database or cache. So the RM must find the value in the intentions list (i.e., step (1) of RM-Read). Doing this efficiently takes some care. One way is to index the intentions list by data item name. On each RM-Read(T_i, x), the RM checks the index for an entry for x. If there is one, it returns the last intentions list value for x. Otherwise, it finds x in the database (i.e., step (2) of RM-Read).

Another way to solve the problem is by using shadowing; see Exercise 6.30.

6.7 THE NO-UNDO/NO-REDO ALGORITHM

The stable database is the portion of the database that is on disk.

To avoid redo, all of a transaction T_i's updates must be in the stable database by the time T_i is committed. To avoid undo, none of T_i's updates can be in the stable database before T_i is committed. Hence, to eliminate both undo and redo, all of T_i's updates must be recorded in the stable database in a single atomic operation, at the time T_i commits. The RM-Commit(T_i) procedure would have to be something like the following:

RM-Commit(T_i)

The after-image of x wrt Ti is the value of x written by Ti

1. In a single atomic action:
   - For each data item x updated by T_i, write the after image of x wrt T_i in the stable database.
   - Insert T_i into the commit list.
2. Acknowledge to the scheduler the processing of RM-Commit(T_i).

Incredible as it may sound, such a procedure is realizable! The difficulty, of course, is to organize the data structures so that an atomic action — a single atomic Write to stable storage — has the entire effect of step (1) in RM-Commit. That is, it must indivisibly install all of a transaction's updates in the stable database and insert T_i into the commit list. It should do this without placing an unreasonable upper bound on the number of updates each transaction may perform.

We can attain these goals by using a form of shadowing. The location of each data item's last committed value is recorded in a directory, stored in stable storage, and possibly cached for fast access. There are also working directories that point to uncommitted versions of some data items. Together, these directories point to all of the before and after images that would ordinarily be stored in a log. We therefore do not maintain a log as a separate sequential file.

The before-image of x wrt Ti is the value of x before x was overwritten by Ti.

When a transaction T_i writes a data item x, a new version of x is created in stable storage. The working directory that defines the database state used by T_i is updated to point to this version. Conceptually, this new version is part of the log until T_i commits. When T_i commits, the directory that defines the committed database state is updated to point to the versions that T_i wrote. This makes

the results of T_i's Writes become part of the committed database state, thereby committing T_i.

With this structure, an RM-Commit procedure with the desired properties requires atomically changing the directory entries for *all* data items written by the transaction that is being committed. If the directory fits in a single data item, then the problem is solved. Otherwise, it seems we have simply moved our problem to a different structure. Instead of atomically installing updates in the stable database, we now have to atomically install updates in the directory.

The critical difference is that since the directory is much smaller than the database, it is feasible to keep two copies of it in stable storage: a *current* directory, pointing to the committed database, and a *scratch* copy. To commit a transaction T_i, the RM updates the scratch directory to represent the stable database state that includes T_i's updates. That is, for each data item x that T_i updates, the RM makes the scratch directory's entry for x point to T_i's new version of x. For data items that T_i did not update, it makes the scratch directory's entries identical to the current directory's entries. Then it swaps the current and scratch directories in an atomic action. This atomic swap action is implemented through a *master record* in stable storage, which has a bit indicating which of the two directory copies is the current one. To swap the directories, the RM simply complements the bit in the master record, which is surely an atomic action! Writing that bit is the operation that commits the transaction. Notice that the RM can only process one Commit at a time. That is, the activity of updating the scratch directory followed by complementing the master record bit is a critical section.

Figure 6–4 illustrates the structures used in the algorithm to commit transaction T_i which updated data items x and y. In Fig. 6–4(a) the transaction has created two new versions, leaving the old versions intact as shadows (appropriately shaded). In Fig. 6–4(b) T_i has set up the scratch directory to reflect the stable database as it should be after its commitment. In Fig. 6–4(c) the master record's bit is flipped, thereby committing T_i and installing its updates in the stable database. Note that there are two levels of indirection to obtain the current value of a data item. First the master record indicates the appropriate directory, and then the directory gives the data item's address in the stable database.

Before describing the five RM procedures, let us define some notation for the stable storage organization used in this algorithm. We have a master record, M, that stores a single bit. We have two directories D^0 and D^1. At any time D^b is the current directory, where b is the present value of M. $D^b[x]$ denotes the entry for data item x in directory D^b. It contains x's address in the stable database. We use $-b$ to denote the complement of b, so D^{-b} is the scratch directory. There may be one or two versions of a data item at any given time: one in the stable database (pointed to by D^b and possibly a new version. All this information — the stable database, the new versions, the two directories, and the master record — must be kept in stable storage. The master record and the directories can also be cached for efficient access.

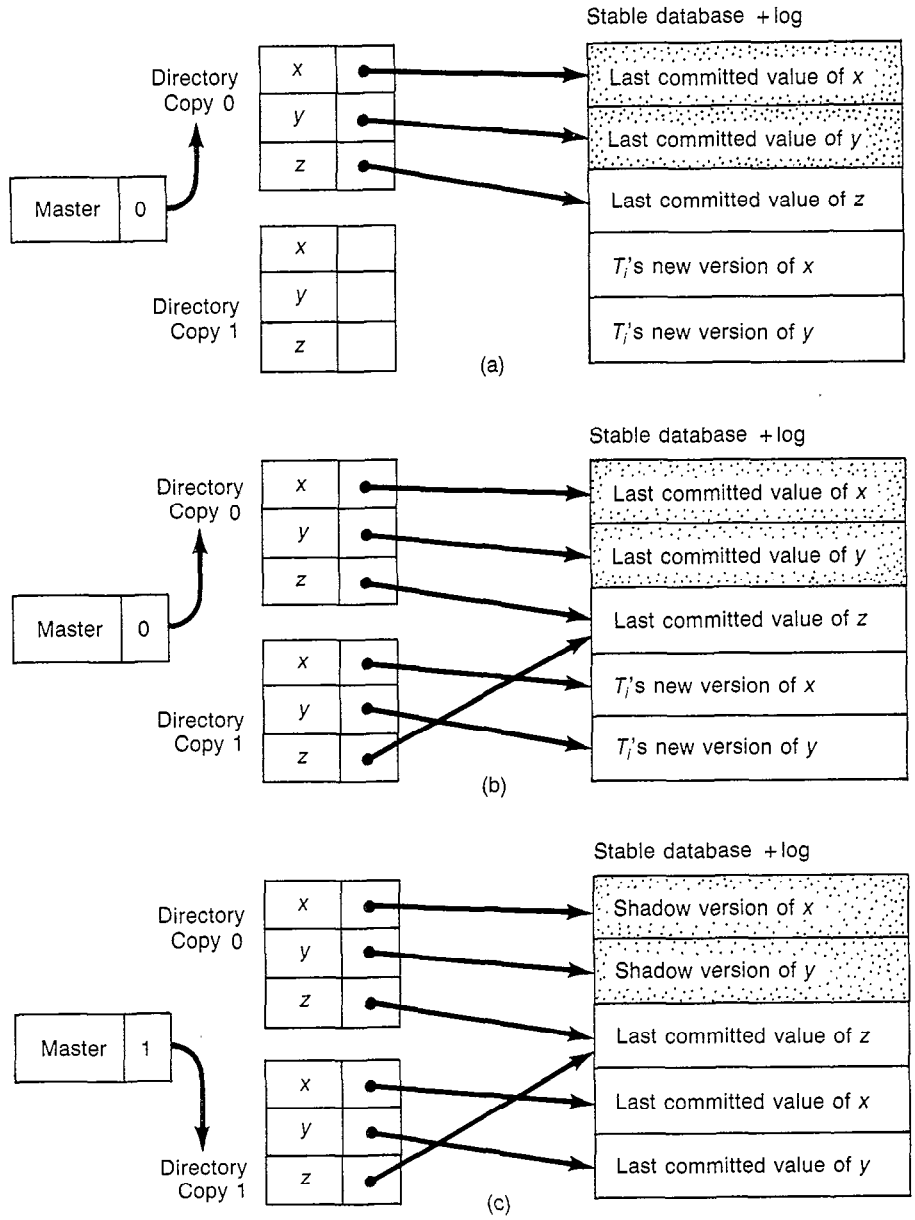

FIGURE 6–4
An Example of the No-Undo/No-Redo Algorithm
(a) Database state after creating new versions for T_i (b) Database state after preparing directory for T_i's commitment (c) Database state after committing T_i

In addition, for each active transaction T_i there is a directory D_i with the addresses of the new versions of the data items written by T_i. $D_i[x]$ denotes the entry of D_i that corresponds to data item x (presumably T_i wrote into x). These directories need not be stored in stable storage. Given this organization of data, the RM procedures are as follows.

RM-Write(T_i, x, v)

1. Write v into an unused location in stable storage and record this location's address in $D_i[x]$.(A)
2. Acknowledge to the scheduler the processing of RM-Write(T_i, x, v).

RM-Read(T_i, x)

1. If T_i has previously written into x, return to the scheduler the value stored in the location pointed to by $D_i[x]$.
2. Otherwise, return to the scheduler the value stored in the location pointed to by $D^b[x]$, where b is the present value of the bit in the master record M.(B)

RM-Commit(T_i)

1. For each x updated by T_i: $D^{-b}[x] := D_i[x]$, where b is the value of M.(C)
2. $M := -b$.(D)
3. For each x updated by T_i: $D^{-b}[x] := D_i[x]$, where b is the (new) value of M.(E)
4. Discard D_i (free any storage used by it).
5. Acknowledge to the scheduler the processing of RM-Commit(T_i).

RM-Abort(T_i)

1. Discard D_i.
2. Acknowledge to the scheduler the processing of RM-Abort(T_i).

Restart

1. Copy D^b into D^{-b}.
2. Free any storage reserved for active transactions' directories and their new versions.
3. Acknowledge to the scheduler the processing of Restart.

Comments

A. [Step (1) of RM-Write] This step creates the new version of x, leaving the shadow version in the stable database untouched.

B. [Step (2) of RM-Read] If T_i has written into x it reads the new version of x that it created when it wrote into x (see step (1) of RM-Write); otherwise it reads the version of x in the stable database.

C. [Step (1) of RM-Commit] This step sets up the scratch directory to reflect the updates of T_i.

D. [Step (2) of RM-Commit] This step complements the bit in the master record, thereby making the scratch directory into the current one (and what used to be the current into the scratch). It is the atomic action that makes T_i committed. Failure before this step will result in T_i's abortion.

E. [Step (3) of RM-Commit] This step records T_i's changes in D^{-b}, which has now become the scratch directory. This ensures that when that directory again becomes the current one, T_i's updates will be properly reflected in the stable database.

You can ignore this paragraph about the Undo Rule and Redo Rule until we cover it in class, in the section on Database Recovery.

The algorithm satisfies the Undo Rule since the stable database never has values written by uncommitted transactions. It also satisfies the Redo Rule because at the time of commitment all of a transaction's updates are in the stable database. In fact, under this algorithm the stable database *always* contains the last committed database state. As a result, virtually no work is needed to abort a transaction or restart the system following a failure.

While Restart is efficient, this algorithm does have three important costs during normal operation. First, accesses to stable storage are indirect and therefore more expensive. However, this cost may be small if the directory is small enough to be stored in cache. Second, finding uncommitted versions and reclaiming their space may be difficult to do efficiently, given the absence of a log. Third, and most importantly, the movement of data to new versions destroys the original layout of the stable database. That is, when a data item is updated, there may not be space for the new copy close to the original data item's (i.e., its shadow's) location. When the update is committed, the data item has changed location from the shadow to the new version. Thus, if the database is designed so that related data items are stored in nearby stable storage locations, that design will be compromised over time as some of those data items are updated. For example, if records of a file are originally stored contiguously on disk for efficient sequential access, they will eventually be spread into other locations thereby slowing down sequential access. This problem is common to many implementations of shadowing.

Because of the organization of the log, this algorithm is also known as the *shadow version algorithm*. And because of the way in which it commits a transaction, by atomically recording all of a transaction's updates in the stable database, it has also been called the *careful replacement algorithm*.