15.1.4 Implementation of Locking

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

Figure 15.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7, and is waiting for a lock on I4.

Although the figure does not show it, the lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

   It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request, and processes the record following it, if any, similarly, and so on.

- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction (see Section 16.3), it releases all locks held by the aborted transaction.
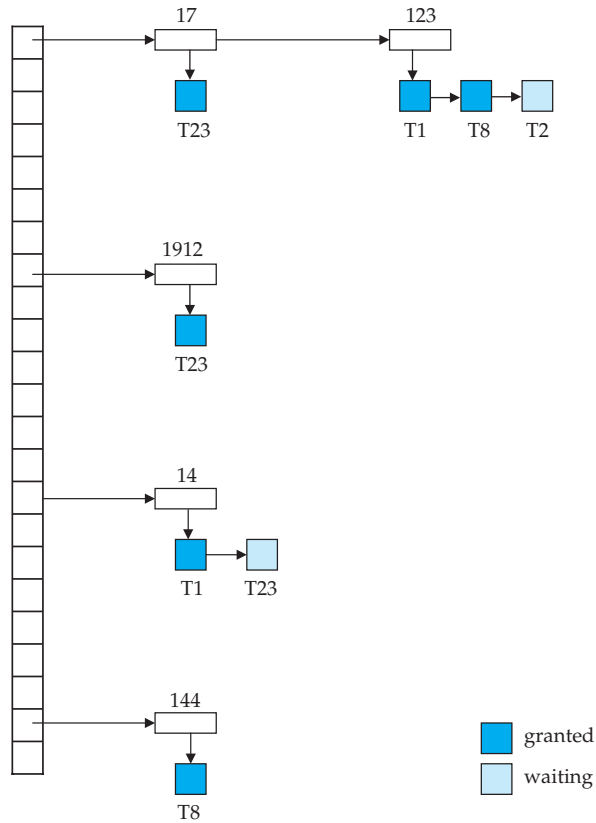
Figure 15.10   Lock table.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted. We study how to detect and handle deadlocks later, in Section 15.2.2. Section 17.2.1 describes an alternative implementation—one that uses shared memory instead of message passing for lock request/grant.

15.1.5   Graph-Based Protocols

As noted in Section 15.1.3, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing