Atomic Transaction Implementation

Atomic Transaction Implementation

Private Workspace,

Writeahead Log,

Two-Phase Commit Protocol.

Private Workspace.

When a process (like a program or task) starts a transaction, it's given its own personal space, called a private workspace. This workspace contains all the files and data that the process needs to work with. Think of it like a copy of the files on your computer that only the process can see and modify. While the process is working, all its changes happen in this private workspace and not on the real files.

Committing or Aborting: The process will either commit or abort the transaction:
If it commits, the changes it made in the private workspace are saved to the real files.
If it aborts, all changes are discarded, and nothing happens to the real files.

Advantages: The advantage of this approach is that each process gets its own workspace as soon as the transaction starts. This prevents other processes from interfering with its work and provides safety until the process is done.

- **Disadvantages and Optimizations:** The downside is that copying everything into a private workspace can be expensive and time-consuming. So, there are optimizations to make it faster:

- If the process reads a file but doesn't change it, there's no need to copy that file into the private workspace. The process can just use the real file directly (as long as it hasn’t been changed by another process since the transaction started).
- Instead of copying all files at the start, the private workspace can begin empty and only get files when the process actually needs them. If a file is opened for writing (modifying), only the part of the file that is being changed gets copied into the private workspace.
- When changes are made, only the modified parts of the file are updated in the private workspace. When the process is done and commits the transaction, only these modified parts are saved back to the original files.
- In short, this system ensures processes can work on files safely in their own space without messing up real files, while using smart tricks to avoid unnecessary copying and save time.

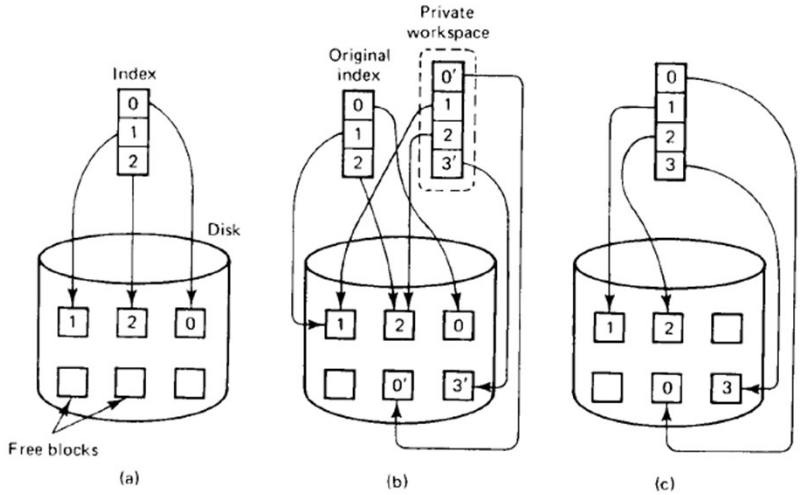

Fig. 3-18. (a) The file index and disk blocks for a three-block file. (b) The situation after a transaction has modified block 0 and appended block 3. (c) After committing.

As can be seen from Fig. 3-18(b), the process running the transaction sees the modified file, but all other processes continue to see the original file. In a

In UNIX, the index is the i-node. Using the private index, the file can be read in the usual way, since the disk addresses it contains are for the original disk blocks. However, when a file block is first modified, a copy of the block is made and the address of the copy inserted into the index, as shown in Fig. 3-18. The block can then be updated without affecting the original. Appended blocks are handled this way too. The new blocks are sometimes called shadow blocks.

In Fig. 3-18.

(a) indexes of file before modifying.

(b) Situation transacion has modified block 0 and new block (block 3) appended to file. here observe private workspace indexes and extra copis.

(c) after comminting transactions

If the transaction aborts, the private workspace is simply deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent's workspace atomically, as shown in Fig. 3-18(c). The blocks that are no longer reachable are put onto the free list.

Writeahead Log(intentions list)

With this method,
Files are actually modified in place, but before any block is changed, a record is written to the writeahead log on stable storage telling which transaction is making the change, which file and block is being changed, and what the old and new values are.
Only after the log has been written successfully is the change made to the file.

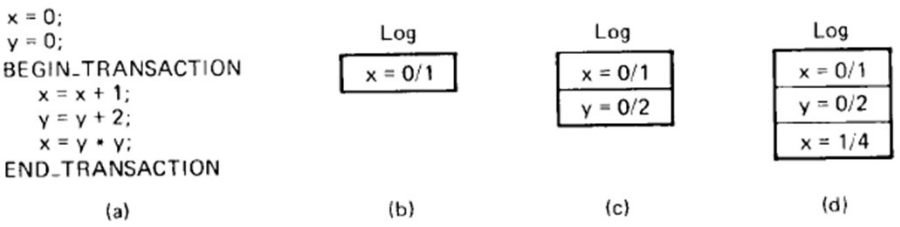

Fig. 3-19. (a) A transaction. (b)–(d) The log before each statement is executed.

Figure 3-19 gives an example of how the log works. In Fig. 3-19(a) we have a simple transaction that uses two shared variables (or other objects), x and y, both initialized to 0.

Fig.3-19 (b),(c) and (d) For each of the three statements inside the transaction, a log record is written before executing the statement, giving the old and new values, separated by a slash.

Case 1: If the transaction succeeds and is committed:

If the transaction succeeds and is committed, a commit record is written to the log, but **the data structures do not have to be changed, as they have already been updated.**

Case 2: If the transaction aborts

If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the change described in it undone. This action is called a rollback.

Case 3: The log can also be used for recovering from crashes.

Suppose that the process doing the transaction crashes just after having written the last log record of Fig. 3-19(d), but before changing x. After the failed machine is rebooted, the log is checked to see if any transactions were in progress at the time of the crash.

When the last record is read and the current value of x is seen to be 1, it is clear that the crash occurred before the update was made, so x is set to 4.

If, on the other hand, x is 4 at the time of recovery, it is equally clear that the crash occurred after the update, so nothing need be changed.

Using the log, it is possible to go forward(do the transaction) or go backward(undo the transaction)

Two-Phase commit Protocol

As we have pointed out repeatedly, the action of committing a transaction must be done atomically, that is, instantaneously and indivisibly.

In a distributed system, the commit may require the cooperation of multiple processes on different machines, each of which holds some of the variables, files, and data bases, and other objects changed by the transaction.

Two-phase commit protocol one of widely used protocol for achieving atomic commit in a distributed system.

The basic idea is illustrated in Fig. 3-20. One of the processes involved functions as the coordinator. Usually, this is the one executing the transaction.

The commit protocol begins when the coordinator writes a log entry saying that it is starting the commit protocol, followed by sending each of the other processes involved (the subordinates) a message telling them to prepare to commit.

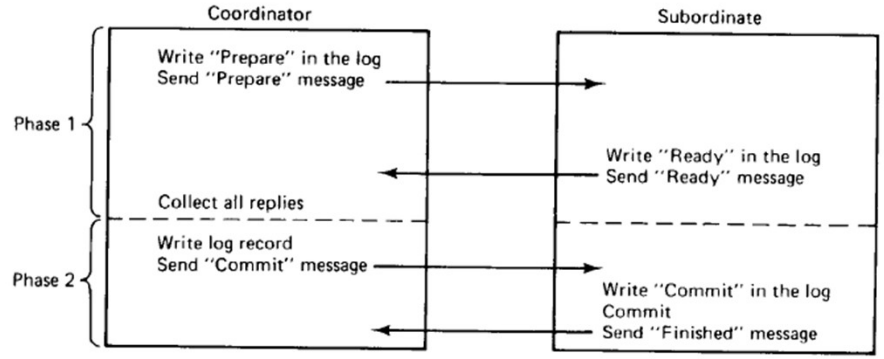

Fig. 3-20. The two-phase commit protocol when it succeeds.

When a subordinate gets the message it checks to see if it is ready to commit, makes a log entry, and sends back its decision.
When the coordinator has received all the responses, it knows whether to commit or abort. If all the processes are prepared to commit, the transaction is committed.

If one or more are unable to commit (or do not respond), the transaction is aborted. Either way, the coordinator writes a log entry and then sends a message to each subordinate informing it of the decision.

It is this write to the log that actually commits the transaction and makes it go forward no matter what happens afterward.

Due to the use of the log on stable storage, this protocol is highly resilient in the face of (multiple) crashes.

Q&A

Thank You!