problems we have two tools. The first is the scheduler, a portion of the database system that arbitrates between conflicting requests. We saw, for example, how a first-come, first-serve scheduler can eliminate livelock. A scheduler can also handle deadlocks and nonserializability by causing one or more transactions to restart, undoing all their actions so far. We shall consider restart of transactions in Section 11.6.

Another approach to handling deadlock and nonserializability is to use one or more protocols, that all transactions must follow. A protocol, in its most general sense, is simply a restriction on the sequences of steps that a transaction may perform. For example, the deadlock-avoiding strategy of requesting locks on items in some fixed order is a protocol. Much of what follows in this chapter concerns the development of protocols that guarantee serializability.

11.2 A SIMPLE TRANSACTION MODEL

Let us begin by introducing what is undoubtedly the simplest model of transactions that still enables us to talk about serializability. In this model, a transaction is viewed as a sequence of lock and unlock statements. Each item locked must subsequently be unlocked. Between a step LOCK A and the next UNLOCK A, a transaction is said to hold a lock on A. We assume a transaction does not try to lock an item if it currently holds a lock on that item, nor does it try to unlock an item on which it does not currently hold a lock.

We further assume that whenever a transaction locks on item A it changes the value of A, and the value that A has when unlocked is essentially unique, in the sense that if v_1 and v_2 are two values A may have before the LOCK A step, then the values held by A after UNLOCK A are always different in the two cases, provided v_1 ≠ v_2.

A more formal way to look at the behavior of transactions is to associate with each pair LOCK A and its following UNLOCK A, a unique function f. Note that one transaction may have more than one such pair for a given A, since, although it is not generally a good idea, we may lock and unlock the same item more than once. Let A_0 be the initial value of A before any transactions are executed. Values that A may assume are formulas of the form f_1 f_2 ··· f_n(A_0), where the f_i’s are functions associated with LOCK A—UNLOCK A pairs of the various transactions. No distinct values are equal. That is, values are regarded as uninterpreted formulas. This definition of “value” is a rigorous treatment of our informal statement in the previous section that we would assume no algebraic laws regarding the effects of transactions on items.

Example 11.5: In Fig. 11.3 we see three transactions and the functions associated with each LOCK—UNLOCK pair. Fig. 11.4 shows a possible schedule of these transactions and the resulting effect on items A, B, and C. We can observe that this schedule is not serializable. In proof, suppose it were. If T_i
11.2 A SIMPLE TRANSACTION MODEL

\[ \text{LOCK } A \xrightarrow{f_1} \text{LOCK } B \xrightarrow{f_2} \text{UNLOCK } A \xrightarrow{f_3} \text{LOCK } C \xrightarrow{f_4} \text{UNLOCK } B \xrightarrow{f_5} \text{LOCK } A \xrightarrow{f_6} \text{UNLOCK } C \xrightarrow{f_7} \text{UNLOCK } A \]

\[ T_1 \xrightarrow{f_2} T_2 \xrightarrow{f_5} T_3 \]

Fig. 11.3. Three transactions.

If \( T_2 \) precedes \( T_3 \), then the final value of \( C \) would be \( f_4(f_3(A)) \), not \( f_5(B) \). If \( T_2 \) precedes \( T_1 \), then the final value of \( A \) would be \( f_6(f_5(A)) \), \( f_4(f_3(A)) \), or \( f_1(f_6(f_5(A))) \), depending on whether the serial order was \( T_2T_1T_3 \), \( T_3T_2T_1 \), or \( T_3T_1T_2 \). As none of these formulas is the actual final value of \( A \) in Fig. 11.4, we see that \( T_3 \) cannot precede \( T_1 \) in an equivalent serial schedule. Since \( T_2 \) can neither precede nor follow \( T_1 \) in an equivalent serial schedule, so a serial schedule does not exist.

Note how our assumption that functions produce unique values is essential in the proof. For example, if it were possible that \( f_3f_2 = f_2f_3 \), then we could not rule out the possibility that \( T_1 \) precedes \( T_2 \). Let us reiterate that our assumption of unique values is not just for mathematical convenience. The work required to enable the database system to examine transactions and detect possibilities such as \( f_3f_2 = f_2f_3 \), and thereby permit a wider class of schedules to be regarded as serializable, is not worth the effort in general.

A Serializability Test

If we consider Example 11.5 and the proof that the schedule of Fig. 11.4 is not serializable, we see the key to a serializability test. We examine a schedule with regard to the order in which the various transactions lock a given item. This order must be consistent with the hypothetical equivalent serial schedule of the transactions. If the orders induced by two different items force two transactions to appear in different order, then we have a paradox, since both orders cannot be consistent with one serial schedule. We can express this test as a problem of finding cycles in a directed graph. The method is described formally in the next algorithm.

Algorithm 11.1: Testing Serializability of a Schedule.

Input: A schedule \( S \) for a set of transactions \( T_1, \ldots, T_k \).
Output: A determination whether \( S \) is serializable, and if so, a serial schedule equivalent to \( S \).
Method: Create a directed graph \( G \) (called a precedence graph), whose nodes correspond to the transactions. To determine the arcs of the graph \( G \), let \( S \) be
Fig. 11.4. A schedule.

\[ a_1; a_2; \ldots; a_n, \text{ where each } a_i \text{ is an action of the form } \]

\[ T_j: \text{LOCK } A_m \text{ or } T_j: \text{UNLOCK } A_m \]

\( T_j \) indicates the transaction to which the step belongs. If \( a_i \) is

\[ T_j: \text{UNLOCK } A_m \]

look for the next action \( a_p \) following \( a_i \) that is of the form \( T_x: \text{LOCK } A_m \). If there is one, then draw an arc from \( T_j \) to \( T_x \). The intuitive meaning of this arc is that in any serial schedule equivalent to \( S \), \( T_j \) must precede \( T_x \).

If \( G \) has a cycle, then \( S \) is not serializable. If \( G \) has no cycles, then find a linear order for the transactions such that \( T_i \) precedes \( T_j \) whenever there is an arc \( T_i \rightarrow T_j \). This can always be done by the process known as topological sorting, defined as follows. There must be some node \( T_i \) with no entering arcs; else we can prove that \( G \) has a cycle. List \( T_i \) and remove \( T_i \) from \( G \). Then repeat the process on the remaining graph until no nodes remain. The order in which the nodes are listed is a serial order for the transactions. \( \square \)

Example 11.6: Consider the schedule of Fig. 11.4. The graph \( G \), shown in Fig. 11.5 has nodes for \( T_1, T_2, \) and \( T_3 \). To find the arcs, we look at each UNLOCK step in Fig. 11.4. For example step (4), \( T_2: \text{UNLOCK } B \), is followed by

\[ T_1: \text{LOCK } B \]

in this case, the lock occurs at the next step. We therefore draw an arc \( T_2 \rightarrow T_1 \). As another example, the action at step (8), \( T_2: \text{UNLOCK } C \), is followed at step (11) by \( T_3: \text{LOCK } C \), and no intervening step locks \( C \). Therefore we draw an
11.2 A simple transaction model

Fig. 11.5. Graph of precedences among transactions.

```
T1
  ↓
  T2
  ↓
  T3
```

Fig. 11.6. A serializable schedule.

```
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCK A</td>
<td>UNLOCK A</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCK B</td>
<td>UNLOCK B</td>
<td></td>
</tr>
<tr>
<td>LOCK A</td>
<td>UNLOCK A</td>
<td></td>
</tr>
</tbody>
</table>
```

```plaintext
Example 11.7: In Fig. 11.6 we see a schedule for three transactions, and Fig. 11.7 shows its precedence graph. As there are no cycles, the schedule of Fig. 11.6 is serializable, and Algorithm 11.1 tells us that the serial order is T1, T2, T3. It is interesting to note that in the serial order, T1 precedes T3, even though in Fig. 11.6, T1 did not commence until T3 had finished.
```

Theorem 11.1: Algorithm 11.1 correctly determines if a schedule is serializable.

Proof: Suppose the precedence graph G has no cycles. Consider the sequence of transactions T_{i1}, T_{i2}, \ldots, T_{in} that in the schedule S lock and unlock item A, in that order. Then in G there are arcs T_{i1} \rightarrow T_{i2} \rightarrow \cdots \rightarrow T_{in}, so the transactions must appear in this order in the constructed serial schedule. As no other transaction locks A, it is easy to check that the value of A after executing S is the same as in the serial schedule constructed by Algorithm 11.1. Since the above holds for any item A, it follows that S is equivalent to the constructed serial schedule, so S is serializable.

Conversely, suppose G has a cycle T_{j1} \rightarrow T_{j2} \rightarrow \cdots \rightarrow T_{jn} \rightarrow T_{j1}. Let there be a serial schedule R equivalent to S, and suppose that in R, T_{j1} appears first.
among the transactions in the cycle. Let the arc $T_{j_{p-1}} \rightarrow T_{j_{p}}$ (take $j_{p-1}$ to be $j_{k}$ if $p = 1$) be in $G$ because of item $A$. Then in $R$, since $T_{j_{p}}$ appears before $T_{j_{p-1}}$, the final formula for $A$ applies a function $f$ associated with some LOCK $A$—UNLOCK $A$ pair in $T_{j_{p}}$ before applying some function $g$ associated with a LOCK $A$—UNLOCK $A$ pair in $T_{j_{p-1}}$. In $S$, however, $T_{j_{p-1}}$ precedes $T_{j_{p}}$, since there is an arc $T_{j_{p-1}} \rightarrow T_{j_{p}}$. Therefore, in $S$, $g$ is applied before $f$. Thus the final value of $A$ differs in $R$ and $S$, in the sense that the two formulas are not the same, and we conclude that $R$ and $S$ are not equivalent. Thus $S$ is equivalent to no serial schedule. 

\[ \square \]

**A Protocol that Guarantees Serializability**

We shall give a simple protocol with the property that any collection of transactions obeying the protocol cannot have a legal, nonserializable schedule. Moreover, this protocol is, in a sense to be discussed subsequently, the best that can be formulated. The protocol is, simply, to require that in any transaction, all locks precede all unlocks.\(^\dagger\)

Transactions obeying this protocol are said to be two-phase; the first phase is the locking phase and the second the unlocking phase. For example, in Fig. 11.3, $T_{1}$ and $T_{3}$ are two-phase; $T_{2}$ is not.

**Theorem 11.2:** If $S$ is any schedule of two-phase transactions, then $S$ is serializable.

**Proof:** Suppose not. Then by Theorem 11.1, the precedence graph $G$ for $S$ has a cycle, $T_{i_{1}} \rightarrow T_{i_{2}} \rightarrow \cdots \rightarrow T_{i_{p}} \rightarrow T_{i_{1}}$. Then some lock by $T_{i_{2}}$ follows an unlock by $T_{i_{1}}$; some lock by $T_{i_{3}}$ follows an unlock by $T_{i_{2}}$, and so on. Finally, some lock by $T_{i_{1}}$ follows an unlock by $T_{i_{p}}$. Therefore, a lock of $T_{i_{1}}$ follows an unlock of $T_{i_{1}}$, contradicting the assumption that $T_{i_{1}}$ is two-phase. \[ \square \]

Another way to see why two-phase transactions must be serializable is to imagine that a two-phase transaction occurs instantaneously at the moment it obtains the last of its locks. Then the order in which the transactions reach this point must be a serial schedule equivalent to the given schedule. For if in the given schedule, transaction $T_{1}$ locks $A$ before $T_{2}$ does, then $T_{1}$ surely obtains the last of its locks before $T_{2}$ does.

We mentioned that the two-phase protocol is in a sense the best that can be done. Precisely, what we can show is that if $T_{1}$ is any transaction that is not two phase, then there is some other transaction $T_{2}$ with which $T_{1}$ could be run in a nonserializable way.

\[ T_{2}; \text{LOCK} \ A; \text{LOCK} \ B \]

Then the schedule of Fig. 11.1 is a nonserializable one.

\[ T_{1}; \text{LOCK} \ A; \text{LOCK} \ B \]

Then the schedule of Fig. 11.1 is a nonserializable one.

\[ T_{2}; \text{LOCK} \ A; \text{LOCK} \ B \]

Note that there are protocols that yield only serial schedules, e.g., the one in Section 11.2, which we assumed to be serializable. In practice, we are usually interested in distinguishing between a read-
11.2 A SIMPLE TRANSACTION MODEL

LOCK A

UNLOCK A

LOCK B

UNLOCK B

T₁ | T₂

Fig. 11.8. A nonserializable schedule.

run in a nonserializable schedule. Suppose T₁ is not two phase. Then there is some step UNLOCK A of T₁ that precedes a step LOCK B. Let T₂ be:

T₂: LOCK A; LOCK B; UNLOCK A; UNLOCK B

Then the schedule of Fig. 11.8 is easily seen to be nonserializable, since the treatment of A requires that T₁ precede T₂, while the treatment of B requires the opposite.

Note that there are particular collections of transactions, not all two-phase, that yield only serial schedules. We shall consider an important example of such a collection in Section 11.5. However, since it is normal not to know the set of all transactions that could ever be executed concurrently with a given transaction, we are usually forced to require all transactions to be two-phase.

11.3 A MODEL WITH READ- AND WRITE-LOCKS

In Section 11.2 we assumed that every time a transaction locked an item it changed that item. In practice, many times a transaction needs only to obtain the value of the item and is guaranteed not to change that value. If we distinguish between a read-only access and a read-write access, we can develop a