Concurrency and correctness

- Concurrency opens the door for potential correctness issues not present in sequential code.

- We need mechanisms to protect data and state to maintain consistency during execution.

Protection mechanisms

- Locks
  - Acquire/release protocol. Blocking acquire if lock unavailable.
- Semaphores [Dijkstra]
  - Counters with increment/decrement rules, blocking decrement unless value is positive. Can be more flexible than boolean locks.
- Monitors [Brinch-Hansen/Hoare]
  - Encapsulation of locks and data within an object.
- Transactions
  - Speculative execution of critical code with rollback and commit capabilities.

Transactions

- Recall the mutual exclusion problem. Why was it important?
- Concurrently executing processes potentially executing some block of code that, if executed by more than one process at the same time, could result in correctness problems.
- So, mutual exclusion was used to protect it.

Transactions

- Transactions are another mechanism to deal with concurrency and sensitive blocks of code.

- The idea originated in the databases community, but has since found applicability in more general contexts.
  - Example: Software transactional memory.
  - Example: Transactional memory hardware
    - Sun "Rock" processor.
      - Adds new instructions for starting, committing, and determining failure of transactions.
      - Fixed bound store queue for transactions.
      - Hardware can detect situations resulting in a failure (eg: context switch, TLB misses, store queue overflow, etc…).

Transactions

- So what is a transaction?

- A sequence of operations that are to be:
  - Free of interference by processes other than the one executing them.
  - Executed as a successful whole, or not at all.
    - No partial execution.
    - The proper term for this is atomicity.

Requirements

- The database community has come up with a set of requirements for transactions:
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- ACID

Atomicity

- The essential property is that of atomicity.
- What is an atomic operation?
  - It is an operation that is indivisible.

- For sequences of operations, they are atomic if to any outsider, they appear to be a single operation.

- Consider the bank account update. An atomic implementation of that would make it appear to external observers that the balances on both accounts changed **simultaneously**, eliminating the possibility of seeing any intermediate, inconsistent state.

Consistency

- The state of the system that starts in a legal state before a transaction will remain in a legal state afterwards.

- This is hard to maintain in a general transaction system beyond just databases.
  - “Legal” state requires too much semantic information from the specific application for a general system to verify.
  - On the other hand, one can set constraints in a database definition to represent what is considered to be “legal”, so there is more hope of enforcing consistency in this more restricted world.

Isolation

- "What happens in the transaction stays in the transaction."

- Any intermediate computations performed by the transaction are not visible outside the transaction. The intermediate computations could represent inconsistent state, and we want them totally hidden.
  - Think of the intermediate balance computations during the bank transfer. We want these totally isolated and never visible to other processes.

Durability

- When the transaction completes, the initiator of the transaction is guaranteed that the result will persist.

- The durability of the result is only as durable as the system it is stored in.
  - Durability doesn't mean that the data can't be destroyed.
  - But, the system will do it's best to keep it around as long as it should be.
    - Examples: RAID storage, replication of servers, writing to nonvolatile storage.
    - Clearly durability typically involves making more than one copy on different storage media.

Transactions

- So what is special about this that plain mutual exclusion doesn’t already do?

- Simple – transactions do NOT force the acquisition of a lock to enter the section.
  - Locking is conservative : make it impossible to do something dangerous.

- Transactions focus on undoing what was intended to be atomic in the event that another process intruded in during the transaction.
  - So, we basically are more optimistic and only worry about cleaning up after conflicts.

Transaction primitives

- **Open**: this tells the underlying support infrastructure that a transaction is to be started.
- **Close**: this tells the support infrastructure that it is done, and the results are to be committed if the transaction was successful.
  - Close yields a success or failure result. Failure means the transaction was aborted.
- **Abort**: This tells the system that the transaction has gone sour and needs to be aborted. Any work done since the open transaction occurred needs to be undone.

Recoverable objects

- The terminology the book uses for persistent objects is “recoverable objects”.
- Recovery means that after a crash, the objects can be resurrected.

- We consider that any data that has successfully lived through a commit is in a place where it can be considered recoverable.