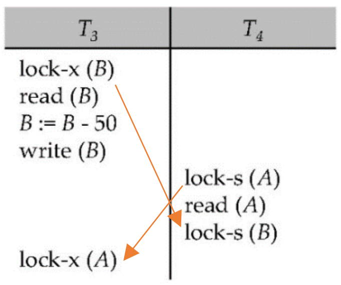
```
lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B)
```

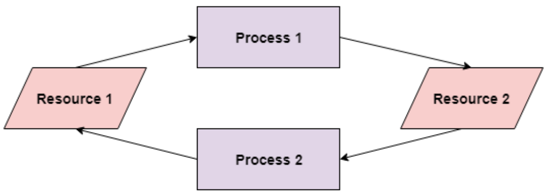
A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

Locking protocols restrict the set of possible schedules.

Attention:

The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.





Deadlock in Operating System

NeitherT3 norT4 can make progress — executing
lock-S(B) causes T4 to wait for T3 to release its lock on B, while executing lock-X(A) causesT3 to wait forT4 to release its lock on A.

To handle a deadlock one of T3 orT4 must be rolled back and its locks released.

Starvation [饥饿] is also possible if concurrency control manager is badly designed. For example:

The most common solution to recover from deadlock is to roll back one or more transactions [从死锁中恢复的最常见解决方案是回滚一个或多个 transaction]

If a transaction is repeatedly chosen as the victim, it will never complete its task, hence starvation. [如果一个 transaction 被反复选择为受害者，它将永远无法完成它的任务，从而导致饥饿]

Concurrency control manager can be designed to prevent starvation.
The most common solution is to include the number of rollbacks in the cost factor for selecting a victim.
More detail about deadlock & starvation: Starvation and Deadlock (tutorialspoint.com)

Two phases of Locking Protocol:

(Two-phase locking does not ensure freedom from deadlocks)

Phase 1: Growing Phase [增长阶段]
- transaction may obtain locks
- transaction may not release locks

Phase 2: Shrinking Phase [缩减阶段]
- transaction may release locks
- transaction may not obtain locks

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).

Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.
Rigorous two-phase locking is even stricter: here all locks (including the shared locks) are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Lock Conversions [锁转换]

– First Phase:
  - can acquire a lock-S on item
  - can acquire a lock-X on item
  - can convert a lock-S to a lock-X (upgrade)

– Second Phase:
  - can release a lock-S
  - can release a lock-X

can convert a lock-X to a lock-S (downgrade)

▲Automatic Acquisition of Locks [自动获取锁] (algorithm)

A transaction Ti issues the standard read/write instruction, without explicit locking calls.

All locks are released after commit or abort

The operation read(D) is processed as:

```
  if Ti has a lock on D then:
        read(D)
  else:
        begin, if necessary, wait until no other transaction has a lock-X on D:
              grant Ti a lock-S on D;
              read(D)
        end
```

The operation write(D) is processed as:

```
  if Ti has a lock-X on D then:
        write(D)
  else:
        begin, if necessary, wait until no other transactions have any lock on D:
           if Ti has a lock-S on D then:
                 upgrade lock on D to lock-X
           else:
                 grant Ti a lock-X on D
                 write(D)
        end
```

▲Implementation of Locking [锁定的实现]

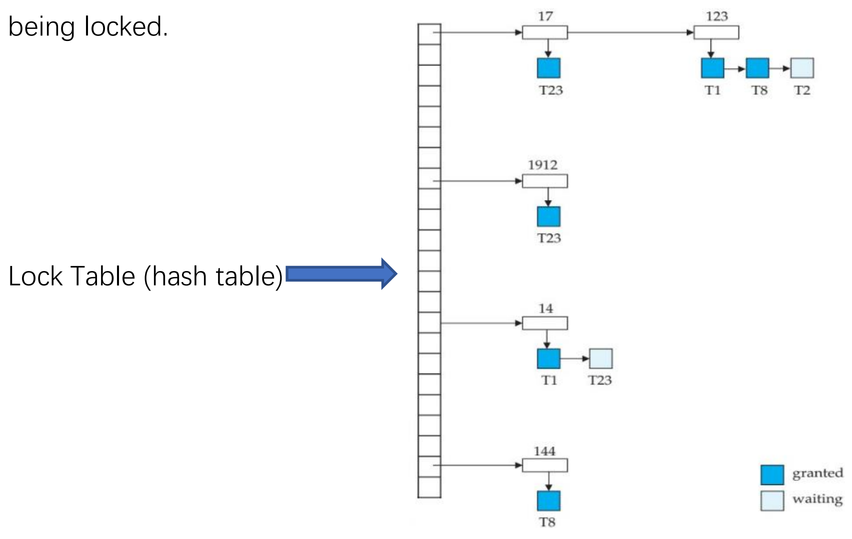
A lock manager can be implemented as a separate process to which transactions send lock and unlock requests.

The lock manager replies to a lock request by sending a lock grant message, or a message asking the transaction to roll back, in case of a deadlock.

The requesting transaction waits until its request is answered.

The lock manager maintains a data structure called a lock table to record granted locks and pending requests.

The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.



▲Deadlock Handling [死结处理]

example

T_1	T_2
lock-X on A write (A)	
	lock-X on B write (B) wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock prevention protocols ensure that the system will never enter into a deadlock state.

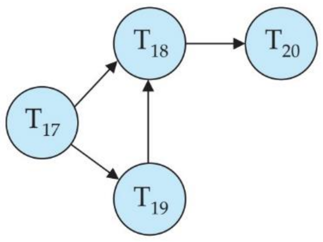
Some prevention strategies:
(1) Require that each transaction locks all its data items before it begins execution (pre-declaration). 锁定
(2) Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol). 排序执行

Use transaction timestamps for the sake of deadlock prevention alone:
wait-die scheme — non-preemptive
older transaction may wait for younger ones but younger transactions never wait for older ones; they are rolled back instead.
Result: a transaction may die several times before acquiring needed data item
wound-wait scheme — preemptive
older transaction wounds (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
Result: may be fewer rollbacks than wait-die scheme.
Both in wait-die and wound-wait schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided. [因此，较老的事务优先于较新的事务，从而避免了饥饿]

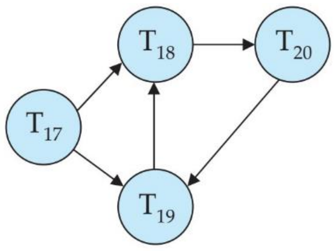
Another approach is the Lock Timeout-Based Schemes:
a transaction waits for a lock only for a specified amount of time; after that, the wait times out and the transaction is rolled back.
In this way, deadlocks are not possible, simple to implement but starvation is possible. Also, difficult to determine good value of the timeout interval.

▲Deadlock Detection [死锁检测]

Deadlocks can be described as a wait-for graph, which consists of a pair G = (V, E), V is a set of vertices (all the transactions in the system), E is a set of edges; each element is an ordered pair Ti ->Tj.
[死锁可以被描述为一个等待图，它由一对 G = (V, E)组成，V 是一组顶点(系统中的所有事务)，E 是一组边;每个元素都是一个有序对 Ti ->Tj]
If Ti -> Tj is in E, then there is a directed edge from Ti to Tj, implying that Ti is waiting for Tj to release a data item.
[如果 Ti -> Tj 在 E 中，那么有一条从 Ti 到 Tj 的有向边，意味着 Ti 正在等待 Tj 释放一个数据项]
When Ti requests a data item currently being held by Tj, then the edge (Ti ->Tj) is inserted in the wait-for graph. This edge is removed only when Tj is no longer holding a data item needed by Ti.
[当 Ti 请求 Tj 当前持有的数据项时，在等待图中插入边(Ti ->Tj)，只有当 Tj 不再持有 Ti 所需要的数据项时，才删除这条边]



Wait-for graph without a cycle



Wait-for graph with a cycle

The system is in a deadlock state if and only if the wait for graph has a cycle. Must invoke a deadlock detection algorithm periodically to look for cycles.

▲Deadlock Recovery

When deadlock is detected, three actions need to be taken:

[1] Some transaction will have to rolled back (made a victim) to break deadlock. Select the transaction as victim that will incur minimum cost. (损失最小化，选择回滚的事务)

[2] Rollback -- determine how far to roll back transaction: (选择回滚的方式)

Total rollback: Abort the transaction and then restart it.

More effective to roll back transaction only as far as necessary to break deadlock.

[3] Starvation happens if same transaction is always chosen as victim. (生成下标日志防止饥饿发生)

Include the number of rollbacks in the cost factor to avoid starvation.