# CS 186 Discussion Section
# Week 14

Peter Alvaro and Kuang Chen

April 27, 2009

## 1 Transactions and Serializibility

- *Serial* and *serializable* schedules
- *Conflict-serializable* schedules: conflict equivalent to some serial schedule (every pair of conflicting actions is ordered the same way)
- *Dependency* (or *Precedence*) graph is acyclic when schedule is conflict-serializable
- *Recoverable* schedules: transaction $T$ commits after all transactions whose changes $T$ read have committed

### 1.1 Types of conflicts

- Write-read (WR): dirty read (reading uncommitted data)

```
T1: R(A), W(A),                    R(B), W(B), Abort
T2:            R(A), W(A), Commit
```

- Read-write (RW): unrepeatable reads

```
T1: R(A),                    R(A), W(A), Commit
T2:      R(A), W(A), Commit
```

- Write-write (WW): overwriting uncommitted data, can result in *lost update*

```
T1: W(A),                    W(B), Commit
T2:      W(A), W(B), Commit
```

### 1.2 Exercises

#### 1.2.1 For each of the following schedules, answer the questions below:

-
  ```
  T1: R(A); W(B);
  T2:             R(B); W(C);
  T3:                         R(C); W(A);
  ```

---

-
  ```
  T1: R(A);      R(B);                 W(A);
  T2:       R(A);      R(B);                W(B);
  T3:                        R(A);
  T4:                              R(B);
  ```

1. What is the precedence graph for the schedule?

   For the first schedule, the precedence graph: $T1 \rightarrow T2, T2 \rightarrow T3, T1 \rightarrow T3$
   For the second: $T2 \rightarrow T1, T3 \rightarrow T1, T1 \rightarrow T2, T4 \rightarrow T2$

2. Is the schedule conflict-serializable? If so, what are all the equivalent serial schedules?

   For the first graph, yes; equivalent schedules: $T1 \rightarrow T2 \rightarrow T3$.
   For the second, the answer is no; there are cycles in the precedence graph ($T2 \rightarrow T1, T1 \rightarrow T2$)

## 2 Locking

Locking protocols can guarantee the properties of schedules we've been discussing.
**Lock types:**

- *(S)hared*: multiple transactions can hold shared locks on an object at the same time
- *e(X)clusive*: only one exclusive lock, and no other locks, can be on an object at a time

Lock compatibility: $S$-$S$ are compatible; $S$-$X$ and $X$-$X$ are not

**The protocols:**

- Two-Phase locking (2PL):
  - Growing phase: acquire $S$ locks for reading and $X$ locks for writing
  - Shrinking phase: release locks
  - Once begun shrinking phase, cannot ask for more locks
- Strict 2PL:
  - 2PL alone is enough to guarantee conflict-serializability, but it is susceptible to *cascading aborts*
  - Cascading aborts example: rollback of T1 requires rollback of T2

    ```
    T1: R(A), W(A),            R(B), W(B), Abort
    T2:             R(A), W(A)
    ```
  - To prevent cascading aborts, Strict 2PL holds all locks until the very end of the transaction
  - Strict 2PL enhances 2PL by requiring the schedule to be *strict*; strict schedules are recoverable

**Deadlocks:**

- A deadlock is a cycle of transactions waiting for locks to be released by other transactions
- You can *prevent* deadlocks by assigning timestamp-based priorities
- You can *detect* by creating a waits-for-graph

### 2.1 Exercises

#### 2.1.1 In general, is it possible to have a deadlock when the regular two-phase-locking (i.e., non-strict) protocol is obeyed? If yes, give an example; if not, explain briefly. What happens with strict 2PL and conservative 2PL?

Yes. For example, consider the following schedule that deadlocks under 2PL:

```
T1: X-Lock(A), W(A),                   X-Lock(B) ...
T2:                 X-Lock(B), W(B),            X-Lock(A) ...
```

This schedule is allowable under strict 2PL as well, so strict 2PL also has the deadlock problem. Conservative 2PL requires that all locks to be used during the transaction be acquired before the transaction starts. If any one lock cannot be acquired, none of the locks are acquired. So, conservative 2PL does avoid the deadlock problem. On the other hand, DBMSs do not generally use conservative 2PL for avoiding deadlock for performance reasons: all data that MAY be accessed by the transaction must be locked, which greatly reduces the concurrency possible. Instead one of the other many solutions for avoiding deadlocks (e.g., looking for cycles in wait-for graphs) is used.