3

TWO PHASE LOCKING

3.1 AGGRESSIVE AND CONSERVATIVE SCHEDULERS

In this chapter we begin our study of practical schedulers by looking at two phase locking schedulers, the most popular type in commercial products. For most of the chapter, we focus on locking in centralized DBSs, using the model presented in Chapter 1. Later sections show how locking schedulers can be modified to handle a distributed system environment. The final section discusses specialized locking protocols for trees and dags.

Recall from Chapter 1 that when a scheduler receives an operation from a TM it has three options:

1. immediately schedule it (by sending it to the DM);
2. delay it (by inserting it into some queue); or
3. reject it (thereby causing the issuing transaction to abort).

Each type of scheduler usually favors one or two of these options. Based on which of these options the scheduler favors, we can make the fuzzy, yet conceptually useful, distinction between *aggressive* and *conservative* schedulers.

An aggressive scheduler tends to avoid delaying operations; it tries to schedule them immediately. But to the extent it does so, it foregoes the opportunity to reorder operations it receives later on. By giving up the opportunity to reorder operations, it may get stuck in a situation in which it has no hope of finishing the execution of all active transactions in a serializable fashion. At this point, it has to resort to rejecting operations of one or more transactions, thereby causing them to abort (option (3) above).

A conservative scheduler, on the other hand, tends to delay operations. This gives it more leeway to reorder operations it receives later on. This leeway makes it less likely to get stuck in a situation where it has to reject operations to produce an SR execution. An extreme case of a conservative scheduler is one that, at any given time, delays the operations of all but one transaction. When that transaction terminates, another one is selected to have its operations processed. Such a scheduler processes transactions serially. It never needs to reject an operation, but avoids such rejections by sometimes excessively delaying operations.

There is an obvious performance trade-off between aggressive and conservative schedulers. Aggressive schedulers avoid delaying operations and thereby risk rejecting them later. Conservative schedulers avoid rejecting operations by deliberately delaying them. Each approach works especially well for certain types of applications.

For example, in an application where transactions that are likely to execute concurrently rarely conflict, an aggressive scheduler might perform better than a conservative one. Since conflicts are rare, conflicts that require the rejection of an operation are even rarer. Thus, the aggressive scheduler would not reject operations very often. By contrast, a conservative scheduler would needlessly delay operations, anticipating conflicts that seldom materialize.

On the other hand, in an application where transactions that are likely to execute concurrently conflict, a conservative scheduler's cautiousness may pay off. An aggressive scheduler might output operations recklessly, frequently placing itself in the undesirable position where rejecting operations is the only alternative to producing incorrect executions.

The rate at which conflicting operations are submitted is not the only factor that affects concurrency control performance. For example, the load on computer resources other than the DBS is also important. Therefore, this discussion of trade-offs between aggressive and conservative approaches to scheduling should be taken with a grain of salt. The intent is to develop some intuition about the operation of schedulers, rather than to suggest precise rules for designing them. Unfortunately, giving such precise rules for tailoring a scheduler to the performance specifications of an application is beyond the state-of-the-art.

Almost all types of schedulers have an aggressive and a conservative version. Generally speaking, a conservative scheduler tries to anticipate the future behavior of transactions in order to prepare for operations that it has not yet received. The main information it needs to know is the set of data items that each transaction will read and write (called, respectively, the *readset* and *writeset* of the transaction). In this way, it can predict which of the operations that it is currently scheduling may conflict with operations that will arrive in the future. By contrast, an aggressive scheduler doesn't need this information, since it schedules operations as early as it can, relying on rejections to correct mistakes.

A very conservative version of any type of scheduler can usually be built if transactions *predeclare their readsets and writesets*. This means that the TM begins processing a transaction by giving the scheduler the transaction's readset and writeset. Predeclaration is more easily and efficiently done if transactions are analyzed by a preprocessor, such as a compiler, before being submitted to the system, rather than being interpreted on the fly.

An impediment to building very conservative schedulers is that different executions of a given program may result in transactions that access different sets of data items. This occurs if programs contain conditional statements. For example, the following program reads either x and y, or x and z, depending on the value of x that it reads.

```
Procedure Fuzzy-readset begin
    Start;
    a := Read(x);
    if (a > 0) then b := Read(y) else b := Read(z);
    Commit
end
```

In this case the transaction must predeclare the set of all data items it *might* read or write. This often causes the transaction to overstate its readset and writeset. For example, a transaction executing Fuzzy-readset would declare its readset to be $\{x, y, z\}$, even though on any single execution it will only access two of those three data items. The same problem may occur if transactions interact with the DBS using a high level (e.g., relational) query language. A high level query may *potentially* access large portions of the database, even though on any *single* execution it only accesses a small portion of the database. When transactions overstate readsets and writesets, the scheduler ends up being even more conservative than it has to be, since it will delay certain operations in anticipation of others that will never be issued.

3.2 BASIC TWO PHASE LOCKING

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. The idea behind locking is intuitively simple. Each data item has a *lock* associated with it. Before a transaction T_1 may access a data item, the scheduler first examines the associated lock. If no transaction holds the lock, then the scheduler obtains the lock on behalf of T_1. If another transaction T_2 does hold the lock, then T_1 has to wait until T_2 gives up the lock. That is, the scheduler will not give T_1 the lock until T_2 releases it. The scheduler thereby ensures that only one transaction can hold the lock at a time, so only one transaction can access the data item at a time.

Locking can be used by a scheduler to ensure serializability. To present such a locking protocol, we need some notation.

Transactions access data items either for reading or for writing them. We therefore associate *two* types of locks with data items: read locks and write locks. We use $rl[x]$ to denote a read lock on data item x and $wl[x]$ to denote a write lock on x. We use $rl_i[x]$ (or $wl_i[x]$) to indicate that transaction T_i has obtained a read (or write) lock on x. As in Chapter 2, we use the letters o, p, and q to denote an arbitrary type of operation, that is, a Read (r) or Write (w). We use $ol_i[x]$ to denote a lock of type o by T_i on x.

Locks can be thought of as entries in a lock table. For example, $rl_i[x]$ corresponds to the entry $[x, r, T_i]$ in the table. For now, the detailed data structure of the table is unimportant. We'll discuss those details in Section 3.6.

Two locks $pl_i[x]$ and $ql_j[y]$ *conflict* if $x = y$, $i \neq j$, and operations p and q are of conflicting type. That is, two locks conflict if they are on the same data item, they are issued by different transactions, and one or both of them are write locks.[1] Thus, two locks on different data items do not conflict, nor do two locks that are on the same data item and are owned by the same transaction, even if they are of conflicting type.

We also use $rl_i[x]$ (or $wl_i[x]$) to denote the *operation* by which T_i *sets* or *obtains* a read (or write) lock on x. It will always be clear from the context whether $rl_i[x]$ and $wl_i[x]$ denote locks or operations that set locks.

We use $ru_i[x]$ (or $wu_i[x]$) to denote the operation by which T_i *releases* its read (or write) lock on x. In this case, we say T_i *unlocks* x (the u in ru and wu means unlock).

It is the job of a two phase locking (2PL) scheduler to manage the locks by controlling when transactions obtain and release their locks. In this section, we'll concentrate on the *Basic* version of 2PL. We'll look at specializations of 2PL in later sections.

Here are the rules according to which a Basic 2PL scheduler manages and uses its locks:

1. When it receives an operation $p_i[x]$ from the TM, the scheduler tests if $pl_i[x]$ conflicts with some $ql_j[x]$ that is already set. If so, it delays $p_i[x]$, forcing T_i to wait until it can set the lock it needs. If not, then the scheduler sets $pl_i[x]$, and then sends $p_i[x]$ to the DM.[2]
2. Once the scheduler has set a lock for T_i, say $pl_i[x]$, it may not release that lock *at least* until after the DM acknowledges that it has processed the lock's corresponding operation, $p_i[x]$.
3. Once the scheduler has released a lock for a transaction, it may not subsequently obtain *any* more locks for that transaction (on *any* data item).

---

[1]We will generalize the notion of lock conflict to operations other than Read and Write in Section 3.8.

[2]The scheduler must be implemented so that setting a lock is atomic relative to setting conflicting locks. This ensures that conflicting locks are never held simultaneously.

Rule (1) prevents two transactions from concurrently accessing a data item in conflicting modes. Thus, conflicting operations are scheduled in the same order in which the corresponding locks are obtained.

Rule (2) supplements rule (1) by ensuring that the DM processes operations on a data item in the order that the scheduler submits them. For example, suppose T_i obtains $rl_i[x]$, which it releases before the DM has confirmed that $r_i[x]$ has been processed. Then it is possible for T_j to obtain a conflicting lock on x, $wl_j[x]$, and send $w_j[x]$ to the DM. Although the scheduler has sent the DM $r_i[x]$ before $w_j[x]$, without rule (2) there is no guarantee that the DM will receive and process the operations in that order.

Rule (3), called the *two phase rule*, is the source of the name *two phase locking*. Each transaction may be divided into two phases: a *growing phase* during which it obtains locks, and a *shrinking phase* during which it releases locks. The intuition behind rule (3) is not obvious. Roughly, its function is to guarantee that *all* pairs of conflicting operations of two transactions are scheduled in the same order. Let's look at an example to see, intuitively, why this might be the case.

Consider two transactions T_1 and T_2:

$$T_1\colon r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_2\colon w_2[x] \rightarrow w_2[y] \rightarrow c_2$$

and suppose they execute as follows:

$$H_1 = rl_1[x]\ r_1[x]\ ru_1[x]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ wu_2[x]\ wu_2[y]\ c_2\ wl_1[y]\ w_1[y]\ wu_1[y]\ c_1$$

Since $r_1[x] < w_2[x]$ and $w_2[y] < w_1[y]$, SG(H_1) consists of the cycle $T_1 \rightarrow T_2 \rightarrow T_1$. Thus, H_1 is not SR.

The problem in H_1 is that T_1 released a lock ($ru_1[x]$) and subsequently set a lock ($wl_1[y]$), in violation of the two phase rule. Between $ru_1[x]$ and $wl_1[y]$, another transaction T_2 wrote into both x and y, thereby appearing to follow T_1 with respect to x and precede it with respect to y. Had T_1 obeyed the two phase rule, this "window" between $ru_1[x]$ and $wl_1[y]$ would not have opened, and T_2 could not have executed as it did in H_1. For example, T_1 and T_2 might have executed as follows.

1. Initially, neither transaction owns any locks.
2. The scheduler receives $r_1[x]$ from the TM. Accordingly, it sets $rl_1[x]$ and submits $r_1[x]$ to the DM. Then the DM acknowledges the processing of $r_1[x]$.
3. The scheduler receives $w_2[x]$ from the TM. The scheduler can't set $wl_2[x]$, which conflicts with $rl_1[x]$, so it delays the execution of $w_2[x]$ by placing it on a queue.
4. The scheduler receives $w_1[y]$ from the TM. It sets $wl_1[y]$ and submits $w_1[y]$ to the DM. Then the DM acknowledges the processing of $w_1[y]$.
5. The scheduler receives c_1 from the TM, signalling that T_1 has terminated. The scheduler sends c_1 to the DM. After the DM acknowledges

processing c_1, the scheduler releases $rl_1[x]$ and $wl_1[y]$. This is safe with respect to rule (2), because $r_1[x]$ and $w_1[y]$ have already been processed, and with respect to rule (3), because T_1 won't request any more locks.

6. The scheduler sets $wl_2[x]$ so that $w_2[x]$, which had been delayed, can now be sent to the DM. Then the DM acknowledges $w_2[x]$.
7. The scheduler receives $w_2[y]$ from the TM. It sets $wl_2[y]$ and sends $w_2[y]$ to the DM. The DM then acknowledges processing $w_2[y]$.
8. T_2 terminates and the TM sends c_2 to the scheduler. The scheduler sends c_2 to the DM. After the DM acknowledges processing c_2, the scheduler releases $wl_2[x]$ and $wl_2[y]$.

This execution is represented by the following history.

$$H_2 = rl_1[x]\ r_1[x]\ wl_1[y]\ w_1[y]\ c_1\ ru_1[x]\ wu_1[y]\ wl_2[x]\ w_2[x]\ wl_2[y]\ w_2[y]\ c_2\ wu_2[x]\ wu_2[y].$$

H_2 is serial and therefore is SR.

An important and unfortunate property of 2PL schedulers is that they are subject to *deadlocks*. For example, suppose a 2PL scheduler is processing transactions T_1 and T_3

$$T_1\text{: } r_1[x] \to w_1[y] \to c_1 \qquad T_3\text{: } w_3[y] \to w_3[x] \to c_3$$

and consider the following sequence of events:

1. Initially, neither transaction holds any locks.
2. The scheduler receives $r_1[x]$ from the TM. It sets $rl_1[x]$ and submits $r_1[x]$ to the DM.
3. The scheduler receives $w_3[y]$ from the TM. It sets $wl_3[y]$ and submits $w_3[y]$ to the DM.
4. The scheduler receives $w_3[x]$ from the TM. The scheduler does not set $wl_3[x]$ because it conflicts with $rl_1[x]$ which is already set. Thus $w_3[x]$ is delayed.
5. The scheduler receives $w_1[y]$ from the TM. As in (4), $w_1[y]$ must be delayed.

Although the scheduler behaved exactly as prescribed by the rules of 2PL schedulers, neither T_1 nor T_3 can complete without violating one of these rules. If the scheduler sends $w_1[y]$ to the DM without setting $wl_1[y]$, it violates rule (1). Similarly for $w_3[x]$. Suppose the scheduler releases $wl_3[y]$, so it can set $wl_1[y]$ and thereby be allowed to send $w_1[y]$ to the DM. In this case, the scheduler will never be able to set $wl_3[x]$ (so it can process $w_3[x]$), or else it would violate rule (3). Similarly if it releases $rl_1[x]$. The scheduler has painted itself into a corner.

This is a classic deadlock situation. Before either of two processes can proceed, one must release a resource that the other needs to proceed.

Deadlock also arises when transactions try to strengthen read locks to write locks. Suppose a transaction T_i reads a data item x and subsequently tries to write it. T_i issues $r_i[x]$ to the scheduler, which sets $rl_i[x]$. When T_i issues $w_i[x]$ to the scheduler, the scheduler must upgrade $rl_i[x]$ to $wl_i[x]$. This upgrading of a lock is called a lock *conversion*. To obey 2PL, the scheduler must not release $rl_i[x]$. This is not a problem, because locks set by the same transaction do not conflict with each other. However, if two transactions concurrently try to convert their read locks on a data item into write locks, the result is deadlock.

For example, suppose T_4 and T_5 issue operations to a 2PL scheduler.

T_4: $r_4[x] \rightarrow w_4[x] \rightarrow c_4$ T_5: $r_5[x] \rightarrow w_5[x] \rightarrow c_5$

The scheduler might be confronted with the following sequence of events:

1. The scheduler receives $r_4[x]$, and therefore sets $rl_4[x]$ and sends $r_4[x]$ to the DM.
2. The scheduler receives $r_5[x]$, and therefore sets $rl_5[x]$ and sends $r_5[x]$ to the DM.
3. The scheduler receives $w_4[x]$. It must delay the operation, because $wl_4[x]$ conflicts with $rl_5[x]$.
4. The scheduler receives $w_5[x]$. It must delay the operation, because $wl_5[x]$ conflicts with $rl_4[x]$.

Since neither transaction can release the $rl[x]$ it owns, and since neither can proceed until it sets $wl[x]$, the transactions are deadlocked. This type of deadlock commonly occurs when a transaction scans a large number of data items looking for data items that contain certain values, and then updates those data items. It sets a read lock on each data item it scans, and converts a read lock into a write lock only when it decides to update a data item.

We will examine ways of dealing with deadlocks in Section 3.4.

3.3 *CORRECTNESS OF BASIC TWO PHASE LOCKING

To prove that a scheduler is correct, we have to prove that all histories representing executions that could be produced by it are SR. Our strategy for proving this has two steps. First, given the scheduler we characterize the properties that all of its histories must have. Second, we prove that any history with these properties must be SR. Typically this last part involves the Serializability Theorem. That is, we prove that for any history H with these properties, SG(H) is acyclic.

To prove the correctness of the 2PL scheduler, we must characterize the set of *2PL histories*, that is, those that represent possible executions of transactions that are synchronized by a 2PL scheduler. To characterize 2PL histories, we'll find it very helpful to include the Lock and Unlock operations. (They

were not in our formal model of Chapter 2.) Examining the order in which Lock and Unlock operations are processed will help us establish the order in which Reads and Writes are executed. This, in turn, will enable us to prove that the SG of any history produced by 2PL is acyclic.

To characterize 2PL histories, let's list all of the orderings of operations that we know must hold. First, we know that a lock is obtained for each database operation before that operation executes. This follows from rule (1) of 2PL. That is, $ol_i[x] < o_i[x]$. From rule (2) of 2PL, we know that each operation is executed by the DM before its corresponding lock is released. In terms of histories, that means $o_i[x] < ou_i[x]$. In particular, if $o_i[x]$ belongs to a committed transaction (all of whose operations are therefore in the history), we have $ol_i[x] < o_i[x] < ou_i[x]$.

> **Proposition 3.1:** Let H be a history produced by a 2PL scheduler. If $o_i[x]$ is in C(H), then $ol_i[x]$ and $ou_i[x]$ are in C(H), and $ol_i[x] < o_i[x] < ou_i[x]$. □

Now suppose we have two operations $p_i[x]$ and $q_j[x]$ that conflict. Thus, the locks that correspond to these operations also conflict. By rule (1) of 2PL, only one of these locks can be held at a time. Therefore, the scheduler must release the lock corresponding to one of the operations before it sets the lock for the other. In terms of histories, we must have $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$.

> **Proposition 3.2:** Let H be a history produced by a 2PL scheduler. If $p_i[x]$ and $q_j[x]$ $(i \neq j)$ are conflicting operations in C(H), then either $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$. □

Finally, let's look at the two phase rule, which says that once a transaction releases a lock it cannot subsequently obtain any other locks. This is equivalent to saying that every lock operation of a transaction executes before every unlock operation of that transaction. In terms of histories, we can write this as $pl_i[x] < qu_i[y]$.

> **Proposition 3.3:** Let H be a complete history produced by a 2PL scheduler. If $p_i[x]$ and $q_i[y]$ are in C(H), then $pl_i[x] < qu_i[y]$. □

Using these properties, we must now show that every 2PL history H has an acyclic SG. The argument has three steps. (Recall that SG(H) contains nodes only for the committed transactions in H.)

1. If $T_i \rightarrow T_j$ is in SG(H), then one of T_i's operations on some data item, say x, executed before and conflicted with one of T_j's operations. Therefore, T_i must have released its lock on x *before* T_j set its lock on x.

2. Suppose $T_i \rightarrow T_j \rightarrow T_k$ is a path in SG(H). From step (1), T_i released some lock before T_j set some lock, and similarly T_j released some lock before T_k set some lock. Moreover, by the two phase rule, T_j set all of its locks before it released any of them. Therefore, by transitivity, T_i released some lock before T_k set some lock. By induction, this argument extends to arbitrarily long paths in SG(H). That is, for any path $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$, T_1 released some lock before T_n set some lock.
3. Suppose SG(H) had a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. Then by step (2), T_1 released a lock before T_1 set a lock. But then T_1 violated the two phase rule, which contradicts the fact that H is a 2PL history. Therefore, the cycle cannot exist. Since SG(H) has no cycles, the Serializability Theorem implies that H is SR.

Notice that in step (2), the lock that T_i released does not necessarily conflict with the one that T_k set, and in general they do not. T_i's lock conflicts with and precedes one that T_j set, and T_j released a lock (possibly a different one) that conflicts with and precedes the one that T_k set. For example, the history that leads to the path $T_i \rightarrow T_j \rightarrow T_k$ could be

$$r_i[x] \rightarrow w_j[x] \rightarrow w_j[y] \rightarrow r_k[y].$$

T_i's lock on x does not conflict with T_k's lock on y.

We formalize this three step argument in the following lemmas and theorem. The two lemmas formalize steps (1) and (2). The theorem formalizes step (3).

Lemma 3.4: Let H be a 2PL history, and suppose $T_i \rightarrow T_j$ is in SG(H). Then, for some data item x and some conflicting operations $p_i[x]$ and $q_j[x]$ in H, $pu_i[x] < ql_j[x]$.

Proof: Since $T_i \rightarrow T_j$, there must exist conflicting operations $p_i[x]$ and $q_j[x]$ such that $p_i[x] < q_j[x]$. By Proposition 3.1,

1. $pl_i[x] < p_i[x] < pu_i[x]$, and
2. $ql_j[x] < q_j[x] < qu_j[x]$.

By Proposition 3.2, either $pu_i[x] < ql_j[x]$ or $qu_j[x] < pl_i[x]$. In the latter case, by (1), (2) and transitivity, we would have $q_j[x] < p_i[x]$, which contradicts $p_i[x] < q_j[x]$. Thus, $pu_i[x] < ql_j[x]$, as desired. □

Lemma 3.5: Let H be a 2PL history, and let $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n$ be a path in SG(H), where $n > 1$. Then, for some data items x *and* y, and some operations $p_1[x]$ and $q_n[y]$ in H, $pu_1[x] < ql_n[y]$.

Proof: The proof is by induction on n. The basis step, for $n = 2$, follows immediately from Lemma 3.4.

For the induction step, suppose the lemma holds for $n = k$ for some $k \geq 2$. We will show that it holds for $n = k + 1$. By the induction hypothesis, the path $T_1 \rightarrow \cdots \rightarrow T_k$ implies that there exist data items x and z, and operations $p_1[x]$ and $o_k[z]$ in H, such that $pu_1[x] < ol_k[z]$. By $T_k \rightarrow T_{k+1}$ and Lemma 3.4, there exists data item y and conflicting operations $o'_k[y]$, and $q_{k+1}[y]$ in H, such that $o'u_k[y] < ql_{k+1}[y]$. By Proposition 3.3, $ol_k[z] < o'u_k[y]$. By the last three precedences and transitivity, $pu_1[x] < ql_{k+1}[y]$, as desired. □

Theorem 3.6: Every 2PL history H is serializable.

Proof: Suppose, by way of contradiction, that SG(H) contains a cycle $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, where $n > 1$. By Lemma 3.5, for some data items x and y, and some operations $p_1[x]$ and $q_1[y]$ in H, $pu_1[x] < ql_1[y]$. But this contradicts Proposition 3.3. Thus, SG(H) has no cycles and so, by the Serializability Theorem, H is SR. □

3.4 DEADLOCKS

The scheduler needs a strategy for detecting deadlocks, so that no transaction is blocked forever. One strategy is *timeout*. If the scheduler finds that a transaction has been waiting too long for a lock, then it simply guesses that there may be a deadlock involving this transaction and therefore aborts it. Since the scheduler is only guessing that a transaction may be involved in a deadlock, it may be making a mistake. It may abort a transaction that isn't really part of a deadlock but is just waiting for a lock owned by another transaction that is taking a long time to finish. There's no harm done by making such an incorrect guess, insofar as correctness is concerned. There *is* certainly a performance penalty to the transaction that was unfairly aborted, though as we'll see in Section 3.12, the overall effect may be to improve transaction throughput.

One can avoid too many of these types of mistakes by using a long timeout period. The longer the timeout period, the more chance that the scheduler is aborting transactions that are actually involved in deadlocks. However, a long timeout period has a liability, too. The scheduler doesn't notice that a transaction might be deadlocked until the timeout period has elapsed. So, should a transaction become involved in a deadlock, it will lose some time waiting for its deadlock to be noticed. The timeout period is therefore a parameter that needs to be tuned. It should be long enough so that most transactions that are aborted are actually deadlocked, but short enough that deadlocked transactions don't wait too long for their deadlocks to be noticed. This tuning activity is tricky but manageable, as evidenced by its use in several commercial products, such as Tandem.

Another approach to deadlocks is to detect them precisely. To do this, the scheduler maintains a directed graph called a *waits-for graph* (*WFG*). The nodes of WFG are labelled with transaction names. There is an edge $T_i \rightarrow T_j$,

from node T_i to node T_j, iff transaction T_i is waiting for transaction T_j to release some lock.[3]

Suppose a WFG has a cycle: $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. Each transaction is waiting for the next transaction in the cycle. So, T_1 is waiting for itself, as is every other transaction in the cycle. Since all of these transactions are blocked waiting for locks, none of the locks they are waiting for are ever going to be released. Thus, the transactions are deadlocked. Exploiting this observation, the scheduler can detect deadlocks by checking for cycles in WFG.

Of course, the scheduler has to maintain a representation of the WFG in order to check for cycles in it. The scheduler can easily do this by adding an edge $T_i \rightarrow T_j$ to the WFG whenever a lock request by T_i is blocked by a conflicting lock owned by T_j. It drops an edge $T_i \rightarrow T_j$ from the WFG whenever it releases the (last) lock owned by T_j that had formerly been blocking a lock request issued by T_i. For example, suppose the scheduler receives $r_i[x]$, but has to delay it because T_j already owns $wl_j[x]$. Then it adds an edge $T_i \rightarrow T_j$ to the WFG. After T_j releases $wl_j[x]$, the scheduler sets $rl_i[x]$, and therefore deletes the edge $T_i \rightarrow T_j$.

How often should the scheduler check for cycles in the WFG? It could check every time a new edge is added, looking for cycles that include this new edge. But this could be quite expensive. For example, if operations are frequently delayed, but deadlocks are relatively rare, then the scheduler is spending a lot of effort looking for deadlocks that are hardly ever there. In such cases, the scheduler should check for cycles less often. Instead of checking every time an edge is added, it waits until a few edges have been added, or until some timeout period has elapsed. There is no danger in checking less frequently, since the scheduler will never miss a deadlock. (Deadlocks don't go away by themselves!) Moreover, by checking less frequently, the scheduler incurs the cost of cycle detection less often. However, a deadlock may go undetected for a longer period this way. In addition, *all* cycles must be found, not just those involving the most recently added edge.

When the scheduler discovers a deadlock, it must break the deadlock by aborting a transaction. The Abort will in turn delete the transaction's node from the WFG. The transaction that it chooses to abort is called the *victim*. Among the transactions involved in a deadlock cycle in WFG, the scheduler should select a victim whose abortion costs the least. Factors that are commonly used to make this determination include:

[3]WFGs are related to SGs in the following sense. If $T_i \rightarrow T_j$ is in the WFG, and both T_i and T_j ultimately commit, then $T_j \rightarrow T_i$ will be in the SG. However, if T_i aborts, then $T_j \rightarrow T_i$ may never appear in the SG. That is, WFGs describe the current state of transactions, which includes waits-for situations involving operations that never execute (due to abortions). SGs only describe dependencies between committed transactions (which arise from operations that actually execute).

- The amount of effort that has already been invested in the transaction. This effort will be lost if the transaction is aborted.
- The cost of aborting the transaction. This cost generally depends on the number of updates the transaction has already performed.
- The amount of effort it will take to finish executing the transaction. The scheduler wants to avoid aborting a transaction that is almost finished. To do this, it must be able to predict the future behavior of active transactions, e.g., based on the transaction's type (Deposits are short, Audits are long).
- The number of cycles that contain the transaction. Since aborting a transaction breaks all cycles that contain it, it is best to abort transactions that are part of more than one cycle (if such transactions exist).

A transaction can repeatedly become involved in deadlocks. In each deadlock, the transaction is selected as the victim, aborts, and restarts its execution, only to become involved in a deadlock again. To avoid such *cyclic restarts*, the victim selection algorithm should also consider the number of times a transaction is aborted due to deadlock. If it has been aborted too many times, then it should not be a candidate for victim selection, unless all transactions involved in the deadlock have reached this state.

3.5 VARIATIONS OF TWO PHASE LOCKING

Conservative 2PL

It is possible to construct a 2PL scheduler that never aborts transactions. This technique is known as *Conservative 2PL* or *Static 2PL*. As we have seen, 2PL causes abortions because of deadlocks. Conservative 2PL avoids deadlocks by requiring each transaction to obtain *all* of its locks before *any* of its operations are submitted to the DM. This is done by having each transaction predeclare its readset and writeset. Specifically, each transaction T_i first tells the scheduler all the data items it will want to Read or Write, for example as part of its Start operation. The scheduler tries to set *all* of the locks needed by T_i. It can do this providing that none of these locks conflicts with a lock held by any other transaction. If the scheduler succeeds in setting all of T_i's locks, then it submits T_i's operations to the DM as soon as it receives them. After the DM acknowledges the processing of T_i's last database operation, the scheduler may release all of T_i's locks.

If, on the other hand, *any* of the locks requested in T_i's Start conflicts with locks presently held by other transactions, then the scheduler does not grant any of T_i's locks. Instead, it inserts T_i along with its lock requests into a waiting queue. Every time the scheduler releases the locks of a completed transaction, it examines the waiting queue to see if it can grant all of the lock requests

of any waiting transactions. If so, it then sets all of the locks for each such transaction and continues processing as just described.

In Conservative 2PL, if a transaction T_i is waiting for a lock held by T_j, then T_i is holding no locks. Therefore, no other transaction T_k can be waiting for T_i, so there can be no WFG edges of the form $T_k \rightarrow T_i$. Since there can be no such edges, T_i cannot be in a WFG cycle, and hence cannot become part of a deadlock. Since deadlock is the only reason that a 2PL scheduler ever rejects an operation and thereby causes the corresponding transaction to abort, Conservative 2PL never aborts a transaction. (Of course, a transaction may abort for other reasons.) This is a classic case of a conservative scheduler. By delaying operations sooner than it has to, namely, when the transaction begins executing, the scheduler avoids abortions that might otherwise be needed for concurrency control reasons.

Strict 2PL

Almost all implementations of 2PL use a variant called *Strict* 2PL. This differs from the Basic 2PL scheduler described in Section 3.2 in that it requires the scheduler to release all of a transaction's locks together, when the transaction terminates. More specifically, T_i's locks are released *after* the DM acknowledges the processing of c_i or a_i, depending on whether T_i commits or aborts (respectively).

There are two reasons for adopting this policy. First, consider when a 2PL scheduler can release some $ol_i[x]$. To do so the scheduler must know that: (1) T_i has set all of the locks it will ever need, and (2) T_i will not subsequently issue operations that refer to x. One point in time at which the scheduler can be sure of (1) and (2) is when T_i terminates, that is, when the scheduler receives the c_i or a_i operation. In fact, in the absence of any information from the TM aside from the operations submitted, this is the earliest time at which the scheduler can be assured that (1) and (2) hold.

A second reason for the scheduler to keep a transaction's locks until it ends, and specifically until *after* the DM processes the transaction's Commit or Abort, is to guarantee a strict execution. To see this, let history H represent an execution produced by a Strict 2PL scheduler and suppose $w_i[x] < o_j[x]$. By rule (1) of 2PL (Proposition 3.1) we must have

1. $wl_i[x] < w_i[x] < wu_i[x]$, and
2. $ol_j[x] < o_j[x] < ou_j[x]$.

Because $wl_i[x]$ and $ol_j[x]$ conflict (whether o is r or w, we must have either $wu_i[x] < ol_j[x]$ or $ou_j[x] < wl_i[x]$ (by Proposition 3.2). The latter, together with (1) and (2), would contradict that $w_i[x] < o_j[x]$ and, therefore,

3. $wu_i[x] < ol_j[x]$.

But because H was produced by a Strict 2PL scheduler we must have that

4. either $a_i < wu_i[x]$ or $c_i < wu_i[x]$.

From (2) - (4), it follows that either $a_i < o_j[x]$ or $c_i < o_j[x]$, proving that H is strict.

Actually, from this argument it follows that it is only necessary to hold *write* locks until after a transaction commits or aborts to ensure strictness. Read locks may be released earlier, subject to the 2PL rules to ensure serializability. Pragmatically, this means that *read locks can be released when the transaction terminates* (i.e., when the scheduler *receives* the transaction's Commit or Abort), *but write locks must be held until after the transaction commits or aborts* (i.e., after the DM *processes* the transaction's Commit or Abort).

Recall that strict histories have nice properties. They are recoverable and avoid cascading aborts. Furthermore, Abort can be implemented by restoring before images. For this reason, 2PL implementations usually take the form of Strict 2PL schedulers, rather than the seemingly more flexible Basic 2PL schedulers.

3.6 IMPLEMENTATION ISSUES

An implementation of 2PL for any particular system depends very much on the overall design of the computer system and on the available operating system facilities. It is therefore difficult to give general guidelines for implementation. However, at the risk of superficiality, we will briefly sketch the issues faced in most implementations of locking.

The scheduler abstraction is usually implemented by a combination of a lock manager (*LM*) and a TM. The LM services the Lock and Unlock operations. When the TM receives a Read or Write from a transaction, it sends the appropriate Lock operation to the LM. When the LM acknowledges that the lock is set, the TM sends the Read or Write to the DM. Thus, the TM subsumes the scheduler function of ensuring that a lock is set before the corresponding operation is performed.

Notice that the control flow here differs somewhat from the scheduler abstraction. In our DBS model, the TM sends the Read or Write directly to the scheduler. The scheduler sets the appropriate lock and forwards the Read or Write to the DM.

The Lock Manager

The LM maintains a table of locks, and supports the operations Lock(transaction-id, data-item, mode) and Unlock(transaction-id, data-item), where

transaction-id is the identifier of the transaction requesting the lock,[4] *data-item* is the name of the data item to be locked, and *mode* is "read" or "write." To process a Lock operation, the LM tries to set the specified lock by adding an entry to the lock table. If another transaction owns a conflicting lock, then the LM adds the lock request to a queue of waiting requests for that data item.[5] Unlock releases the specified lock, and grants any waiting lock requests that are no longer blocked.

Lock and Unlock operations are invoked very frequently. In most transaction processing systems (such as airline reservation and on-line banking), each transaction does little computing for each data item it accesses. Therefore, unless locking is very fast, it consumes a significant fraction of the processor's time. As a rule of thumb, it should take on the average no more than several hundred machine language instructions to set or release a lock (including potential overhead for a monitor call, supervisor call, context switch, etc.). To reach this speed, the LM is often optimized carefully for special cases that occur frequently, such as setting a lock that conflicts with no other locks, and releasing all of a transaction's locks at once.

The lock table is usually implemented as a hash table with the data item identifier as key, because hash tables are especially fast for content-based retrieval. An entry in the table for data item x contains a queue header, which points to a list of locks on x that have been set and a list of lock requests that are waiting. Each lock or lock request contains a transaction-id and a lock mode. Since a very large number of data items can potentially be locked, the LM limits the size of the lock table by only allocating entries for those data items that actually *are* locked. When it releases the last lock for a data item x, it deallocates the entry for x.

Since the TM normally releases all of a transaction's read locks as soon as the transaction terminates, releasing a transaction's read locks should be a basic LM operation. Similarly, if the scheduler is strict, the TM releases a transaction's write locks as soon as the DM acknowledges committing the transaction. So releasing write locks should also be a basic LM operation. To make these operations fast, a common practice is to link together in the lock table all of the read lock entries and all of the write lock entries of each transaction. If the Commit operation is very efficient, then it may not be cost effective to release read locks before the Commit and write locks afterwards. Instead, it may be satisfactory to release write *and* read locks after the Commit. This saves the overhead of one call to the LM, at the expense of some lost concurrency by holding read locks a little longer than necessary.

---

[4]This is the transaction identifier discussed in Section 1.1.

[5]One could add another parameter to Lock that specifies whether, in the event that the lock request cannot be granted, the request should be queued or cancelled (i.e., return immediately to the caller).

The lock table should be protected. It should only be accessed by the programs that implement Lock and Unlock, so that it cannot be easily corrupted. Clearly, a stray update to memory that compromises the integrity of the lock table's data structures is likely to cause the entire system to malfunction (by executing in a non-SR manner) or die. This goal of protection may be accomplished by making the LM part of the operating system itself, thereby providing strong protection against corruption by user programs. Alternatively, it might be implemented as a monitor or device driver. The latter technique is a common workaround in systems that do not support shared memory between processes.

Blocking and Aborting Transactions

When the LM releases a lock for x, it may be able to grant other lock requests that are waiting on x's lock queue. If there are such waiting requests, then the LM must schedule them "fairly"; otherwise, it runs the risk of delaying some forever.

For example, suppose there is a queue of lock requests [$rl_2[x]$, $rl_3[x]$, $wl_4[x]$, $rl_5[x]$] waiting for $wl_1[x]$ to be released, where $rl_2[x]$ is at the head of the queue. After it releases $wl_1[x]$, the LM can now set $rl_2[x]$ and $rl_3[x]$. It can also set $rl_5[x]$. This is unfair, in the sense that $rl_5[x]$ has jumped ahead of $wl_4[x]$. It can also lead to the indefinite postponement of $wl_4[x]$, since a steady stream of read lock requests may continually jump ahead of $wl_4[x]$, preventing it from being set. This danger of indefinite postponement can be avoided by servicing the queue first-come-first-served, thereby never letting a read lock request jump ahead. Or, the LM can allow read lock requests to jump ahead only if no write lock request has been waiting too long, where the maximum waiting time is a tunable parameter.

The mechanism by which the LM causes transaction T_i to wait and later unblocks it depends on the process synchronization primitives provided by the operating system, and on the way transactions and DBS modules are structured. For example, suppose each transaction executes as a process, and the DBS is also a process. The DBS receives Read and Write requests as messages. For each such message, it invokes its LM to set the appropriate lock. By not responding to a transaction T_i's message, the DBS has effectively blocked T_i, if T_i is waiting for the response. It eventually unblocks T_i by sending the response message. Alternatively, suppose T_i calls the DBS as a procedure (e.g., a monitor) that executes in T_i's process context. Then the DBS can block T_i simply by blocking the process in which T_i is executing (e.g., by waiting on a condition that is assigned uniquely to T_i). When the LM executing (in the DBS) in another transaction's context releases the relevant lock, it can signal the event for T_i (e.g., by signaling T_i's condition), thereby unblocking T_i and allowing it to complete the lock request.

Similar issues arise in aborting the victim of a detected deadlock. The TM must be informed of the forced abortion, so it can either notify the transaction or automatically restart it. When processing an Abort (for deadlock or any other reason), the TM must delete the transaction's legacy from the lock table. Performing such activities outside the normal flow of control of the transaction again depends on several factors: how transactions, the TM, the LM, and the deadlock detector are structured as processes; if and how they share memory; and what synchronization and process control primitives they can exercise on each other.

System dependencies are often paramount in designing solutions to these control problems. The quality of those solutions can be among the most important factors affecting the robustness and performance of the concurrency control implementation.

Atomicity of Reads and Writes

A critical assumption in our model of histories is that every operation in the history is atomic. Since our correctness proof of 2PL is built on the history model, our implementation of 2PL must honor its assumptions. There are four operations that we used in histories to argue the correctness of 2PL: Lock, Unlock, Read, and Write. These four operations must be implemented atomically.

To ensure that concurrent executions of Locks and Unlocks are atomic, accesses to the lock table must be synchronized using an operating system synchronization mechanism, such as semaphores or monitors. If accesses to the lock table are synchronized by a single semaphore (or lock, monitor, etc.), then a very high locking rate may cause that semaphore to become a bottleneck. This bottleneck can be relieved by partitioning the lock table into several component tables, with a different semaphore serializing accesses to each component. To insert a lock entry, the Lock operation selects the appropriate component by analyzing its parameters (e.g., by hashing the data item name), and then requesting the semaphore that regulates access to that component.

Since most databases are stored on disk, Reads and Writes on data items are usually implemented by Reads and Writes of fixed-size disk blocks. If the granularity of a data item is a disk block, then it is straightforward to implement Read and Write atomically. Each Read or Write on a data item is implemented as an atomic Read or Write on a disk block. If the granularity of data items is not a disk block, then extra care is needed to ensure that Reads and Writes on data items execute atomically.

For example, suppose the granularity of data items is a record, where many records fit on each disk block, but no record is spread over more than one disk block. A program that implements Read must read the disk block that contains the record, extract the record from that block, and return the record to the calling program. A program that implements Write must read the disk

Accounts

Account#	Location	Balance
339	Marlboro	750
914	Tyngsboro	2308
22	Tyngsboro	1550

Assets

Location	Total
Marlboro	750
Tyngsboro	3858

FIGURE 3–1
A Banking Database

block that contains the record, update the relevant record in that block, and then write the block back to disk.[6]

An uncontrolled concurrent execution of two of these operations may not be atomic. For example, suppose that two Writes execute concurrently on different records stored in the same block. Suppose each Write begins by independently and concurrently reading a copy of the block into its local memory area. Each Write now updates the copy of its record in its private copy of the disk block, and then writes that private copy back to disk. Since each Write was working on an independent copy of the block, one of the record updates gets lost. If the two Writes had executed serially (with their associated Reads), then neither update would have been lost. Therefore, the implementation is not atomic.

A solution to this problem is to require that each block be locked while a Write is being applied to a record contained in that block. Depending on how Read and Write are implemented, Reads may need to lock disk blocks as well.

Notice that these locks on blocks that make record operations atomic are generally not the same as the record locks that are used to make transactions serializable. A lock on a block can be released as soon as the operation that was using it completes. A lock on a record must follow the 2PL locking protocol.

3.7 THE PHANTOM PROBLEM

We have been modelling a database as a fixed set of data items, which can be accessed by Reads and Writes. Most real databases can dynamically grow and shrink. In addition to Read and Write, they usually support operations to Insert new data and Delete existing data. Does 2PL extend naturally to support dynamic databases? The answer is yes, but the following example would suggest *no*.

[6]As we will see in Chapter 6, the disk Writes do not have to take place physically to complete the transaction.

Suppose we have two files representing banking information (see Fig. 3-1): an Accounts file that, for each account, gives the account number (Account#), the bank branch that holds the account (Location), and the amount of money in the account (Balance); and an Assets file that, for each bank branch (Location) gives the total assets of that branch (Total).

We have two transactions to execute on this database. Transaction T_1 reads all of the accounts in Tyngsboro[7] from the Accounts file, adds up their Balances, and compares that sum to the Total assets in Tyngsboro (in Assets). Transaction T_2 adds a new account [99, Tyngsboro, 50] by inserting a record into the Accounts file and then adding the Balance of that account to the Total assets in Tyngsboro. Here is one possible execution of these transactions:

Read_1(Accounts[339], Accounts[914], Accounts[22]);

Insert_2(Accounts[99, Tyngsboro, 50]);

Read_2(Assets[Tyngsboro]); /* returns 3858 */

Write_2(Assets[Tyngsboro]); /* writes 3908 */

Read_1(Assets[Tyngsboro]); /* returns 3908 */

This execution could have resulted from an execution in which both T_1 and T_2 were two phase locked. T_1 begins by locking the three Accounts records that it wants to Read. (It has to read all of the Accounts records to determine which ones are in Tyngsboro.) Then T_2 locks the record it is about to insert and, after inserting the record, locks the Tyngsboro record in Assets. After finishing with its update to Assets, T_2 releases both of its locks. Now T_1 can finish up by locking the Tyngsboro record in Assets, reading the Total for that Branch, and terminating.

Unfortunately, the execution is not SR. T_1 reads Accounts 339, 914, and 22, but when it reads the Assets of Tyngsboro, it gets a Total that includes Account 99. If T_1 had executed serially before T_2, then it would have correctly read the old Total for Tyngsboro, 3858. If T_1 had executed serially after T_2, then it would have read all four Accounts records and the correct new Total for Tyngsboro.

The problem is record 99 in Accounts. When T_1 first looks in Accounts, it doesn't find that record. However, when it looks in Assets a little later, it finds a Total that reflects the insertion of record 99. Record 99 is called a *phantom* record, because it seems to appear and disappear like a ghost.

The *phantom problem* is the concurrency control problem for dynamic databases. The example seems to show that 2PL does not guarantee correct executions for dynamic databases. Fortunately, the example is misleading and 2PL actually is a good method for synchronizing accesses to dynamic databases.

[7]Rhymes with "Kingsborough."

To see how the example is misleading, let's return to first principles. A basic assumption of our model is that a transaction communicates with other transactions *only* through Reads and Writes, and that all such Reads and Writes are synchronized by the scheduler. In the example under discussion, there is a hidden conflict between a Read from T_1 and a Write from T_2. Since T_1 read *all* of the records in Accounts, it must have *read* some control information that told it which records to read, and since T_2 inserted a record into Accounts, it must have *written* that control information. These Read and Write operations on the control information must be locked, just like any other accesses to shared data.

Suppose we adopt a straightforward implementation that locks control information. For example, suppose each file has an end-of-file marker (EOF) after the last record. To determine which records to read, T_1 reads (and locks) records until it reads (and locks) EOF. To insert a record, T_2 must move EOF, so it write locks it. T_1's and T_2's locks on EOF would prevent the incorrect execution under discussion: if T_1 reads EOF before T_2 tries to write it, then T_2 is unable to insert the new record until after T_1 reads Assets[Tyngsboro]; if T_2 writes EOF before T_1 tries to read it, then T_1 cannot finish scanning records in Accounts until after T_2 adds the new record and updates Assets[Tyngsboro].

Unfortunately, this straightforward implementation may perform poorly. Every transaction that inserts a record locks EOF, thereby preventing any other transaction from scanning or inserting into the file. In some cases this is unavoidable. For instance, in the example T_1 is scanning for Accounts records in Tyngsboro, and T_2 is inserting such a record. However, if T_2 were inserting a record into Marlboro instead of Tyngsboro, then T_1 and T_2 would not be accessing any records in common, so their conflict between accesses to EOF would be unnecessary. We can exploit this observation by using a technique called *index locking*.

Index Locking

Suppose each file has one or more *indices* associated with it. Each index is defined on exactly one field of the file, and contains a set of *index entries*. Each index entry has one value of the field on which it's defined and a list of pointers to the records that have that field value. Indices are commonly used in DBSs to speed up access to sets of records whose fields have given values.[8] In the example database of Fig. 3–1, we might create an index on the Location field of Accounts for this purpose.

When a transaction such as T_1 scans Accounts for records in Tyngsboro, it reads and locks the index entry for Tyngsboro. Since any transaction that

---

[8]Section 3.13 describes locking schedulers that are specialized for synchronizing access to tree-structured indices.

inserts a new record r in Tyngsboro must add a pointer to r in the index entry for Tyngsboro, it will try to lock that index entry and thereby conflict with T_1 as desired. If a transaction inserts a record into any other Location, it will access a different index entry and therefore won't conflict with T_1. Thus, T_1 only conflicts with transactions that insert records that it wants to read.

Recall that data item names are not interpreted by the LM. Therefore, a transaction can set a lock on an index entry even though the index does not physically exist. Thus, we can obtain the benefit of index locking without requiring the indices to exist. Each transaction that scans records sets a lock on an index entry that "covers" all of the records it's reading, and each transaction that inserts a record sets a lock on every index entry that would include the new record (if the index existed) and that may be locked by a scanning transaction. The effect is the same as if the transactions were locking indices that physically exist.

In a sense, a transaction that locks the Tyngsboro index entry is effectively locking all records that satisfy the predicate (Location = "Tyngsboro").

One can generalize index locking by allowing more complex predicates to be locked, such as conjunctions of such predicates. That is, the data item name stored in the lock table is actually a predicate. Two locks conflict if there could be a record that satisfies both predicates, that is, if the lock predicates are mutually satisfiable. This is called *predicate locking*. While more general than index locking, it is also more expensive, since it requires the LM to detect conflicts between arbitrary predicates. It is therefore not widely used.

3.8 LOCKING ADDITIONAL OPERATIONS

In some applications, there are periods with heavy Write traffic on certain data items, called *hot spots*. For example, in the banking database of Fig. 3–1, every deposit and withdrawal transaction for Accounts at a given branch Location requires updating the Assets record for that Location; during periods of peak load, the Assets records may become hot spots. When hot spots are present, many transactions may be delayed waiting for locks on hot spot data items. This performance problem can often be avoided by adding other types of operations to the standard repertoire of Read and Write.

For example, in deposits and withdrawals, Writes are used to add and subtract from the Total assets of Locations. If we implement Increment and Decrement as atomic operations, then most Writes can be replaced by these operations. Since Increment and Decrement commute, they can set weaker locks than Write operations, which do not commute. These weaker locks allow transactions to execute concurrently in situations where ordinary write locks would have them delay one another, thereby helping to relieve the bottleneck created by the hot spot. A version of this scheme is implemented in the Main Storage Data Base (MSDB) feature of IBM's IMS Fast Path.

The generalized locking scheme has four types of operations: Read, Write, Increment, and Decrement. It therefore has four types of locks: read locks, write locks, increment locks, and decrement locks. To define the conflict relation between lock types, we examine the corresponding operation types to determine which ones commute. To determine this, we have to be precise about what Increment and Decrement actually do. Let us define them this way:

Increment(x, *val*): add *val* to data item x.

Decrement(x, *val*): subtract *val* from data item x.

We assume that data item values can be arbitrary positive or negative numbers. To ensure that Increment and Decrement commute, we assume that neither operation returns a value to the transaction that issued it. Therefore, for any data item x and values val_I and val_D, the sequence of operations [Increment(x, val_I), Decrement(x, val_D)] produces exactly the same result as the sequence [Decrement(x, val_D), Increment(x, val_I)]. That is, each operation returns the same result (i.e., nothing) and the two operations leave x in the same final state, independent of the order in which they execute. Since they commute independent of the value of val_I and val_D, we drop the *val* parameters in what follows.

Increment(x) and Decrement(x) *do* conflict with Read(x) and Write(x). For example, Read(x) returns a different value of x depending on whether it precedes or follows Increment(x). Increment(x) produces a different value depending on whether it precedes or follows Write(x). Lock types should be defined to conflict in the same way that their corresponding operations conflict (we'll explain why in a moment). Therefore, the compatibility matrix for the lock types is as shown in Fig. 3–2. A "y" (yes) entry means that two locks of the types specified by the row and column labels for that entry can be simultaneously held on a data item by two different transactions, i.e., the two lock types do *not* conflict. An "n" (no) entry means that the two lock types cannot be concurrently held on a data item by distinct transactions, i.e., the lock types conflict.

Since increment and decrement locks do not conflict, different transactions can concurrently set these locks, most importantly on hot spot data items. Transactions that use these new lock types will therefore be delayed less frequently than if they had only used write locks, which do conflict.

This technique requires that Increment(x) and Decrement(x) be implemented atomically. Each of these operations must read x, update the value appropriately, and then write the result back into x. To ensure that this read-update-write process is atomic, no other operation can access the data item while the process is going on. Thus, while an Increment or Decrement is operating on a data item, the data item is effectively locked (against any other operations on that data item). This lock is only held for the duration of the Increment or Decrement operation. Once the operation is completed, this lock

	Read	Write	Increment	Decrement
Read	y	n	n	n
Write	n	n	n	n
Increment	n	n	y	y
Decrement	n	n	y	y

FIGURE 3–2
A Compatibility Matrix

can be released. However, the increment or decrement lock must be held until the transaction commits to satisfy the two phase rule and strictness.

To understand why this generalized form of 2PL works correctly, we need to revisit the proof of correctness, Lemmas 3.4 and 3.5 and Theorem 3.6. Relative to this issue, the critical step in the proof is: If $T_i \rightarrow T_j$ is in SG(H), then T_i and T_j had conflicting accesses to some data item, and T_i unlocked that data item before T_j locked it. As long as every pair of conflicting (i.e., noncommutative) operation types have associated lock types that conflict, then this argument is valid. Since we defined the lock types so that this property holds, this generalized form of 2PL is correct. We don't even need to modify the proof to handle this case, since it is expressed in terms of arbitrary conflicting operations, such as p and q.

So, we can easily add new operation types by following these simple rules:

1. Ensure that the implementation of each new operation type is atomic with respect to all other operation types.
2. Define a lock type for each new operation type.
3. Define a compatibility matrix for the lock types (for both the old and new operations) so that two lock types conflict iff the corresponding operation types on the same data item do not commute.

3.9 MULTIGRANULARITY LOCKING

So far we have viewed the database as an unstructured collection of data items. This is a very abstract view. In reality a data item could be a block or page of data, a file, a record of a file, or a field of a record. The *granularity* of a data item refers to that item's relative size. For instance, the granularity of a file is *coarser*, and the granularity of a field *finer*, than that of a record.

The granularity of data items is unimportant as far as correctness is concerned. The granularity *is* important, however, when it comes to performance. Suppose, for instance, that we use some version of 2PL. Using coarse granules incurs low overhead due to locking, since there are fewer locks to

FIGURE 3–3
A Lock Type Graph

manage. At the same time, it reduces concurrency, since operations are more likely to conflict. For example, if we lock files, two transactions that update the same file cannot proceed concurrently even if they access disjoint sets of records. Fine granularity locks improve concurrency by allowing a transaction to lock only those data items it accesses. But fine granularity involves higher locking overhead, since more locks are requested.[9]

Selecting a granularity for locks requires striking a balance between locking overhead and amount of concurrency. We can do even better than choosing a uniform "optimal" granule size for all data items by means of *multigranularity locking* (*MGL*). MGL allows each transaction to use granule sizes most appropriate to its mode of operation. *Long* transactions, those that access many items, can lock coarse granules. For example, if a transaction accesses many records of a file, it simply locks the entire file, instead of locking each individual record. *Short* transactions can lock at a finer granularity. In this way, long transactions don't waste time setting too many locks, and short transactions don't block others by locking large portions of the database that they don't access.

MGL requires an LM that prevents two transactions from setting conflicting locks on two granules that overlap. For example, a file should not be read locked by a long transaction if a record of that file is write locked by a short transaction. An unsatisfactory solution would be to require that the long transaction look at each record of the file to find out whether it may lock the file. This would defeat the very purpose of locking at coarse granularity — namely, to reduce locking overhead.

A better solution is possible by exploiting the natural hierarchical relationship between locks of different granularity. We represent these relationships by

[9]There are other factors that could cause concurrency to actually decrease when finer granularity is used. For a more detailed discussion of the effects of granularity on performance, see Section 3.12.

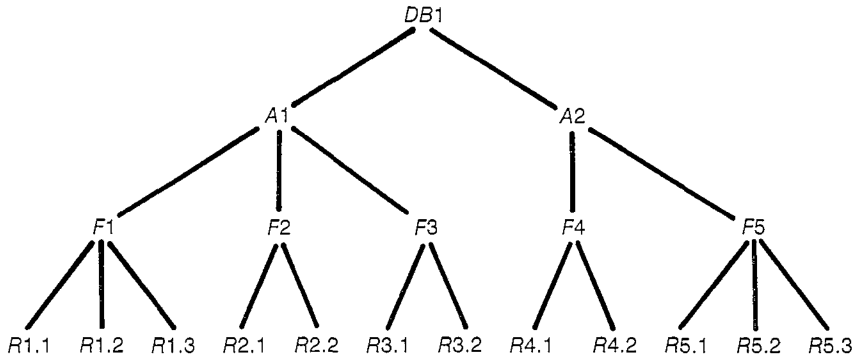

FIGURE 3–4
A Lock Instance Graph

	r	*w*	*ir*	*iw*	*riw*
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

FIGURE 3–5
A Compatibility Matrix for Multigranularity Locking

a *lock type graph*. Each edge in the graph connects a data type of coarser granularity to one of finer granularity. For example, in Fig. 3–3 areas (i.e., regions of disks) are of coarser granularity than files, which are of coarser granularity than records.

A set of data items that is structured according to a lock type graph is called a *lock instance graph* (see Fig. 3–4). The graph represents an abstract structure that is used only by the scheduler to manage locks of different granularities. It need not correspond to the physical structure of the data items themselves.

We'll assume that the lock instance graph is a tree. (Later we'll consider more general types of lock instance graphs.) Then a lock on a coarse granule x *explicitly* locks x and *implicitly* locks all of x's proper descendants, which are finer granules "contained in" x. For example, a read lock on an area implicitly read locks the files and records in that area.

It is also necessary to propagate the effects of fine granule locking activity to the coarse granules that "contain" them. To do this, each lock type has an associated *intention lock* type. So, in addition to read and write locks, we have *intention read* (*ir*) *locks* and *intention write* (*iw*) *locks*. Before it locks x, the scheduler must ensure that there are no locks on ancestors of x that implicitly lock x in a conflicting mode. To accomplish this, it sets intention locks on those ancestors. For example, before setting $rl[x]$ on record x, it sets *ir* locks on x's database, area, and file ancestors (in that order). For any y, $irl[y]$ and $wl[y]$ conflict. Thus, by setting $irl[y]$ on every ancestor y of x, the scheduler ensures that there is no $wl[y]$ that implicitly write locks x. For the same reason, $iwl[x]$ conflicts with $rl[x]$ and $wl[x]$ (see Fig. 3–5).

Suppose a transaction reads every record of a file and writes into a few of those records. Such a transaction needs both a read lock on the file (so it can read all records) and an *iw* lock (so it can write lock some of them). Since this is a common situation, it is useful to define an *riw* lock type. An $riwl[x]$ is logically the same as owning both $rl[x]$ and $iwl[x]$ (see Fig. 3–5).

For a given lock instance graph G that is a tree, the scheduler sets and releases locks for each transaction T_i according to the following *MGL protocol*:

1. If x is not the root of G, then to set $rl_i[x]$ or $irl_i[x]$, T_i must have an *ir* or *iw* lock on x's parent.
2. If x is not the root of G, then to set $wl_i[x]$ or $iwl_i[x]$, T_i must have an *iw* lock on x's parent.
3. To read (or write) x, T_i must own an r or w (or w) lock on some ancestor of x. A lock on x itself is an *explicit lock* for x; a lock on a proper ancestor of x is an *implicit lock* for x.
4. A transaction may not release an intention lock on a data item x, if it is currently holding a lock on any child of x.

Rules (1) and (2) imply that to set $rl_i[x]$ or $wl_i[x]$, T_i must first set the appropriate intention locks on all ancestors of x. Rule (3) implies that by locking x, a transaction has implicitly locked all of x's descendants in G. This implicit locking relieves a transaction from having to set explicit locks on x's descendants, which is the main reason for MGL. Rule (4) says that locks should be released in leaf-to-root order, which is the reverse of the root-to-leaf direction in which they were obtained. This ensures that a transaction never owns a read or write lock on x without owning the corresponding intention locks on ancestors of x.

For example, referring to Fig. 3–4, suppose that transaction T_1 wants to set $rl_1[F3]$. It must first set $irl_1[DB1]$, then $irl_1[A1]$, and finally $rl_1[F3]$. Now suppose T_2 tries to set $wl_2[R3.2]$. It must set $iwl_2[DB1]$, $iwl_2[A1]$, and $iwl_2[F3]$. It can obtain the first two locks, but not $iwl_2[F3]$, because it conflicts with $rl_1[F3]$. After T_1 releases $rl_1[F3]$, T_2 can set $iwl_2[F3]$ and $wl_2[R3.2]$. Now

suppose T_3 comes along and tries to set $rl_3[A1]$. It must set $irl_3[DB1]$, which it can do immediately, and then set $rl_3[A1]$. It cannot do this before T_2 releases $iwl_2[A1]$.

Correctness

The goal of the MGL protocol is to ensure that transactions never hold conflicting (explicit or implicit) locks on the same data item (i.e., node of the lock instance graph).

Theorem 3.7: Suppose all transactions obey the MGL protocol with respect to a given lock instance data graph, G, that is a tree. If a transaction owns an explicit or implicit lock on a node of G, then no other transaction owns a conflicting explicit or implicit lock on that node.

Proof: It is enough to prove the theorem for *leaf* nodes. For, if two transactions held conflicting (explicit or implicit) locks on a nonleaf node x, they would be holding conflicting (implicit) locks on all descendants and, in particular, all *leaf* descendants of x. Suppose then that transactions T_i and T_j own conflicting locks on leaf x. There are seven cases:

	transaction T_i	*transaction* T_j
1.	implicit r lock	explicit w lock
2.	implicit r lock	implicit w lock
3.	explicit r lock	explicit w lock
4.	explicit r lock	implicit w lock
5.	implicit w lock	explicit w lock
6.	implicit w lock	implicit w lock
7.	explicit w lock	explicit w lock

Case 1. By rule (3) of the MGL protocol, T_i owns $rl_i[y]$ for some ancestor y of x. By rule (2) of the MGL protocol (and induction), T_j must own an iw lock on every ancestor of x. In particular, it owns $iwl_i[y]$, which is impossible because the lock types iw and r conflict.

Case 2. By rule (3) of the MGL protocol, T_i owns $rl_i[y]$ for some ancestor y of x, and T_j owns $wl_j[y']$ for some ancestor y' of x. There are three subcases: (a) $y = y'$, (b) y is an ancestor of y', and (c) y' is an ancestor of y. Case (a) is impossible, because T_i and T_j are holding conflicting read and write locks (respectively) on $y = y'$. Case (b) is impossible because T_j must own $iwl_j[y]$, which conflicts with $rl_i[y]$. And case (c) is impossible because T_i must own $irl_i[y']$, which conflicts with $wl_j[y']$. Thus, the assumed conflict is impossible.

Cases (3) and (7) are obviously impossible. Cases (4) and (5) follow the same argument as case (1), and (6) follows the argument of (2). □

Implementation Issues

Theorem 3.7 says that the MGL protocol prevents transactions from owning conflicting locks. However, this is not sufficient for serializability. To ensure serializability, a scheduler that manages data items of varying granularities must use the MGL protocol *in conjunction with* 2PL. One way of framing the relationship between these two techniques is to say that 2PL gives rules for *when* to lock and unlock data items. The MGL protocol tells *how* to set or release a lock on a data item, given that data items of different granularities are being locked. For example, for a transaction T_i to read a record, 2PL requires that T_i set a read lock on the record. To set that read lock, MGL requires setting *ir* locks on the appropriate database, area, and file, and setting an *r* lock on the record.

Using MGL, the LM services commands to set and release locks in a conventional manner. When it gets a lock request, it checks that no other transaction owns a conflicting lock on the same data item, where the data item could be a database, area, file, or record. If no conflicting locks are set, then it grants the lock request by setting the lock. Otherwise, it blocks the transaction until either the lock request can be granted or a deadlock forces it to reject the request, thereby causing the transaction to abort. The LM need not know about the lock type graph, the lock instance graph, the MGL protocol, or implicit locks. Its only new feature is that it handles more lock types (namely, intention locks) using the expanded compatibility matrix.

Given the larger number of lock types, there are more types of lock conversion than simply converting a read lock to a write lock. For example, one might convert $irl_i[x]$ to $rl_i[x]$ or $riwl_i[x]$. To simplify this lock conversion activity, it is helpful to define the strength of lock types: lock type p is *stronger* than lock type q if for every lock type o, $ol_i[x]$ conflicts with $ql_j[x]$ implies that $ol_i[x]$ conflicts with $pl_j[x]$. For example, *riw* is stronger than r, and r is stronger than *ir*, but r and *iw* have incomparable strengths.

If a transaction owns $pl_i[x]$ and requests $ql_i[x]$, then the LM should convert $pl_i[x]$ into a lock type that is at least as strong as both p and q. For example, if $p = r$ and $q = iw$, then the LM should convert $rl_i[x]$ into $riwl_i[x]$. The strengths of lock types and the lock conversion rules that they imply can be derived from the compatibility matrix. However, this is too inefficient to do at run-time for each lock request. It is better to derive the lock conversion rules statically, and store them in a table that the LM can use at run-time (see Fig. 3–6). The lock conversion problems of deadlocks (cf. Section 3.2) and fair scheduling (cf. Section 3.6) must also be generalized to this expanded set of lock types.

		Old lock type				
		ir	*iw*	*r*	*riw*	*w*
Requested lock type	*ir*	*ir*	*iw*	*r*	*riw*	*w*
	iw	*iw*	*iw*	*riw*	*riw*	*w*
	r	*r*	*riw*	*r*	*riw*	*w*
	riw	*riw*	*riw*	*riw*	*riw*	*w*
	w	*w*	*w*	*w*	*w*	*w*

FIGURE 3–6
Lock Conversion Table
If a transaction has a lock of "old lock type" and requests a lock of "requested lock type," then the table entry defines the lock type into which the "old lock type" should be converted.

Lock Escalation

A system that employs MGL must decide the level of granularity at which a given transaction should be locking data items. Fine granularity locks are no problem. The TM or scheduler simply requests them one by one as it receives operations from the transaction. Coarse granularity locks are another matter. A decision to set a coarse lock is based on a prediction that the transaction is likely to access many of the data items covered by the lock. A compiler may be able to make such predictions by analyzing a transaction's program and thereby generating coarse granularity lock requests that will be explicitly issued by the transaction at run-time. If transactions send high level (e.g., relational) queries to the TM, the TM may be able to tell that the query will generate many record accesses to certain files.

The past history of a transaction's locking behavior can also be used to predict the need for coarse granularity locks. The scheduler may only be able to make such predictions based on the transaction's recent behavior, using a technique called *escalation*. In this case, transactions start locking items of fine granularity (e.g., records). If a transaction obtains more than a certain number of locks of a given granularity, then the scheduler starts requesting locks at the next higher level of granularity (e.g., files), that is, it escalates the granularity of the locks it requests. The scheduler may escalate the granularity of a transaction's lock requests more than once.

In Section 3.2, we showed that a deadlock results when two transactions holding read locks on a data item try to convert them to write locks. Lock escalation can have the same effect. For example, suppose two transactions are holding *iw* locks on a file and are setting *w* locks on records, one by one. If they both escalate their record locking activity to a file lock, they will both try to convert their *iw* lock to a *w* lock. The result is deadlock.

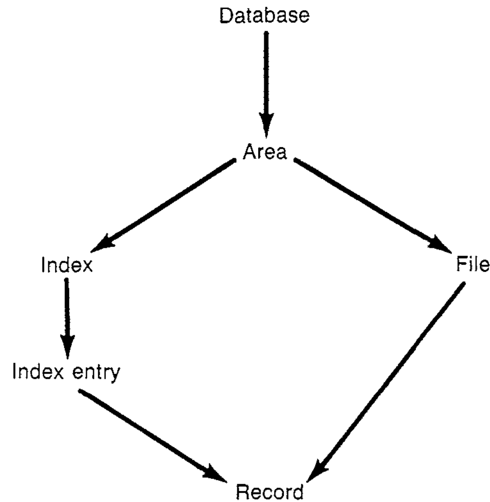

FIGURE 3–7
A Dag Structured Lock Type Graph

In some applications, lock escalations have a high probability of leading to lock conversions that cause deadlocks. In such cases lock escalation may be inappropriate. Instead, if a transaction gets too many fine granularity locks, it should be aborted and restarted, setting coarser granularity locks in its second incarnation. This may be less expensive than lock escalation, which may cause a deadlock.

Generalized Locking Graph

So far, we have assumed that the lock type graph is a tree. This is too restrictive for some applications. In particular, if we use indices, then we would like to be able to lock records of a file by locking indices, index entries, the file, or records. Thus, we are led to the locking type graph of Fig. 3–7, which is a rooted dag,[10] not a tree.

However, we cannot use the MGL protocol's rules for *w* and *iw* locks, as illustrated by the following example. Consider the banking database of Fig. 3–1, structured using the lock type graph of Fig. 3–7. Suppose there is an index on Location in Accounts, and that the Location index and the Accounts file are stored in area A1. Suppose T_1 has an *ir* lock on the database, on A1, and on the Location index for Accounts, and an *r* lock on the Tyngsboro index

[10] See Appendix, Section A.3, for the definition of rooted dag.

entry. Suppose another transaction T_2 has an *iw* lock on the database and A1 and a *w* lock on the Accounts file. This is an error, because T_1 has implicitly read locked all records pointed to by the Tyngsboro index entry and T_2 has implicitly write locked those very same records.

One solution to this problem is to require a transaction to set a *w* or *iw* lock on x only if it owns an *iw* lock on *all* parents of x. For example, to obtain a write lock on record 14, a transaction must have an *iw* lock on (say) the Tyngsboro index entry *and* the Accounts file. To implicitly write lock x, a transaction must explicitly or implicitly write lock *all* parents of x. So, to implicitly write lock the records in the Accounts file, it is not enough to set a write lock on the Accounts file. One must also write lock the Location index (or all of the index's entries). This prevents the problem of the last paragraph; if T_1 has an *ir* lock on the Location index, then T_2 cannot obtain a *w* lock on that index and therefore cannot implicitly lock any of the records in the Accounts file.

For a given lock instance graph G that is a dag, the scheduler sets and releases locks for each transaction T_i as follows:

1. If x is not the root of G, then to set $rl_i[x]$ or $irl_i[x]$, T_i must have an *ir* or *iw* lock on some parent of x.
2. If x is not the root of G, then to set $wl_i[x]$ or $iwl_i[x]$, T_i must have an *iw* lock on *all* of x's parents.
3. To read x, T_i must own an *r* or *w* lock on *some* ancestor of x. To write x, T_i must own, for every path from the root of G to x, a *w* lock for some ancestor of x along that path (i.e., it may own different locks for different paths). A lock on x itself is an *explicit lock* for x; locks on proper ancestors of x are *implicit locks* for x.
4. A transaction may not release an intention lock on a data item x if it is currently holding a lock on any child of x.

The proof that this protocol prevents transactions from owning conflicting (explicit or implicit) locks is similar to that of Theorem 3.7 (see Exercise 3.23).

3.10 DISTRIBUTED TWO PHASE LOCKING

Two phase locking can also be used in a distributed DBS. Recall from Section 1.4 that a distributed DBS consists of a collection of communicating sites, each of which is a centralized DBS. Each data item is stored at exactly one site. We say that the scheduler *manages* the data items stored at its site. This means that the scheduler is responsible for controlling access to these (and only these) items.

A transaction submits its operations to a TM. The TM then delivers each Read(x) or Write(x) operation of that transaction to the scheduler that manages x. When (and if) a scheduler decides to process the Read(x) or

Write(x), it sends the operation to its local DM, which can access x and return its value (for a Read) or update it (for a Write). The Commit or Abort operation is sent to all sites where the transaction accessed data items.

The schedulers at all sites, taken together, constitute a *distributed scheduler.* The task of the distributed scheduler is to process the operations submitted by the TMs in a (globally) serializable and recoverable manner.

We can build a distributed scheduler based on 2PL. Each scheduler maintains the locks for the data items stored at its site and manages them according to the 2PL rules. In 2PL, a Read(x) or Write(x) is processed when the appropriate lock on x can be obtained, which only depends on what other locks on x are presently owned. Therefore, each local 2PL scheduler has all the information it needs to decide when to process an operation, without communicating with the other sites. Somewhat more problematic is the issue of when to release a lock. To enforce the two phase rule, a scheduler cannot release a transaction T_i's lock until it knows that T_i will not submit any more operations to it *or any other scheduler.* Otherwise, one scheduler might release T_i's lock and some time later another scheduler might set a lock for T_i, thereby violating the two phase rule (see Exercise 3.26).

It would appear that enforcing the two phase rule requires communication among the schedulers at different sites. However, if schedulers use *Strict* 2PL, then they can avoid such communication. Here is why. As we said previously, the TM that manages transaction T_i sends T_i's Commit to all sites where T_i accessed data items. By the time the TM decides to send T_i's Commit to all those sites, it must have received acknowledgments to all of T_i's operations. Therefore, T_i has surely obtained all the locks it will ever need. Thus, if a scheduler releases T_i's locks after it has processed T_i's Commit (as it must under Strict 2PL), it knows that no scheduler will subsequently set any locks for T_i.

To prove that the distributed 2PL scheduler is correct we simply note that any history H it could have produced satisfies the properties of 2PL histories described in Propositions 3.1–3.3. By Theorem 3.6 then, H is SR. Moreover if the local schedulers use Strict 2PL, H is ST and therefore RC.[11]

The simplicity of this argument is a consequence of the fact that histories model centralized and distributed executions equally well. Since we'll typically specify the properties of histories generated by a scheduler without referring to whether it is a centralized or distributed one, the proof of correctness will apply to both cases.

---

[11]One may question the legitimacy of this argument, given that the Commit or Abort of a distributed transaction is processed by several sites, yet is represented as a single atomic event in a history. For the time being, we can view c_i or a_i as the atomic event corresponding to the moment T_i's TM received acknowledgments of the processing of T_i's Commit or Abort by *all* sites where T_i accessed data items. In Chapter 7 we'll have a lot more to say about why (and how) the commitment and abortion of a distributed transaction can be viewed as an atomic event.

3.11 DISTRIBUTED DEADLOCKS

As in the centralized case, a distributed 2PL scheduler must detect and resolve deadlocks. Timeouts can be used to guess the existence of deadlocks. Or we can explicitly detect deadlocks using WFGs.

The scheduler at site i can maintain a *local* waits-for graph, WFG_i, recording the transactions that wait for other transactions to release a lock on data items managed by that scheduler. Each WFG_i is maintained as described for centralized 2PL. The *global* waits-for graph, WFG, is the union of the local WFG_is.

Unfortunately, it is possible that WFG contains a cycle, and therefore the system is deadlocked, even though each WFG_i is acyclic. For example, consider a distributed scheduler consisting of two 2PL schedulers: scheduler A manages x and scheduler B manages y. Suppose that we have two transactions:

$$T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_2 = r_2[y] \rightarrow w_2[x] \rightarrow c_2$$

Now consider the following sequence of events:

1. Scheduler A receives $r_1[x]$ and sets $rl_1[x]$.
2. Scheduler B receives $r_2[y]$ and sets $rl_2[y]$.
3. Scheduler B receives $w_1[y]$. Since $wl_1[y]$ conflicts with $rl_2[y]$, the scheduler makes $w_1[y]$ wait and adds the edge $T_1 \rightarrow T_2$ to WFG_B.
4. Scheduler A receives $w_2[x]$. That scheduler delays $w_2[x]$ and adds $T_2 \rightarrow T_1$ to WFG_A.

The union of WFG_A and WFG_B contains the cycle $T_1 \rightarrow T_2 \rightarrow T_1$, and we therefore have a deadlock. But this deadlock is not detected by either scheduler's local WFG, since both WFG_A and WFG_B are acyclic. Such a deadlock is called a *distributed deadlock*. To discover such deadlocks, all the schedulers must put their local WFGs together and check the resulting global WFG for cycles.

Global Deadlock Detection

A simple way to do this is for each scheduler to send changes to its WFG_i to a special process, the *global deadlock detector*. The global deadlock detector keeps the latest copy of the local WFG that it has received from each scheduler. It periodically takes the union of these local WFGs to produce a global WFG, and checks it for cycles.

Since the global WFG is only periodically analyzed for cycles, deadlocks may go undetected for a while. As in centralized DBSs, the main penalty for the delay in detecting deadlocks is that deadlocked transactions are holding resources that they aren't using and won't use until the deadlock is broken. The communications delays in shipping around the local WFGs contribute to

the delay of distributed deadlock detection, so this delay may be longer than in a centralized DBS.

Once the global deadlock detector finds a deadlock, it must select a victim to abort. This is done based on the same considerations as centralized deadlocks. Therefore, in addition to receiving local WFGs, the global deadlock detector needs information from each site to help it make good victim selections. Moving this information around costs more messages. A technique called *piggybacking* can be used to reduce this message cost.

In piggybacking, n messages that originate at one site and that are all addressed to a common other site are packaged up in one large message. Thus, the number of messages is reduced by a factor of n. Since communication cost is generally a function of the number of messages exchanged (as well as the amount of information), reducing the number of messages in this way can significantly reduce communication cost.

This technique can be applied to global deadlock detection. Each site has its local WFG to send to the global deadlock detector. It also has information for victim selection to send, such as each transaction's resource consumption or abortion cost. Combining this information into one message reduces the cost of sending it to the global deadlock detector.

Phantom Deadlocks

Another problem with distributed deadlock detection relates again to the delay in detecting deadlocks. Clearly, every deadlock will eventually be detected. It may take a while before all of the edges in the deadlock cycle are sent to the global deadlock detector. But since deadlocks don't disappear spontaneously, eventually all of the edges in the cycle will propagate to the deadlock detector, which will then detect the deadlock.

But what about edges in the global WFG that are out-of-date, due to the delay in sending local WFGs to the global deadlock detector? Might the global deadlock detector find a WFG cycle that isn't really a deadlock? Such incorrectly detected deadlocks are called *phantom deadlocks*.

For example, suppose a scheduler sends its local WFG containing some edge $T_i \rightarrow T_j$ to the global deadlock detector. Suppose that shortly after this local WFG is sent, T_j releases its locks, thereby unblocking T_i. Since the scheduler only sends its local WFG periodically, the global detector may use the copy of the graph containing $T_i \rightarrow T_j$ to look for cycles. If it finds a cycle containing that edge, it may believe it has found a genuine deadlock. But the deadlock is a *phantom deadlock*. It isn't real, because one edge in the cycle has gone away, unknown to the global deadlock detector.

Phantom deadlocks can surely happen if a transaction that was involved in a real deadlock spontaneously aborts. For example, a deadlocked transaction might be aborted because some hardware resource (e.g., a terminal) being used by the transaction failed. Although the deadlock wasn't detected, it was

broken by the spontaneous abortion. If the global deadlock detector finds the deadlock before it learns of the abortion, it may unnecessarily abort another transaction.

It is interesting that phantom deadlocks can *only* occur due to spontaneous abortions, as long as all transactions are two phase locked. To see this, assume that each lock operation corresponds to a database operation (i.e., no intention locks), and no transaction spontaneously aborts. Suppose the global deadlock detector found a cycle, $T_1 \rightarrow T_2 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$, but there really is no deadlock. Since no transaction spontaneously aborts, and there is no deadlock, all transactions eventually commit. In the resulting execution, since each lock operation corresponds to a database operation, for each edge $T_i \rightarrow T_{i+1}$ in this WFG cycle, there must be an edge $T_{i+1} \rightarrow T_i$ in the SG of the execution. This is true even if the deadlock cycle is a phantom cycle; each edge in the cycle existed at *some* time, so the conflict between the database operations for each edge is real and must produce an SG edge. However, that means that the SG has the same cycle as the WFG, but in the opposite direction. This is impossible, because all transactions were two phase locked. We leave the extension of this argument for intention locks as an exercise (Exercise 3.25).

Distributed Cycle Detection

Most WFG cycles are of length two. To see why, consider how a WFG grows. Suppose we start with all active transactions waiting for no locks, so the WFG has no edges. As transactions execute, they become blocked waiting for locks, so edges begin to appear. Early in the execution, most transactions are not blocked, so most edges will correspond to a transaction's being blocked by a lock owned by an unblocked transaction. But as more transactions become blocked, there is an increased chance that a transaction T_i will be blocked by a lock owned by a blocked transaction T_j. Such an event corresponds to creating a path of length two (i.e., T_i waits for T_j, which is waiting for some other transaction).

Suppose all transactions access the same number of data items, and all data items are accessed with equal probability. Then it can be shown that, on the average, blocked and unblocked transactions own about the same number of locks. This implies that if transactions randomly access data items, then *all* transactions (both blocked and unblocked ones) are equally likely to block a given unblocked transaction. So the probability that an edge creates a path of length two (or three, four, etc.) is proportional to the fraction of blocked transactions that are on the ends of paths of length one (or two, three, etc.). Since initially there are no paths, this must mean that short paths predominate. That is, most transactions are unblocked, many fewer are blocked at the ends of paths of length two, many fewer still are at the ends of paths of length three, and so forth. Hence, an edge that completes a cycle has a much higher chance

of connecting an unblocked transaction to one at the end of a path of length one than to one at the end of longer paths. Therefore, most WFG cycles are of length two. For typical applications, over 90% of WFG cycles are of length two.

This observation that cycles are short may make global deadlock detection a less attractive choice than it first appears to be. With global deadlock detection there may be a significant delay and overhead in assembling all of the local WFGs at the global deadlock detector. Thus, a distributed deadlock might go undetected for quite a while. This is especially annoying because most deadlocks involve only two transactions. If the two sites that participate in the deadlock communicate directly, they can detect the deadlock faster by exchanging WFGs with each other. But if every pair of sites behaved this way, then they would all be functioning as global deadlock detectors, leading to much unnecessary communication.

Path pushing is a distributed deadlock detection algorithm that allows all sites to exchange deadlock information without too much communication. Using path pushing, each site looks for cycles in its local WFG *and* lists all paths in its WFG. It selectively sends portions of the list of paths to other sites that may need them to find cycles. Suppose site A has a path $T_i \rightarrow \cdots \rightarrow T_j$. It sends this path to every site at which T_j might be blocked, waiting for a lock. When a site, say B, receives this path, it adds the path's edges to its WFG. Site B's WFG may now have a cycle. If not, B still may find some new and longer paths that neither A nor B had seen before. It lists these paths and sends them to sites that may have more edges to add to the paths.

Every cycle in the global WFG can be decomposed into paths, each of which exists in one local WFG. Using this algorithm, each site sends its paths to other sites that may be able to extend them, by concatenating them with paths that (only) it knows about. Eventually, each path in a cycle will be "pushed" all the way around the cycle and the cycle will be detected by some site.

For example, suppose sites A, B, and C have the following WFGs.

$$\begin{aligned} WFG_A &= T_1 \rightarrow T_5 \rightarrow T_3 \\ WFG_B &= T_3 \rightarrow T_4 \\ WFG_C &= T_4 \rightarrow T_1 \end{aligned}$$

Site A sends the one and only path in WFG_A to site B, whose graph is now

$$WFG_B = T_1 \rightarrow T_5 \rightarrow T_3 \rightarrow T_4$$

Since WFG_B has changed, site B sends its one and only path to C, whose WFG becomes

$$WFG_C = T_1 \rightarrow T_5 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$$

which contains a cycle.

This method usually detects short cycles faster than global deadlock detection. If T_1 is waiting for T_2 at site A and T_2 is waiting for T_1 at site B, then as soon as either A or B sends its paths to the other site, the receiving site will detect the deadlock. By contrast, using global deadlock detection, both A and B would have to wait until both of them send their WFGs to the global deadlock detector, which then has to report the deadlock back to both A and B.

Path pushing sounds fine, as long as each site knows where to send its paths. It could send them to all sites, and in some cases that is the best it can do. But this does involve a lot of communication. The communications cost could easily overshadow the benefit of detecting short cycles more quickly.

One can avoid some of the communication by observing that not all paths need to be sent around. Consider a deadlock cycle that is a concatenation of paths, $p_1, \ldots, p_n$, each of that is local to one site's WFG, say $\text{site}_1, \ldots, \text{site}_n$ (respectively). So far, we have been pushing all of those paths around the cycle. So to start, site_1 sends p_1 to site_2, site_2 sends p_2 to site_3, etc. Now each site knows about longer paths, so site_1 sends $[p_n, p_1]$ to site 2, site 2 sends $[p_1, p_2]$ to site_3, etc. Using this approach, *every* site will end up detecting the deadlock, which is clearly more than what's necessary. Even worse, two sites that detect the same deadlock might choose different victims.

To reduce the traffic, suppose that (1) each transaction, T_i, has a unique name, $Id(T_i)$, which identifies it, and (2) *Id*s are totally ordered. In every cycle, at least one path $T_i \rightarrow \cdots \rightarrow T_j$ has $Id(T_i) < Id(T_j)$. (If no path had this property, then $T_i \rightarrow \cdots \rightarrow T_i$ implies $Id(T_i) > Id(T_i)$, a contradiction.) If we only send around paths that have this property, we will still find every cycle. But on average, we will only be sending half as many paths. Therefore, after a site produces a list of paths, it should only send those that have the property.

Communications traffic can be controlled further if each transaction is only active at one site at a time. Suppose that when a transaction T_j executing at site A wants to access data at another site B, it sends a message to B, stops executing at A, and begins executing at B. (Essentially, T_j is making a remote procedure call from A to B.) It does not continue executing at A until B replies to A that it is finished executing its part of the transaction.

When site A finds a path in its WFG from T_i to T_j, it need only send it to B. A knows that B is the only place where T_j could be executing and thereby be blocked waiting for a lock. Of course, T_j at B may have sent a message to C, and so may be stopped at B and now executing at C. But then B will send the path $T_i \rightarrow \cdots \rightarrow T_j$ to C the next time it performs deadlock detection. Eventually, the path $T_i \rightarrow \cdots \rightarrow T_j$ will make its way to every site at which T_j could be waiting for a lock.

In the resulting algorithm, each site, say A, performs the following steps. Site A periodically detects cycles in its local WFG. For each cycle, it selects a victim and aborts it. It then lists all paths not in cycles. For each such path, $T_i \rightarrow \cdots \rightarrow T_j$, if

1. $Id(T_i) < Id(T_j)$, and
2. T_j was formerly active at site A, but is now stopped, waiting for a response from another site B,

then A sends the path to B. When a site receives a list of paths from another site, it adds those edges to its WFG and performs the above steps.

We explore additional simplifications to this algorithm in Exercises 3.27 and 3.28.

Timestamp-based Deadlock Prevention

Deadlock prevention is a cautious scheme in which the scheduler aborts a transaction when it determines that a deadlock might occur. In a sense, the timeout technique described earlier is a deadlock prevention scheme. The system doesn't know that there is a deadlock, but suspects there might be one and therefore aborts a transaction.

Another deadlock prevention method is to run a test at the time that the scheduler is about to block T_i because it is requesting a lock that conflicts with one owned by T_j. The test should guarantee that if the scheduler allows T_i to wait for T_j, then deadlock cannot result. Of course, one could never let T_i wait for T_j. This trivially prevents deadlock but forces many unnecessary abortions. The idea is to produce a test that allows waiting as often as possible without ever allowing a deadlock.

A better test uses a priority that TMs assign to each transaction. Before allowing T_i to wait for T_j, a scheduler compares the transactions' priorities. If T_i has higher priority than T_j, then T_i is allowed to wait; otherwise, it is aborted. In this scheme, T_i waits for T_j only if T_i has higher priority than T_j. Therefore, for each edge $T_i \rightarrow T_j$ in WFG, T_i has higher priority than T_j. The same is true of longer paths that connect T_i to T_j. If there were a cycle in the WFG connecting T_i to itself, then T_i would have a priority higher than itself, which is impossible because each transaction has a single priority. Thus, deadlock is impossible.

However, this scheme may be subject to a different misfortune that prevents a transaction from terminating. If priorities are not assigned carefully, it is possible that every time a transaction tries to lock a certain data item, it is aborted because its priority isn't high enough. This is called *livelock* or *cyclic restart*.

For example, suppose each TM uses a counter to assign a priority to each transaction when it begins executing or when it restarts after being aborted. Suppose a TM supervises the execution of T_i and T_j as follows.

1. The TM assigns T_i a priority of 1.
2. T_i issues $w_i[x]$, causing it to set $wl_i[x]$.
3. The TM assigns T_j a priority of 2.

4. T_j issues $w_j[y]$, causing it to set $wl_j[y]$.
5. T_i issues $w_i[y]$. Since T_j has set $wl_j[y]$, we have to decide whether to allow T_i to wait. Since T_i's priority is lower than T_j's, T_i is aborted.
6. The TM restarts T_i, assigning it a priority of 3.
7. T_i issues $w_i[x]$, causing it to set $wl_i[x]$.
8. T_j issues $w_j[x]$. Since T_i has set $wl_i[x]$ and has a higher priority than T_i, T_j is aborted.
9. The TM restarts T_j, assigning it a priority of 4.
10. T_j issues $w_j[y]$, causing it to set $wl_j[y]$.

We are now in exactly the same situation as step (4). If the transactions follow the same sequence of requests that they did before, they will each cause the other one to abort as before, perhaps forever. They are in a livelock.

Livelock differs from deadlock because it doesn't prevent a transaction from executing. It just prevents the transaction from completing because it is continually aborted. One way to avoid livelock is to ensure that each transaction eventually has a high enough priority to obtain all of the locks that it needs without being aborted. This can be accomplished by using a special type of priority called timestamps.

Timestamps are values drawn from a totally ordered domain. Each transaction T_i is assigned a timestamp, denoted $\text{ts}(T_i)$, such that if $T_i \neq T_j$ then either $\text{ts}(T_i) < \text{ts}(T_j)$ or $\text{ts}(T_j) < \text{ts}(T_i)$.

Usually, TMs assign timestamps to transactions. If there is only one TM in the entire system, then it can easily generate timestamps by maintaining a counter. To generate a new timestamp, it simply increments the counter and uses the resulting value. If there are many TMs, as in distributed DBSs, then a method is needed to guarantee the total ordering of timestamps generated by different TMs. It is desirable to find a method that doesn't require the TMs to communicate with each other, which would make the timestamp generation activity more expensive.

The following technique is usually used to make this guarantee. Each TM is assigned a unique number (its process or site identifier, for example). In addition, each TM maintains a counter as before, which it increments every time it generates a new timestamp. However, a timestamp is now an ordered pair consisting of the current value of the counter followed by the TM's unique number. The pairs are totally ordered, first by their counter value and second, in case of ties, by their unique TM numbers.

The local counter used by each TM can be an actual clock. If a clock is used, then the TM obviously should not increment it to guarantee uniqueness. Instead, it should simply check that the clock has ticked between the assignment of any two timestamps.

Since timestamps increase monotonically with time and are unique, if a transaction lives long enough it will eventually have the smallest timestamp

(i.e., will be the oldest) in the system. We can use this fact to avoid livelock by using a rule for priority-based deadlock prevention that never aborts the oldest active transaction. Every transaction that is having trouble finishing due to livelock will eventually be the oldest active transaction, at which point it is guaranteed to finish.

Suppose we define a transaction's priority to be the inverse of its timestamp. Thus, the older a transaction, the higher its priority. We can then use timestamps for deadlock prevention, without risking livelock, as follows. Suppose the scheduler discovers that a transaction T_i may not obtain a lock because some other transaction T_j has a conflicting lock. The scheduler can use several strategies, two of which are:

Wait-Die: **if** $\text{ts}(T_i) < \text{ts}(T_j)$ **then** T_i waits **else** abort T_i.

Wound-Wait: **if** $\text{ts}(T_i) < \text{ts}(T_j)$ **then** abort T_j **else** T_i waits.

The words *wound*, *wait*, and *die* are used from T_i's viewpoint; T_i wounds T_j, causing T_j to abort; T_i waits; and T_i aborts and therefore dies. In both methods, only the younger of the two transactions is aborted. Thus, the oldest active transaction is never aborted by either method.

To ensure these methods are not subject to livelock, two other restrictions are needed. First, the timestamp generator must guarantee that it only generates a finite number of timestamps smaller than any given timestamp. If this were not true, then a transaction could remain in the system indefinitely without ever becoming the oldest transaction. Second, when an aborted transaction is restarted, it uses its old timestamp. If it were reassigned a new timestamp every time it was restarted, then it might never become the oldest transaction in the system.

Notice that wounding a transaction might not cause it to abort. The definition of Wound-Wait should really be:

if $\text{ts}(T_i) < \text{ts}(T_j)$ **then** *try to* abort T_j **else** T_i waits.

The scheduler can only *try* to abort T_j because T_j may have already terminated and committed before the scheduler has a chance to abort it. Thus, the abort may be ineffective in killing the transaction. That's why it's called *wound* and not *kill*; the scheduler wounds T_j, in a (possibly unsuccessful) attempt to kill it. But this still avoids the deadlock, because the wounded transaction releases its locks whether it commits or aborts.

Wound-Wait and Wait-Die behave rather differently. In Wound-Wait, an old transaction T_i pushes itself through the system, wounding every younger transaction T_j that it conflicts with. Even if T_j has nearly terminated and has no more locks to request, it is still vulnerable to T_i. After T_i aborts T_j and T_j restarts, T_j may again conflict with T_i, but this time T_j waits.

By contrast, in Wait-Die an old transaction T_i waits for each younger transaction it encounters. So as T_i ages, it tends to wait for more younger transactions. Thus, Wait-Die favors younger transactions while Wound-Wait

favors older ones. When a younger transaction T_j conflicts with T_i, it aborts. After it restarts, it may again conflict with T_i and therefore abort again — a disadvantage relative to Wound-Wait. However, once a transaction has obtained all of its locks, it will not be aborted for deadlock reasons — an advantage over Wound-Wait.

Comparing Deadlock Management Techniques

Each approach to deadlock management has its proponents. Many centralized DBSs, such as IBM's DB2 and RTI's INGRES, use WFG cycle detection. Each of the distributed techniques we have described is implemented in a commercial product, and each putatively works well: Tandem uses timeout; Distributed INGRES uses centralized deadlock detection; IBM's System R* prototype uses path pushing; and GE's MADMAN uses timestamp-based prevention.

3.12 LOCKING PERFORMANCE[12]

To accept published guidelines on locking performance requires a leap of faith, because the results are derived with simplistic assumptions and the state-of-the-art is unsettled. Nevertheless, an understanding of locking performance is pivotal to quality system design. This section should be read as preliminary results of an immature field.

Throughout this section, we will assume that all transactions require the same number of locks, all data items are accessed with equal probability, and all locks are write locks. The transactions use Strict 2PL: data items are locked before they are accessed, and locks are released only after the transactions commit (or abort). The DBS is centralized, so there is no communication cost. However, the DBS may be running on a machine with two or more tightly coupled processors.

Resource and Data Contention

In any multiprogramming system, the amount of work done on the system cannot increase linearly with the number of users. When there is *resource contention* over memory space, processor time, or I/O channels, queues form and time is wasted waiting in the queues. In DBSs that use locking, queues also form because of delays due to lock conflicts, called *data contention*.

Locking can cause *thrashing*. That is, if one increases the number of transactions in the system, throughput will increase up to a point, then drop. The

[12]Written by Dr. Y.C. Tay, Mathemathics Department, National University of Singapore. Unlike other sections, this section mentions results that are not derived in the book. For adequate justifications of these conclusions, we refer the reader to the bibliographic notes.

user usually observes this as a sudden increase in response time. This phenomenon is similar to thrashing in operating systems. There, the throughput drops due to time wasted in page faults when too many processes each have too little space. It is the result of resource contention — too many processes fighting over main memory. In DBSs, thrashing *can be caused by data contention alone*. In an idealized system with unlimited hardware resources, so that transactions queue only for data and never for resources, thrashing may still occur.

We distinguish two forms of thrashing: *RC-thrashing*, which occurs in systems with resource contention and no data contention, and *DC-thrashing*, which occurs in an idealized system with data contention and no resource contention. Although all systems have some resource contention, DC-thrashing is a useful concept.

Thrashing

Resource and data contention produce rather different forms of thrashing. With RC-thrashing, the system is busy transferring pages in and out of memory, so user processes make little progress. This suggests that DC-thrashing may be caused by transaction restarts induced by deadlocks. If the deadlock rate is high, then transactions are busy being repeatedly restarted, so transactions make little progress. However, DC-thrashing is in fact not caused by restarts, but by blocking.

Measurements of experimental and commercial DBSs indicate that deadlocks are much rarer than conflicts. Simulations also show that, up to the DC-thrashing point, transactions spend much more time waiting in lock queues than in being restarted. Moreover, the restart rate can be as low as 1-2% of throughput when DC-thrashing happens. But the most conclusive evidence is that beyond the DC-thrashing point, increasing the number of transactions actually decreases the number of transactions that are not blocked. That is, adding one more transaction causes more than one transaction (on average) to be blocked. Thus, whereas RC-thrashing happens because the system is busy doing wasteful work, DC-thrashing happens because too many transactions are tied up in lock queues, thus reducing system utilization.

It does not even take much blocking to cause DC-thrashing. At the DC-thrashing point, the average length of a lock queue could be less than one, and the average depth of a tree in the waits-for graph less than two. (The latter implies that, up to the DC-thrashing point, most deadlock cycles have only two transactions.) Hence, if half the transactions are blocked, the system is probably thrashing.

Although blocking is the dominant performance factor up to the DC-thrashing point, the effect of deadlocks does increase at a much faster rate than blocking. Beyond the thrashing point, restarts rapidly overtake blocking as the dominant factor.

Blocking and Restarts

Locking resolves conflicts either by blocking a transaction or by aborting and restarting it. Restarts are obviously undesirable, since work is wasted. The way blocking degrades performance is more subtle. Blocking lets a transaction hold locks without doing anything with them, even while other transactions are waiting to acquire those locks. Through DC-thrashing, we have seen how seriously this can affect performance.

Both restarts and blocking are bad for performance. But which is worse? Since Strict 2PL may be thrashing and yet have a very low deadlock rate, it resolves almost all conflicts by blocking. Therefore, let us call Strict 2PL a *blocking policy.* Alternatively, a *pure restart policy* simply aborts a transaction whenever it requests a lock that is already held by another transaction, and restarts the aborted transaction when the other releases the lock. Thus, a pure restart policy resolves all conflicts by restarts. Comparing these two policies is a way of comparing the performance effect of blocking and restarts.

Intuitively, a pure restart policy is very severe. One might expect it to perform badly compared with a blocking policy. Surprisingly, this is not necessarily so. Let the *multiprogramming level* (MPL) refer to the number of active transactions. Since aborted transactions waiting to restart consume minimal resources, we exclude them from MPL. However, since transactions blocked in lock queues by a blocking policy consume resources (mainly, memory space) as transactions in resource queues do, we include them in MPL.

Given the same MPL and under two conditions (see the next paragraph), a pure restart policy has a throughput that is only slightly lower than that of a blocking policy before the latter's DC-thrashing point. Furthermore, when DC-thrashing sets in for the blocking policy, the pure restart policy has a higher throughput. (See Fig. 3–8. Note that this comparison does not take resource contention into account yet; if there were no conflicts, the throughput would increase linearly with MPL in this figure.)

The two required conditions are quick transaction abortion and low resource contention. If abortions take too long, they will slow down the throughput of a pure restart policy, thus making it inferior to a blocking policy, which is only marginally affected by abort time since it has a low deadlock rate. Resource contention also hurts a pure restart policy more than a blocking policy. With the latter, some transactions are blocked in lock queues, so fewer transactions compete for resources. (Thus, data contention alleviates resource contention for a blocking policy.) Since resource contention causes transactions to waste time waiting in resource queues, it degrades the throughput of a blocking policy less than that of a pure restart policy. This too can make a pure restart policy consistently inferior (see Fig. 3–9).

Therefore, our intuition that a pure restart policy has worse throughput than a blocking policy is based on the assumption that restarts either take a long time or add too much resource contention. However, both assumptions

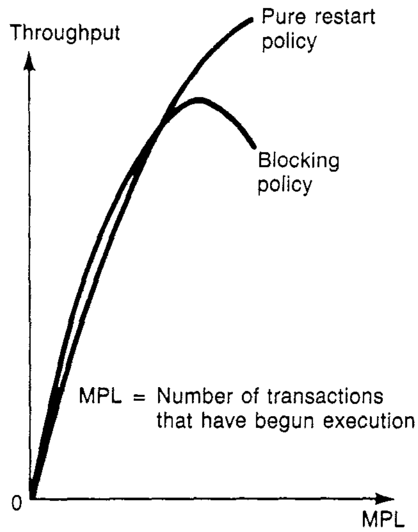

FIGURE 3–8
Throughput of Blocking and Pure Restart Policies with No Resource Contention

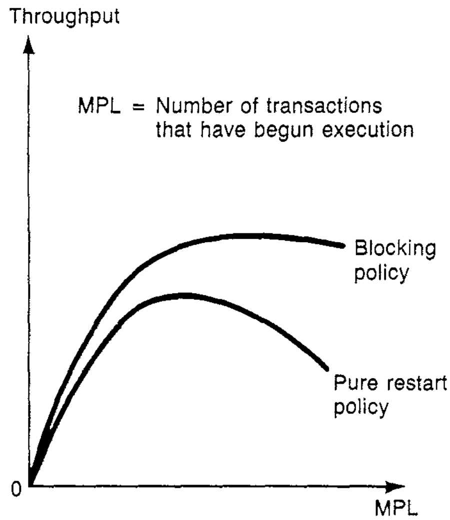

FIGURE 3–9
Possible Effect of Resource Contention on Fig. 3–8

could be violated. A clever implementation can make transaction abortion fast. And resource contention can perhaps be minimized by giving each transaction a dedicated microprocessor, so there is no contention for CPU cycles. Hence, a pure restart policy may be feasible.

Still, a pure restart policy has a longer response time than a blocking policy, even when their throughputs are similar. This is because an aborted transaction must wait for the conflicting transaction to release the lock before restarting, thus increasing its response time.

Therefore, except for the response time difference, blocking a transaction for a conflict may not be better than restarting it. Blocking is selfish. A blocked transaction can preserve what it has done, and prevent transactions that need its locks from making progress. Restarting is self-sacrificing. Since a conflict prevents a transaction from proceeding, restarting it frees its locks, so that it will not hinder others. When data contention becomes intense, altruism is the better policy. This is why a pure restart policy has a higher throughput than a blocking policy when the latter suffers from DC-thrashing, provided the two conditions hold. One may therefore consider restarts as a means of overcoming the upper bound that blocking imposes on the throughput through DC-thrashing.

Predeclaration

Another way to exceed the throughput limit that blocking imposes is to replace *Basic 2PL*, where a transaction sets locks as it needs them, by *Conservative 2PL*, where it obtains them before it begins. As the number of transactions increases, the throughput under Basic 2PL is initially higher than under Conservative 2PL, but eventually becomes lower. However, resource contention can change this. Under light data contention, Basic 2PL delays fewer transactions, and so suffers more resource contention. Its throughput is therefore reduced more, and may become consistently lower than Conservative 2PL.

Resource contention aside, how can Conservative 2PL have a higher throughput than Basic 2PL? Intuitively, since Basic 2PL only sets locks as they are needed, it should have more concurrency and therefore higher throughput than Conservative 2PL. This is true when data contention is light. But when it becomes heavy, as when DC-thrashing sets in, Basic 2PL in fact causes transactions to hold locks longer than under Conservative 2PL, thereby lowering throughput.

Conservative 2PL is sometimes favored because it avoids deadlocks. However, DC-thrashing occurs even if deadlocks are rare. Conservative 2PL should therefore be first considered as a means of bringing throughput above the limit set by blocking through DC-thrashing. Its advantage in deadlock avoidance is secondary.

A Bound on Workload

We have said little about performance beyond the DC-thrashing point. Indeed, we have assumed that a system will not be driven beyond that point, since there is no performance gain to be had. DC-thrashing thus defines an operating region for the combination of parameters that affect the performance. What is this region?

Suppose N is the MPL, k the number of locks a transaction requires, and D the number of data items, where data item is the locking granularity. (Note that, in general, k is less than the number of data accesses a transaction makes. For instance, if a data item is a file, two writes on one particular file would require only one lock.) Then a measure of the data contention is the *DC-workload* $W = k^2N/D$. DC-thrashing occurs at about $W = 1.5$, so the operating region is roughly bounded by $k^2N/D < 1.5$. (This number 1.5 was numerically obtained from a performance model, and confirmed through simulations. It is not known why DC-thrashing occurs at this particular value of W.)

The value 1.5 is almost surely optimistic. It is based on the assumption that accesses are uniform over the database. In reality, access patterns are skewed, which causes DC-thrashing to occur earlier. Furthermore, DC-thrashing does not account for resource contention, which further reduces the throughput. Therefore, real DBSs thrash before the DC-thrashing point. Hence, the value 1.5 only indicates the order of magnitude of the bound on the DC-workload.

MPL, Transaction Length, and Granularity

Bearing the caveat in mind, MPL should therefore be less than $1.5D/k^2$ for given k and D. This bound should only act as a guide in planning a system. The true bound will quickly reveal itself once the system is built.

Other than thrashing, there is another constraint on the MPL. Although throughput increases with N (up to the thrashing point), the deadlock rate increases even faster. If restarts are expensive, they may further reduce the number of active transactions that can be handled.

As expected, increasing the number of locks per transaction reduces the throughput. It also increases the number of deadlocks per transaction completion. Transactions should therefore be kept short. Long transactions should be broken into smaller ones, if possible. For a quantitative but simplistic argument, suppose N transactions requiring k locks each are broken into $2N$ transactions requiring k^2 locks each. The DC-workload then drops from k^2N/D to $(k/2)^2(2N)/D = k^2N/2D$, thus reducing the data contention.

Besides the number of transactions and the number of locks they need, another parameter in the DC-workload is the number of data items D. A small D (coarse granularity) implies more data is covered by each lock. A large D (fine granularity) implies the opposite.

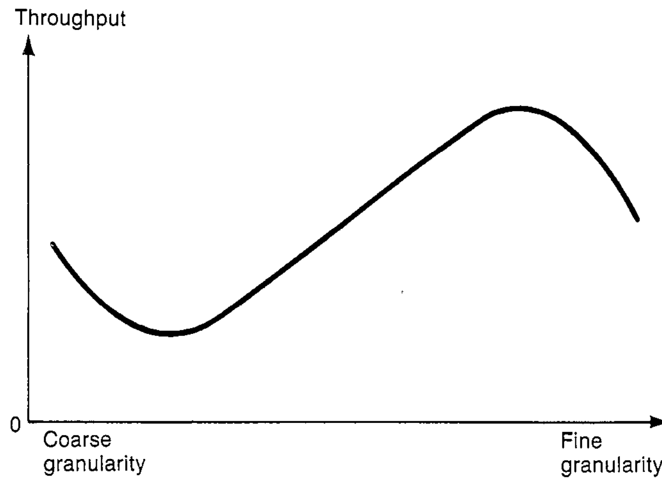

FIGURE 3–10
The General Granularity Curve

Three factors determine the effect of granularity on performance. One is locking overhead. The finer the granularity, the more locks a transaction must set, thus incurring more overhead.

Another factor is data contention. Intuitively, the finer the granularity, the more potential concurrency, so the better the performance. Actually, this intuition is not entirely correct. Finer granularity does reduce the probability of conflict *per request*. However, more locks are needed too, so the number of conflicts a transaction encounters may increase. One can see this from the DC-workload k^2N/D. If an increase in D causes a proportionate increase in k, then the DC-workload increases, so there is more data contention.

The third factor is resource contention. Recall that data contention alleviates resource contention by blocking some transactions. Refining the granularity may therefore release so many transactions from lock queues that they end up spending even more time in resource queues.

These three factors combine to shape the *granularity curve* in Fig. 3–10. The initial drop in throughput as granularity is refined is caused by an increase in k when D is increased, leading to increased locking overhead and data contention. As granules shrink, the number of locks a transaction requires approaches the maximum of one new lock per data access. Now k becomes insensitive to D, the DC-workload decreases, and throughput picks up if granularity is further refined. The final drop in the granularity curve is caused by resource contention. Suppose there are enough transactions in the system to cause RC-thrashing if some transactions are not blocked in lock queues. Then

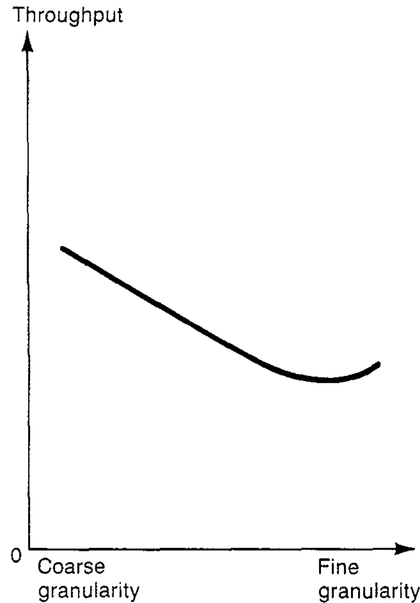

FIGURE 3–11
Possible Granularity Curve for Long Transactions

refining granularity reduces the data contention, unblocks the transactions, and causes a drop in throughput through RC-thrashing.

A given system may not see the entire granularity curve. For instance, for *long* transactions, which access a significant portion of the database, even the finest granularity may not bring the throughput above the initial drop, as in Fig. 3–11. Each transaction should then lock the entire database, thus using the coarsest granularity. For short transactions, k may quickly become insensitive to D, so the initial drop in the granularity curve is minimal. If, in addition, N is not excessive, then the final drop in the granularity curve will not occur, so the curve may look like Fig. 3–12. In this case, the curve suggests that the granularity should be as fine as possible.

Read Locks and Nonuniform Access

We have so far assumed that all locks are write locks. Suppose now that a fraction s of the lock requests are for read locks, and the rest are for write locks. Then the DC-workload drops from k^2N/D to $(1-s^2)\ k^2N/D$. Equivalently, it is as if the granules have been refined, with D increased to $D/(1-s^2)$.

Contrary to our assumption, transactions do not really access all data with equal probability. In particular, a portion p of the database may contain high-

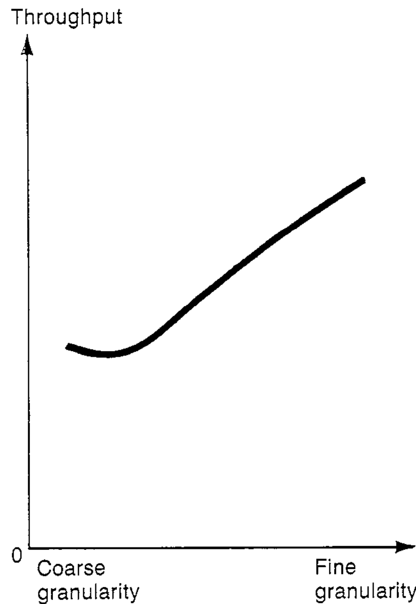

FIGURE 3–12
Possible Granularity Curve for Short Transactions

traffic data that transactions access with a higher probability q than the rest of the database. For example, if $p = 0.2$ and $q = 0.7$, then 70% of a transaction's requests fall within 20% of the database. If we assume uniform access among high-traffic data, and uniform access among the rest of the data as well, then this skewed access pattern increases the DC-workload from k^2N/D to $(1 + (q-p)^2/p(1-p))(k^2N/D)$. Equivalently, it is as if the number of granules has been reduced from D to $D/(1 + (q-p)^2/p(1-p)$.

3.13 TREE LOCKING

Suppose data items are known to be structured as nodes of a tree, and transactions always access data items by following paths in the tree. The scheduler can exploit the transactions' predictable access behavior by using locking protocols that are not two phase. That is, in certain cases, a transaction can release a lock and subsequently obtain another lock. This can lead to better performance.

To simplify the discussion we shall not distinguish between Reads and Writes. Instead we have just one type of operation, "transaction T_i accesses data item x," denoted $a_i[x]$. The Access operation $a_i[x]$ can read and/or write into x. Hence, two Access operations on the same data item conflict.

We associate a lock type a with the operation type a. We use $al_i[x]$ to denote an *access lock* on x by transaction T_i. Since two Access operations on x conflict, two access locks on x also conflict.

In *tree locking* (*TL*), we assume that a hierarchical structure has been imposed on the set of data items.[13] That is, there is a tree *DT*, called the *data tree*, whose nodes are labelled by the data items. The TL scheduler enforces the following rules:

1. Before submitting $a_i[x]$ to the DM, the scheduler must set $al_i[x]$.
2. The scheduler can set $al_i[x]$ only if no $al_j[x]$ is set, for all $j \neq i$.
3. If x is not the root of *DT*, then the scheduler can set $al_i[x]$ only if $al_i[y]$ is already set, where y is x's parent in *DT*.
4. The scheduler must not release $al_i[x]$ until *at least* after the DM has acknowledged that $a_i[x]$ has been processed.
5. Once the scheduler releases a lock on behalf of T_i, it may not subsequently obtain that same lock again for T_i.

Rules (3) and (5) imply that the scheduler can release $al_i[x]$ only after it has obtained the locks T_i needs on x's children. This handshake between locking children and unlocking their parent is called *lock coupling*. Notice that lock coupling implies that locks are obtained in root-to-leaf order.

This leads to the key fact about TL: If T_i locks x before T_j, then for every descendant v of x in *DT*, if T_i and T_j both lock v, then T_i locks v before T_j. To see this, let $(x, z_1, \ldots, z_n, v)$ be the path of nodes connecting x to v ($n \geq 0$). By rules (3) and (5), T_i must lock z_1 before releasing its lock on x. Since T_i locks x before T_j, that means T_i must lock z_1 before T_j. A simple induction argument shows that the same must be true for every node on the path. This gives us the following proposition.

Proposition 3.8: If T_i locks x before T_j, then for every descendant v of x in *DT*, if T_i and T_j both lock v, then T_i locks v before T_j. □

Consider any edge $T_i \rightarrow T_j$ in the SG of some history produced by TL. By the definition of SG, there is a pair of conflicting operations $a_i[x] < a_j[x]$. By rules (1), (2), and (4), T_i unlocked x before T_j locked x. By Proposition 3.8, it immediately follows that T_i locked the root before T_j, since x is a descendant of the root and, by rule (3), all transactions must lock the root. By rule (2), this implies that T_i unlocked the root before T_j locked the root. A simple induction argument shows that this property also holds for paths in the SG. That is, if there is a path from T_i to T_j in SG, then T_i unlocked the root before T_j locked the root.

[13]This should not be confused with the lock instance graph, used in connection with multigranularity locking.

Suppose the SG has a cycle $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. By the previous paragraph, it follows that T_1 unlocked the root before T_1 locked the root. This violates rule (5), so the cycle cannot exist. Thus, we have proved the following theorem.

Theorem 3.9: The tree locking scheduler produces serializable executions. □

TL scheduling is reminiscent of a scheduling policy used to avoid deadlocks in operating systems where processes must obtain resources in a predefined linear order. TL schedulers share the property of deadlock freedom with this policy. To see this, first note that if T_i is waiting to lock the root, it can't be involved in a deadlock (since it has no locks and therefore no transaction could be waiting for it). Now, suppose T_i is waiting for a lock currently held by T_j on a node other than the root. By the argument just given, T_j unlocks the root before T_i locks it. Thus, by induction, if the WFG has a cycle containing T_i, T_i unlocks the root before it locks the root, a contradiction. So, TL schedulers are not prone to deadlocks.

In addition to deadlock avoidance, another benefit of TL is that locks can be released earlier than in 2PL. For any data item x, once a transaction has locked all of x's children that it will ever lock, it no longer needs $al_i[x]$ to satisfy rule (3), and can therefore release it. The problem is, how does the scheduler know that T_i has locked all of x's children that it needs? Clearly, if T_i has locked *all* children of x, then it has locked all those that it needs. However, other than this special case, the scheduler cannot determine that T_i no longer needs $al_i[x]$ unless it receives some advice from T_i's TM. Without this help, it can only safely release a transaction T_i's locks when the transaction terminates. In this case transactions are (strictly) two phase locked and there is little point in enforcing the additional restriction of tree locking — except that we also get deadlock freedom. Therefore, TL only makes sense in those cases where the TM knows transactions' access patterns well enough to tell the scheduler when to release locks.

Releasing locks earlier than the end of the transaction is valuable for performance reasons. By holding locks for shorter periods, transactions block each other less frequently. Thus transactions are delayed less often due to locking conflicts, and thereby have better response time.

However, this benefit is only realized if transactions normally access nodes in DT in root-to-leaf order. If they don't, then TL is imposing an unnatural ordering on their accesses, thereby forcing them to lock nodes before they're ready to use them or to lock nodes that they don't use at all. In this sense, TL could be reducing the concurrency among transactions.

In addition, we may need to strengthen TL to ensure recoverability, strictness, or avoidance of cascading aborts. For example, to avoid cascading aborts, a transaction should hold its lock on each data item it writes until it

commits, which is more than what TL requires. For internal nodes, holding locks for longer periods can have a serious performance impact, since transactions must lock an internal node x to access any of x's descendants. Fortunately, in many practical applications, most updates are to leaves of DT, which transactions can lock until commitment with little performance impact. We'll look at one such application, B-trees, later in the section.

Variations of Tree Locking

TL can be generalized in several ways. First, we need not restrict a transaction to set its first lock on the root of DT. It is safe for it to begin by locking *any* data item in DT. However, once it sets its first lock on some data item x, rule (3) implies that it can subsequently only lock data items in the subtree rooted at x (see Exercise 3.39).

TL can also be generalized to distinguish between read and write locks. If each transaction sets either only read locks or only write locks, then the ordinary conflict rules between these locks are satisfactory for producing SR executions. However, if a transaction can set both read and write locks, then problems can arise, because read locks can allow transactions to "pass" each other while moving down the tree. For example, suppose x is the root of the tree, y is a child of x, and z is a child of y. Consider the following sequence of events:[14]

$$wl_1[x]\ rl_1[y]\ wu_1[x]\ wl_2[x]\ rl_2[y]\ wu_2[x]\ wl_2[z]\ ru_2[y]\ wu_2[z]\ wl_1[z]\ ru_1[y]\ wu_1[z].$$

In this execution, T_1 write locked x before T_2, but T_2 write locked z before T_1, producing a non-SR execution. This was possible because T_2 "passed" T_1 when they both held read locks on y. A solution to this problem is to require that for every path of data items $x_1, \ldots, x_n$, if T_i sets write locks on x_1 and x_n, and sets read locks on the other data items on the path, then it obtains locks on all data items on the path before it releases locks on any of them. By holding locks this long, a transaction ensures that other transactions cannot pass it along this path. Another solution is to require that locks set by a transaction along a path are of nondecreasing strength. (See Exercise 3.40.)

A third generalization of tree locking is *dag locking* (*DL*), in which data items are organized into a partial order rather than a hierarchy. That is, there is a rooted dag whose nodes are labelled by the data items. The DL scheduler must enforce the same rules (1) – (5), the only difference being

3. Unless x is the root, to obtain a lock on x, T_i must be holding (at that time) a lock on *some* parent of x, and there must have been a time at which T_i held locks on *all* parents of x.

---

[14]Recall that $rl_i[x]$, $wl_i[x]$, $ru_i[x]$ and $wu_i[x]$ mean that T_i has set a read lock on x, set a write lock on x, released its read lock on x, and released its write lock on x, respectively.

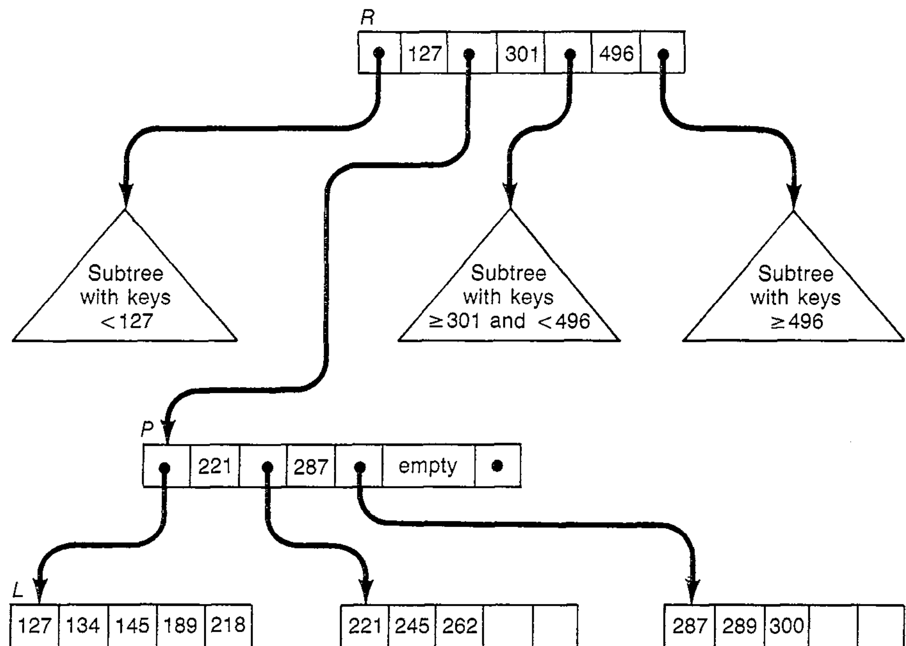

FIGURE 3–13
A B-tree
In this example, there is room for up to three keys in each internal node and up to five keys in each leaf.

As with TL, the DL scheduler produces SR executions and is not prone to deadlocks (see Exercise 3.41). The two previous generalizations apply to this case too.

B-Tree Locking[15]

An important application of tree locking is to concurrency control in tree-structured indices. The most popular type of search tree in database systems is B-trees. There are several specialized tree locking protocols specifically designed for B-trees, some of which we will describe. These protocols can also be applied to other types of search structures, such as binary trees and dynamic hash tables.

[15]This subsection assumes a basic knowledge of B-trees; see [Bayer, McCreight 72], and [Comer 79]. We use the B^+-tree variation in this section, which is the variation of choice for commercial products, such as IBM's VSAM.

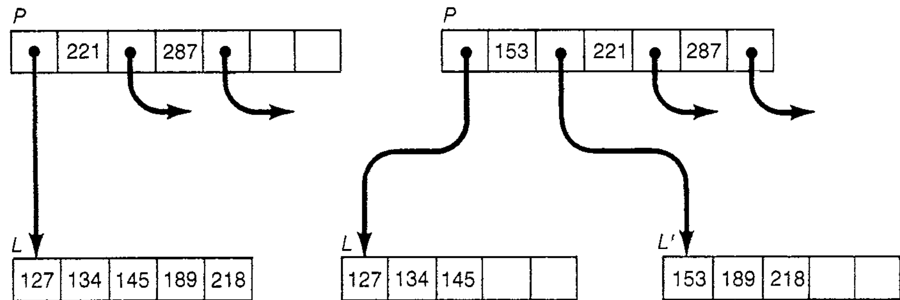

FIGURE 3–14
A B-tree Insertion Causing a Split
(a) Before inserting key 153.
(b) After splitting L to make room for key 153.

A B-tree consists of a set of nodes structured as a tree. Its purpose is to index a set of records of the form [key, data], where the key values are totally ordered and may consist, for example, of numbers or alphanumeric strings.

Each node of a B-tree contains a sorted list of key values. For internal nodes, each pair of consecutive field values defines a range of key values between two keys, k_i and k_{i+1} (see Fig. 3–13). For each such pair of consecutive values, there is a pointer to a subtree that contains all of the records whose key values are in the range defined by that pair. For leaf nodes, each key has the data part of the record associated with that key value (but no pointer). Since the data portion of records is uninterpreted by the B-tree algorithms of interest to us, we will ignore them in the following discussion and examples.

The two B-tree algorithms that are important for this discussion are Search and Insert. (Delete leads to problems similar to those of Insert, so we will not treat it here.) The *search* of a B-tree for a key value begins at the root. Each node has information that directs the search to the appropriate child of that node. The search proceeds down one path until it reaches a leaf, which contains the desired key value. For example, a search for key 134 in Fig. 3–13 (1) finds the key range [127, 301) in the root *R*, (2) follows the pointer to *P*, (3) finds the key range [127, 221) in node *P*, (4) follows the pointer to *L*, and (5) finds the key 134 in *L*.[16]

To *insert* a record *R* in a B-tree, a program first searches for the leaf *L* that should contain the record. If there is room for *R* in *L*, then it inserts *R* there (properly sequenced). Otherwise, it allocates another node *L'*, moves half of *L*'s records from *L* to *L'*, and adds the minumum key in *L'* and a pointer to *L'*

[16]The notation [*a*, *b*) means the range of values from *a* to *b* that includes the value *a* but not the value *b*.

in L's parent (see Fig. 3–14). If L's parent has no room for the extra key and pointer, then the program splits L's parent in the same way that it split L. This splitting activity continues recursively up the tree until it reaches a node that has room for the pointer being added, or it splits the root, in which case it adds a new root.

We want to implement Search and Insert as transactions.[17] We could use 2PL for this purpose. But since Search and Insert begin by accessing the root of the tree, each such operation would effectively lock the entire tree.

We can do somewhat better by using tree locking. Since Search only *reads* nodes, it sets read locks. Since Insert writes into a leaf L, it should certainly set a write lock on L. If L was full before the Insert began, then Insert will also write into some of L's ancestors, and must set write locks on them as well. Unfortunately, Insert cannot determine whether it will need to write into nonleaf nodes until it actually reads L to determine if L is full. Since this happens at the end of its search down the tree, it doesn't know which nodes to write lock until it has read all the nodes it will read. Herein lies the critical problem of B-tree locking.

Exactly which nodes does Insert have to write lock? If L isn't full, then it only write locks L. If L *is* full, then it will write into L's parent, P. If P is full, then it will write into P's parent, and so on. In general, if L is full, then Insert should set write locks on the path $P_1, \ldots, P_n, L$ of ancestors of L ($n \geq 1$) such that P_1 is not full and P_2 through P_n are full.

One way for Insert to do this is to set write locks during its search down the tree. It releases each write lock when it realizes that the lock is not needed. Insert does this as follows. Before reading a node N, it sets a write lock on that node. If N is not full, then N won't be split. It therefore releases all locks it owns on N's ancestors, since the insertion will not cause any of them to be written. After it has read L, it has write locked the appropriate path, and can proceed with its updating activity.

This approach requires setting write locks before they are actually known to be needed. If internal nodes are not full, as is often the case in B-trees, then all of the write locks on internal nodes will be released. These locks needlessly generate conflicts during the search, thereby delaying the transaction. For this reason, it is probably better to delay the acquisition of write locks until it is known that they are needed. This more aggressive approach can be accomplished by doing lock conversions.

---

[17]A user may want to regard Search and Insert as atomic operations nested within a larger transaction. This means that a Search or Insert must be atomic with respect to other Searches or Inserts issued by the same or different transactions. In this sense, Search and Insert are independent transactions. However, larger transactions that invoke multiple Searches and Inserts have other synchronization requirements. This opens a broader collection of issues, that of *nested transactions*, which is not treated in this book. See the Bibliographic Notes in Chapter 1 for references.

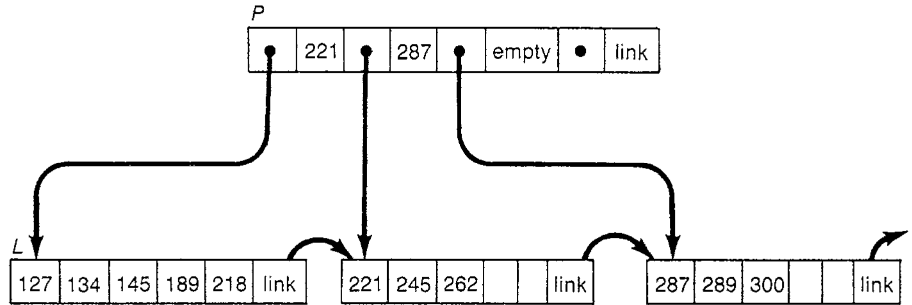

FIGURE 3–15
Links in a B-tree

During its initial search procedure, Insert only sets read locks on internal nodes. It concludes the search by setting a write lock on L. If it discovers that L is full, then it converts the necessary read locks into write locks. So, starting at the node closest to the root that it must write lock, it proceeds down the tree converting its read locks to write locks. In this way, it only sets write locks on nodes that actually have to be written.

Unfortunately, this modified protocol can lead to deadlock. For instance, two transactions may both be holding a read lock on a node and wanting to convert it to a write lock — a deadlock situation. One can avoid such a deadlock by introducing a new lock type, called might-write. A might-write lock conflicts with a write or might-write lock, but not with a read lock. Instead of obtaining read locks on its first pass down the tree, Insert obtains might-write locks. When it reaches a non-full node, it releases its ancestors' locks as before. After it reaches the desired leaf, it converts the might-write locks that it still owns to Write locks. This prevents two Inserts from locking the same node, and therefore prevents deadlocks on lock conversion.

B-Tree Locking Using Links

Lock contention can be reduced even further by departing from the lock coupling requirement of TL. Insert can be designed to write into each B-tree node independently, without owning a lock on the node's parent.

This algorithm requires that each node N have a link to its right sibling, denoted link(N). That is, link(N) points to the child of N's parent, P, that follows N, as in Fig. 3–15. If N is P's rightmost child, then link(N) points to the first child of P's right sibling (or, if P has no right sibling, then link(N) points to the first grandchild of the right sibling of P's parent, etc.). Thus, all of the nodes in each level are linked in key order through link(N). Only the rightmost node on each level has a null link.

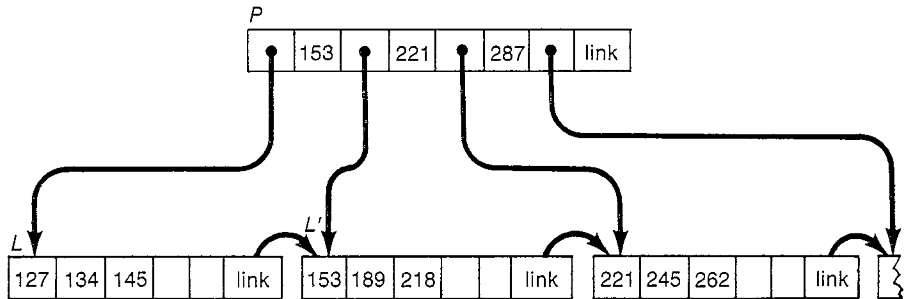

FIGURE 3–16
The B-tree of Fig. 3–15 after Inserting Key 153

Insert keeps these links up-to-date by adjusting them when it splits a node. Suppose Insert obtains a lock on node N, is ready to insert a value into N, but discovers that N is full. Then it splits N by moving the rightmost half of its contents to a newly allocated node N'. It sets link(N') := link(N) and link(N) := N', and then releases its lock on N (see Fig. 3–16, where $N = L$ and $N' = L'$). Notice that at this point, Insert owns no locks at all. Yet it now can obtain a lock on N's parent, P, and add a pointer to N' in P. If it can't add this pointer because P is full, it repeats the splitting process just described. Otherwise, it can simply insert the pointer and release the lock on P.

A Search proceeds down the tree as before, but without lock coupling. That is, after it reads an internal node N to obtain the pointer that directs it to the appropriate child C of N, it can release its read lock on N *before* obtaining its read lock on C. This obviously creates a window during which an Insert can come along and update both N and C, thereby appearing to both precede the Search (with respect to C) and follow it (with respect to N). Normally, this would be considered non-SR. However, by exploiting the semantics of B-trees and the link fields, we can modify the Search procedure slightly to avoid this apparent nonserializability.

The only way that Insert can upset Search's activity is to modify N in a way that would have caused Search to follow a different path than the one it is about to take to C. Any other update to N is irrelevant to Search's behavior. This can only happen if Insert splits C, thereby moving some of C's contents to a new right neighbor C' of C, and updating N to include a pointer to C'. If this occurred, then when Search looks in C, it may not find what it was looking for. However, in that case Search can look at C' by following link(C). For example, suppose a Search is searching for key 189 in Fig. 3–15. It reads P, and releases its locks. Now an Insert inserts 153, producing the B-tree in Fig. 3–16. Key 189 is now no longer in the node L where Search will look for it, based on the state of P that it read.

We therefore modify Search as follows. If Search is looking for value v in a node N and discovers that v is larger than any value in N, then it reads link(N), releases its lock on N, locks the node N' pointed to by link(N), and reads N'. It continues to follow links in this way until it reaches a node N such that it either finds in N the value v it is looking for or determines that v is smaller than the largest value in N and therefore is not present in the tree. In the example of the preceding paragraph, Search will discover that the largest key in L is 145, so it will follow link(L), just in case L split after Search read P. It will thereby find 189 in L' as desired.

Notice that since Search and Insert each requests a lock only when it owns no locks, it cannot be a party to a deadlock.

BIBLIOGRAPHIC NOTES

Two phase locking was introduced in [Eswaran et al.76]. Proofs of correctness have appeared in [Bernstein, Shipman, Wong 79], [Eswaran et al. 76], and [Papadimitriou 79]. Our proof in Section 3.3 is based on one in [Ullman 82]. Treating deadlock detection as cycle detection in a digraph is from [Holt 72]. Other work on centralized deadlock detection includes [Coffman, Elphick, Shoshani 71], [King, Collmeyer 74], and [Macri 76]. The implementation issues presented in Section 3.6 are largely from Gray's classic work on transaction management implementation [Gray 78], and from many discussions with implementors; see also [Weinberger 82]. The phantom problem was introduced in [Eswaran et al, 76], which suggested predicate locks to solve it. Predicate locks were later developed in [Wong, Edelberg 77] and [Hunt, Rosenkrantz 79]. Versions of the hot spot technique of Section 3.8 appeared in [Reuter 82] and [Gawlick, Kinkade 85]. Multigranularity locking was introduced in [Gray et al. 75] and [Gray, Lori, Putzolu 75]; further work appeared in [Korth 82]. Path pushing deadlock detection is described in [Gligor, Shattuck 80] and [Obermarck 82]. Timestamp-based prevention appears in [Rosenkrantz, Stearns, Lewis 78] and [Stearns, Lewis, Rosenkrantz 76]. Other distributed deadlock techniques are described in [Beeri, Obermarck 81], [Chandy, Misra 82], [Chandy, Misra, Haas 83], [Isloor, Marsland 80], [Kawazu et al. 79], [Korth et al. 83], [Lomet 79], [Lomet 80a], [Lomet 80b], [Marsland, Isloor 80], [Menasce, Muntz 79], [Obermarck 82], [Stonebraker 79b], and [Tirri 83].

The performance effect of transaction characteristics and locking protocols other than those mentioned in Section 3.12 has mostly been studied in the literature through simulations. Locking performance in a distributed system was examined in [Ries, Stonebraker 77] [Ries, Stonebraker 79] [Thanos, Carlesi, Bertino 81], and [Garcia-Molina 79] for replicated data. Ries and Stonebraker also considered hierarchical locking, as did [Carey, Stonebraker 84]. Multiversion systems were studied in [Carey, Stonebraker 84], [Kiessling, Landherr 83], and [Peinl, Reuter 83] (see Chapter 5). Mixtures of queries and updaters were studied in [Lin, Nolte 82a].

The results we have presented about locking performance can be found in [Tay, Goodman, Suri 85], but most of them can be found elsewhere too. Some were previously known to other researchers, and some were corroborated by later work. (See [Tay,

Goodman, Suri 84] for an account of the agreements and contradictions among the papers.) For instance, thrashing caused by locking was observed in [Balter, Berard, Decitre 82], [Franaszek, Robinson 85], [Lin, Nolte 82b], and [Ryu, Thomasian 86]; blocking was identified as the main cause of this thrashing in the first and fourth papers. That most deadlock cycles have only two transactions was pointed out in [Devor, Carlson 82] and [Gray et al. 81b]. The former, as well as [Beeri, Obermarck 81], also observed that deadlocks are rare in System R and IMS/VS. A pure restart policy was first studied in [Shum, Spirakis 81]; it is also a special case of the protocol in [Chesnais, Gelenbe, Mitrani 83], and was compared to a blocking policy under various levels of resource contention in [Agrawal, Carey, Livny 85]. But the locking protocol that has received the most attention is conservative 2PL [Galler, Bos 83], [Mitra, Weinberger 84], [Morris, Wong 85], [Potier, Leblanc 80], and [Thomasian, Ryu 83]. The problem of choosing the appropriate granularity was addressed in [Carey, Stonebraker 84], [Ries, Stonebraker 77], [Ries, Stonebraker 79], and [Ryu, Thomasian 86]. The effect of shared locks was evaluated in [Lavenberg 84] and [Mitra 85]. The model of nonuniform access we have described was introduced in [Munz, Krenz 77], and used in [Lin, Nolte 82].

Lock coupling protocols for tree locking appeared in [Bayer, Schkolnick 77], [Kedem, Silberschatz 81], [Samadi 76], and [Silberschatz, Kedem 76]. The linking method appeared in [Kung, Lehman 80] for binary trees, and was extended for B-trees in [Lehman, Yao 81]. Other work on locking protocols for dynamic search structures includes [Buckley, Silberschatz 84], [Ellis 82], [Ford, Schultz, Jipping 84], [Goodman, Shasha 85], [Kedem 83], [Kwong, Wood 82], and [Manber, Ladner 84].

EXERCISES

3.1 Give an example of a serializable history that could not have been produced by a 2PL scheduler.

3.2 Give an example of a non-SR execution that is two phased locked, except for one transaction that converts a write lock to a read lock.

3.3 Suppose all transactions that write into the database are two phase locked, but read-only transactions may violate the two phase rule. In what sense will the database be kept in a consistent state? Is there any sense in which it will be inconsistent? If we dropped all read-only transactions from a history, would the resulting history be SR? Do queries read consistent data?

3.4* Prove that every 2PL history H has the following property: There exists a serial history H_s such that for every two transactions T_i and T_j in H, if T_i and T_j are not interleaved (see Exercise 2.12) in H and T_i precedes T_j in H, then T_i precedes T_j in H_s.

3.5* Give a serializability theoretic proof that if each transaction is two phase locked, releases its read locks before it terminates, and releases its write locks after it commits, then the resulting execution is strict.

3.6* Define a *locked point* of a transaction to be any moment at which it owns all of its locks; that is, it is a moment after it has performed its last lock operation and before it has released any lock. Using serializability theory, prove that for every history H produced by a 2PL scheduler there is an equivalent serial history in which transactions are in the order of their locked points in H.

3.7 Design an efficient algorithm that finds and lists all cycles in a WFG. (The algorithm should be efficient in the sense that its running time is polynomial in the size of the graph and in the number of cycles in the graph.)

3.8 Suppose T_j is waiting for a lock on x held by T_i. Now suppose T_k requests that lock on x and it too must wait. In general, is it better to add $T_k \rightarrow T_j$, $T_k \rightarrow T_i$, or both to the WFG? Discuss the effect of this decision on the algorithm that schedules waiting requests after a lock is released.

3.9 Consider a centralized DBS that uses 2PL and in which all transactions are sequential programs. Thus, no transaction can have more than one outstanding Read or Write request that is blocked. Could a transaction be involved in more than one deadlock? Prove your answer.

3.10 Suppose that if a lock request for x cannot be granted immediately, edges are added to the WFG from the blocked transaction to every transaction that owns a conflicting lock on x. Deadlock detection is then performed. If no deadlock is detected, then the request is added to the end of x's lock queue. The queue is serviced in a first-come-first-served manner. Show that this method does not detect all deadlocks. Propose a modified method that does.

3.11 Let $T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ be a cycle in a WFG. An edge $T_i \rightarrow T_j$ is a *chord* of the cycle if T_i and T_j are nodes of the cycle and $T_i \rightarrow T_j$ is not an edge of the cycle. A cycle is *elementary* if it has no chords. Suppose we use a deadlock detector that finds all of the elementary cycles in a WFG, and breaks each such deadlock by aborting a victim in the cycle. Prove that this breaks *all* cycles in the WFG.

3.12 In our description of Conservative 2PL, we assumed that a transaction predeclares the set of data items it reads or writes. Describe a 2PL scheduler that does *not* predeclare its readset and writeset, yet is not subject to deadlocks.

3.13 Write a program that implements a Strict 2PL scheduler. Prove that your program satisfies all of the conditions in Propositions 3.1–3.3.

3.14* To prove the correctness of 2PL, in Propositions 3.1–3-3 stated conditions that every 2PL history must satisfy. State the additional conditions that must be satisfied by every 2PL history that represents an execution of a Strict 2PL scheduler and of a Conservative 2PL scheduler.

3.15 Suppose we partition the lock table, and we assign a distinct semaphore to each partition to ensure it is accessed atomically. Suppose each lock is assigned to a partition based on its data item name, so each transaction may own locks in multiple partitions. To release a transaction's locks, the LM must access more than one partition. Since the LM may acquire more than one semaphore to do this, it may deadlock if two transactions release their locks concurrently. Give an example of the problem, and propose a solution to it.

3.16 Instead of using an end-of-file marker as in Section 3.7, suppose we use fixed length records and maintain a *count* of the number of records in a file F. Give an example of a non-SR execution where transaction T_1 scans F, T_2 inserts a record into F, T_1 reads some other data item x, T_2 writes x, both transactions are two phase locked, and neither transaction locks count. Explain why this is an example of the phantom problem.

3.17 Consider a database consisting of one file, F. Each transaction begins by issuing a command "getlock where Q," where Q is a qualification (i.e., a Boolean formula) that's true for some records and false for others. The scheduler processes the Getlock command by write locking the set of all records in F that satisfy Q. The transaction can only read and modify records that were locked by its Getlock command. The transaction can insert a new record, which the scheduler write locks just before it inserts it. The scheduler holds a transaction's locks until it commits. Does this locking algorithm prevent phantoms? If so, prove it correct. If not, show a non-SR execution.

3.18 Suppose we modify the MGL protocol for dags so that T_i can set $iwl_i[x]$ as long as it owns an *iw* lock on some parent (rather than all parents) of x. Prove or disprove that the resulting protocol is correct.

3.19 Suppose we reverse the MGL protocol for dags: to set $rl_i[x]$ or $irl_i[x]$, T_i must have an *ir* or *iw* lock on *all* parents of x, and to set $wl_i[x]$ or $iwl_i[x]$, T_i must have an *iw* lock on *some* parent of x. Prove that the resulting protocol is correct. Under what conditions would you expect this protocol to outperform the MGL protocol for dags in Section 3.9?

3.20 The MGL protocol for lock instance graphs that are trees is limited to read and write locks. Generalize the protocol so that it will work for arbitrary lock types (e.g., Increment and Decrement).

3.21 Rule (2) of the MGL protocol requires that if a transaction has a w lock or *iw* lock on a data item x, then it must have an *iw* lock on x's parent. Is it correct for it to hold a w lock on x's parent instead? Is there a case in which it would be useful to set such a w lock if the lock instance graph is a tree? What about dags?

3.22 In the dag lock type graph in Fig. 3–7, a lock on an index entry locks all fields of all records that the entry points to. Suppose we distinguish

indexed fields from non-indexed fields. A lock on an index entry should only lock the indexed field of the records it points to. Other fields of these records can be concurrently locked. Design a lock instance dag that implements this approach and argue that it has the intended effect, assuming the dag MGL protocol.

3.23 Prove that the MGL protocol for dag lock instance graphs is correct in the sense of Theorem 3.7.

3.24 In the MGL protocol for lock instance graphs that are trees, suppose we allow a transaction to release a lock on a data item x before it releases its lock on some child of x. Assuming the scheduler uses Basic 2PL, give an example of an incorrect execution that can result from this.

3.25 In Section 3.11, we argued that if no transaction spontaneously aborts and every lock obtained by a transaction corresponds to a database operation (i.e., no intention locks), then there are no phantom deadlocks. Are phantom deadlocks possible if we allow intention locks? Prove your answer.

3.26 Consider a distributed DBS. Give an example execution of two transactions which is not SR and satisfies the 2PL rules locally (i.e., at each site, considered individually) though not globally (i.e., considering all sites together). In your example, be sure to give the precise sequence in which locks are set and released by the schedulers as well as the sequence in which Reads and Writes are executed.

3.27 In this problem, we will simplify the path pushing algorithm for distributed deadlock detection. Assume that each transaction is a sequential process; thus, it is only active (unblocked) at one site at a time. Suppose we augment the WFG at each site, say A, with an additional node labelled EX for "external." For each transaction T_i, if T_i was formerly executing at site A and now is stopped waiting for a response from some other site, then add an edge $T_i \rightarrow EX$. If T_i is executing at A and was formerly executing at any other site, then add an edge $EX \rightarrow T_i$.

We now modify the algorithm as follows. Site A periodically detects cycles in its WFG. If a cycle does not contain the node EX, then a transaction is aborted. For each cycle $EX \rightarrow T_i \rightarrow \cdots \rightarrow T_j \rightarrow EX$, site A sends the path $T_i \rightarrow \cdots \rightarrow T_j$ to the site from which T_j is waiting for a response. When a site receives a list of paths from another site, it adds those edges to its WFG and performs the above steps. Prove that this algorithm detects all deadlock cycles.

3.28 Design an approach to deleting edges from each site's WFG in the path pushing deadlock detection algorithm of Section 3.11.

3.29 In timestamp-based deadlock detection, we assigned each transaction a unique timestamp. Suppose we do not require transactions' timestamps to be unique. Do the Wait-Die and Wound-Wait methods still prevent deadlock? Do they prevent cyclic restart?

3.30 Suppose we alter the definition of Wait-Die as follows:

if ts(T_i) > ts(T_j) **then** T_i waits **else** abort T_i.

Does this method prevent deadlock? Does it prevent cyclic restart? Back up your claims with proofs or counterexamples. Compare the dynamic behavior of this method with standard Wait-Die and Wound-Wait.

3.31 Suppose a transaction is assigned a new timestamp every time it is aborted and restarted. Using Wound-Wait for deadlock detection, give an example of two transactions T_i and T_j that cyclically restart each other using this method. If possible, design them so they "self-synchronize" in the sense that even with small variations in transaction execution time and communications delay, they still experience cyclic restart.

3.32 Design a hybrid deadlock detection and prevention algorithm that, to the extent possible, uses WFG cycle detection locally at each site, and uses timestamp-based prevention to avoid global deadlocks.

3.33 In Wound-Wait timestamp-based deadlock prevention, suppose that when T_i wounds T_j, T_j aborts only if it is waiting, or later tries to wait, for another lock. Does this version of Wound-Wait prevent deadlock? Prove your answer.

3.34 Suppose we partition the set of sites in a distributed DBS into regions. Each site has a local deadlock detector. Each region has a global deadlock detector to which its sites send their WFGs. There is also a system-wide deadlock detector to which all the regional deadlock detectors send their WFGs. This arrangement of deadlock detectors is called *hierarchical deadlock detection*. Under what circumstances would you expect hierarchical deadlock detection to perform better or worse than a single global deadlock detector? Is hierarchical deadlock detection subject to phantom deadlocks under different conditions than a single global deadlock detector?

3.35 Suppose we modify the hierarchical deadlock detector of the previous problem as follows. Define a transaction to be *local* if it only accesses data at one site. Each site constructs a *locally compressed* WFG by taking the transitive closure of its WFG and then deleting all nodes corresponding to local transactions (along with edges that are incident with those nodes). Each site periodically sends its locally compressed WFG (not its full WFG) to its regional deadlock detector. Each regional deadlock detector also does "local" compression, where in this case "local" means local to the set of sites in the region. Each regional deadlock detector periodically sends its locally compressed WFG to the system-wide deadlock detector. Does this deadlock detection scheme detect all deadlocks? Might it detect a phantom deadlock? Prove your answers.

3.36 Let k, N, and D be as defined in Section 3.12. If the deadlock rate is low, so that most transactions terminate without restarts, then a transac-

tion has $k/2$ locks on average. Assume all locks are write locks, and lock requests are uniformly distributed over the database.

a. What is the probability of conflict per request?
b. What is the probability that a transaction will encounter a conflict?
c. Suppose $k = 1$ if $D = 2$, and $k = 2$ if $D = 5$. Let $N = 2$. Compute the probabilities in (a) and (b) for $D = 2$ and $D = 5$. Note that one probability decreases while the other increases.
d. Show that, if $kN/2D$ is small, then the probability in (b) is $k^2N/2D$ approximately. Note the relationship with the DC-workload.

3.37 Assume again the conditions in Exercise 3.37. Let R be the response time of a transaction. If we assume that when a transaction is blocked, there is no other transaction waiting for the same lock, then the waiting time for the lock is $R/2$ on average (since the deadlock rate is low). Now let T be the response time of a transaction if the concurrency control were switched off.

a. Show that (with the concurrency control switched on)

$$R = T + \frac{R}{2} \sum_{r=1}^{k} r \binom{k}{r} p^r (1-p)^{k-r}$$

where p is the probability of conflict per request, and $\binom{k}{r} = \frac{k!(k-r)!}{r!}$.

b. Using the identity $\sum_{r=1}^{k} r \binom{k}{r} p^r (1-p)^{k-r} = kp$, and the probability of conflict from Problem 3.37, deduce that $R = T/\left(1 - \frac{k^2N}{4D}\right)$.

c. Little's Law from elementary queueing theory now implies that the throughput is $\frac{N}{T}\left(1 - \frac{k^2N}{4D}\right)$. Does this formula predict DC-thrashing?

d. How does resource contention affect the formulas in (b) and (c)?

3.38 Consider a DBS that uses a Strict 2PL scheduler. In the following, throughput (i.e., user demand) is the same before and after the change.

a. The code of the transactions running on a particular system is changed, but the number of locks (all of which are write locks) required by a transaction is unaffected. The change results in an increase in response time. Give two possible reasons.
b. A system is running a mixture of queries and updates. (Queries only set read locks, whereas updates set write locks.) Whenever the proportion of queries increases, overall response time becomes worse. Give three possible reasons.

c. A certain portion of a database is identified as a high contention area, so the granularity for this portion was refined. However, response time becomes worse. Give three possible reasons.

3.39 Modify the TL protocol so that a transaction need not begin by locking the root of DT. Prove that the modified protocol is correct.

3.40 Extend the TL protocol to handle read locks and write locks. Prove the resulting protocol produces SR executions and is free of deadlocks.

3.41 Prove that the DL protocol produces SR executions and is free of deadlocks.

3.42 Do Exercise 3.41 for an arbitrary set of lock types with its associated compatibility matrix.

3.43 Extend the various versions of B-tree locking in Secion 3.13 to handle the deletion of nodes.