Concurrency Control Algorithms in Atomic Transaction

Concurrency Control

When multiple transactions are happening at the same time, they need to be managed properly to avoid conflicts. This is where concurrency control comes in.

Locking

Locking is a way to manage access to files (or other data). A process locks a file before reading or writing it, so other processes cannot access it until the lock is released.

Types of Locks:
Read Lock: Multiple processes can read a file at the same time, but no one can write to it.
Write Lock: Only one process can write to a file, and no other locks (read or write) are allowed.

This system ensures that a file doesn’t get changed while a transaction is still using it.

Granularity of Locking:

Locking can apply to different levels, such as:

Entire file (larger granularity).
Individual records or pages (smaller granularity).

Smaller granularity allows more processes to work simultaneously but can lead to more deadlocks.

Two-Phase Locking (2PL)

To avoid problems like inconsistency and deadlocks, two-phase locking (2PL) is used. In 2PL, the process works in two phases:

Growing Phase: The process collects all the locks it needs.
Shrinking Phase: Once it has everything, it starts releasing the locks after completing the transaction.

If all transactions follow 2PL, it guarantees that they will run correctly without conflicts.

Strict Two-Phase Locking is a common version of 2PL. Here, the process holds onto all locks until it either completes or cancels the transaction. This ensures:

No transaction reads files it shouldn’t.

The system can manage locks automatically without programmer intervention.

Deadlocks

A deadlock happens when two processes block each other, each waiting for the other to release a lock. Here's how to handle deadlocks:

- Acquire locks in a specific order to avoid circular waiting.
- Use deadlock detection, where the system monitors lock requests and checks for cycles in the process graph.
- Implement timeouts, meaning a lock is released if it stays locked for too long.

These methods help prevent or fix deadlocks, ensuring smooth transaction execution.

Optimistic Concurrency Control is another way to handle multiple transactions at the same time. The idea is simple: let each transaction run as if no other transactions exist, and only check for problems at the end.

How It Works:

- Each transaction reads and writes files as if it's the only one doing so.
- When the transaction is ready to commit (finalize), the system checks if any of the files it worked on have been changed by another transaction since it started.
- If no changes have been made by others, the transaction is committed.
- If changes were made, the transaction is aborted (canceled), and it may have to start over.

Advantages:

- Deadlock-free: Since no locks are used, there's no risk of deadlocks (where two processes block each other).
- Maximum parallelism: Multiple transactions can happen at once without having to wait for one another.

Disadvantages:

- Failure risk: Sometimes, a transaction may fail if it finds that files it worked on were changed by others. In such cases, the entire transaction has to be repeated.
- Heavy load problems: When many transactions are running at the same time (heavy load), the chances of failure increase, making optimistic concurrency control less efficient.

Timestamp-Based Concurrency Control

Timestamp-based concurrency control is another method to manage transactions running at the same time. Each transaction is given a unique timestamp when it begins, ensuring that transactions are processed in the correct order.

How It Works:

Every file has two timestamps:
Read timestamp (TRD): The time when a committed transaction last read the file.
Write timestamp (TWR): The time when a committed transaction last wrote to the file.

When a transaction tries to read or write a file, it checks these timestamps:

- If the file's timestamps are older than the transaction’s timestamp, the transaction proceeds.
- If the file’s timestamps are newer, it means another transaction has already modified the file, and the current transaction is aborted.

Read Timestamp (TRD): For each data item, the TRD indicates the largest timestamp of any transaction that has successfully read the item.

Write Timestamp (TWR): For each data item, the TWR indicates the largest timestamp of any transaction that has successfully written to the item.

Read Operation

- When a transaction Ti with timestamp T(Ti) attempts to read a data item X
- If T(Ti) < TWR(X) then Ti is aborted and restarted with a new timestamp. This is because Ti is
- trying to read a version of X that has already been overwritten by a newer transaction.
- If T(Ti) >= TWR(X) then Ti is allowed to proceed. Then TR(X) is updated to max(TR(X),T(Ti))

Write Operation

- When a transaction Ti with timestamp T(Ti) attempts to write to a data item X:
- If T(Ti) < TR(X) then Ti is aborted and restarted. Ti is trying to overwrite a value that has already
- been read by a newer transaction.
- If T(Ti)< TWR(X) then Ti is also aborted and restarted, as it is attempting to write an outdated value.
- If T(Ti) ≥ max(TR(X), TWR(X)) then the write is allowed to proceed, and TWR(X) is updated to TR(Ti).

Example

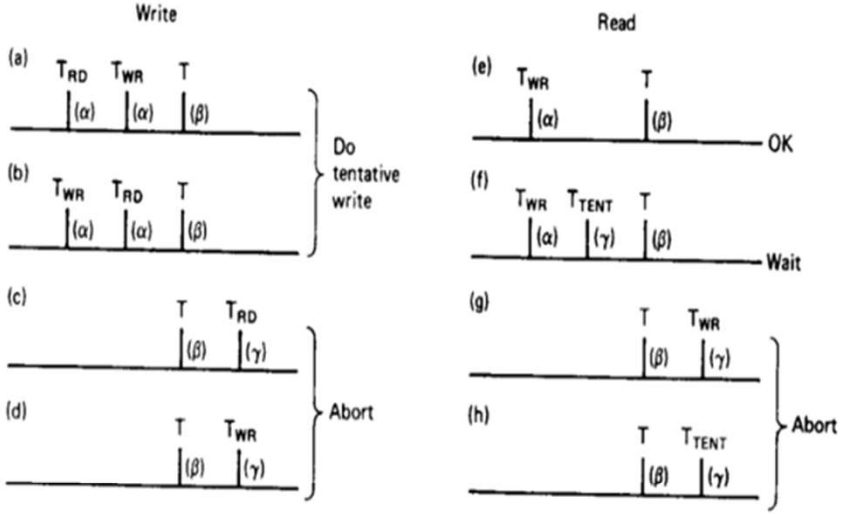

Fig. 3-22. Concurrency control using timestamps.

- There are three transaction alpha, beta and gamma. Alpha ran a long time ago, and used every file needed by beta and gamma.
- So, all files have read and write timestamps set to alpha's time stamps.
- Beta and gamma start concurrently, with beta having a lower timestamp than gamma .

- Let T be the timestamp of beta.
- TRD and TWR be read and write time stamp of the file which is the time stamp of alpha.

- Let beta writing a file.
- T > TRD and TWR (Fig.a,b) (gamma is not committed)
- write is accepted and done tentatively.(will be permanent when beta committed)
- Beta's timestamp is recorded in the file as a tentative write.

Contd...

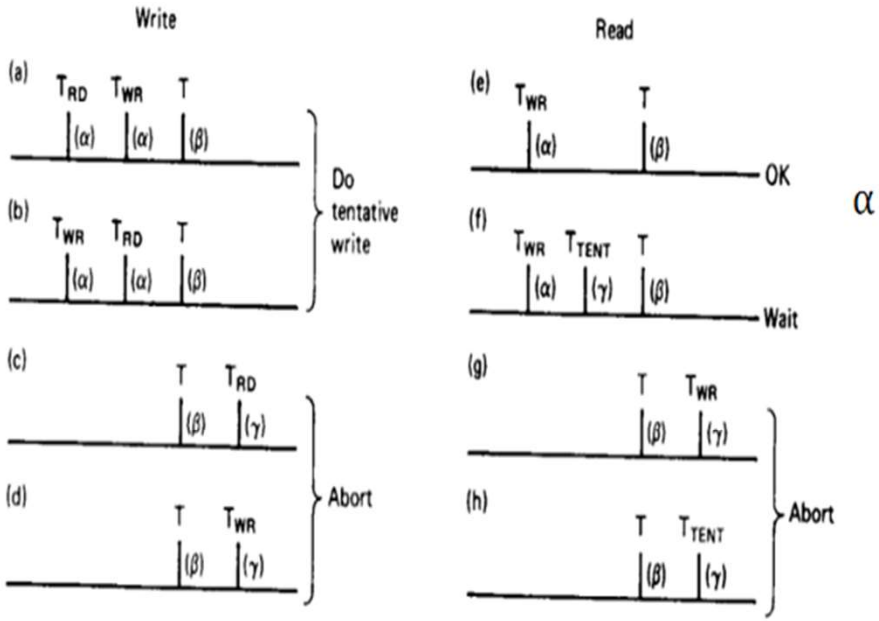

Fig. 3-22. Concurrency control using timestamps.

Fig. c, d.
Gamma has either read or write the file and committed.
Beta's transaction is aborted.

Beta wants to read the file.
Fig. e. $T > TWR(\alpha)$ read done immediately.

Fig. f. Some interloper with timestamp lower than beta's is trying to write the file.
Beta waits until the interloper commits.

Fig. g. Gamma has changed the file and committed. Beta will abort.

Fig. h. Gamma is in the process of changing the file and has not committed yet, beta is too late and aborted.

Conclusion

- Transaction has many advantages and thus are a promising technique for building reliable distributed system.
- It has great implementation complexity which yield low performance.