Atomic Transactions

- The Transaction Model / Primitives
- Serializability
- Implementation
  - Serialization Graphs
  - 2-Phase Locking
  - Optimistic Concurrency Control
- Transactional Memory
- *Reading: Silberschatz et al.: Operating Systems Concepts, 8th ed., Chapter 6.9*

Atomic Transactions

- Example: Online bank transaction:
  `withdraw(amount, account1)`
  `deposit(amount, account2)`
- Q1: What if network fails before deposit?
- Q2: What if sequence is interrupted by another sequence?
- Solution: Group operations in an atomic transaction.
- Primitives:
  - **`BEGIN_TRANSACTION`**
  - **`END_TRANSACTION`**
  - **`ABORT_TRANSACTION`**
  - **`READ`**
  - **`WRITE`**

ACID Properties

Atomic:	transactions happen indivisibly
Consistent:	no violation of system invariants
Isolated:	no interference between concurrent transactions
Durable:	after transaction commits, changes are permanent

Serializability

Schedule is serial if the steps of each transaction occur consecutively.

Schedule is serializable if its effect is "equivalent" to some serial schedule..

```
BEGIN TRANSACTION       BEGIN TRANSACTION       BEGIN TRANSACTION
  x := 0;                 x := 0;                 x := 0;
  x := x + 1;             x := x + 2;             x := x + 3;
END TRANSACTION         END TRANSACTION         END TRANSACTION
```

schedule 1	x=0	x=x+1	x=0	x=x+2	x=0	x=x+3	legal
schedule 2	x=0	x=0	x=x+1	x=x+2	x=0	x=x+3	legal
schedule 3	x=0	x=0	x=x+1	x=0	x=x+2	x=x+3	illegal

Testing for Serializability the hard way: Serialization Graphs

- Input: Schedule *S* for set of transactions $T_1, T_2, \ldots, T_k$.
- Output: Determination whether *S* is serializable.
- Method:
  - Create *serialization graph G*:
    - Nodes: correspond to transactions
    - Arcs: G has an arc from T_i to T_j if there is a T_i:*UNLOCK*(A_m) operation followed by a T_j:*LOCK*(A_m) operation in the schedule.
  - Perform topological sorting of the graph.
    - If graph has cycles, then *S* is not serializable.
    - If graph has no cycles, then topological order is a serial order for transactions.

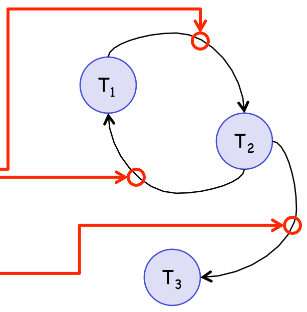
Theorem:

This algorithm correctly determines if a schedule is serializable.

Non-Serializable Schedule: Example

[ref: J.D. Ullman: Principles of Database and Knowledge-Base Systems]

Step	T1	T2	T3
(1)	LOCK A		
(2)		LOCK B	
(3)		LOCK C	
(4)		UNLOCK B	
(5)	LOCK B		
(6)	UNLOCK A		
(7)		LOCK A	
(8)		UNLOCK C	
(9)		UNLOCK A	
(10)			LOCK A
(11)			LOCK C
(12)	UNLOCK B		
(13)			UNLOCK C
(14)			UNLOCK A



Transactions: Implementation Issues

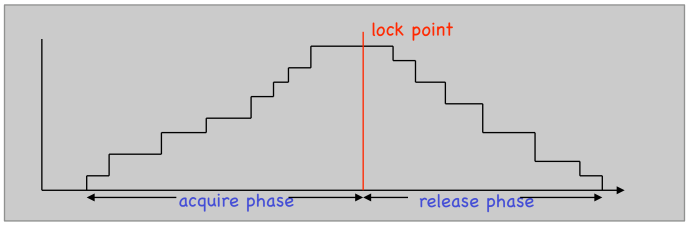
1. How to maintain information from not-yet committed transactions: "Prepare for aborts"
   - private workspace
   - write-ahead log / "intention lists with rollback
   - transactions commit data into database

2. Concurrency control:
   - pessimistic -> lock-based: 2-Phase Locking
   - optimistic -> Timestamp-Based with rollback

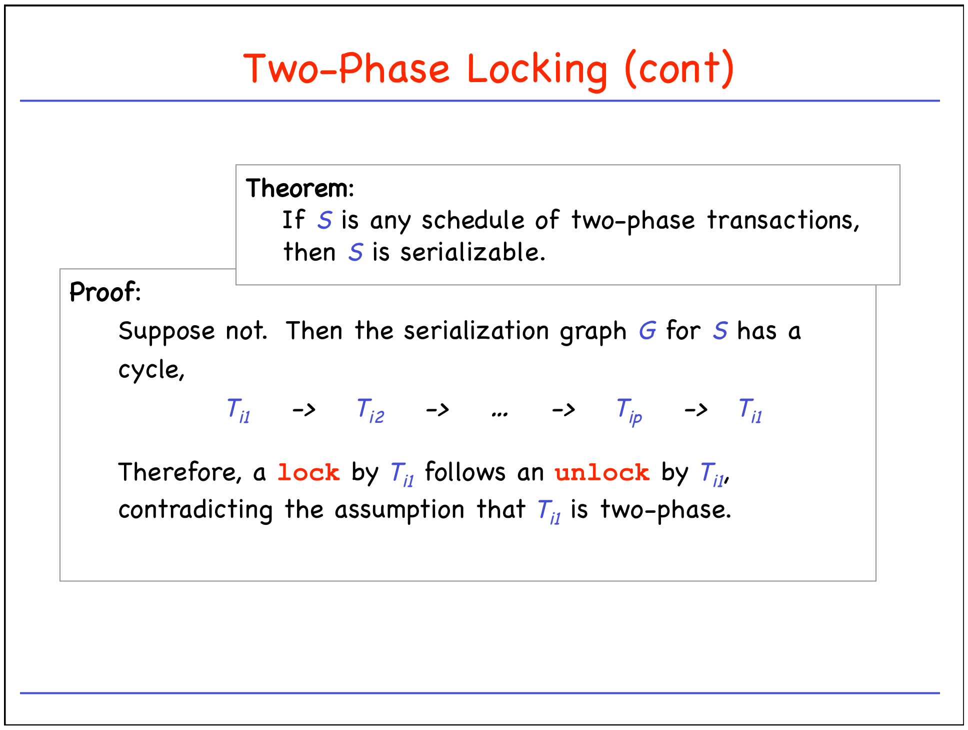
3. Commit protocol
   - 2-Phase Commit Protocol.

Serializability through Two-Phase Locking

- We allow for two types of locks:
  - read locks (non-exclusive)
  - write locks (require exclusive access)
- Enforce serializability through appropriate locking:
  - release locks (and modify data) items only after lock point



- All Two-Phase-Locking schedules are serializable.
- Problems:
  - deadlock prone!
  - allows only a subset of all serializable schedules.



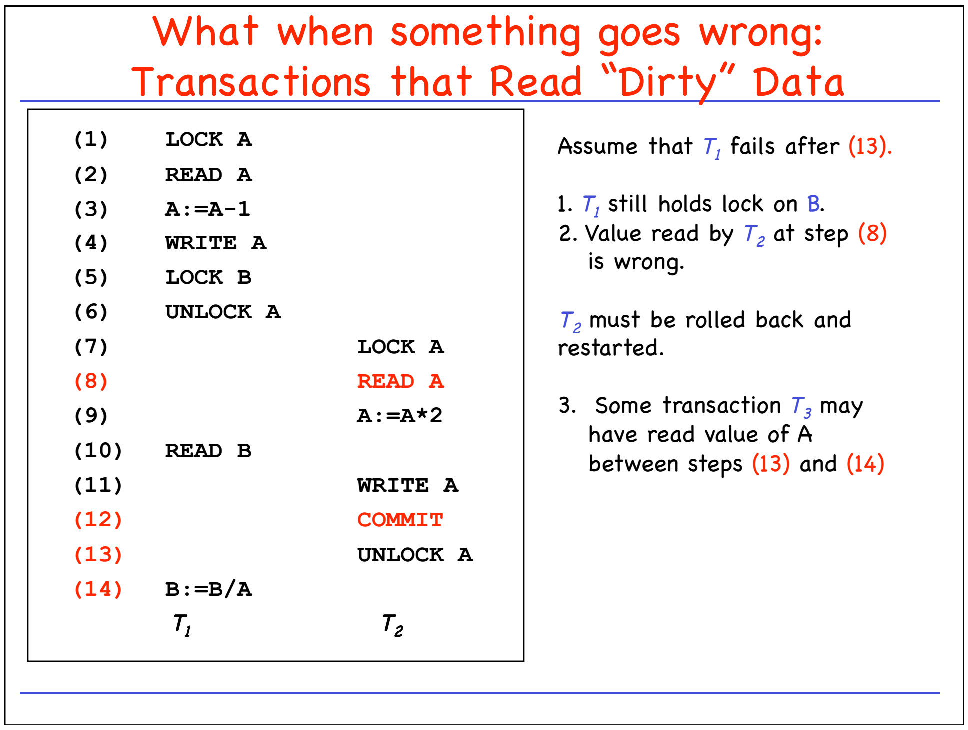
Two-Phase Locking (cont)

Theorem:
If S is any schedule of two-phase transactions, then S is serializable.

Proof:
Suppose not. Then the serialization graph G for S has a cycle,

$$T_{i1} \rightarrow T_{i2} \rightarrow \dots \rightarrow T_{ip} \rightarrow T_{i1}$$

Therefore, a **lock** by T_{i1} follows an **unlock** by T_{i1}, contradicting the assumption that T_{i1} is two-phase.



What when something goes wrong: Transactions that Read "Dirty" Data

	T_1	T_2
(1)	LOCK A	
(2)	READ A	
(3)	A:=A-1	
(4)	WRITE A	
(5)	LOCK B	
(6)	UNLOCK A	
(7)		LOCK A
(8)		READ A
(9)		A:=A*2
(10)	READ B	
(11)		WRITE A
(12)		COMMIT
(13)		UNLOCK A
(14)	B:=B/A	

Assume that T_1 fails after (13).

1. T_1 still holds lock on B.
2. Value read by T_2 at step (8) is wrong.

T_2 must be rolled back and restarted.

3. Some transaction T_3 may have read value of A between steps (13) and (14)

Strict Two-Phase Locking

- Strict two-phase locking:
  - A transaction cannot write into the database until it has reached its commit point.
  - A transaction cannot release any locks until it has finished writing into the database; therefore locks are not released until after the commit point.
- pros:
  - transaction read only values of committed transactions
  - no cascaded aborts
- cons:
  - limited concurrency
  - deadlocks

Optimistic Concurrency Control

"Forgiveness is easier to get than permission"

- Basic idea:
  - Process transaction without attention to serializability.
  - Keep track of accessed data items.
  - At commit point, check for conflicts with other transactions.
  - Abort if conflicts occurred.
- Approach:
  - Assign timestamp to each transaction.
  - Make sure that schedule has the same effect of a serial schedule in order of assigned timestamps.

Timestamp-based Optimistic Concurrency Control

- Data items are tagged with read-time and write-time.

1. Transaction cannot read value of item if that value has not been written until after the transaction executed.
   Transaction with T.S. t_1 cannot read item with write-time t_2 if $t_2 > t_1$.
   (abort and try with new timestamp)
2. Transaction cannot write item if item has value read at later time.
   Transaction with T.S. t_1 cannot write item with read-time t_2 if $t_2 > t_1$.
   (abort and try with new timestamp)

- Other possible conflicts:
  - Two transactions can read the same item at different times.
  - What about transaction with T.S. t_1 that wants to write to item with write-time t_2 and $t_2>t_1$?

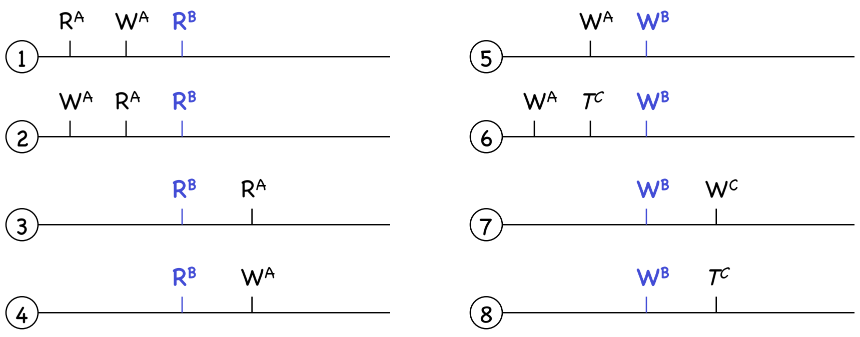
Timestamp-Based Conc. Control (cont)

Rules for preserving serial order using timestamps:

```
a) Perform the operation X
        if X = READ    and t >= $t_w$
     or if X = WRITE   and t >= $t_r$ and t >= $t_w$.

   if X = READ : set read-time  to t if t > $t_r$.
   if X = WRITE: set write-time to t if t > $t_w$.

b) Do nothing if X = WRITE and $t_r$ <= t < $t_w$.

c) Abort transaction if X = READ and t < $t_w$
                     or X = WRITE and t < $t_r$.
```

Timestamp-based Optimistic Concurrency Control

- Accesses to data items are tagged with timestamp
- Examples:



Transactional Memory

- Transactional Memory borrows the concept of an atomic transaction from databases.
- Rather than locking resources, a code block is marked as atomic, and when it runs, the reads and writes are done against a transaction log instead of global memory.
- When the code is complete, the runtime re-checks all of the reads to make sure they are unchanged and then commits all of the changes to memory at once.
- If any of the reads are dirty, the transaction is rolled back and re-executed.
- This, when combined with additional tools for blocking and choice, allows program to remain simple, correct, and composable while scaling to many threads without the additional overhead that course grained locking incurs.

[Phil Windley, Technometria]

Problems with Locking in OS's

[C. J. Rossbach et al. :
TxLinux: Using and Managing Hardware Transactional Memory in an Operating System, SOSP 2007]

- In 2001 study of Linux bugs, 346 of 1025 bugs (34%) involved synchronization.
- 2003 study of Linux 2.5 kernel found 4 confirmed and 8 unconfirmed deadlock bugs.
- Linux source file `mm/filemap.c` has a 50 line comment on the top of the file describing the lock ordering used in the file. The comment describes locks used at a calling depth of 4 from functions in the file.
- Locking is not modular; a component must know about the locks taken by another component in order to avoid deadlocks.
- Other known disadvantages: priority inversion, convoys, lack of composability, and failure to scale with problem size and complexity

Transactional Memory: Primitives (conceptually)

[**M. Herlihy, J. Eliot, B. Mossin: "Transactional memory: architectural support for lock-free data structures,"** Proceedings of the 20th Annual International Symposium on Computer Architecture (1993)]

- Load-transactional (LT): reads the value of a shared memory location into a private register.
- Load-transactional-exclusive (LTX): reads the value of a shared memory location into a private register, "hinting" that the location is likely to be updated.
- Store-transactional (ST): tentatively writes a value from a private register to a shared memory location. This new value does not become visible to other processors until the transaction successfully commits.

Transactional Memory: Primitives (conceptually)

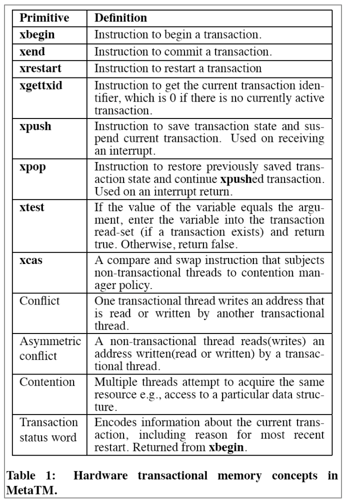
- Commit (COMMIT): attempts to make the transaction's tentative changes permanent. It succeeds only if no other transaction has updated any location in the transaction's data set, and no other transaction has read any location in this transaction's write set.
  - Successful: The transaction's changes to its write set become visible to other processes.
  - Unsuccessful: All changes to the write set are discarded.

  Either way, COMMIT returns an indication of success or failure.

- Abort (ABORT): discards all updates to the write set.

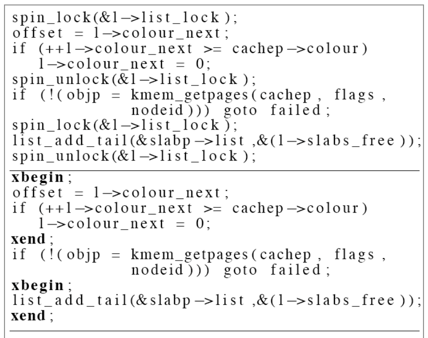
- Validate (VALIDATE): tests the current transaction status.
  - Successful: The current transaction has not aborted (although it may do so later).
  - Unsuccessful: The current transaction has aborted, and VALIDATE discards the transaction's tentative updates.

Transactional Memory in Practice: MetaTM



Primitive	Definition
xbegin	Instruction to begin a transaction.
xend	Instruction to commit a transaction.
xrestart	Instruction to restart a transaction
xgettxid	Instruction to get the current transaction identifier, which is 0 if there is no currently active transaction.
xpush	Instruction to save transaction state and suspend current transaction. Used on receiving an interrupt.
xpop	Instruction to restore previously saved transaction state and continue **xpush**ed transaction. Used on an interrupt return.
xtest	If the value of the variable equals the argument, enter the variable into the transaction read-set (if a transaction exists) and return true. Otherwise, return false.
xcas	A compare and swap instruction that subjects non-transactional threads to contention manager policy.
Conflict	One transactional thread writes an address that is read or written by another transactional thread.
Asymmetric conflict	A non-transactional thread reads(writes) an address written(read or written) by a transactional thread.
Contention	Multiple threads attempt to acquire the same resource e.g., access to a particular data structure.
Transaction status word	Encodes information about the current transaction, including reason for most recent restart. Returned from **xbegin**.

Table 1: Hardware transactional memory concepts in MetaTM.



```
spin_lock(&l->list_lock);
offset = l->colour_next;
if (++l->colour_next >= cachep->colour)
   l->colour_next = 0;
spin_unlock(&l->list_lock);
if (!(objp = kmem_getpages(cachep, flags,
               nodeid))) goto failed;
spin_lock(&l->list_lock);
list_add_tail(&slabp->list,&(l->slabs_free));
spin_unlock(&l->list_lock);
```

```
xbegin;
offset = l->colour_next;
if (++l->colour_next >= cachep->colour)
   l->colour_next = 0;
xend;
if (!(objp = kmem_getpages(cachep, flags,
               nodeid))) goto failed;
xbegin;
list_add_tail(&slabp->list,&(l->slabs_free));
xend;
```

Figure 1: A simplified example of fine-grained locking from the Linux function `cache_grow` in `mm/slab.c`, and its transactional equivalent.

Over 2000 static occurrences of spinlocks in Linux!

Transactional Memory in Practice

- In order to ensure isolation, TM systems must be able to roll back the effects of a transaction that has lost a conflict.
- TM can only roll back processor state and the contents of physical memory.
  - e.g., the effects of I/O cannot be rolled back!
  - Executing I/O operations as part of a transaction can break the atomicity and isolation.
  - This is known as the "output commit problem".
- Critical sections protected by locks will not restart and so may freely perform I/O.
- In practice, therefore, transactions have to run besides lock-based portions.
  - This must be supported by the system.

Simple SW Transactions: Sequence Locks

- Sequence locks (seqlocks) are a form of software transaction in the Linux kernel.
- Readers loop reading the seqlock counter at the start of the loop, performing any read of the data structure that they need, and then read the seqlock counter at the end of the loop.
- If the counter values match, the read loop exits.
- Writers lock each other out and they increment the counter both before they start updating the data and after they end.
- Readers fail if they read an odd counter value as it means a writer was doing an update concurrent with their reading.