2PL is not deadlock-free!

We refer to such a blockage situation as a *deadlock*. The above example has sketched a situation where we employed two-phase commit and have reached a deadlock.

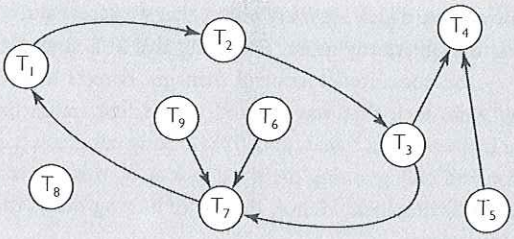
Deadlock Detection

Transaction waiting graphs record the 'waits-for' relationship between transactions.

Deadlocks not only occur between two transactions but they can also occur when three or more transactions are mutually waiting for each other. In order to characterize deadlocks more precisely, we need to define the *transaction waiting graph*. The nodes in this graph are the transactions that are currently active. For every conflict between a lock that is requested by transaction T_1 and another lock that has been granted to transaction T_2, the graph includes an edge from T_1 to T_2 to denote the waiting relationship. We note that deadlocks can involve more than one transaction (see Example 11.12).

We can find deadlocks by detecting cycles in waiting graphs.

Example 11.12
Deadlock Involving More Than Two Transactions



In this transaction waiting graph, the four transactions T_1, T_2, T_3 and T_7 are in a deadlock.

In order to detect the occurrence of deadlocks efficiently, the concurrency control manager has to maintain an up-to-date representation of this transaction waiting graph. It has to create a new node for every transaction that is started. It has to delete the node for every transaction that is committed or aborted. The concurrency control manager then needs to record every locking conflict that it detects by inserting an edge into the graph. Furthermore whenever a transaction resolves a locking conflict by releasing a conflicting lock the respective edge has to be deleted.

Deadlock detection is linear in relation to the number of concurrent transactions.

The concurrency control manager can use the transaction waiting graph to detect deadlocks. This is done by checking for cycles in the graph. The complexity of this operation is linear in relation to the number of nodes, that is the number of transactions. This complexity is so low that the concurrency control manager can perform this operation on a regular basis, for example, whenever a new locking conflict has been recognized.

Deadlock Resolution

Deadlocks are resolved by aborting involved transactions.

When a deadlock occurs, the transactions involved in the deadlock cannot proceed any more and therefore they cannot resolve the deadlock themselves. The isolation property of transactions, in fact, means that they do not know about the other transactions that are involved in the deadlock. Deadlocks therefore have to be resolved by the concurrency control manager. The transactions that participate in a conflict are represented in the transaction waiting graph as those nodes that are connected by the cycle. In order to resolve the

deadlock, the concurrency control manager has to break this cycle by selecting a transaction from the cycle and aborting it. A good heuristic for doing this is to select the transaction that has the maximum incoming or outgoing edges so as to reduce the chance of further deadlocks. In Example 11.12 this would be transaction T_7. Selecting it would not only resolve the deadlock, but also facilitate progress of T_6 and T_9.

We have seen in the first section that transactions can be aborted implicitly due to hardware or operating system failures, or explicitly if they cannot reach a consistent state. We now add the occurrence of deadlocks as another reason why transactions may be aborted. We note that depending on the application, deadlocks may actually be a regular occurrence and therefore transactions have to be implemented in such a way that they abort properly, even in a distributed setting.

11.2.4 Hierarchical Locking

2PL is inefficient if transactions need large number of objects.

While two-phase locking is fully appropriate to transactions that only access a few objects, it is inefficient for those transactions that have to access a high number of objects. At the close of business of a bank branch, for example, the branch has to perform a transaction summing up all balances of all account objects managed at this branch in order to check whether the bank's overall accounts balance. This transaction might have to access many thousands of account objects. Inefficiencies with accessing large numbers of objects arise for two reasons: transactions have to acquire and release a lock for every object; if there are many concurrent transactions that have a similar profile, the chances of deadlock occurrence are high.

Locking Granularity

We can observe that objects that are accessed by transactions concurrently are often contained in more coarse-grained composite objects. For example,

1. files are contained in directories, which are contained in other directories;
2. relational databases contain a set of tables, which contain a set of tuples, which contain a set of attributes; or
3. distributed composite objects may act as containers for component objects, which may again contain other objects.

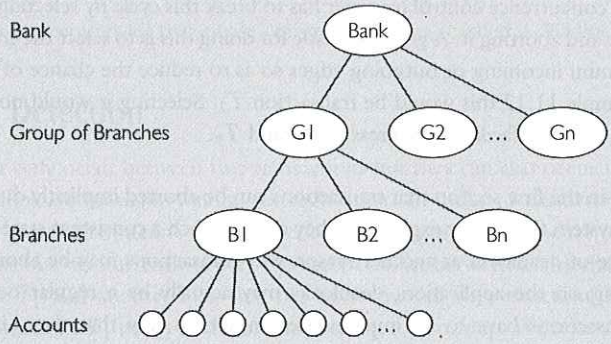
A common access pattern for transactions is to visit all or a large subset of the objects that are contained in a container. The concurrency control manager can save a lot of effort if it manages to exploit these containment hierarchies for concurrency control purposes.

Hierarchical Locking Protocol

Hierarchical locking protocols lock all component objects with one locking operation applied to the composite object.

The basic idea of the *hierarchical locking protocol* is to lock all objects that are contained in a composite object simply by locking the composite object. Then a large number of objects can be locked or unlocked in a single locking operation. Nevertheless, some transactions will still need to be able to lock single objects. It is necessary for these transaction to also record at the more coarse-grained objects that they are locking some objects contained in the

Example 11.13
Different Locking Granularities in Financial Transactions



Consider the containment hierarchy of account objects of a bank as shown above. The bank can be seen as a set of groups of branches, each group has a set of branches, and each branch manages its own account objects.

composite objects. To be able to record this, concurrency control protocols use intention locks.

An intention lock on a composite object highlights that a transaction has acquired a real lock on a component.

An *intention lock* is a lock acquired for a composite object before a transaction requests a real lock for an object that is contained in the composite object. Intention locks are used to signal to those transactions that wish to lock the entire composite object that some other transaction currently has locks for objects that are contained in the composite object. Hence, the complexity of the decision about whether or not a concurrency control manager can grant a lock on a composite object is independent of the number of objects contained in the composite object. In practice, concurrency control schemes will use different modes of intention locks, for the same reason as there are different modes for real locks.

The hierarchical locking protocol locks every composite object of a component object.

In order to be able to read or write an object, it is necessary in the hierarchical locking protocol to acquire intention locks on all composite objects in which the object is contained. This is done from the root of the containment hierarchy to the object that is to be locked. In Example 11.13, in order to obtain a write lock on a particular account object, a transaction obtains intention write locks on the bank, the group of branches and the branch object that the object is contained in.

Figure 11.5
Lock Compatibility for Hierarchical Locking

	Intention Read	Read	Intention Write	Write
Intention Read	+	+	+	−
Read	+	+	−	−
Intention Write	+	−	+	−
Write	−	−	−	−

Figure 11.5 shows the lock compatibility matrix between the locking modes that are used for hierarchical locking. This matrix extends the minimal lock matrix that we presented in Figure 11.4 with two new locking modes: intention read and intention write. The *intention read* locks indicate that some transaction has acquired or is about to acquire read locks on the

objects in the composite object. The *intention write* locks indicate that some other transaction has or is about to acquire write locks on the objects in the composite object. Intention read and intention write locks are compatible among themselves because they do not actually correspond to any locks. An intention read lock is also compatible with a read lock because a read access to all elements of the composite object can be done while some other transactions are accessing components of that object. However, an intention read lock is incompatible with a write lock because it is not possible to modify every element of the composite object while some other transaction is reading the state of an object of the composite. Likewise, it is not possible to write all objects while some other transaction is writing some objects.

Example 11.14
Hierarchical Locking of Bank Accounts

Let us assume that there are three transactions. T_1 produces a sum of the balances of a particular customer, whose accounts are managed at branch B1. T_2 performs a funds transfer between two accounts held at branch B1. T_3 produces a sum of the balances of all accounts held at a group of branches to which B1 belongs. T_3 locks the root note in intention read and the group of branches in read mode. T_1 locks the root node, the group of branches and the branch objects in intention read and the account objects of the customer in read mode. T_2 locks the root, the group of branches and the branch object in intention write mode and the two account objects in write mode. The hierarchical lock compatibility matrix determines that T_1 can be performed at the same time as T_3. T_1 and T_2 can be performed concurrently if the set of account objects is disjoint. T_2 and T_3, however, cannot be performed at the same time.

Hierarchical locking has enabled us to lock all account objects of a group of branches with two locking operations. The overhead is that for every individual object, we also have to use intention locks on every composite object in which the object is contained. This overhead is justifiable if there are some transactions that access a large number of objects and the composition hierarchy is not very deep.

11.2.5 The CORBA Concurrency Control Service

Now that we have seen the principles of two-phase and hierarchical locking, let us now review how they are applied to distributed objects in practice. To do so, we review the CORBA Concurrency Control service. It was adopted in 1994 as part of the CORBAservices RFP2 and specifies standard interfaces for controlling the concurrency between distributed CORBA objects.

Lock Compatibility

The Concurrency Control service supports hierarchical two-phase locking of CORBA objects. Figure 11.6 shows the lock compatibility matrix.

In addition to the locking modes that we have discussed already, the service defines upgrade locks. *Upgrade locks* are used when a transaction initially needs read access to an object, but already knows that it will at a later point also need to acquire a write lock for the object.

Upgrade locks are not compatible with themselves.