Table 18.1 Examples of threats.

Threat	Theft and fraud	Loss of confidentiality	Loss of privacy	Loss of integrity	Loss of availability
Using another person's means of access	✓	✓	✓		
Unauthorized amendment or copying of data	✓			✓	
Program alteration	✓			✓	✓
Inadequate policies and procedures that allow a mix of confidential and normal output	✓	✓	✓		
Wire tapping	✓	✓	✓		
Illegal entry by hacker	✓	✓	✓		
Blackmail	✓	✓	✓		
Creating 'trapdoor' into system	✓	✓	✓		
Theft of data, programs, and equipment	✓	✓	✓		✓
Failure of security mechanisms, giving greater access than normal		✓	✓	✓	
Staff shortages or strikes				✓	✓
Inadequate staff training		✓	✓	✓	✓
Viewing and disclosing unauthorized data	✓	✓	✓		
Electronic interference and radiation				✓	✓
Data corruption owing to power loss or surge				✓	✓
Fire (electrical fault, lightning strike, arson), flood, bomb				✓	✓
Physical damage to equipment				✓	✓
Breaking cables or disconnection of cables				✓	✓
Introduction of viruses				✓	✓

The extent that an organization suffers as a result of a threat's succeeding depends upon a number of factors, such as the existence of countermeasures and contingency plans. For example, if a hardware failure occurs corrupting secondary storage, all processing activity must cease until the problem is resolved. The recovery will depend upon a number of factors, which include when the last backups were taken and the time needed to restore the system.

An organization needs to identify the types of threat it may be subjected to and initiate appropriate plans and countermeasures, bearing in mind the costs of implementing them. Obviously, it may not be cost-effective to spend considerable time, effort, and money on potential threats that may result only in minor inconvenience. The organization's business may also influence the types of threat that should be considered, some of which may be rare. However, rare events should be taken into account, particularly if their impact would be significant. A summary of the potential threats to computer systems is represented in Figure 18.1.

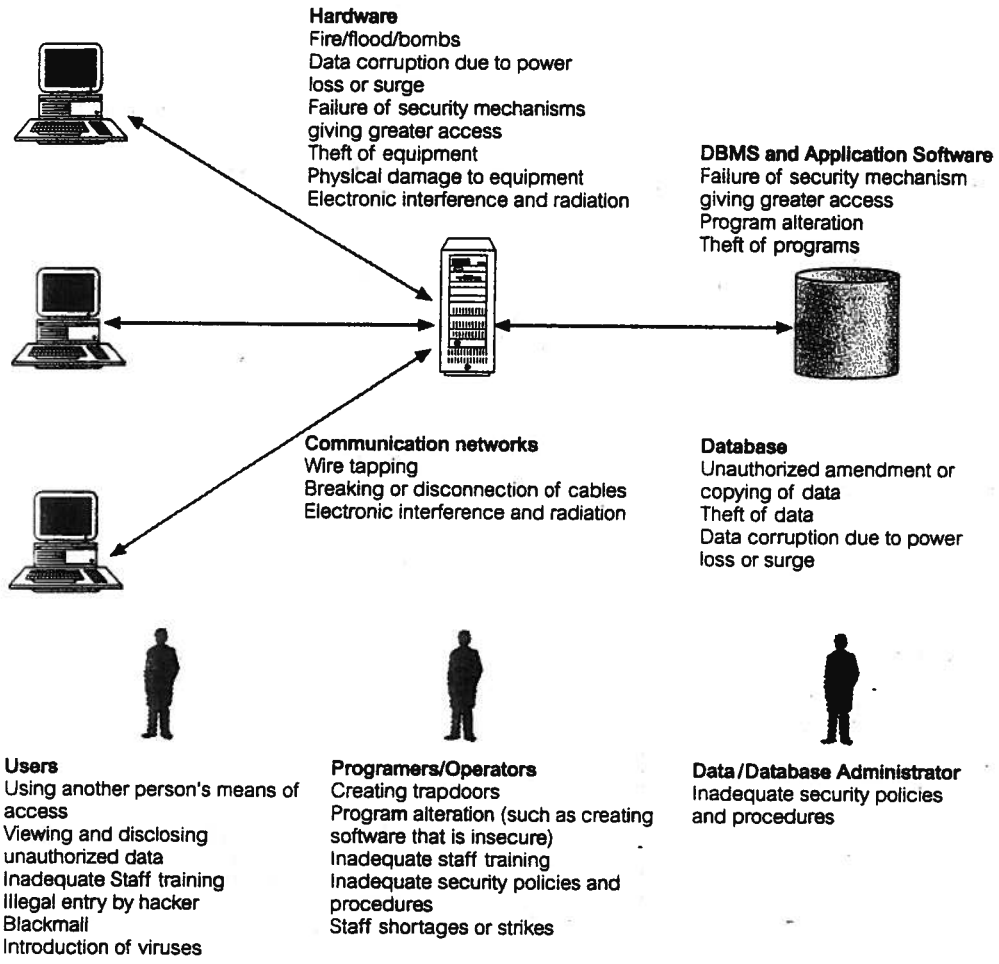


Figure 18.1 Summary of potential threats to computer systems.

Countermeasures – Computer-Based Controls

18.2

The types of countermeasure to threats on computer systems range from physical controls to administrative procedures. Despite the range of computer-based controls that are available, it is worth noting that, generally, the security of a DBMS is only as good as that of the operating system, owing to their close association. Representation of a typical multi-user computer environment is shown in Figure 18.2. In this section we focus on the following computer-based security controls for a multi-user environment (some of which may not be available in the PC environment):

5:

T

- authorization
- views
- backup and recovery
- integrity
- encryption
- RAID technology.

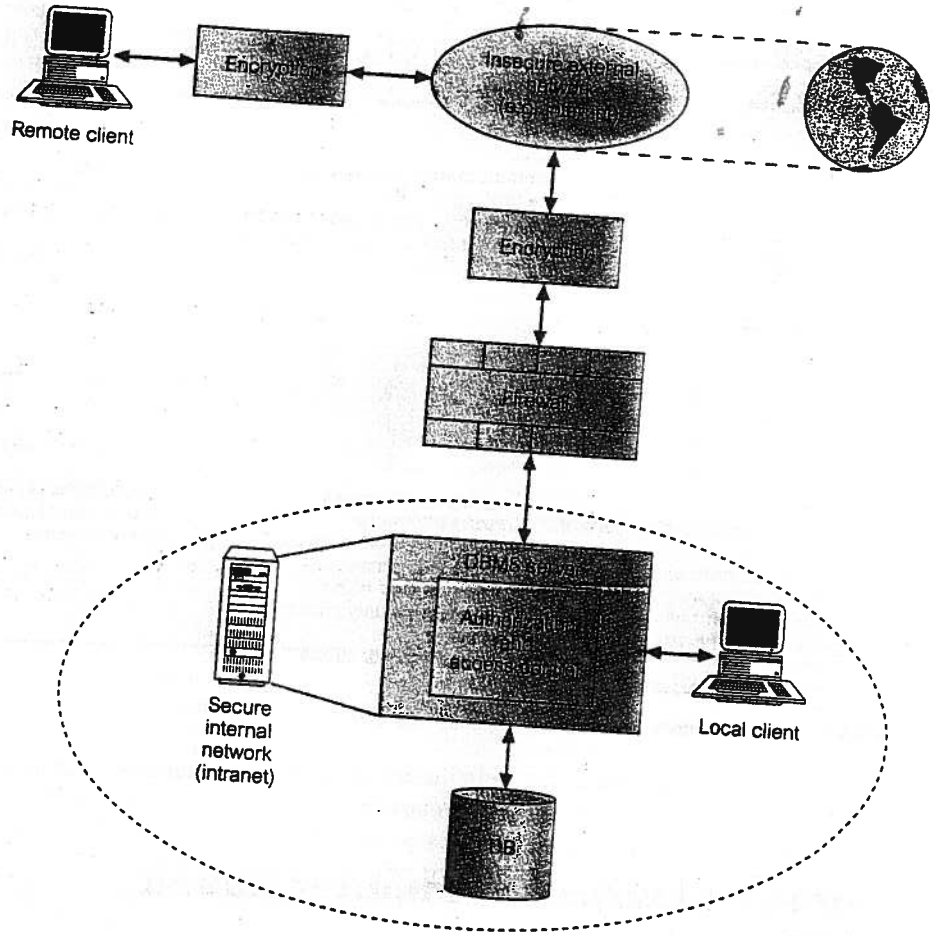


Figure 18.2 Representation of a typical multi-user computer environment.

18.2.1 Authorization

Authorization The granting of a right or privilege that enables a subject to have legitimate access to a system or a system's object.

Table 18.3 Access control matrix.

User identifier	propertyNo	type	price	ownerNo	staffNo	branchNo	Query row limits
Sales	0001	0001	0001	0000	0000	0000	15
SG37	0101	0101	0111	0101	0111	0000	100
SG5	1111	1111	1111	1111	1111	1111	none

propertyNo, type, and ownerNo attributes and Select, Update, and Insert privileges (shown as 0001 + 0010 + 0100 = 0111) for the price and staffNo attributes, with a limit of 100 rows for any query result set. Finally, user SG5 (Susan Brand) has Select, Update, Insert, and Delete privileges (shown as 0001 + 0010 + 0100 + 1000 = 1111), in other words All privileges for all attributes, with no limit set on the number of rows for any query result set.

DBMSs use similar matrices to implement access control, although the precise details of implementation vary from one system to another. On some DBMSs, a user has to tell the system under which identifier he or she is operating, especially if the user is a member of more than one group. It is essential to become familiar with the available authorization and other control mechanisms provided by the DBMS, particularly where priorities may be applied to different authorization identifiers and where privileges can be passed on. This will enable the correct types of privileges to be granted to users based on their requirements and those of the application programs that many of them will use.

Views (Subschemas) 18.2.2

View A view is the dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a *virtual relation* that does not actually exist in the database, but is produced upon request by a particular user, at the time of request.

The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users. The user is not aware of the existence of any attributes or rows that are missing from the view. A view can be defined over several relations with a user being granted the appropriate privilege to use it, but not to use the base relations. In this way, using a view is more restrictive than simply having certain privileges granted to a user on the base relation(s). We discussed views in detail in Sections 3.4 and 6.4.

Backup and Recovery 18.2.3

Backup The process of periodically taking a copy of the database and log file (and possibly programs) on to offline storage media.

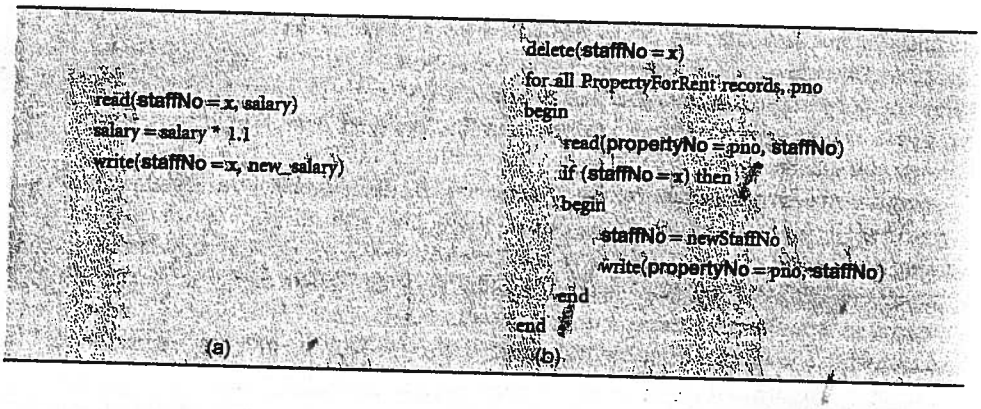


Figure 19.1 Example transactions.

Staff (staffNo, fName, lName, position, sex, DOB, salary, branchNo)
PropertyForRent (propertyNo, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

A simple transaction against this database is to update the salary of a particular member of staff given the staff number, x. At a high level, we could write this transaction as shown in Figure 19.1(a). In this chapter we denote a database read or write operation on a data item x as read(x) or write(x). Additional qualifiers may be added as necessary; for example, in Figure 19.1(a), we have used the notation read(staffNo = x, salary) to indicate that we want to read the data item salary for the tuple with primary key value x. In this example, we have a **transaction** consisting of two database operations (read and write) and a non-database operation (salary = salary*1.1).

A more complicated transaction is to delete the member of staff with a given staff number x, as shown in Figure 19.1(b). In this case, as well as having to delete the tuple in the Staff relation, we also need to find all the PropertyForRent tuples that this member of staff managed and reassign them to a different member of staff, *newStaffNo* say. If all these updates are not made, referential integrity will be lost and the database will be in an **inconsistent state**: a property will be managed by a member of staff who no longer exists in the database.

A transaction should always transform the database from one consistent state to another, although we accept that consistency may be violated while the transaction is in progress. For example, during the transaction in Figure 19.1(b), there may be some moment when one tuple of PropertyForRent contains the new *newStaffNo* value and another still contains the old one, x. However, at the end of the transaction, all necessary tuples should have the new *newStaffNo* value.

A transaction can have one of two outcomes. If it completes successfully, the transaction is said to have **committed** and the database reaches a new consistent state. On the other hand, if the transaction does not execute successfully, the transaction is **aborted**. If a transaction is aborted, the database must be restored to the consistent state it was in before the transaction started. Such a transaction is **rolled back** or **undone**. A committed transaction cannot be aborted. If we decide that the committed transaction was a mistake, we must perform another **compensating transaction** to reverse its effects (as we discuss in Section 19.4.2). However, an aborted transaction that is rolled back can be restarted later and, depending on the cause of the failure, may successfully execute and commit at that time.

Figure 19.4
The lost update problem.

Time	T_1	T_2	bal_x
t_1			100
t_2		begin_transaction	100
t_3	begin_transaction	read(bal_x)	100
t_4	read(bal_x)	$bal_x = bal_x + 100$	100
t_5	$bal_x = bal_x - 10$	write(bal_x)	200
t_6	write(bal_x)	commit	90
	commit		90

Figure 19.5
The uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1			100
t_2		begin_transaction	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	read(bal_x)	write(bal_x)	200
t_6	$bal_x = bal_x - 10$		200
t_7	write(bal_x)	rollback	100
t_8	commit		190
			190

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	$bal_x = bal_x - 10$	sum = sum + bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum = sum + bal_y	90	50	25	150
t_7	$bal_z = bal_z + 10$		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum = sum + bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

Figure 19.6
The inconsistent analysis problem.

The problems described in Examples 19.1–19.3 resulted from the mismanagement of concurrency, which left the database in an inconsistent state in the first two examples and presented the user with the wrong result in the third. Serial execution prevents such problems occurring. No matter which serial schedule is chosen, serial execution never leaves the database in an inconsistent state, so every serial execution is considered correct, although different results may be produced. The objective of **serializability** is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.

If a set of transactions executes concurrently, we say that the (nonserial) schedule is correct if it *produces the same results as some serial execution*. Such a schedule is called **serializable**. To prevent inconsistency from transactions interfering with one another, it is essential to guarantee serializability of concurrent transactions. In serializability, the ordering of read and write operations is important:

- If two transactions only read a data item, they do not conflict and order is not important.
- If two transactions either read or write completely separate data items, they do not conflict and order is not important.
- If one transaction writes a data item and another either reads or writes the same data item, the order of execution is important.

Consider the schedule S_1 shown in Figure 19.7(a) containing operations from two concurrently executing transactions T_7 and T_8. Since the write operation on bal_x in T_8 does not conflict with the subsequent read operation on bal_y in T_7, we can change the order of these operations to produce the equivalent schedule S_2 shown in Figure 19.7(b). If we also now change the order of the following non-conflicting operations, we produce the equivalent serial schedule S_3 shown in Figure 19.7(c):

- Change the order of the write(bal_x) of T_8 with the write(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the read(bal_y) of T_7.
- Change the order of the read(bal_x) of T_8 with the write(bal_y) of T_7.

Figure 19.7 Equivalent schedules: (a) nonserial schedule S_1; (b) nonserial schedule S_2 equivalent to S_1; (c) serial schedule S_3, equivalent to S_1 and S_2.

(a)

Time	T_7	T_8
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	write(bal_x)	
t_4		begin_transaction
t_5		read(bal_x)
t_6		write(bal_x)
t_7	read(bal_y)	
t_8	write(bal_y)	
t_9	commit	
t_{10}		read(bal_y)
t_{11}		write(bal_y)
t_{12}		commit

(b)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
	begin_transaction
	read(bal_x)
read(bal_y)	
	write(bal_x)
write(bal_y)	
commit	
	read(bal_y)
	write(bal_y)
	commit

(c)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
read(bal_y)	
write(bal_y)	
commit	
	begin_transaction
	read(bal_x)
	write(bal_x)
	read(bal_y)
	write(bal_y)
	commit

Schedule S_3 is a serial schedule and, since S_1 and S_2 are equivalent to S_3, S_1 and S_2 are serializable schedules.

This type of serializability is known as **conflict serializability**. A conflict serializable schedule orders any conflicting operations in the same way as some serial execution. Under the **constrained write rule** (that is, a transaction updates a data item based on its old value, which is first read by the transaction), a **precedence** (or **serialization**) **graph** can be produced to test for conflict serializability. For a schedule S, a precedence graph is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

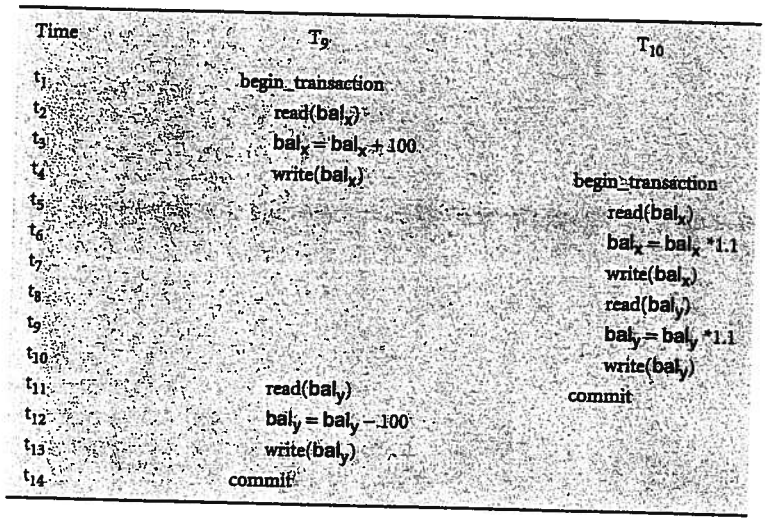
- Create a node for each transaction.
- Create a directed edge $T_i \rightarrow T_j$, if T_j reads the value of an item written by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been read by T_i.
- Create a directed edge $T_i \rightarrow T_j$, if T_j writes a value into an item after it has been written by T_i.

If an edge $T_i \rightarrow T_j$ exists in the precedence graph for S, then in any serial schedule S′ equivalent to S, T_i must appear before T_j. If the precedence graph contains a cycle the schedule is not conflict serializable.

Example 19.4 Non-conflict serializable schedule

Consider the two transactions shown in Figure 19.8. Transaction T_9 is transferring £100 from one account with balance bal_x to another account with balance bal_y, while T_{10} is increasing the balance of these two accounts by 10%. The precedence graph for this schedule, shown in Figure 19.9, has a cycle and so is not conflict serializable.

Figure 19.8
Two concurrent update transactions.



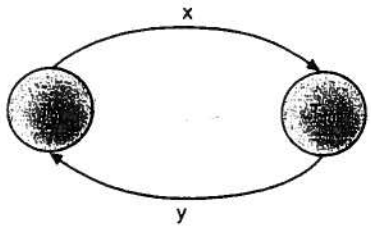


Figure 19.9
Precedence graph for Figure 19.8.

View serializability

There are several other types of serializability that offer less stringent definitions of schedule equivalence than that offered by conflict serializability. One less restrictive definition is called **view serializability**. Two schedules S_1 and S_2 consisting of the same operations from n transactions $T_1, T_2, \ldots, T_n$ are view equivalent if the following three conditions hold:

- For each data item x, if transaction T_i reads the initial value of x in schedule S_1, then transaction T_i must also read the initial value of x in schedule S_2.
- For each read operation on data item x by transaction T_i in schedule S_1, if the value read by x has been written by transaction T_j, then transaction T_i must also read the value of x produced by transaction T_j in schedule S_2.
- For each data item x, if the last write operation on x was performed by transaction T_i in schedule S_1, the same transaction must perform the final write on data item x in schedule S_2.

A schedule is view serializable if it is view equivalent to a serial schedule. Every conflict serializable schedule is view serializable, although the converse is not true. For example, the schedule shown in Figure 19.10 is view serializable, although it is not conflict serializable. In this example, transactions T_{12} and T_{13} do not conform to the constrained write rule; in other words, they perform *blind writes*. It can be shown that any view serializable schedule that is not conflict serializable contains one or more blind writes.

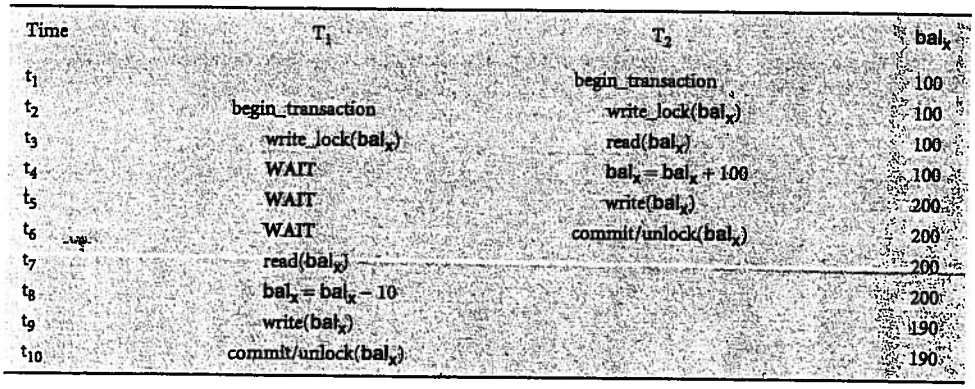
Time	T_{11}	T_{12}	T_{13}
t_1	begin_transaction		
t_2	read(bal_x)		
t_3		begin_transaction	
t_4		write(bal_x)	
t_5		commit	
t_6	write(bal_x)		
t_7	commit		
t_8			begin_transaction
t_9			write(bal_x)
t_{10}			commit

Figure 19.10
View serializable schedule that is not conflict serializable.

If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item. Downgrading can take place only during the shrinking phase. We now look at how two-phase locking is used to resolve the three problems identified in Section 19.2.1.

Example 19.6 Preventing the lost update problem using 2PL

A solution to the lost update problem is shown in Figure 19.11. To prevent the lost update problem occurring, T_2 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database. When T_1 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_2, the request is not immediately granted and T_1 has to **wait** until the lock is released by T_2. This occurs only once the commit of T_2 has been completed.



Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	write_lock(bal_x)	100
t_3	write_lock(bal_x)	read(bal_x)	100
t_4	WAIT	$bal_x = bal_x + 100$	100
t_5	WAIT	write(bal_x)	200
t_6	WAIT	commit/unlock(bal_x)	200
t_7	read(bal_x)		200
t_8	$bal_x = bal_x - 10$		200
t_9	write(bal_x)		190
t_{10}	commit/unlock(bal_x)		190

Figure 19.11 Preventing the lost update problem.

Example 19.7 Preventing the uncommitted dependency problem using 2PL

A solution to the uncommitted dependency problem is shown in Figure 19.12. To prevent this problem occurring, T_4 first requests an exclusive lock on bal_x. It can then proceed to read the value of bal_x from the database, increment it by £100, and write the new value back to the database. When the rollback is executed, the updates of transaction T_4 are undone and the value of bal_x in the database is returned to its original value of £100. When T_3 starts, it also requests an exclusive lock on bal_x. However, because the data item bal_x is currently exclusively locked by T_4, the request is not immediately granted and T_3 has to wait until the lock is released by T_4. This occurs only once the rollback of T_4 has been completed.

Figure 19.12 Preventing the uncommitted dependency problem.

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		write_lock(bal_x)	100
t_3		read(bal_x)	100
t_4	begin_transaction	$bal_x = bal_x + 100$	100
t_5	write_lock(bal_x)	write(bal_x)	200
t_6	WAIT	rollback/unlock(bal_x)	100
t_7	read(bal_x)		100
t_8	$bal_x = bal_x - 10$		100
t_9	write(bal_x)		90
t_{10}	commit/unlock(bal_x)		90

Example 19.8 Preventing the inconsistent analysis problem using 2PL

A solution to the inconsistent analysis problem is shown in Figure 19.13. To prevent this problem occurring, T_5 must precede its reads by exclusive locks, and T_6 must precede its reads with shared locks. Therefore, when T_5 starts it requests and obtains an exclusive lock on bal_x. Now, when T_6 tries to share lock bal_x the request is not immediately granted and T_6 has to wait until the lock is released, which is when T_5 commits.

Figure 19.13 Preventing the inconsistent analysis problem.

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum = 0	100	50	25	0
t_3	write_lock(bal_x)		100	50	25	0
t_4	read(bal_x)	read_lock(bal_x)	100	50	25	0
t_5	$bal_x = bal_x - 10$	WAIT	100	50	25	0
t_6	write(bal_x)	WAIT	90	50	25	0
t_7	write_lock(bal_z)	WAIT	90	50	25	0
t_8	read(bal_z)	WAIT	90	50	25	0
t_9	$bal_z = bal_z + 10$	WAIT	90	50	25	0
t_{10}	write(bal_z)	WAIT	90	50	35	0
t_{11}	commit/unlock(bal_x, bal_z)	WAIT	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum = sum + bal_x	90	50	35	90
t_{14}		read_lock(bal_y)	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum = sum + bal_y	90	50	35	140
t_{17}		read_lock(bal_z)	90	50	35	140
t_{18}		read(bal_z)	90	50	35	140
t_{19}		sum = sum + bal_z	90	50	35	175
t_{20}		commit/unlock(bal_x, bal_y, bal_z)	90	50	35	175

It can be proved that if *every* transaction in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable (Eswaran *et al.*, 1976). However, while the two-phase locking protocol guarantees serializability, problems can occur with the interpretation of when locks can be released, as the next example shows.

Example 19.9 Cascading rollback

Consider a schedule consisting of the three transactions shown in Figure 19.14, which conforms to the two-phase locking protocol. Transaction T_{14} obtains an exclusive lock on bal_x then updates it using bal_y, which has been obtained with a shared lock, and writes the value of bal_x back to the database before releasing the lock on bal_x. Transaction T_{15} then obtains an exclusive lock on bal_x, reads the value of bal_x from the database, updates it, and writes the new value back to the database before releasing the lock. Finally, T_{16} share locks bal_x and reads it from the database. By now, T_{14} has failed and has been rolled back. However, since T_{15} is dependent on T_{14} (it has read an item that has been updated by T_{14}), T_{15} must also be rolled back. Similarly, T_{16} is dependent on T_{15}, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **cascading rollback**.

Time	T_{14}	T_{15}	T_{16}
t_1	begin_transaction		
t_2	write_lock(bal_x)		
t_3	read(bal_x)		
t_4	read_lock(bal_y)		
t_5	read(bal_y)		
t_6	$bal_x = bal_y + bal_x$		
t_7	write(bal_x)		
t_8	unlock(bal_x)	begin_transaction	
t_9		write_lock(bal_x)	
t_{10}		read(bal_x)	
t_{11}		$bal_x = bal_x + 100$	
t_{12}		write(bal_x)	
t_{13}		unlock(bal_x)	
t_{14}			
t_{15}	rollback		
t_{16}			begin_transaction
t_{17}			read_lock(bal_x)
t_{18}		rollback	
t_{19}			rollback

Figure 19.14 Cascading rollback with 2PL.