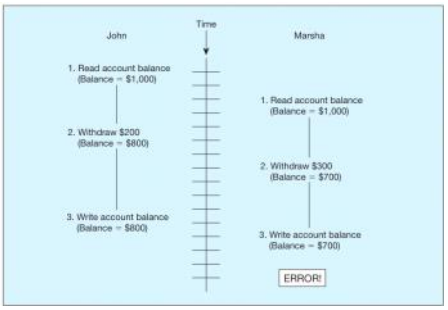
CONTROLLING CONCURRENT ACCESS

Databases are shared resources. Database administrators must expect and plan for the likelihood that several users will attempt to access and manipulate data at the same time.

- With concurrent processing involving updates, a database without concurrency control will be compromised due to interference between users.
- There are two basic approaches to concurrency control: a pessimistic approach (involving locking) and an optimistic approach (involving versioning).
- If users are only reading data, no data integrity problems will be encountered because no changes will be made in the database.
- However, if one or more users are updating data, then potential problems with maintaining data integrity arise.
- When more than one transaction is being processed against a database at the same time, the transactions are considered to be concurrent.
- The actions that must be taken to ensure that data integrity is maintained are called currency control actions.

The Problem of Lost Updates

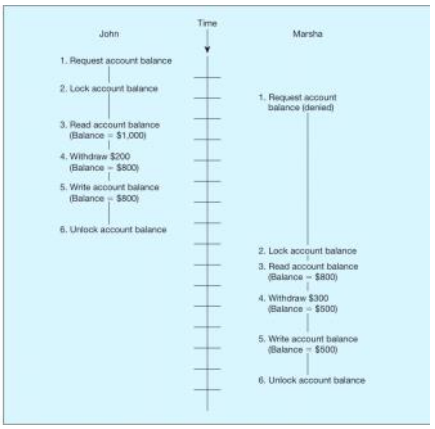
- The most common problem encountered when multiple users attempt to update a database without adequate concurrency control is lost updates.
- Example:
  John and Marsha have a joint checking account, and both want to withdraw some cash at the same time, each using an ATM terminal in a different location. Figure 7-20 shows the sequence of events that might occur in the absence of a concurrency control mechanism. John's transaction reads the account balance (which is $1,000), and he proceeds to withdraw $200. Before the transaction writes the new account balance ($800), Marsha's transaction reads the account balance (which is still $1,000). She then withdraws $300, leaving a balance of $700. Her transaction then writes this account balance, which replaces the one written by John's transaction. The effect of John's update has been lost due to interference between the transactions, and the bank is unhappy.



- Another similar type of problem that may occur when concurrency control is not established is the inconsistent read problem.
  - This problem occurs when one user reads data that have been partially updated by another user. The read will be incorrect and is sometimes referred to as a dirty read or an unrepeatable read.

Serializability

- Concurrent transactions need to be processed in isolation so that they do not interfere with each other.
  - If one transaction were entirely processed before another transaction, no interference would occur.
- Procedures that process transactions so that the outcome is the same as this are called serializable.
  - Processing transactions using a serializable schedule will give the same results as if the transactions had been processed one after the other.
- Schedules are designed so that transactions that will not interfere with each other can still be run in parallel.
  - For example, transactions that request data from different tables in a database will not conflict with each other and can be run concurrently without causing data integrity problems.
- Serializability is achieved by different means, but locking mechanisms are the most common type of concurrency control mechanism.
  - With locking, any data that are retrieved by a user for updating must be locked, or denied to other users, until the update is complete or aborted.
  - Locking data is much like checking a book out of the library; it is unavailable to others until the borrower returns it.



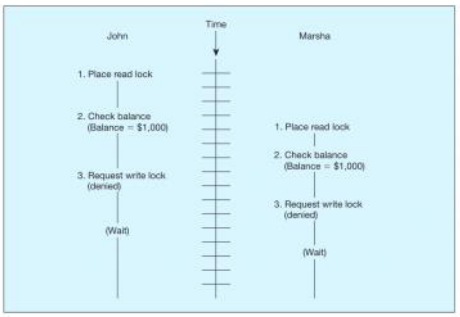
Locking Mechanisms

Figure 7-21 shows the use of record locks to maintain data integrity. John initiates a withdrawal transaction from an ATM. Because John's transaction will update this record, the application program locks this record before reading it into main memory. John proceeds to withdraw $200, and the new balance ($800) is computed. Marsha has

- initiated a withdrawal transaction shortly after John, but her transaction cannot access the account record until John's transaction has returned the updated record to the database and unlocked the record. The locking mechanism thus enforces a sequential updating process that prevents erroneous updates.

- LOCKING LEVEL
  - is the extent of the data base resource that is included with each lock.
  - • ***Database*** The entire database is locked and becomes unavailable to other users. This level has limited application, such as during a backup of the entire database (Rodgers, 1989).

• ***Table*** The entire table containing a requested record is locked. This level is appropriate mainly for bulk updates that will update the entire table, such as giving all employees a two percent raise.
• ***Block or page*** The physical storage block (or page) containing a requested record is locked. This level is the most commonly implemented locking level. A page will be a fixed size (4K, 8K, and so forth) and may contain records of more than one type.
• ***Record*** Only the requested record (or row) is locked. All other records, even within a table, are available to other users. It does impose some overhead at run time when several records are involved in an update.
• ***Field*** Only the particular field (or column) in a requested record is locked. This level may be appropriate when most updates affect only one or two fields in a record. For example, in inventory control applications, the quantity-on-hand field changes frequently, but other fields (e.g., description and bin location) are rarely updated. Field-level locks require considerable overhead and are seldom used.

- TYPES OF LOCKS
  - So far, we have discussed only locks that prevent all access to locked items. But in reality, there are two types of locks:
    - 1. *Shared locks* **Shared locks (S locks or read locks)** allow other transactions to read (but not update) a record or other resource. A transaction should place a shared lock on a record or data resource when it will only read but not update that record. Placing a shared lock on a record prevents another user from placing an exclusive lock (but not a shared lock) on that record.
    - 2. *Exclusive locks* **Exclusive locks (X locks or write locks)** prevent another transaction from reading (and therefore updating) a record until it is unlocked. A transaction should place an exclusive lock on a record when it is about to update that record (Descollonges, 1993). Placing an exclusive lock on a record prevents another user from placing any type of lock on that record.
    - 

      When John initiates his transaction, the program places a read lock on his account record because he is reading the record to check the account balance. When John requests a withdrawal, the program attempts to place an exclusive lock (write lock) on the record because this is an update operation. However, as you can see in the figure, Marsha has already initiated a transaction that has placed a read lock on the same record. As a result, his request is denied; remember that if a record is a read lock, another user cannot obtain a write lock.

- DEADLOCK: an impasse that results when two or more transactions have locked a common resource and each must wait for the other to unlock that resource.
  - Example: Figure 7-22 shows a simple example of deadlock. John's transaction is waiting for Marsha's transaction to remove the read lock from the account record and vice versa. Neither person can withdraw money from the account even though the balance is more than adequate.
    - The image directly above this is the example being referenced
- MANAGING DEADLOCK
  - There are two basic ways to resolve deadlocks: deadlock prevention and deadlock resolution.
    - When deadlock prevention is employed, user programs must lock all records they will require at the beginning of a transaction rather than one at a time.

      Two-phase locking protocol

      A procedure for acquiring the necessary locks for a transaction in which all necessary locks are acquired before any locks are released, resulting in a growing phase when locks are acquired and a shrinking phase when they are released.

      Deadlock prevention

      A method for resolving deadlocks in which user programs must lock all records they require at the beginning of a transaction (rather than one at a time).

      - A good option but can often lead to issues still because we wont really know always in advance how many locks we are going to need
    - deadlock resolution mechanisms work as follows: The DBMS maintains a matrix of resource usage, which, at a given instant, indicates what subjects (users) are using what objects (resources).
      - By scanning this matrix, the computer can detect deadlocks as they occur. The DBMS then resolves the deadlocks by "backing out" one of the deadlocked transactions. Any changes made by that transaction up to the time of deadlock are removed, and the transaction is restarted when the required resources become available.

      Deadlock resolution

      An approach to dealing with deadlocks that allows deadlocks to occur but builds mechanisms into the DBMS for detecting and breaking the deadlocks.

Versioning: With versioning, there is no form of locking. Each transaction is restricted to a view of the database as of the time that transaction started, and when a transaction modifies a record, the DBMS creates a new record version instead of overwriting the old record.

- Locking mechanism rebuttal:
  Locking, as described here, is often referred to as a pessimistic concurrency control mechanism because each time a record is required, the DBMS takes the highly cautious approach of locking the record so that other programs cannot use it. In reality, in most cases other users will not request the same documents, or they may only want to read them, which is not a problem. Thus, conflicts are rare.
- takes the optimistic approach that most of the time other users do not want the same record, or, if they do, they want to only read (but not update) the record.
- Analogy for understanding versioning

  The best way to understand versioning is to imagine a central records room, corresponding to the database (Celko, 1992). The records room has a service window. Users (corresponding to transactions) arrive at the window and request documents (corresponding to database records). However, the original documents never leave the records room. Instead, the clerk (corresponding to the DBMS) makes copies of the requested documents and time stamps them. Users then take their private copies (or versions) of the documents to their own workplace and read them and/or make changes. When finished, they return their marked-up copies to the clerk. The clerk merges the changes from marked-up copies into the central database. When there is no conflict (e.g., when only one user has made changes to a set of database records), that user's changes are merged directly into the public (or central) database.

- What happens when there is conflict?

Suppose instead that there is a conflict; for example, say that two users have made conflicting changes to their private copy of the database. In this case, the changes made by one of the users are committed to the database. (Remember that the transactions are time stamped so that the earlier transaction can be given priority.) The other user must be told that there was a conflict, and his work cannot be committed (or incorporated into the central database). She must check out another copy of the data records and repeat the previous work. Under the optimistic assumption, this type of rework will be the exception rather than the rule.

- Example

Figure 7-24 shows a simple example of the use of versioning for the checking account example. John reads the record containing the account balance and successfully withdraws $200, and the new balance ($800) is posted to the account with a COMMIT statement. Meanwhile, Marsha has also read the account record and requested a withdrawal, which is posted to her local version of the account record. However, when the transaction attempts to COMMIT, it discovers the update conflict, and her transaction is aborted (perhaps with a message such as "Cannot complete transaction at this time"). Marsha can then restart the transaction, working from the correct starting balance of $800.