It is sufficient to cover Sections 18.1, 18.5, 18.6, and 18.7, and possibly 18.3.2, if the main emphasis is on introducing the concurrency control techniques that are used most often in practice. The other techniques are mainly of theoretical interest.

18.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 18.1.1 we discuss the nature and types of locks. Then, in Section 18.1.2 we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 18.1.3 we discuss two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled.

18.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple but restrictive and so are not used in practice. Then we discuss shared/exclusive locks, which provide more general locking capabilities and are used in practical database locking schemes. In Section 18.3.2 we describe a certify lock and show how it can be used to improve performance of locking protocols.

Binary Locks. A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested. We refer to the current value (or state) of the lock associated with item X as **lock(X)**.

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a **lock_item(X)** operation. If LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock_item(X)** operation, which sets LOCK(X) to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(X) and unlock_item(X) operations is shown in Figure 18.1.

Notice that the lock_item and unlock_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no

```
lock_item(X):
B: if LOCK(X) = 0              (* item is unlocked *)
       then LOCK(X) ← 1        (* lock the item *)
   else
       begin
       wait (until LOCK(X) = 0
           and the lock manager wakes up the transaction);
       go to B
       end;

unlock_item(X):
   LOCK(X) ← 0;                (* unlock the item *)
   if any transactions are waiting
       then wakeup one of the waiting transactions;
```

Figure 18.1
Lock and unlock operations for binary locks.

interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 18.1, the wait command within the lock_item(X) operation is usually implemented by putting the transaction on a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed on the same queue. Hence, the wait command is considered to be outside the lock_item operation.

Notice that it is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain only these records for the items that are currently locked in a **lock table**, which could be organized as a hash file. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.[1]
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

1. This rule may be removed if we modify the lock_item (X) operation in Figure 18.1 so that if the item is currently locked *by the requesting transaction*, the lock is granted.

These rules can be enforced by the lock manager module of the DBMS. **Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to hold the lock on item X. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.**

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for *reading purposes only*. However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: *read-locked, write-locked,* or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have four fields:<Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(X)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on X. If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. The three operations read_lock(X), write_lock(X), and unlock(X) are described in Figure 18.2.[2] As before, each of the three operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed on a waiting queue for the item.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.[3]

2. These algorithms do not allow *upgrading* or *downgrading* of locks, as described later in this section. The reader can extend the algorithms to allow these additional operations.

3. This rule may be relaxed to allow a transaction to unlock an item, then lock it again later.

```
read_lock(X):

B: if LOCK(X) = "unlocked"
       then  begin LOCK(X) ← "read-locked";
             no_of_reads(X) ← 1
             end
   else if LOCK(X) = "read-locked"
       then no_of_reads(X) ← no_of_reads(X) + 1
   else  begin
         wait (until LOCK(X) = "unlocked"
            and the lock manager wakes up the transaction);
         go to B
         end;

write_lock(X):

B: if LOCK(X) = "unlocked"
       then LOCK(X) ← "write-locked"
   else  begin
         wait (until LOCK(X) = "unlocked"
            and the lock manager wakes up the transaction);
         go to B
         end;

unlock (X):

   if LOCK(X) = "write-locked"
      then  begin LOCK(X) ← "unlocked";
            wakeup one of the waiting transactions, if any
            end
   else it LOCK(X) = "read-locked"
      then  begin
            no_of_reads(X) ← no_of_reads(X) − 1;
            if no_of_reads(X) = 0
               then  begin LOCK(X) = "unlocked";
                     wakeup one of the waiting transactions, if any
                     end
            end;
```

Figure 18.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

4. A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. This rule may be relaxed, as we discuss shortly.
5. A transaction *T* will not issue a write_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*. This rule may be relaxed, as we discuss shortly.

6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read_lock(X) and write_lock(X) operations in Figure 18.2 must be changed appropriately. We leave this as an exercise for the reader.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 18.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 18.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 18.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best known protocol, two-phase locking, is described in the next section.

18.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding** or **growing** (first) phase, during which new locks on items can be acquired but none can be released; and a **shrinking** (second) phase, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Transactions T_1 and T_2 of Figure 18.3(a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2', as

4. This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 25).