

## 9.5.2 Simple Locking

The second locking discipline, known as *simple locking*, is similar in spirit to, though not quite identical with, the mark-point discipline. The simple locking discipline has two rules. First, each transaction must acquire a lock for every shared data object it intends to read or write before doing any actual reading and writing. Second, it may release its locks only after the transaction installs its last update and commits or completely restores the data and aborts. Analogous to the mark point, the transaction has what is called a *lock point*: the first instant at which it has acquired all of its locks. The collection of locks it has acquired when it reaches its lock point is called its *lock set*. A lock manager can enforce simple locking by requiring that each transaction supply its intended lock set as an argument to the **begin\_transaction** operation, which acquires all of the locks of the lock set, if necessary waiting for them to become available. The lock manager can also interpose itself on all calls to read data and to log changes, to verify that they refer to variables that are in the lock set. The lock manager also intercepts the call to commit or abort (or, if the application uses roll-forward recovery, to log an END record) at which time it automatically releases all of the locks of the lock set.

The simple locking discipline correctly coordinates concurrent transactions. We can make that claim using a line of argument analogous to the one used for correctness of the mark-point discipline. Imagine that an all-seeing outside observer maintains an ordered list to which it adds each transaction identifier as soon as the transaction reaches its lock point and removes it from the list when it begins to release its locks. Under the simple locking discipline each transaction has agreed not to read or write anything until that transaction has been added to the observer's list. We also know that all transactions that precede this one in the list must have already passed their lock point. Since no data object can appear in the lock sets of two transactions, no data object in any transaction's lock set appears in the lock set of the transaction preceding it in the list, and by induction to any transaction earlier in the list. Thus all of this transaction's input values are the same as they will be when the preceding transaction in the list commits or aborts. The same argument applies to the transaction before the preceding one, so all inputs to any transaction are identical to the inputs that would be available if all the transactions ahead of it in the list ran serially, in the order of the list. Thus the simple locking discipline ensures that this transaction runs completely after the preceding one and completely before the next one. Concurrent transactions will produce results as if they had been serialized in the order that they reached their lock points.

As with the mark-point discipline, simple locking can miss some opportunities for concurrency. In addition, the simple locking discipline creates a problem that can be significant in some applications. Because it requires the transaction to acquire a lock on every shared object that it will either read *or* write (recall that the mark-point discipline requires marking only of shared objects that the transaction will write), applications that discover which objects need to be read by reading other shared data objects have no alternative but to lock every object that they *might* need to read. To the extent that the set of objects that an application *might* need to read is larger than the set for which it eventually

*does* read, the simple locking discipline can interfere with opportunities for concurrency. On the other hand, when the transaction is straightforward (such as the `TRANSFER` transaction of Figure 9.16, which needs to lock only two records, both of which are known at the outset) simple locking can be effective.

### 9.5.3 Two-Phase Locking

The third locking discipline, called *two-phase locking*, like the read-capture discipline, avoids the requirement that a transaction must know in advance which locks to acquire. Two-phase locking is widely used, but it is harder to argue that it is correct. The two-phase locking discipline allows a transaction to acquire locks as it proceeds, and the transaction may read or write a data object as soon as it acquires a lock on that object. The primary constraint is that the transaction may not release any locks until it passes its lock point. Further, the transaction can release a lock on an object that it only reads any time after it reaches its lock point *if* it will never need to read that object again, even to abort. The name of the discipline comes about because the number of locks acquired by a transaction monotonically increases up to the lock point (the first phase), after which it monotonically decreases (the second phase). Just as with simple locking, two-phase locking orders concurrent transactions so that they produce results as if they had been serialized in the order they reach their lock points. A lock manager can implement two-phase locking by intercepting all calls to read and write data; it acquires a lock (perhaps having to wait) on the first use of each shared variable. As with simple locking, it then holds the locks until it intercepts the call to commit, abort, or log the `END` record of the transaction, at which time it releases them all at once.

The extra flexibility of two-phase locking makes it harder to argue that it guarantees before-or-after atomicity. Informally, once a transaction has acquired a lock on a data object, the value of that object is the same as it will be when the transaction reaches its lock point, so reading that value now must yield the same result as waiting till then to read it. Furthermore, releasing a lock on an object that it hasn't modified must be harmless if this transaction will never look at the object again, even to abort. A formal argument that two-phase locking leads to correct before-or-after atomicity can be found in most advanced texts on concurrency control and transactions. See, for example, *Transaction Processing*, by Gray and Reuter [Suggestions for Further Reading 1.1.5].

The two-phase locking discipline can potentially allow more concurrency than the simple locking discipline, but it still unnecessarily blocks certain serializable, and therefore correct, action orderings. For example, suppose transaction  $T_1$  reads  $X$  and writes  $Y$ , while transaction  $T_2$  just does a (blind) write to  $Y$ . Because the lock sets of  $T_1$  and  $T_2$  intersect at variable  $Y$ , the two-phase locking discipline will force transaction  $T_2$  to run either completely before or completely after  $T_1$ . But the sequence

```
T1: READ X
T2: WRITE Y
T1: WRITE Y
```

in which the write of  $T_2$  occurs between the two steps of  $T_1$ , yields the same result as running  $T_2$  completely before  $T_1$ , so the result is always correct, even though this sequence would be prevented by two-phase locking. Disciplines that allow all possible concurrency while at the same time ensuring before-or-after atomicity are quite difficult to devise. (Theorists identify the problem as NP-complete.)

There are two interactions between locks and logs that require some thought: (1) individual transactions that abort, and (2) system recovery. Aborts are the easiest to deal with. Since we require that an aborting transaction restore its changed data objects to their original values before releasing any locks, no special account need be taken of aborted transactions. For purposes of before-or-after atomicity they look just like committed transactions that didn't change anything. The rule about not releasing any locks on modified data before the end of the transaction is essential to accomplishing an abort. If a lock on some modified object were released, and then the transaction decided to abort, it might find that some other transaction has now acquired that lock and changed the object again. Backing out an aborted change is likely to be impossible unless the locks on modified objects have been held.

The interaction between log-based recovery and locks is less obvious. The question is whether locks themselves are data objects for which changes should be logged. To analyze this question, suppose there is a system crash. At the completion of crash recovery there should be no pending transactions because any transactions that were pending at the time of the crash should have been rolled back by the recovery procedure, and recovery does not allow any new transactions to begin until it completes. Since locks exist only to coordinate pending transactions, it would clearly be an error if there were locks still set when crash recovery is complete. That observation suggests that locks belong in volatile storage, where they will automatically disappear on a crash, rather than in non-volatile storage, where the recovery procedure would have to hunt them down to release them. The bigger question, however, is whether or not the log-based recovery algorithm will construct a correct system state—correct in the sense that it could have arisen from some serial ordering of those transactions that committed before the crash.

Continue to assume that the locks are in volatile memory, and at the instant of a crash all record of the locks is lost. Some set of transactions—the ones that logged a `BEGIN` record but have not yet logged an `END` record—may not have been completed. But we know that the transactions that were not complete at the instant of the crash had non-overlapping lock sets at the moment that the lock values vanished. The recovery algorithm of Figure 9.23 will systematically `UNDO` or `REDO` installs for the incomplete transactions, but every such `UNDO` or `REDO` must modify a variable whose lock was in some transaction's lock set at the time of the crash. Because those lock sets must have been non-overlapping, those particular actions can safely be redone or undone without concern for before-or-after atomicity during recovery. Put another way, the locks created a particular serialization of the transactions and the log has captured that serialization. Since `RECOVER` performs `UNDO` actions in reverse order as specified in the log, and it performs `REDO` actions in forward order, again as specified in the log, `RECOVER` reconstructs exactly that same serialization. Thus even a recovery algorithm that reconstructs the

entire database from the log is guaranteed to produce the same serialization as when the transactions were originally performed. So long as no new transactions begin until recovery is complete, there is no danger of miscoordination, despite the absence of locks during recovery.

### 9.5.4 Performance Optimizations

Most logging-locking systems are substantially more complex than the description so far might lead one to expect. The complications primarily arise from attempts to gain performance. In Section 9.3.6 we saw how buffering of disk I/O in a volatile memory cache, to allow reading, writing, and computation to go on concurrently, can complicate a logging system. Designers sometimes apply two performance-enhancing complexities to locking systems: physical locking and adding lock compatibility modes.

A performance-enhancing technique driven by buffering of disk I/O and physical media considerations is to choose a particular lock granularity known as *physical locking*. If a transaction makes a change to a six-byte object in the middle of a 1000-byte disk sector, or to a 1500-byte object that occupies parts of two disk sectors, there is a question about which “variable” should be locked: the object, or the disk sector(s)? If two concurrent threads make updates to unrelated data objects that happen to be stored in the same disk sector, then the two disk writes must be coordinated. Choosing the right locking granularity can make a big performance difference.

Locking application-defined objects without consideration of their mapping to physical disk sectors is appealing because it is understandable to the application writer. For that reason, it is usually called *logical locking*. In addition, if the objects are small, it apparently allows more concurrency: if another transaction is interested in a different object that is in the same disk sector, it could proceed in parallel. However, a consequence of logical locking is that logging must also be done on the same logical objects. Different parts of the same disk sector may be modified by different transactions that are running concurrently, and if one transaction commits but the other aborts neither the old nor the new disk sector is the correct one to restore following a crash; the log entries must record the old and new values of the individual data objects that are stored in the sector. Finally, recall that a high-performance logging system with a cache must, at commit time, force the log to disk and keep track of which objects in the cache it is safe to write to disk without violating the write-ahead log protocol. So logical locking with small objects can escalate cache record-keeping.

Backing away from the details, high-performance disk management systems typically require that the argument of a `PUT` call be a block whose size is commensurate with the size of a disk sector. Thus the real impact of logical locking is to create a layer between the application and the disk management system that presents a logical, rather than a physical, interface to its transaction clients; such things as data object management and garbage collection within disk sectors would go into this layer. The alternative is to tailor the logging and locking design to match the native granularity of the disk management system. Since matching the logging and locking granularity to the disk write granularity