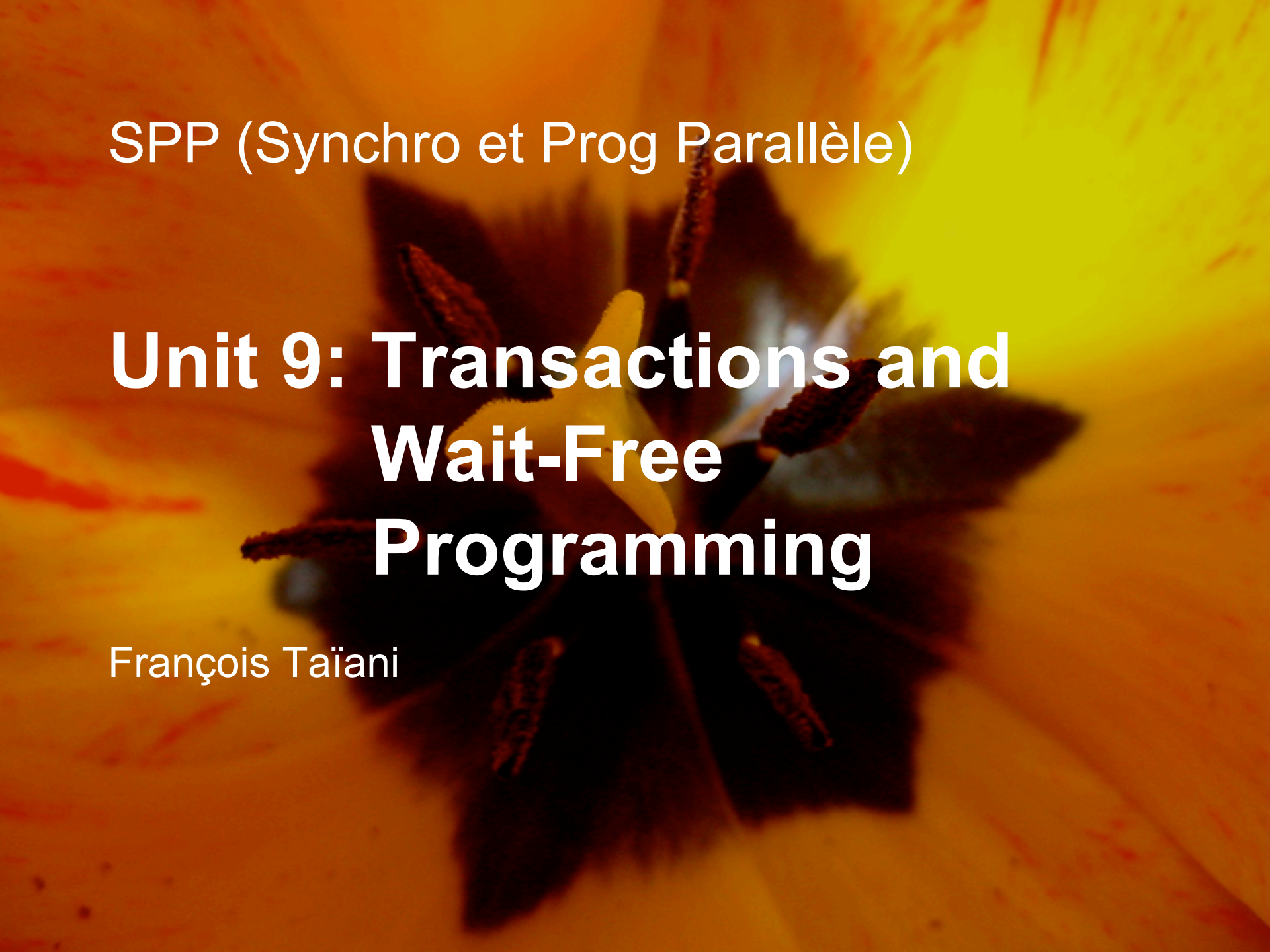


SPP (Synchro et Prog Parallèle)

Unit 9: Transactions and Wait-Free Programming

François Taïani

Transactions

- Main Synchronization Mechanisms used in DB
- Principle: 2 new operations
  - ➔ dbStore.startTransaction()
  - ➔ dbStore.endTransaction() (aka commit)
- Often completed with 3rd operation
  - ➔ dbStore.abortTransaction()
- Use (dbStore dropped for brevity)
  - ➔ startTransaction

    some sequence of actions (reads, writes, computation)

    endTransaction

Semantic

- Transactions
  - ➔ either execute in totality or not at all (“atomicness”)
  - ➔ concurrent transactions don’t interfere (“isolation”)
- Notes on DB
  - ➔ Other usual properties: Consistent (app) / Durable (FT)
- Notes on atomicity
  - ➔ “Atomicness” ≠ from “atomic” concurrent objects
  - ➔ “Atomicness” -> transactions must be able to roll back
- Notes on Isolation
  - ➔ Different levels / semantics of “isolation”
  - ➔ “Isolation” closely related to linearizability

Atomicness of Transactions

- Vocabulary: “Atomic” is overloaded
  - ➔ usual name in DB = atomicity, but confusing here
  - ➔ “atomicness” avoids confusion w/ atomic objects
- Key principles to provide atomicness
  - ➔ must be able to cancel transactions halfway through
  - ➔ if cancelled, any effect of the transaction should disappear
  - ➔ “as if” never happened -> might cause cascading roll backs
- Two main strategies
  - ➔ work on shadow copies of variables, only write if success
  - ➔ log all write operations, undo if abort
  - ➔ (not that easy, but we’ll ignore details here)

Isolation of Transactions

- Key problem: schedule operations of transactions
- Naïve approach: serialise all transactions
  - ➔ transactions execute one after the other
  - ➔ easy, but poor concurrency, poor resource utilisation
- General approach:
  - ➔ interleave actions of different transactions
  - ➔ use scheduling techniques to insure “correct” outcome
  - ➔ (might include aborting transaction that don’t fit)

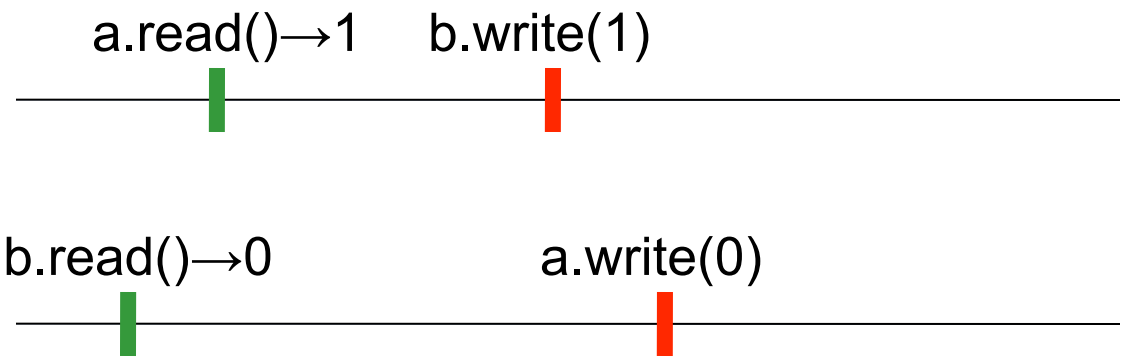
Isolation of Transactions

What is a correct outcome? Three main semantics

- Serializability: A schedule S is serializable iff
  - ➔ ∃ sequential schedule, same result of chosen schedule
  - ➔ (Note: similar to sequential consistency for conc. objects)
- Strict Serializability: S is strictly serializable iff
  - ➔ ∃ sequential schedule S', same result as S
  - ➔ S' respects precedence order of S ($<_S \subseteq <_{S'}$)
  - ➔ (Note: similar to atomicity for concurrent objects)
- Snapshot isolation: S respects snapshot isolation iff
  - ➔ each transaction = work on personal snapshot of DB
  - ➔ successful provided updated value not in conflict

Example

- Consider this example
  - Transaction 1: x := a.read() ; b.write(x) ;
  - Transaction 2: y := b.read() ; a.write(y) ;
  - x and y are local variables
  - the system is initialised as a = 1 ; b = 0
- What semantic(s) does the following schedule follow?



Comment on Example

- A case of write-skew anomaly
  - ➔ write-sets of T1 & T2 disjoint: no conflicting updates
  - ➔ so no risk of one overwriting the results of the other
  - ➔ but read-set of T2 (b) updated by T1 before end of T2
  - ➔ impossible under (strict) serializability semantic
- In General
  - ➔ snap. isolation $<_{weaker}$ seria^{lizability} $<_{weaker}$ strict serial^{izability}
  - ➔ $<_{weaker}$ means “allows more schedules than”
  - ➔ consequence: provide less guarantees

Implementing Isolation

- 2PL: 2 phase locking (provides serializability)
  - ➔ growth phase: take locks on resources
  - ➔ skrinking phase: release locks on resources
  - ➔ detect deadlocks and abort
  - ➔ or avoid deadlocks by order on resources

- Other approaches (not discussed)
  - ➔ Timestamp Ordering (provides serializability)
  - ➔ Multiversion Concurrency Control (for snapshot isolation)

Wait-Free Programming

- Principle: avoid locks!
  - ➔ dangerous: deadlocks, starvation
  - ➔ inefficient: diminish concurrency
- Ideal: Wait-Free concurrent object
  - ➔ all operation can always progress (only limited by CPU)
- Quiz: Does wait-freedom makes sense for a queue?

Wait-Free Programming

- Principle: avoid locks!
  - ➔ dangerous: deadlocks, starvation
  - ➔ inefficient: diminish concurrency
- Ideal: Wait-Free concurrent object
  - ➔ all operation can always progress (only limited by CPU)
- Quiz: Does wait-freedom makes sense for a queue?
  - ➔ Yes, but need to allow “undefined” return value (⊥)
- Difficulty:
  - ➔ very complex to design and prove
  - ➔ but some practical algorithms do exists (for queues in particular)

Summary

- Broad brush presentation of transactions
  - ➔ Very rich fields, of crucial importance
  - ➔ Both for DB, but also transactional engine (J2EE…)
  - ➔ A large range of semantics for isolation
  - ➔ A large range of scheduling algorithms
- Wait free concurrent objects
  - ➔ still an area of research for practicable implementations
  - ➔ important to be aware of its existence: performance ↗
  - ➔ variant properties: non-blocking, lock-free
- We have only touched the surface on both subjects