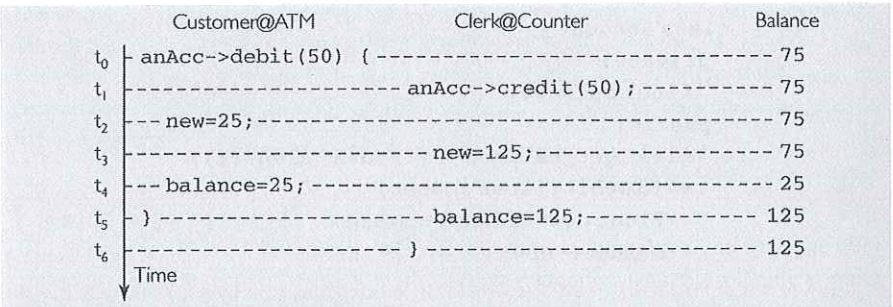
Example 11.9

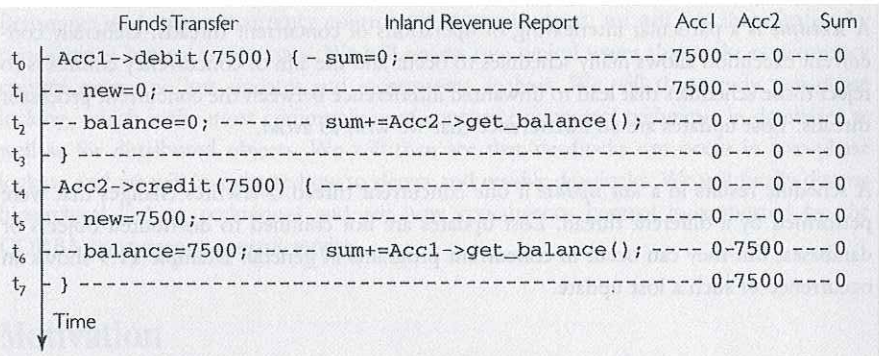
Concurrent Schedule Leading to Lost Updates



The above figure shows `credit` and `debit` operations that are performed concurrently on the same `Account` object. The column on the right-hand side traces the evolution of the `balance` attribute of the account. Our implementation stores the result of the subtraction and addition operations in the temporary variable `new`. Programmers might not write code like this, however most processors perform the addition and subtraction in this way and we have made that explicit for illustration purposes. The time passes from the top to the bottom. The schedule interleaves operations in such a way that the modification performed by the `debit` operation is overwritten by the `credit` operation; the update performed by `debit` is lost.

Example 11.10

Concurrent Schedule Leading to Inconsistent Analysis



Assume that a customer has two accounts. The first thread performs a funds transfer between the two accounts by debiting an amount of 7,500 from the first account and crediting it to the second account. The second thread adds the account balances for a report about the customer's assets to the Inland Revenue. The analysis reveals a result of a balance of 0, though the customer commands assets in the amount of 7,500. The reason for this inconsistent analysis is that the accounts were modified by the fund transfer thread while the sums were computed.

Transactions are serializable if the same result can be achieved by executing them in sequential order.

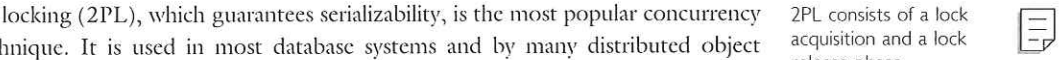
Two transactions are *serializable* if the same result can be achieved by executing them one after the other, that is in serial order. Serializability is the objective of every concurrency control technique. Concurrency control techniques that achieve serializability inhibit every non-serializable schedule by delaying operations that would cause non-serializability. It is rather complex for a concurrency control technique to reject every non-serializable and

admit every serializable schedule. For the sake of efficiency, concurrency control techniques often employ simplifications, which might also reject some serializable schedules. However, they never permit non-serializable schedules.

11.2.2 Two-Phase Locking

Two-phase locking (2PL), which guarantees serializability, is the most popular concurrency control technique. It is used in most database systems and by many distributed object systems. *Two-phase locking* is based on the idea that transactions ask the concurrency control manager for a lock for a shared resource; they *acquire* locks prior to using resources. In the realm of this book, resources are objects, but the mechanisms apply as well to tuples of relational databases or pages, the storage granularity of object databases. The concurrency control manager only *grants* the lock if the use of the object does not conflict with locks granted previously to concurrent transactions. If the transaction does not access an object any more, it will *release* the lock so that it can be acquired by other transactions.

2PL consists of a lock acquisition and a lock release phase.



Two-phase locking guarantees serializability.

Locking Profile

Two-phase locking demands that once a transaction has released a lock, it cannot acquire other locks. The two phases of 2PL are consequently referred to the *lock acquisition* and the *lock release* phases. If these phases are strictly distinguished, serializability is guaranteed by 2PL. For a proof of this theorem we refer to [Bernstein et al., 1987].

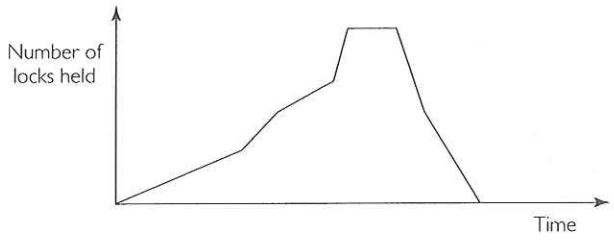


Figure 11.3
Typical Two-Phase Locking Profile

Figure 11.3 shows a typical locking profile of a transaction. The number of locks held by the transaction grows up to a certain maximum. The transaction can release the lock for an object only when it knows that it will no longer need the object and has acquired all the locks that it needs.

2PL never acquires a lock after it has started releasing locks.

It is desirable to release locks as early as possible so that concurrent transactions that need access to the object can acquire their lock. However, due to the complexity involved in deciding whether the it is safe to release a lock, transactions often retain locks to objects and only release them together at the end.

Locks

There are two perspectives on locking that have to be distinguished. The above locking profile showed the perspective of a transaction, which knows which locks it holds for the

objects that it wants to access. The concurrency control manager, on the other hand, has to decide which locks to grant. It has to base this decision on the locks that have already been granted to concurrent transactions for that object.

Binary locks on a resource can either be held or not held.

A very simplistic locking approach would be to consider locks as binary information where objects are either locked or not. We refer to these locks as *binary locks*. The implementation of `synchronized` methods and blocks in Java use such binary locks. If a Java thread tries to acquire a lock on an object that is already locked the concurrency control manager will force that thread to wait until the existing lock has been released.

Binary locks are overly restrictive.

This approach, however, restricts concurrency unnecessarily. Many transactions only read the state of an object, which does not interfere with other transactions that are reading the object as well. The concurrency control manager can permit multiple transactions to read the object's state, but it has to prevent a transaction that wishes to write to an object's state to proceed if other transactions are reading or writing to the object. To avoid unnecessary concurrency restrictions between transactions, concurrency control managers distinguish locks with different *modes*.

Lock modes determine operations that transactions can request from objects.

A *read lock* is acquired by a transaction that wishes to access the state of an object. A transaction that wishes to perform a modification to the state of an object will require a *write lock*. Following the principle of information hiding, the state of objects is often hidden and only exposed by operations. These operations therefore often also perform the lock acquisition and then locking can be kept transparent to clients.

Lock Compatibility

Lock compatibility matrices determine compatibility and conflicts between locking modes.

Given that there are different locking modes, the definition of a concurrency control scheme always includes the definition of compatibility between locks. Figure 11.4 shows the lock compatibility matrix for a concurrency control scheme that includes only read and write locks. This is the minimal lock compatibility matrix. More complicated ones are used in practice. The scheme defines that a read lock is compatible with another read lock, but incompatible with a write lock. Furthermore, write locks are incompatible with each other.

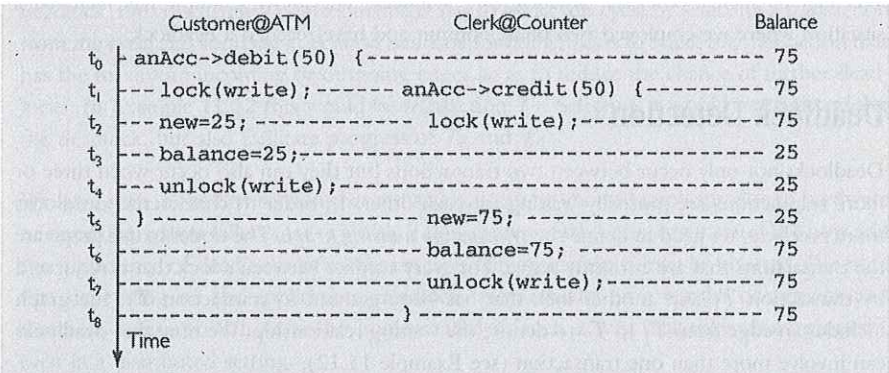
Figure 11.4
Minimal Lock Compatibility Matrix

	Read	Write
Read	+	–
Write	–	–

Given that concurrency control managers may grant multiple locks to different transactions, they have to keep track of which locks they have granted. This is necessary for them to have a basis for future decisions on lock acquisition requests. Hence concurrency control managers associate *locksets* with every shared object. The lockset will then include a lock for every transaction that has been granted access to an object in a particular mode.

The situation when access cannot be granted due to an incompatibility between the requested lock and a previously-granted lock is referred to as a *locking conflict*. The previously-granted locks that cause the conflict are referred to as *conflicting locks*. There are

Example 11.11
Preventing Lost Updates through Locking



The above figure continues Example 11.9 on Page 298. The difference between the two figures is that we now acquire a lock for every object that we access and release it when we do not need to access the object any more. Following this 2PL strategy, lost updates are prevented. At time t_2 the concurrency control manager detects a concurrency control conflict between the write lock that was granted to the first transaction at t_1 and the lock that is requested by the second transaction. This locking conflict is handled by delaying the second transaction and only granting the write lock at t_5, that is after the first transaction has released the conflicting lock. Hence, the use of locking delays the second transaction and avoids the lost update.

different options as to how a concurrency control manager handles locking conflicts. The first approach is to make the requesting transaction wait until the conflicting lock is released by the other transaction. The second approach is to return control to the requesting transaction indicating that the requested lock cannot be granted. The third approach, which is most commonly used in practice, is a combination of the two. The concurrency control manager returns control to the requesting transaction only when a certain amount of time has passed (time-out) and the conflicting lock has not been released.

11.2.3 Deadlocks

Forcing a transaction that requests a conflicting lock to wait solves the problem of lost updates and inconsistent analysis; it does however, introduce a new class of *liveness* problem.

Liveness refers to the property of a concurrent system that something desirable will eventually happen.

It could happen, that while the transaction is waiting for the conflicting lock to be released, another transaction comes along that obtains the lock. *Starvation* problems can be solved by the concurrency control manager. It has to ensure fairness, which means that every transaction will eventually obtain the resources that it wants.

A transaction starves if it waits indefinitely to obtain a shared resource.

A different class of liveness problem is deadlocks. Transactions may request locks for more than one object. It may then happen that a transaction T_1 has a lock on one object O_1 and waits for a conflicting lock on a second object O_2 to be released. If this conflicting lock is held by another transaction T_2 and that transaction requests a lock on O_1 then the two transactions are mutually waiting for each other's locks to be released.

Transactions that are waiting for each other to release locks are in a deadlock.