Theory results in Transactional Memory

Panagiota Fatourou

FORTH ICS &
University of Crete

HTDC 2013
La Plagne, France, March 2013

Topics to be discussed

- Basics of TM
- Safety
- Liveness
- Universal Constructions & their relation to TM
- Lower Bounds
- TM Algorithms

PART 1: Basics of TM

Software Transactional Memory (STM)

- An alternative parallel programming paradigm
- Relieves naive programmer from
  - using locks
  - developing non-blocking algorithms

- **Key Idea**:
  - STM system is developed by **expert programmers**
  - Implementation details are hidden from the user
  - Parallel programming for average user becomes easier
    - parallel code resembles to its sequential analog, so it is easily understandable, verifiable

STM

- Supports execution of **transactions**
  - blocks of sequential code, each of which contain accesses to pieces of data, called data items
  - In a concurrent environment, the data items may be accessed concurrently by several processes and, therefore, synchronization is needed
- STM guarantees that operations of some transaction
  - will be atomically executed, if transaction **commits**
  - will never become apparent, if transaction **aborts**
- To ensure consistency, STMs employ **ownerships** on data items
  - a transaction must acquire the ownership of some data item before accessing it
- Each STMs maintains additional information (metadata)
  - for each transaction, e.g. its status (pending or completed)
  - for each data item, e.g. the id of the transaction holding its ownership

Useful Definitions

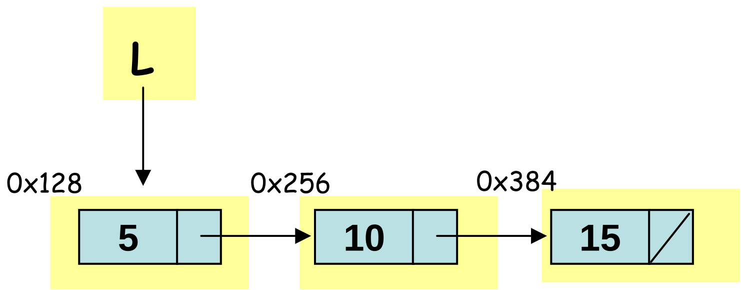
- A TM algorithm provides , for each process, an implementation for the following functions:
  - T.ReadDI(x), where x is a handle for a data item
  - T.WriteDI(x,v), where x is a handle for a data item and v is a value
  - T.BeginTrans()
  - T.CommitTrans()
  - T.CreateDI(size), T.AbortTrans()
- Each time a transaction T calls one of these routines we say that it invokes an operation. Each operation returns a response.

Example: An ordered linked list

Sequential Code

```
typedef struct node {
    int num;
    struct node *next;
} NODE;

NODE *L;
```



Transactional Code

```
typedef struct node {
    int num;
    TmVar next;
} NODE;

TmVar L;
```

Example: An ordered linked list

Sequential Code

```
Node *Insert (Node *L, int x) {
        NODE *p , *prevp = NULL, *newNode ;
        p = L;
        while ( p!=null && p->num < x) {
            prevp = p;
            p = p->next;
        }
        if (p!=null && p->num==x)
            return L;
        newNode = malloc(sizeof(Node));
        newNode->num = x;
        newNode->next = p;
        If (prevp != NULL)
                prevp->next = newNode;
        else return p;
```

Transactional Code

```
<TmVar,boolean> Insert (TmVar L, int x) {
    NODE *p, *prevp = NULL, *newNode;
    TmVar pTmVar, prevpTmVar = NULL, newNodeTmVar;
    boolean abort;

    while (1) {
        BeginTrans ();
        pTmVar = L;
        (abort, p) = ReadDI (t, pTmVar);
        while  (! abort && p != null && p->num < x) {
                prevpTmVar = pTmVar;
                prevp = p;
                pTmVar = p->next;
               (abort, p) = ReadDI (t, pTmVar);
        }
        if (abort) { AbortTrans ();  return <ABORT, L>; }
        if (p!=null && p->num==x)
            If (CommitTransaction()) return <COMMIT, L>;
            else return <ABORT, L>;
        …
    } // while } // Insert
```

Example: An ordered linked list

Sequential Code

```
boolean Insert (Node **L, int x) {
    NODE *p , *prevp, *newNode ;
    p = L;
    while ( P!=null && P->num < x)
       prevp = p;
       p = p->next;
    if (p!=null && p->num==x)
       return (FALSE);
    newNode = malloc(sizeof(Node));
    newNode->num = x;
    newNode->next = p;
    If (prevp != NULL)
         prevp->next = newNode;
    else return L = p;
    return L;
```

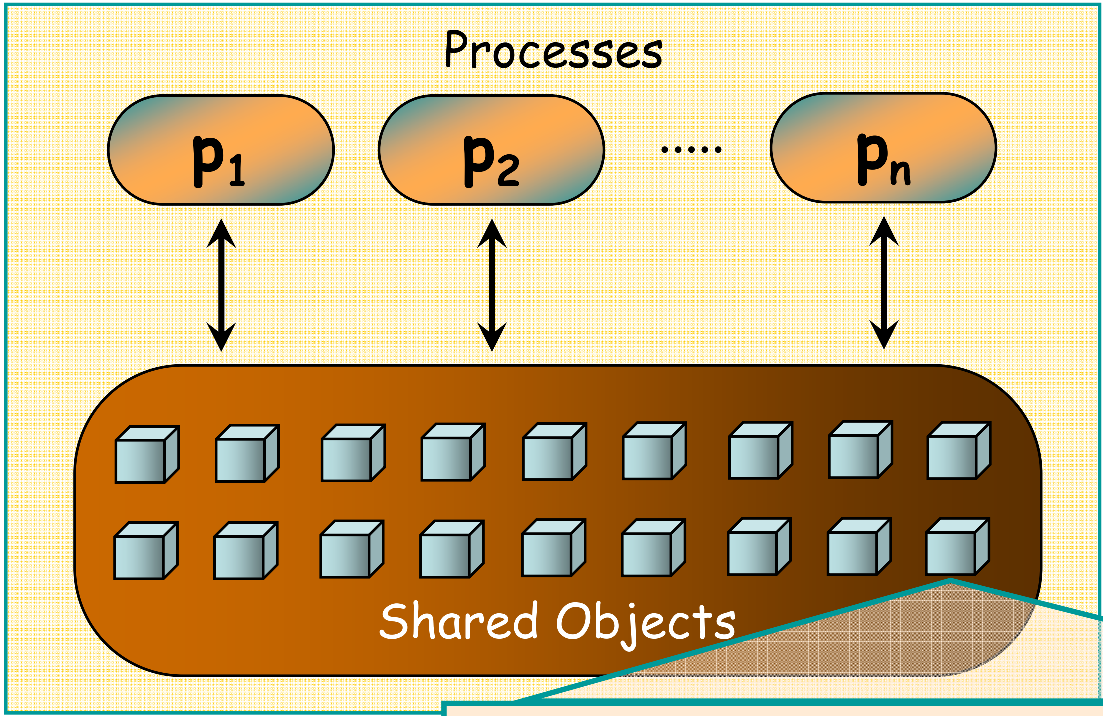
Transactional Code

```
void Insert (TmVar L, int x) {
    …
    newNode = (Node *)malloc (sizeof(Node));
    newNode->num = num;
    newNode->next = pTmVar;
    newNodeTmVar = CreateDI (sizeof(NODE));
    abort = WriteDI (newNodeTmVar, newNode);
    if (abort) return <ABORT,L>;
    If (prevp != NULL) {
        newprev.num = prevp->num;
        newprev.next = newNodeTmVar;
        abort = WriteDI (prevpTmVar, &newprev);
    }
    else abort = WriteDI (L, newNodeTmVar);
    if (abort) return <ABORT,L>;
    if (CommitTransaction()) return<COMMIT, L>;
} // Insert
```

PART 2: Safety

- strict serializability
- serializability
- opacity
- snapshot isolation
- other consistency conditions found in TM

Model



Register R

- supports **read(R)** and **write(R,v)**

LL/SC object R

- supports **LL(R)** and **SC(R,v)**

CAS object R

- supports **read(R)** and **CAS(R, v_{old}, v_{new})**

- the operations supported by base objects are called **primitives**
- processes may experience **crash failures**

Model

- A configuration is a vector consisting of the state of each process and the value of each base object.
- A step of a process consists of a primitive on a base object, the response to this primitive, and possibly some local computation by the process.
- An execution is a sequence of steps taken by the processes.

Useful Definitions

a,b,d: data items, initially 0

T_1 by p_1: W(a,1) W(b,1)

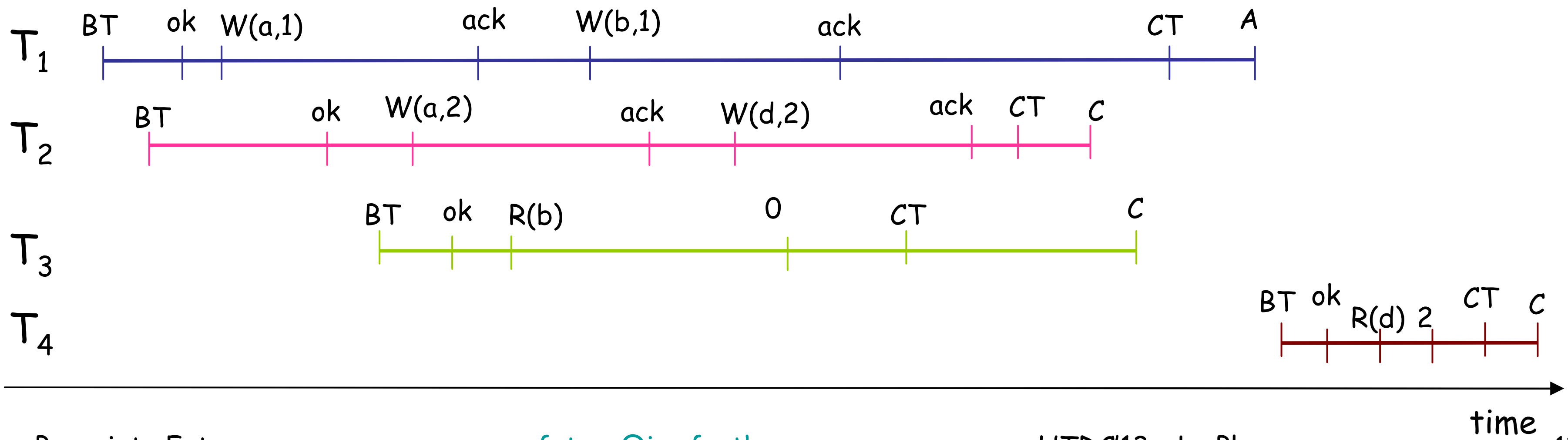
T_2 by p_2: W(a,2) W(d,2)

T_3 by p_3: R(b)

T_4 by p_4: R(d)

- A history H is a sequence of operation invocations and responses performed by transactions.

H: T_1.BeginTrans(), T_2.BeginTrans(), T_1.ok, T_1.W(a,1), T_2.ok, T_3.BeginTrans(), T_2.W(a,2), T_3.ok, T_1.ack, T_3.R(b), T_1.W(b,1), T_2.ack, T_2.W(d,2), T_3.<0>, T_1.ack, T_3.CommitTrans(), T_2.ack, T_2.CommitTrans(), T_2.C_2, T_3.C_3, T_1.CommitTrans(), T_1.A_1, T4.BeginTrans(), T_4.ok, T_4.R(d), T_4.<2>, T_4.CommitTrans, T_4.C_4



Useful Definitions

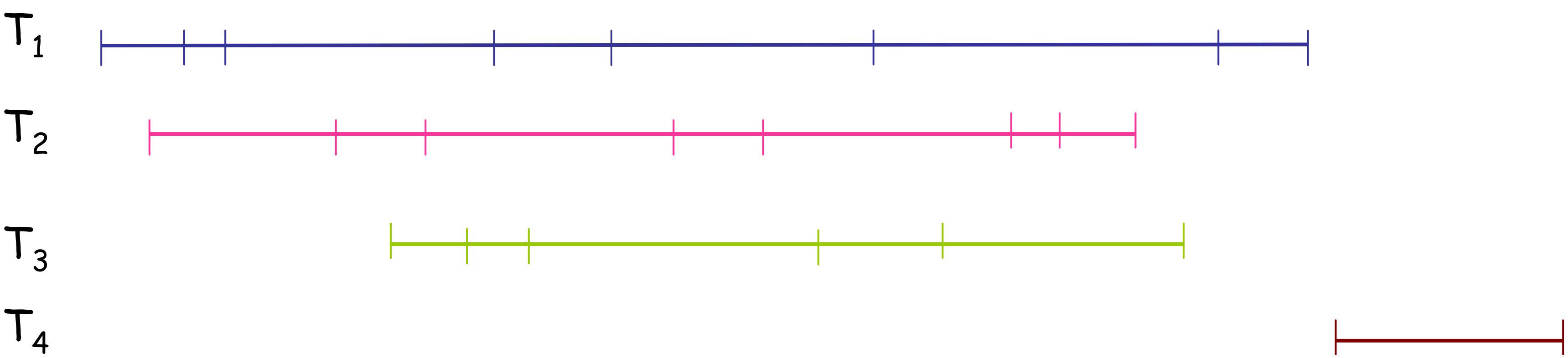
- H|T: longest subsequence of H consisting only of invocations and responses of T
  - H | T_1: T_1.BeginTrans(), T_1.ok, T_1.W(a,1), T_1.ack, T_1.W(b,1), T_1.ack, T_1.CommitTrans(), $T_1.A_1$,
  - H | T_4: T_4.BeginTrans(), T4.ok, T_4.R(d), T_4.<2>, T_4.CommitTrans, $T_4.C_4$
- T is live in H if H|T is not empty and does not end with T.A or T.C
- H|p: longest subsequence of H consisting only of invocations and responses of transactions executed by process p
- H is well-formed if ∀ T in H:
  - H|T is a sequence of invocations and responses starting with BeginTrans(), ok
  - every invocation of ReadDI is followed either by a value or by A_T
  - each invocation of WriteDI is followed either by an ack or A_T
  - each invocation of CommitTrans is followed by C_T or A_T
  - no invocation follows after C_T or A_T
- For each execution a of a TM algorithm, we denote by H_a the sequence of invocations and responses performed by the transactions executed in a.

Partial Orders induced by histories

- A history H induces an irreflexive partial order $<_H$ on transactions:
  - $T_1 <_H T_2$ if T_1.A or T_1.C appears in H before T_2.BeginTrans()
- Informally, $<_H$ captures the real time precedence ordering of transactions in H.
- Transactions unrelated by $<_H$ are said to be concurrent.
- If H is sequential, $<_H$ is a total order on transactions.

Example

$T_1 <_H T_4$, $T_2 <_H T_4$, $T_3 <_H T_4$
T1, T2 and T3 are concurrent



Useful Definitions

- A history is sequential if no two transactions are concurrent in it.
  - S: T_2.BeginTrans(), T_2.ok, T_2.W(a,2), T_2.ack, T_2.W(d,2), T_2.ack, T_2.CommitTrans(), $T_2.C_2$, T_3.BeginTrans(), T_3.ok, T_3.R(b), T_3.<0>, T_3.CommitTrans(), $T_3.C_3$, T_1.BeginTrans(), T_1.ok, T_1.W(a,1), T_1.ack, T_1.W(b,1), T_1.ack, T_1.CommitTrans(), $T_1.A_1$, T_4.BeginTrans(), T4.ok, T_4.R(d), T_4.<2>, T_4.CommitTrans(), $T_4.C_4$
- Histories H and H' are equivalent if they contain the same transactions and for every transaction T in H, H|T = H'|T.
  - H and S are equivalent.

Useful Definitions

- comp(H): set of all histories which can be derived from H by:
  - appending to H an A_T response for each transaction T which has invoked an operation other than CommitTrans and has not received a response for it
  - appending either a C_T or an A_T response for every transaction T that has invoked CommitTrans without having received a response for it
  - appending an invocation of an AbortTrans() and the response A_T for each other transaction T.
- A sequential history S is legal if its restriction to committed and live transactions (i.e., comm(S)) respects the sequential specification of the data items accessed in comm(S).

Strict Serializability

Papadimitirou, J. ACM, 1979

We say that an execution α is strictly serializable if it is possible to do all of the following:

- For each committed transaction T, to insert a serialization point $*_T$ somewhere between T's first invocation and T's last response in α.
- To choose a subset A of the live transactions in α and, for each transaction $T \in A$, insert a serialization point $*_T$ somewhere after T's first invocation.
- These serialization points should be inserted, so that, in the sequential execution σ constructed by serially executing each transaction T at the point that its serialization point has been inserted:
  - if $T \notin A$, the same operations, as in α, are invoked by T and the response of each such operation is the same as that in α, and
  - if $T \in A$, a prefix of the operations invoked by T in σ are the same as all operations invoked by T in α and the response of each such operation is the same as that in α.

Strict Serializability

- A history H is final-state strictly serializable if there exists a history $H' \in comp(H)$ s.t.:
  (1) comm(H') is equivalent to some legal sequential history S,
  (2) $<_{comm(H')} \subseteq <_{comm(S)}$

- A history H is strictly serializable if each of its finite prefixes is final-state strictly serializable.

- The above definition of strict serializability is not equivalent to that provided in the previous slide.

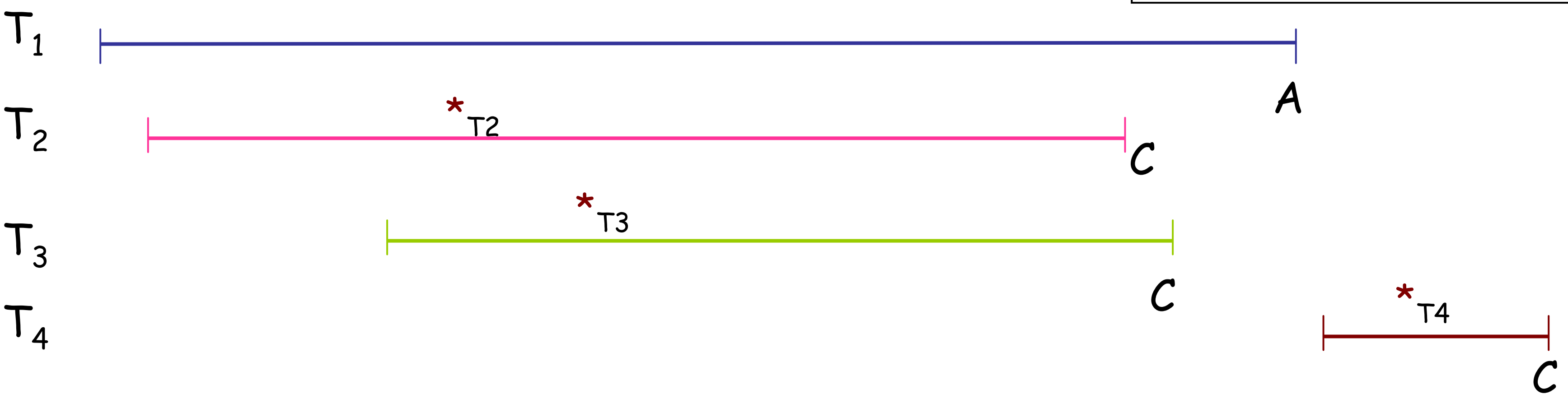
Strict Serializability - Example

T_1 by p_1: W(a,1) W(b,1)

T_2 by p_2: W(a,2) W(d,2)

T_3 by p_3: R(b) -> 0

T_4 by p_4: R(d) -> 2



H: T1.BeginTrans(), T2.BeginTrans(), T1.ok, T1.W(a,1), T2.ok, T3.BeginTrans(), T2.W(a,2), T3.ok, T1.ack, T3.R(b), T1.W(b,1), T2.ack, T2.W(d,2), T3.<0>, T1.ack, T3.CommitTrans(), T2.ack, T2.CommitTrans(), T2.C2, T3.C3, T1.CommitTrans(), T1.A1, T4.BeginTrans(), T4.ok, T4.R(d), T4.<2>, T4.CommitTrans, T4.C4

Restriction of H to committed transactions: T2.BeginTrans(), T2.ok, T3.BeginTrans(), T2.W(a,2), T3.ok, T3.R(b), T2.ack, T2.W(d,2), T3.<0>, T3.CommitTrans(), T2.ack, T2.CommitTrans(), T2.C2, T3.C3, T4.BeginTrans(), T4.ok, T4.R(d), T4.<2>, T4.CommitTrans, T4.C4

S: T2.BeginTrans(), T2.ok, T2.W(a,2), T2.ack, T2.W(d,2), T2.ack, T2.CommitTrans(), T2.C2, T3.BeginTrans(), T3.ok, T3.R(b), T3.<0>, T3.CommitTrans(), T3.C3, T4.BeginTrans(), T4.ok, T4.R(d), T4.<2>, T4.CommitTrans(), T4.C4

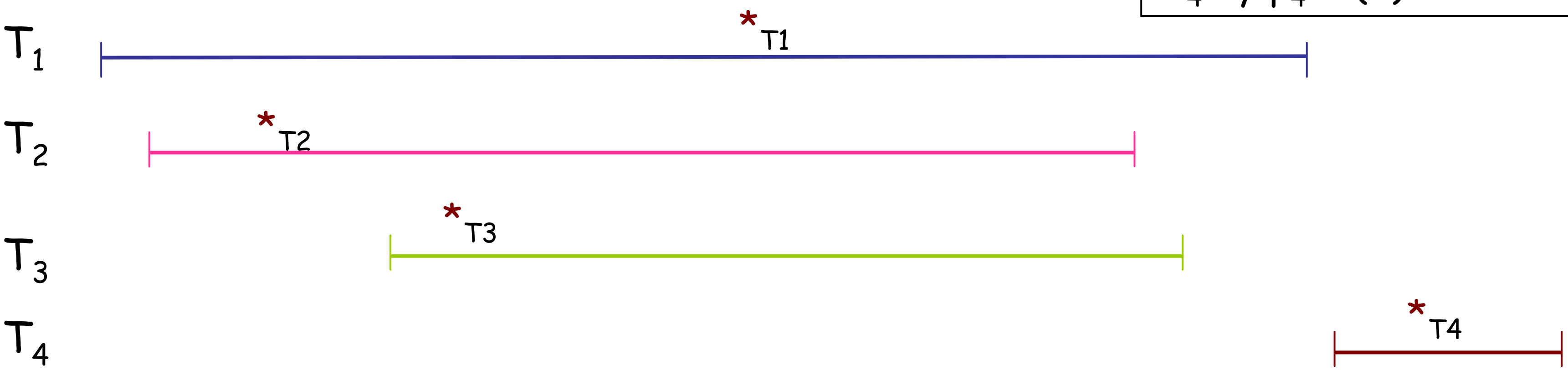
What if T1 also commits?

T_1 by p_1: W(a,1) W(b,1)

T_2 by p_2: W(a,2) W(d,2)

T_3 by p_3: R(b) -> 0

T_4 by p_4: R(d) -> 2



H: T1.BeginTrans(), T2.BeginTrans(), T1.ok, T1.W(a,1), T2.ok, T3.BeginTrans(), T2.W(a,2), T3.ok, T1.ack, T3.R(b), T1.W(b,1), T2.ack, T2.W(d,2), T3.<0>, T1.ack, T3.CommitTrans(), T2.ack, T2.CommitTrans(), T2.C2, T3.C3, T1.CommitTrans(), T1.C, T4.BeginTrans(), T4.ok, T4.R(d), T4.<2>, T4.CommitTrans, T4.C4

S: T2.BeginTrans(), T2.ok, T2.W(a,2), T2.ack, T2.W(d,2), T2.ack, T2.CommitTrans(), T2.C2, T3.BeginTrans(), T3.ok, T3.R(b), T3.<0>, T3.CommitTrans(), T3.C3, T1.BeginTrans(), T1.ok, T1.W(a,1), T1.ack, T1.W(b,1), T1.ack, T1.CommitTrans(), T1.C, T4.BeginTrans(), T4.ok, T4.R(d), T4.<2>, T4.CommitTrans(), T4.C4

Serializability

Papadimitirou, J. ACM, 1979

- We say that an execution α is serializable if it is possible to do all of the following:
  - For each committed transaction T, to insert a serialization point $*_T$ ~~somewhere between T's first invocation and T's last response in α.~~
  - To choose a subset A of the live transactions in α and, for each transaction $T \in A$, insert a serialization point $*_T$ ~~somewhere after T's first invocation.~~
  - These serialization points should be inserted, so that, in the sequential execution σ constructed by serially executing each transaction T at the point that its serialization point has been inserted:
    - if $T \notin A$ the same operations, as in α, are invoked by T and the response of each such operation is the same as that in α.
    - if $T \in A$, a prefix of the operations invoked by T in σ are the same as all operations invoked by T in α and the response of each such operation is the same as that in α.

Serializability

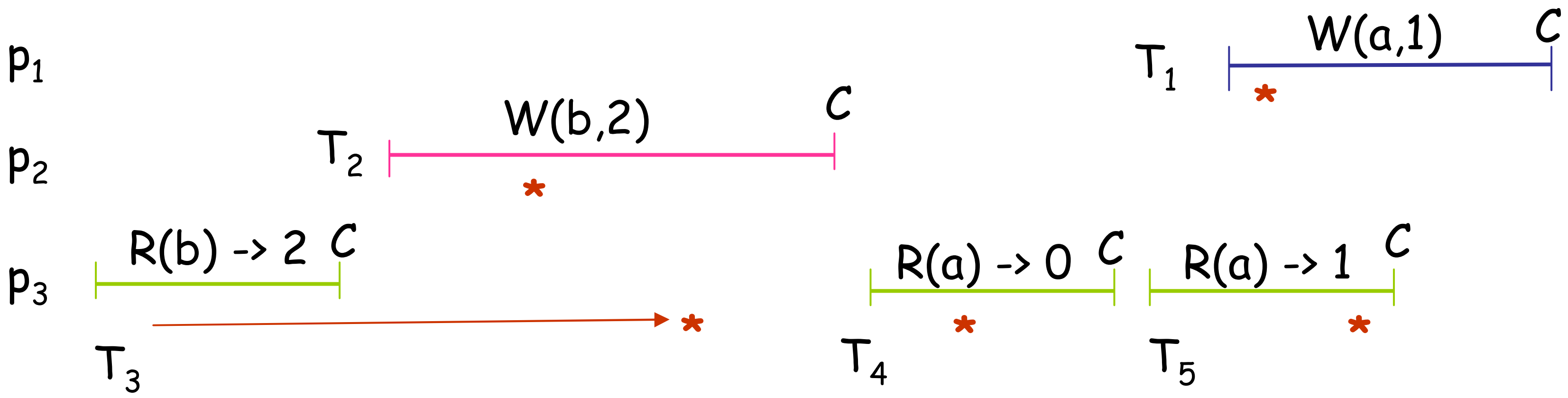
- A history H is final-state serializable if there exists a history H' $\in$ comp(H) s.t.:
  (1) comm(H') is equivalent to some legal sequential history S,
  (2) $\overleftarrow{\text{comm}(H')} \subseteq \overleftrightarrow{\text{comm}(S)}$ for each process p, comm(H') | p = S|p

- A history H is serializable if each of its finite prefixes is final-state serializable.

- The above definition of serializability is not equivalent to that provided in the previous slide.

Serializability - Example

T_1 by p_1: W(a,1)
T_2 by p_2: W(b,2)
T_3 by p_3: R(b) -> 2
T_4 by p_3: R(a) -> 0
T_5 by p_3: R(a) -> 1



The above execution is not strictly serializable.

However, it is serializable.

Strict serializability is a stronger condition than serializability:

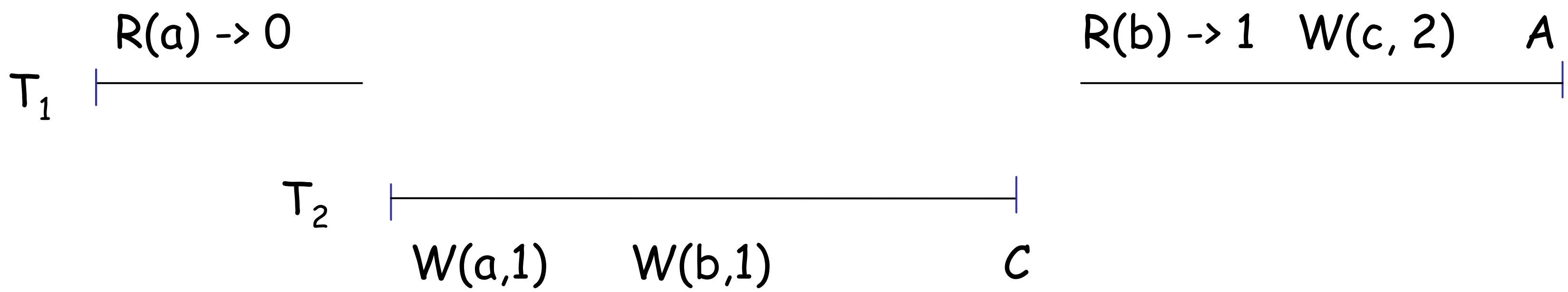
Any strictly serializable execution is serializable.
The opposite is not TRUE!

Opacity

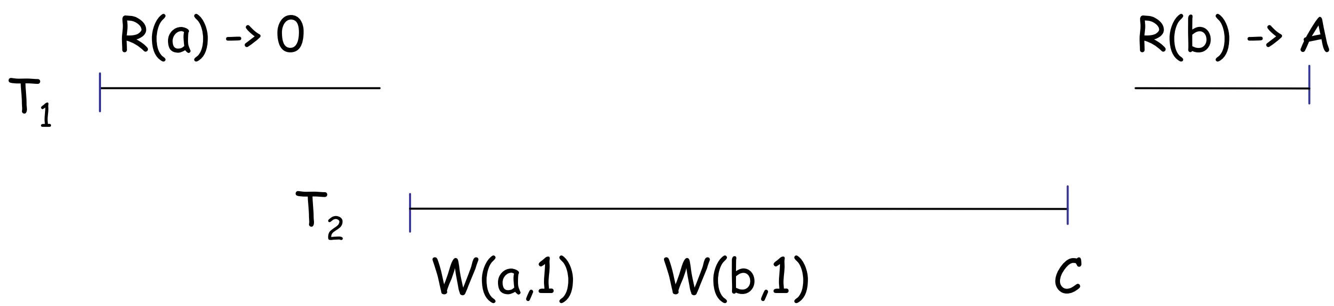
Guerraoui & Kapalka, PPoPP 2008

- In addition to ensuring conditions on the behavior of committed (or ready to commit) transactions, opacity places conditions on the behavior of non-committed transactions as well:
  - non-committed transactions should not behave inappropriately, i.e., their execution should not result in:
    - entering infinite loops, or
    - abnormal termination (e.g. segmentation fault)
- A history H is final-state opaque if there exists a history $H' \in comp(H)$ s.t.:
  - H' is equivalent to some legal sequential history S,
  - $<_{H'} \subseteq <_S$
  - for every transaction $T \in H'$, the longest subsequence of S made of (1) all transactions preceding T in S, and (2) every prefix of S|T s.t. T is live in S|T, is a legal history.
- A history H is opaque if each of its finite prefixes is final-state opaque.

Opacity - Example



This execution is not opaque.



This execution is opaque.

Snapshot Isolation (SI)

Lu et al., IEEE Trans. on Knowledge & Data Eng. (DB), Riegel et al., Transact 2006 (TM)

- Snapshot isolation requires that transactions should be executed as if every readDI operation reads from some snapshot of the memory that was taken when the transaction started.
- Snapshot isolation is appealing for TM:
  - it provides the potential to increase throughput for workloads with long transactions

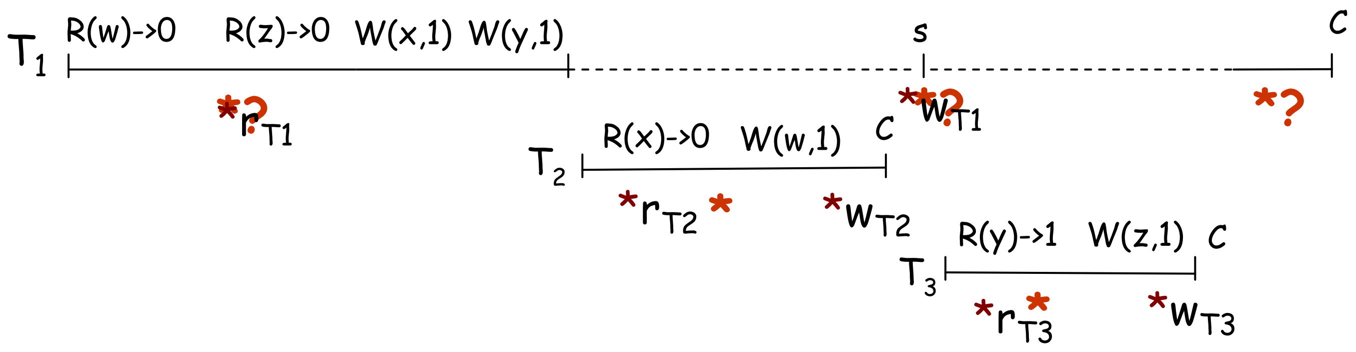
Snapshot Isolation

- Let H be a history and T be a transaction in H.
- T| read: longest subsequence of T consisting only of read invocations
- T| write: longest subsequence of T consisting only of write invocations
- if T is commiting (or live):
  - T_r = T|read, CommitTrans(T_r), C_{Tr} if T|read $\neq \Lambda$, and Λ otherwise
  - T_w = T|write, CommitTrans(T_w), C_{Tw} if T|write $\neq \Lambda$, and Λ otherwise
- If T is aborting:
  - T_r = T|read if T|read contains A_{Tr} or if T|read = Λ, and
    T_r = T|read CommitTrans(T_r) A_{Tr} otherwise
  - $T_w = \Lambda$

Snapshot Isolation (roughly speaking)

- An execution a satisfies snapshot isolation, if for every completed transaction T in a (and for some of the live transactions) it is possible to insert a read serialization point $*_{T,r}$ and a write serialization point $*_{T,w}$ s.t:
  - $*_{T,r}$ precedes $*_{T,w}$
  - both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of T,
  - if σ_α is the sequence defined by these serialization points, in order, and $H_{\sigma\alpha}$ is the history we get by replacing each $*_{T,r}$ with T_r and each $*_{T,w}$ with T_w in σ_α, then $H_{\sigma\alpha}$ is legal.

Snapshot Isolation - Example



x,y,w,z: data items, initially 0

T_1 by p_1: R(w) R(z) W(x,1) W(y,1)

T_2 by p_2: R(x) W(w,1)

T_3 by p_3: R(y) W(z,1)

This execution is not strictly serializable.

However, it ensures snapshot isolation.

Other Consistency Conditions in TM

- **Causal Consistency** (Imbs et al., INRIA TR, 2008)
- Virtual World Consistency: Weaker version of opacity where for non-committed transactions only causal consistency is ensured (Imbs et al., INRIA TR, 2008)
- **TMS1 and TMS2:** variants of opacity (Doherty et. al, WTTM 2012)
- **Z-linearizability:** transactions are partitioned into two sets, long and short transactions, and different consistency conditions are ensured for each set. (Riegel et. al, Universite de Neuchatel TR, 2007)
- Many consistency conditions have previously appeared in the DB literature and probably make sense for TM computing as well: (Attiya et. al, Transact 2012)
  - Recoverability, Strictness, Rigorousness, Avoiding Cascading aborts
  - I am sure that there are more…
- etc.

Privatization

- strong atomicity/isolation
- weak atomicity/isolation
- single lock atomicity
- disjoint lock atomicity
- asymmetric flow ordering
- encounter time lock atomicity
- selective strict serializability
- internal consistency
- race-free atomicity

PART 2: Liveness

Conventional Progress Conditions

Blocking algorithms

- An arbitrary and unexpected delay by a process may prevent all other processes from making progress.
- e.g., a process that holds a lock
- Deadlock-freedom: no deadlock occurs
  - the system as a whole makes progress but progress to individual processes is not guaranteed
- Starvation-freedom: no starvation occurs
  - Every process eventually makes progress (e.g., it acquires the lock).

➡ Blocking algorithms do not ensure progress if processes crash

Conventional Progress Conditions

Non-blocking algorithms

- An arbitrary and unexpected delay by any process does not prevent other processes from making progress.
- Obstruction-freedom
  - Every process makes progress if it executes solo for long enough (starting from any reachable configuration).
- Lock-freedom
  - Progress is ensured for some of the active processes
- Wait-freedom
  - Progress is ensured for each active process

➡ Non-blocking algorithms ensure progress even if processes crash

TM Progress

✓ A TM progress property should additionally ensure that some transactions commit

- Local Progress*
  - Each process that keeps executing a transaction (e.g. restarts it in case it aborts) eventually commits it.
  - Similar to wait-freedom
- Global Progress*
  - Some process that keeps executing a transaction eventually commits it.
  - Similar to lock-freedom
- Solo Progress*
  - Each process which eventually runs solo while it keeps executing a transaction eventually commits it.

*V. Bushkov, R. Guerraoui, and M. Kapalka, "On the Liveness of Transactional Memory", PODC 2012

- Obstruction-freedom#
  - A transaction that does not encounter step contention during the course of its execution must commit.

R. Guerraoui and M. Kapalka, "On Obstruction-free Transactions", SPAA 2008.

TM Progress

- Permissiveness: a transaction must commit unless doing so violates correctness
  - R. Guerraoui, T. Henzinger, and V. Singh, "Permissiveness in TM", DISC 2008.
- Progressiveness: a transaction must commit if it doesn't conflict with any other transaction
  - M. Scott, "Sequential Specification of Transactional Memory Semantics", Transact 2006.
- Strong Progressiveness: progressiveness + if a number of transactions conflict on a single t-var, then at least one of them must be able to commit.
  - R. Guerraoui and M. Kapalka, "The Semantics of Progress in Lock-Based Transactional Memory", POPL 2009.

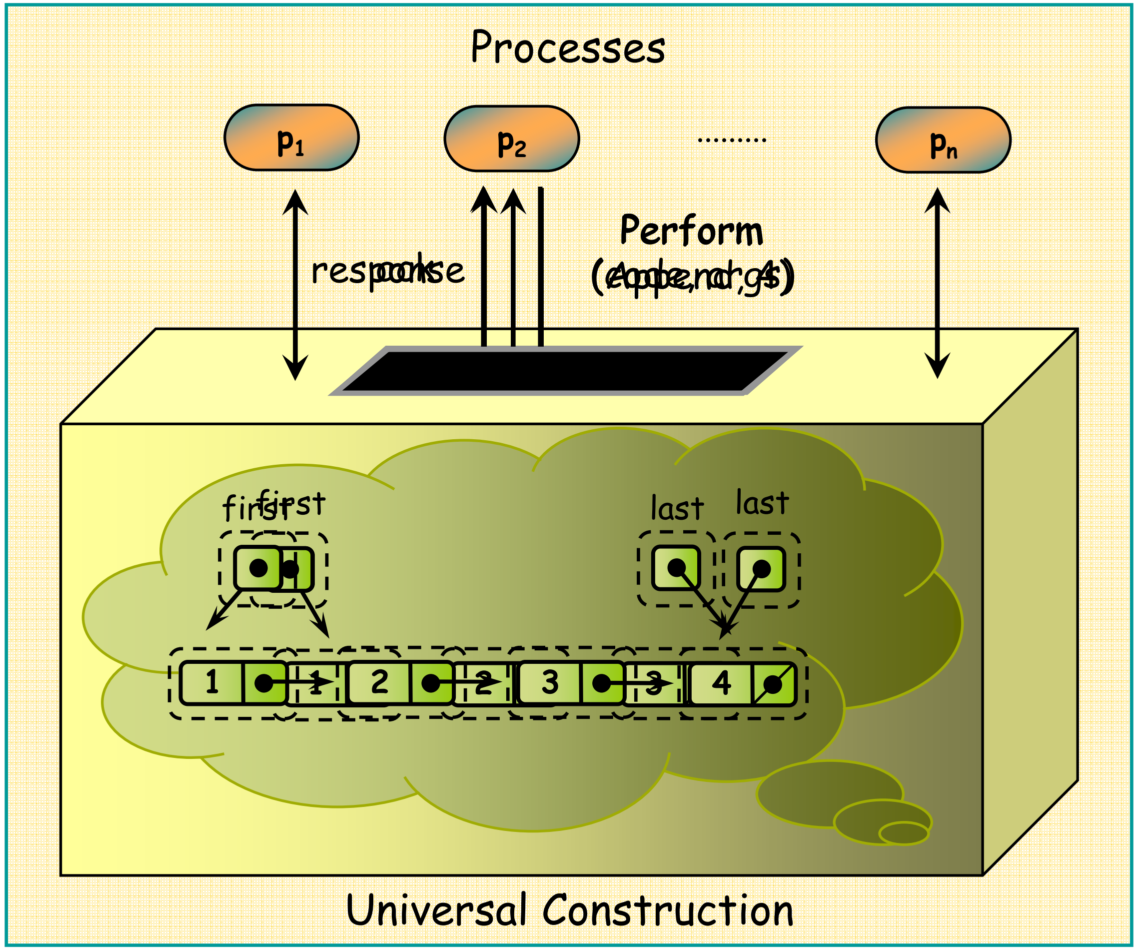
TM Progress

- Read-only transactions never abort
  - H. Attiya and E. Hillel, "Single-version STMs can be multiversion permissive", ICDCN 2011.
  - D. Pelerman, R. Fan, and I. Keidar, "On maintaining multiple versions in STM", PODC 2012.
    - update transactions may abort and they require locks to execute some of the transactional instructions
- No transaction ever aborts
  - Y. Afek, A. Matveev, and N. Shavit, "Pessimistic Software Lock-Elision", DISC 2012.
  - A. Matveev and N. Shavit, "Towards a fully pessimistic STM model", Transact 2012.
    - read-only transactions are wait-free
    - write-only transactions are blocking and they are executing one-after-the-other using a global lock
  - S. Dolev, P. Fatourou, and E. Kosmas, "Abort-Free SemanticTM by Dependency Aware Scheduling of Transactional Instructions", Transact 2013.
    - no conflict ever occurs, so no transaction ever aborts
    - parallelism is fine-grained; it is achieved at the level of transactional instructions instead of transactions themselves

Universal Constructions

- Provides a general mechanism to **automatically execute** pieces of **sequential code** in a **concurrent environment**

- For each process p, an operation **Perform** is supported
- takes as **parameters**
  - piece of sequential code
  - input arguments
- **applies** code to the simulated state
- **returns** a response to p



Universal constructions and TM algorithms are closely related

Same Goal

Simplify parallel programming by providing mechanisms to efficiently execute code in a concurrent environment.

Main Differences

1. Support of "Abort"
   - A TM algorithm informs the external environment when a transaction is aborted.
   - Universal constructions, thus far, did not have the facility to abort the execution of the piece of code that was passed as a parameter.
2. Access to transaction's code
   - A TM system invokes actions for reading or writing a data item, initiating, committing or aborting a transaction.
     - In this talk we will call this model tcode-unaware
   - A universal construction requires that the transaction's code is placed in a routine and a pointer to this routine is passed as an argument to Perform.
     - in many cases, e.g. in data structures, this is indeed the case
     - a lot of TM implementations (DSTM, TL II) assume that a pointer to a function containing the transaction's code is passed as an argument to the implementation.
     - We will call this model tcode-aware

➡ Programming in the tcode-unaware model is probably easier in some cases

Results

- There does not exist a tcode-unawareTM implementation that ensures both local progress and opacity. (Bushkov et al, PODC 2012)
- There exist several universal constructions that ensure wait-freedom and can serve as wait-free tcode-aware TM systems
  - Afek et al., STOC 1995
  - Anderson & Moir, PODC 1995
  - Chuong et al., SPAA 2010
  - Fatourou et al., DISC 2009, SPAA 2011
  - Herlihy, PPoPP 1990, TOPLAS 1991

➡ It is easy to ensure that no transaction ever aborts by allowing waiting or helping.

Results

Opacity + progressiveness Imbs et al., OPODIS 2008	Possible
Opacity + permissiveness + invisible reads Crain et al., IRISA TR, 2010	Impossible
Opacity + permissiveness Guerraoui et al., DISC 2008	NP-hard
Opacity + probabilistic permissiveness + lock freedom Guerraoui et al., DISC 2008	Possible
Virtual world consistency + probabilistic permissiveness + invisible reads Crain et al., IRISA TR, 2010	Possible

PART 3: Impossibility Results & Lower Bounds

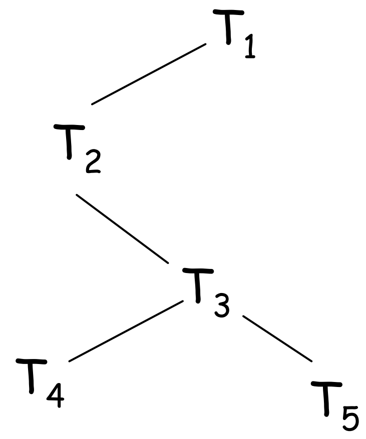
Useful Definitions

- An execution a is legal starting from a configuration C if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in C), and for each object, the responses to the operations performed on the object are in accordance with its specification (and the value stored in the object at C).
- An execution a is indistinguishable from another execution a' for some processes, if each of these processes take the same steps in a and a', and each of these steps has the same response in a and a'.

Parallelizability

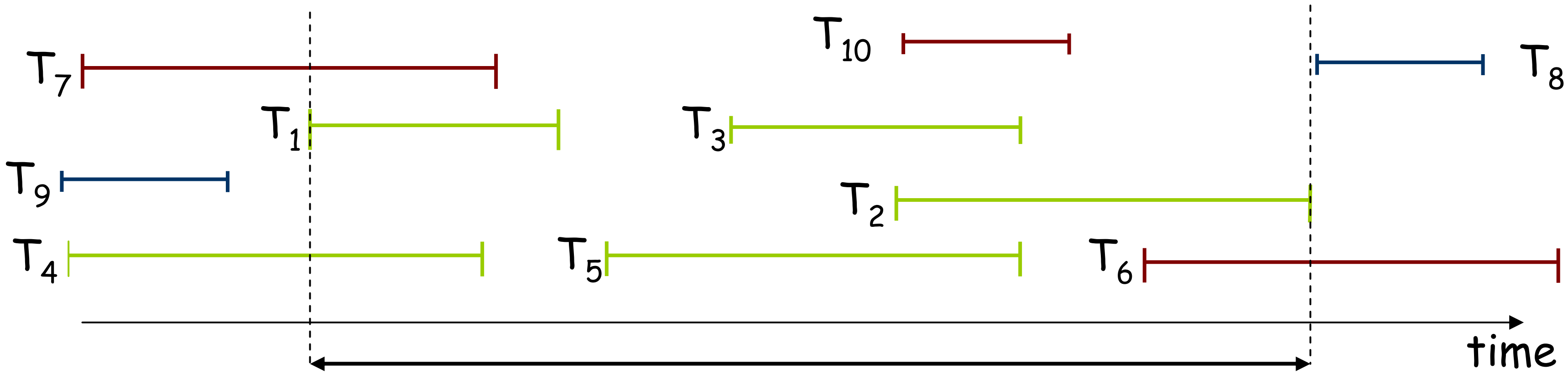
- We say that two transactions conflict in an execution α, if they both invoke an operation on a common data item
- We say that two executions contend on a base object o if they both contain a primitive on o and one of these primitives is non-trivial (i.e. it may update o).
- We say that a TM implementation is strictly disjoint-access parallel (DAP) if, in each execution α, and for every two transactions T1 and T2 in α, if α|T1 and α|T2 contend on some base object, then T1 conflicts with T2 in α.

Parallelizability



Conflict Graph of an execution interval

- vertices represent transactions whose execution overlaps with the interval
- edges connect transactions that conflict
- A TM implementation I is disjoint-access-parallel (dap) if, for every execution a of I which contains two transactions T1 and T2, $a|T_1$ and $a|T_2$ contend on some base object, only if there is a path between T_1 and T_2 the conflict graph of the minimal execution interval a containing T_1 and T_2.



Impossibility Results

1. No TM implementation ensures obstruction-freedom, strict DAPism and strict serializability

   Guerraoui & Kapalka, "On obstruction-free Transactions", SPAA 2008

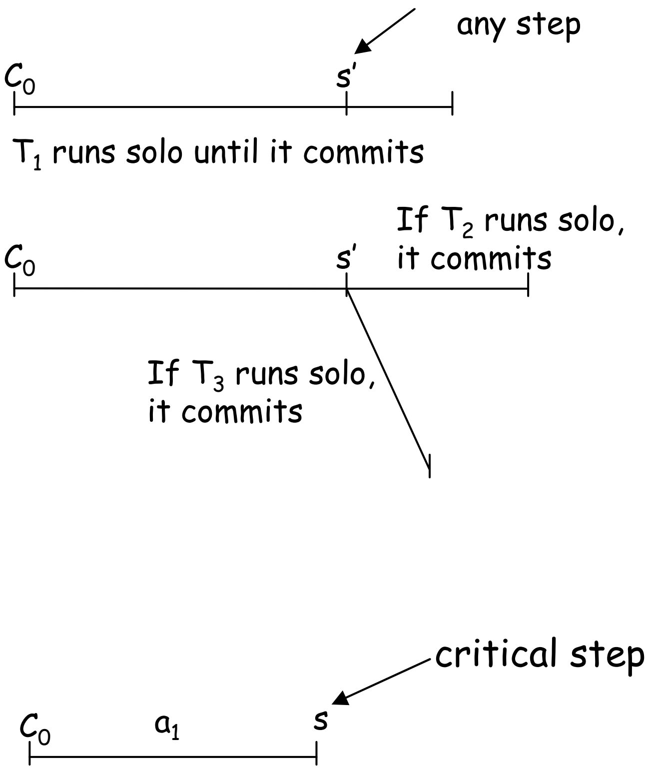
2. No TM implementation ensures obstruction-freedom, strict DAPism and snapshot isolation.

   Busjhkov, Dziuma, Fatourou, & Guerraoui, "Snapshot Isolation does not Scale Either", 2013.

3. No TM implementation (where each aborted transaction is restarted) ensures DAPism and wait-freedom.

   Ellen, Fatourou, Kosmas, Milani, & Travers, "Universal Constructions that Ensure Disjoint-Access-Parallelism and Wait-freedom", PODC 2012.

Obstruction-freedom, Strict serializability and Strict DAPism: Impossible



x,y,w,z: data items, initially 0

T_1 by p_1: R(w) R(z) W(x,1) W(y,1)

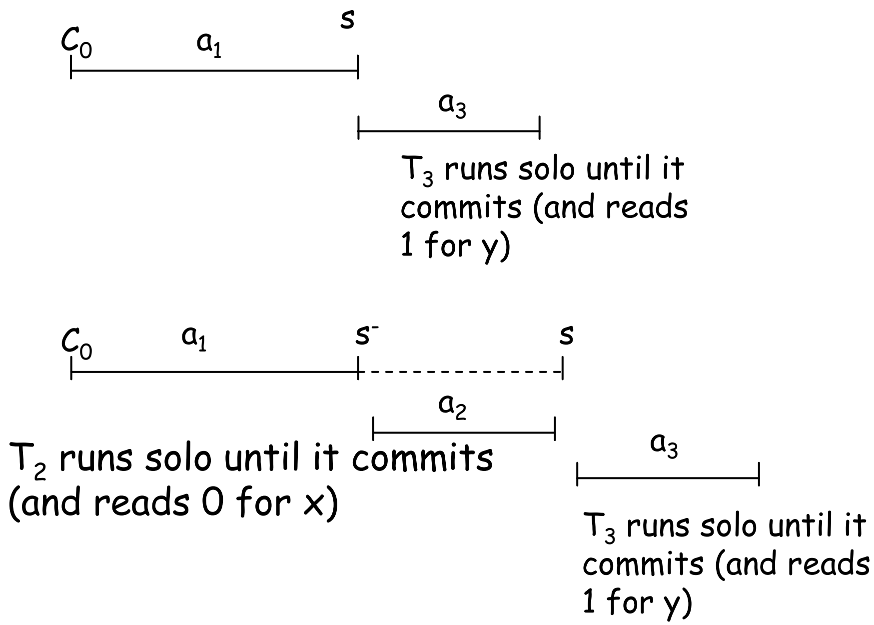
T_2 by p_2: R(x) W(w,1)

T_3 by p_3: R(y) W(z,1)

Critical step

- If each of T_2, T_3 runs solo starting from the configuration before s, its read returns 0
- If each of T_2, T_3 runs solo starting from the configuration after s, the read of at least one of T_2 or T_3 returns 1; wlog, assume that it is T_3 that reads the value 1 for y.

Obstruction-freedom, Strict serializability & Strict DAPism: Impossible



T_1 by p_1: R(w) R(z) W(x,1) W(y,1)
T_2 by p_2: R(x) W(w,1)
T_3 by p_3: R(y) W(z,1)

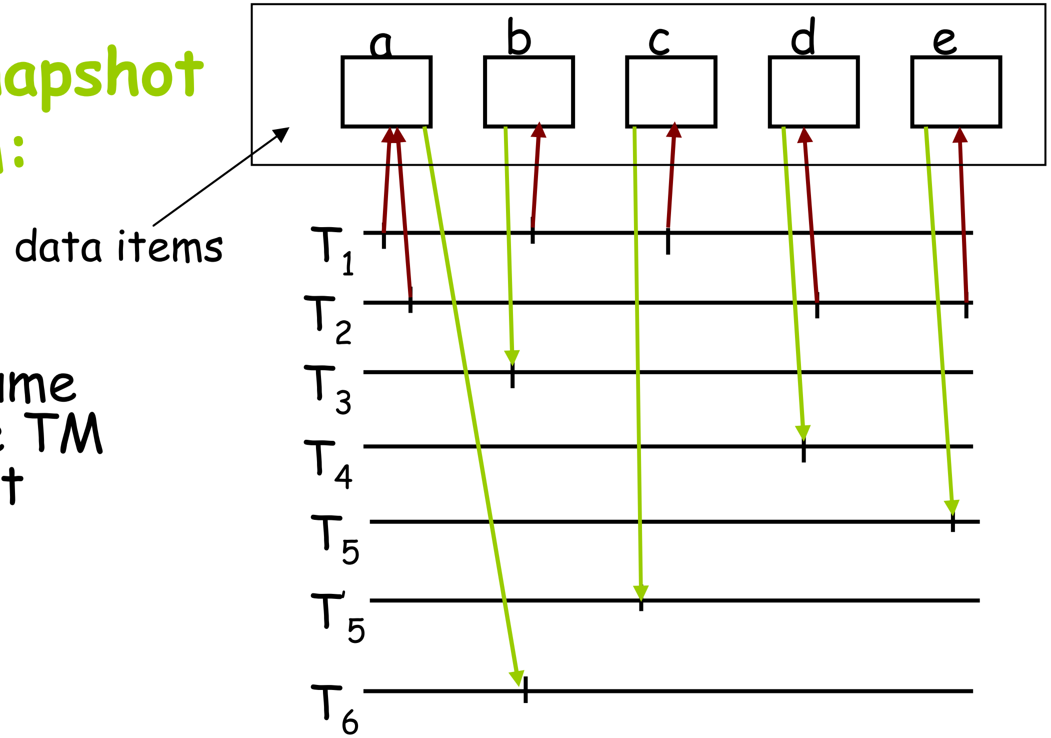
- sa_3 is legal after a_1a_2
  - s is a non-trivial step on a base object o read in a_3
  - a_2 and a_3 do not conflict, so they do not contend on o or on any other base object (otherwise strict dapism would be violated)
  - a_2 does not perform any non-trivial operation on o or any other base object read in a_3
    - s is legal after a_1a_2
    - a_3 is legal after a_1a_2s

This execution violates strict serializability:

- Since T_2 commits, T_1 must abort.
- Since T_3 commits, T1 must commit.

A contradiction!

Obstruction-freedom, Snapshot Isolation & strict DAPism: Impossible



Main Ideas

- Proof by contradiction. Assume there is an obstruction-free TM alg A which ensures snapshot isolation and strict DAPism.
- We will construct two executions:

  $\delta_1 = \alpha_1\ \alpha_2\ s_1\ s_2\ \gamma_6$, and

  $\delta_2 = \alpha_1\ \alpha_2\ s_2\ s_1\ \gamma'_6$, s.t.:
  - γ_6 and γ'_6 are solo executions of T_6
  - in γ_6, T_6 reads the value 2 for a
  - in γ'_6, T_6 reads the value 1 for a
  - γ_6 and γ'_6 are indistinguishable.
- Thus, T_6 should read the same value for a in both executions. A contradiction!

T_1 by p_1: W(a,1) W(b,1) W(c,1)

T_2 by p_2: W(a,2) W(d,2) W(e,2)

T_3 by p_3: R(b)

T_4 by p_4: R(d)

T_5 by p_5: R(e)

T_5 by p_5: R(c)

T_6 by p_6: R(a)

Definition of α_1, α_2, s_1 and s_2

T_1 by p_1: W(a,1) W(b,1) W(c,1)

T_2 by p_2: W(a,2) W(d,2) W(e,2)

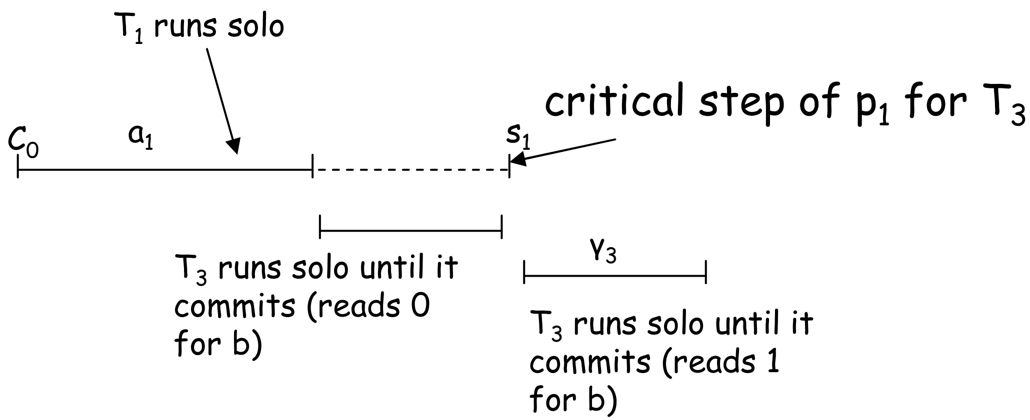
T_3 by p_3: R(b)

T_4 by p_4: R(d)

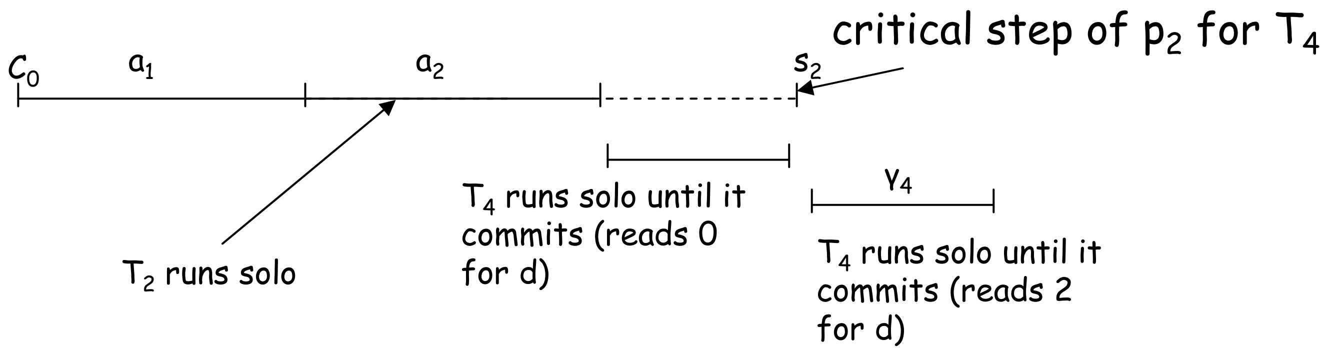
T_5 by p_5: R(e)

T'_5 by p_5: R(c)

T_6 by p_6: R(a)

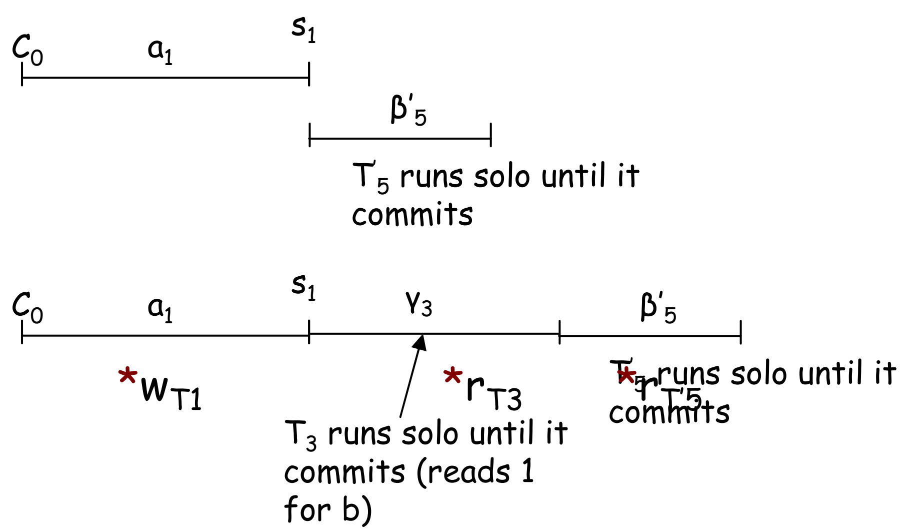


1. s_1 applies a non-trivial operation on some base object o_1 s.t T_3 reads o_1 in γ_3



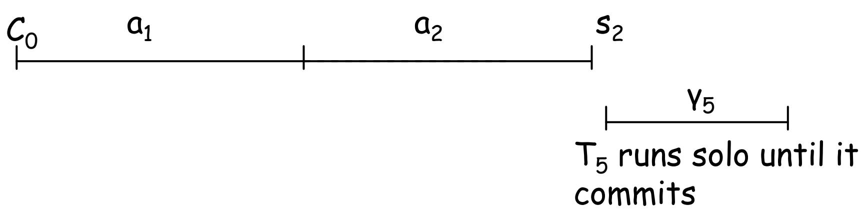
2. s_2 applies a non-trivial operation on some base object o_2 s.t T_4 reads o_2 in γ_4

Useful Claims



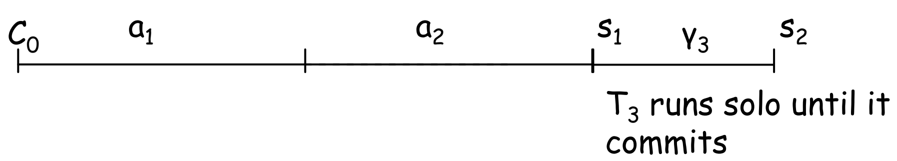
T1 by p1: W(a,1) W(b,1) W(c,1)
T2 by p2: W(a,2) W(d,2) W(e,2)
T3 by p3: R(b)
T4 by p4: R(d)
T5 by p5: R(e)
T'5 by p5: R(c)
T6 by p6: R(a)

Claim 1: T'_5 reads 1 for c in β'_5.



Claim 2: T_5 reads 2 for e in γ_5.

- Proved in a similar way as Claim 1



Claim 3: $o_1 \neq o_2$.

- γ_3 is legal after a_1 a_2 s_1
- γ_3 reads o_1
- γ_3 does not read any base object written by a_2 s_2

What is our goal?

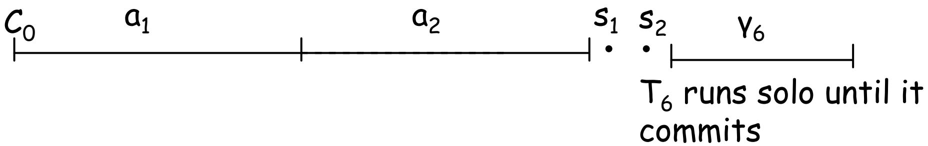
Goal

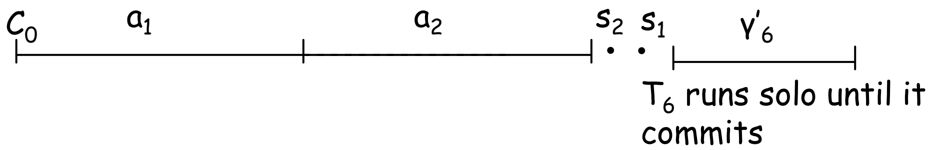
- To construct two executions:

  $\delta_1 = \alpha_1\ \alpha_2\ s_1\ s_2\ \gamma_6$, and

  $\delta_2 = \alpha_1\ \alpha_2\ s_2\ s_1\ \gamma'_6$, s.t.:

  - γ_6 and γ'_6 are solo executions of T_6
  - in γ_6, T_6 reads the value 2 for a
  - in γ'_6, T_6 reads the value 1 for a
  - γ_6 and γ'_6 are indistinguishable.





Lemma 1: γ_6 and γ'_6 are indistinguishable

T_1 by p_1: W(a,1) W(b,1) W(c,1)

T_2 by p_2: W(a,2) W(d,2) W(e,2)

T_3 by p_3: R(b)

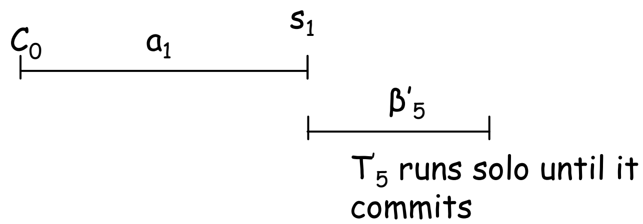
T_4 by p_4: R(d)

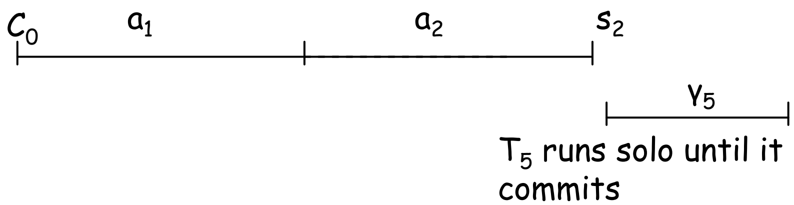
T_5 by p_5: R(e)

T_5 by p_5: R(c)

T_6 by p_6: R(a)

Claim 1: T'_5 reads 1 for c in β'_5.
Claim 2: T_5 reads 2 for e in γ_5.
Claim 3: $o_1 \neq o_2$.





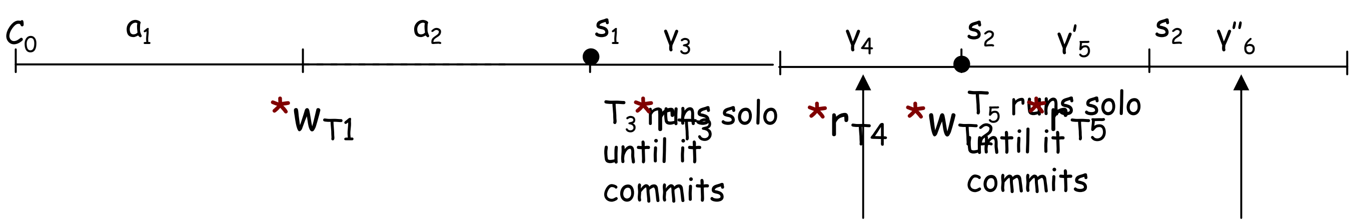
T_6 reads 2 for a in γ_6

Recall that (1) T_3 reads 1 for b in γ_3, and (2) T_3 reads o_1

Claim 4: T_4 reads 0 for d in γ_4

- γ'_4: solo execution of T_4 after $a_1\ a_2$
- γ_4 and γ'_4 are indistinguishable

T_1 by p_1: W(a,1) W(b,1) W(c,1)
T_2 by p_2: W(a,2) W(d,2) W(e,2)
T_3 by p_3: R(b) -> 1
T_4 by p_4: R(d) -> 0
T_5 by p_5: R(e) -> 2
T_5 by p_5: R(c)
T_6 by p_6: R(a)



T_4 runs solo until it commits

T_6 runs solo until it commits

T_6 reads 2 for a in γ''_6

- Recall that γ_5 is the solo execution of T_5 after a1 a2 s2

Recall that (1) T_5 reads 2 for e in γ_5 and (2) $o_1 \neq o_2$

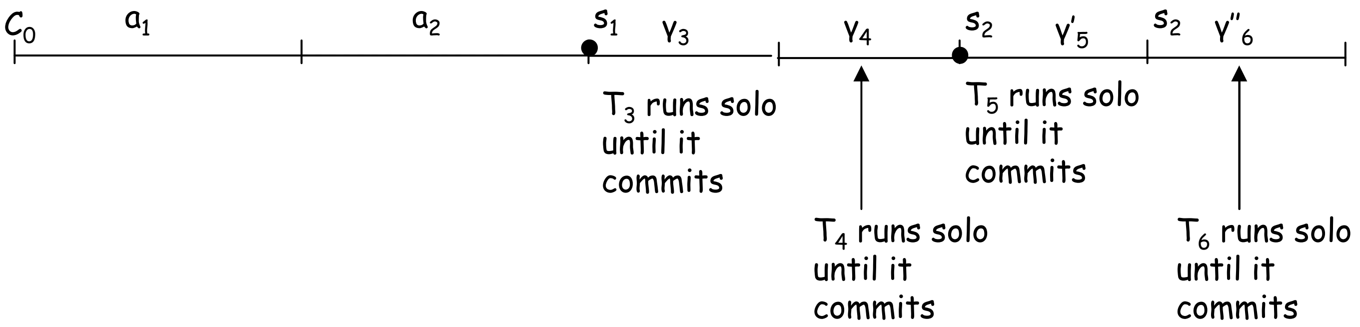
Claim 5: T_5 reads 2 for e in γ'_5

- γ_5 and γ'_5 are indistinguishable

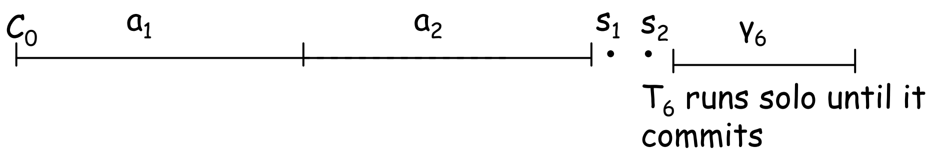
If γ'_5 does not read o_2 -> γ_5 and γ'_5 are indistinguishable.

Otherwise, γ_4 does not write o_2, since then γ_4 and γ'_5 conflict which violates strict DAPism.

T_6 reads 2 for a in γ_6



T_6 reads 2 for a in γ''_6



- γ_6 and γ''_6 are indistinguishable
- ➡ T_6 reads 2 for a in γ_6

T_1 by p_1: W(a,1) W(b,1) W(c,1)

T_2 by p_2: W(a,2) W(d,2) W(e,2)

T_3 by p_3: R(b)

T_4 by p_4: R(d)

T_5 by p_5: R(e)

T_5 by p_5: R(c)

T_6 by p_6: R(a)

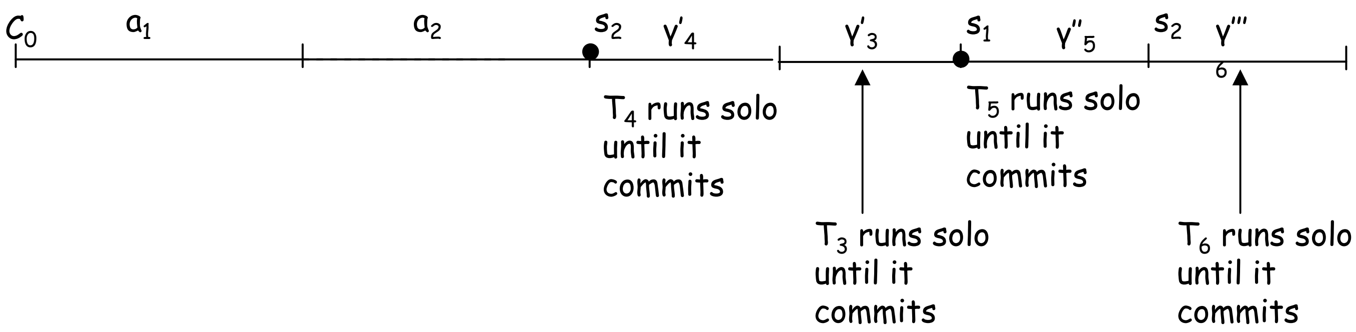
T_6 reads 1 for a in γ'_6

Claim 6: T_5 reads 2 for e in γ'_5
- γ_5 and γ'_5 are indistinguishable

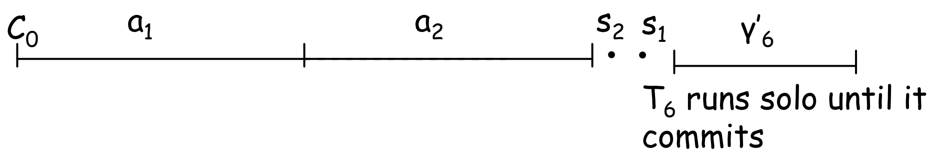
Claim 7: γ'_6 and γ'''_6 are indistinguishable

➧ T_6 reads 1 for a in γ'_6

T_1 by p_1: W(a,1) W(b,1) W(c,1)
T_2 by p_2: W(a,2) W(d,2) W(e,2)
T_3 by p_3: R(b)
T_4 by p_4: R(d)
T_5 by p_5: R(e)
T_5 by p_5: R(c)
T_6 by p_6: R(a)



T_6 reads 1 for a in γ'''_6



Putting it all together

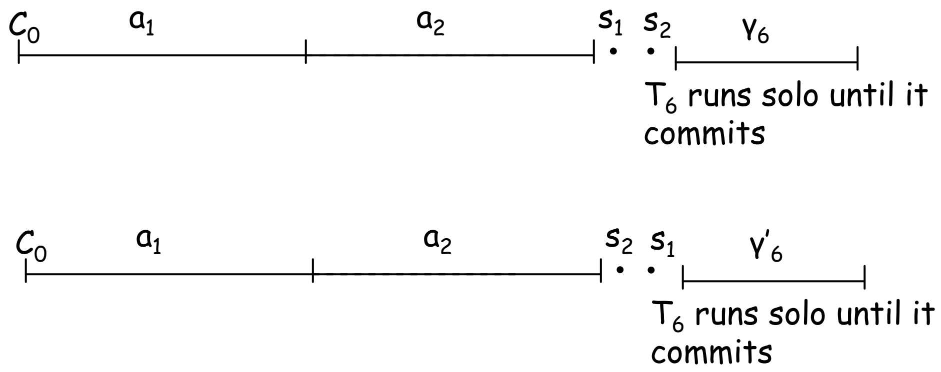
We have constructed two executions:

$$\delta 1 = \alpha 1\ \alpha 2\ s1\ s2\ \gamma 6, \text{ and}$$

$$\delta 2 = \alpha 1\ \alpha 2\ s2\ s1\ \gamma' 6, \text{ s.t.:}$$

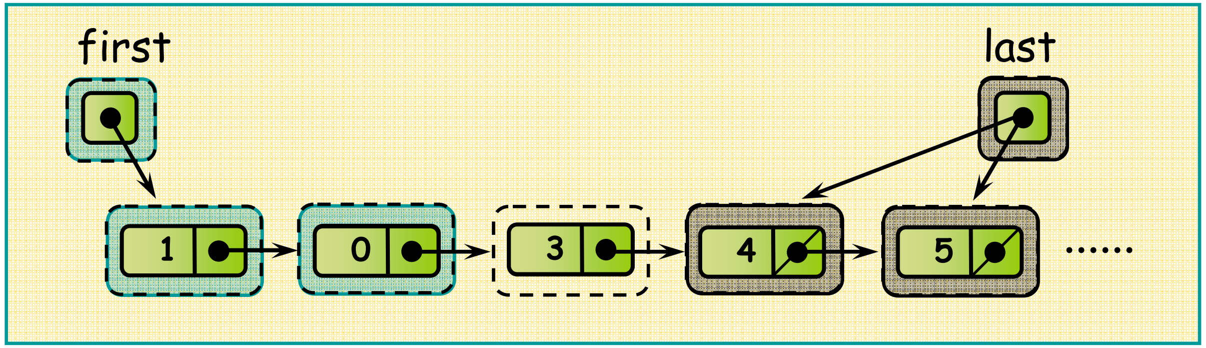
- $\gamma 6$ and $\gamma' 6$ are solo executions of T6
- in $\gamma 6$, T6 reads the value 2 for a
- in $\gamma' 6$, T6 reads the value 1 for a
- $\gamma 6$ and $\gamma' 6$ are indistinguishable.

It follows that T6 should read the same value for a in both executions.
A contradiction!



DAPism & wait-freedom: Impossible

Proof intuition: there is an execution where
- process **p** performs a **single** instance of Search(0) that never terminates
- process **q** performs instances of Append(i), for i ≥ 1



dap prevents q from discovering and helping Search(0)

Search(0) may terminate … it accesses node

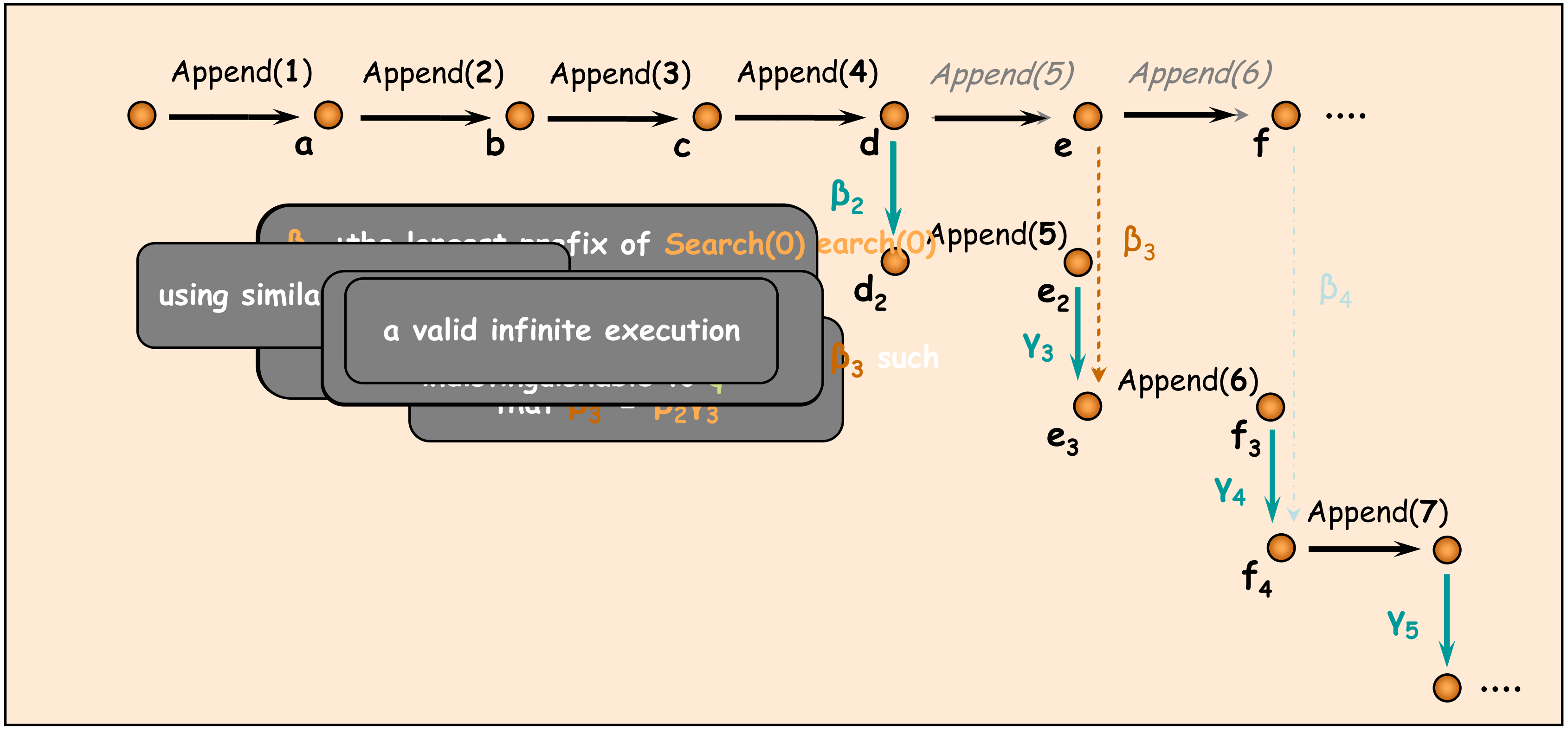
the same reasoning is applied repeatedly …

…s p from discovering whether node 5 has been appended before or after Seach(0) started

DAPism & wait-freedom: Impossible

Proof

1. Construction of a valid infinite execution
2. We prove that for each **i ≥ 3**, at least one of γ_i, γ_{i+1}, and γ_{i+2} is nonempty

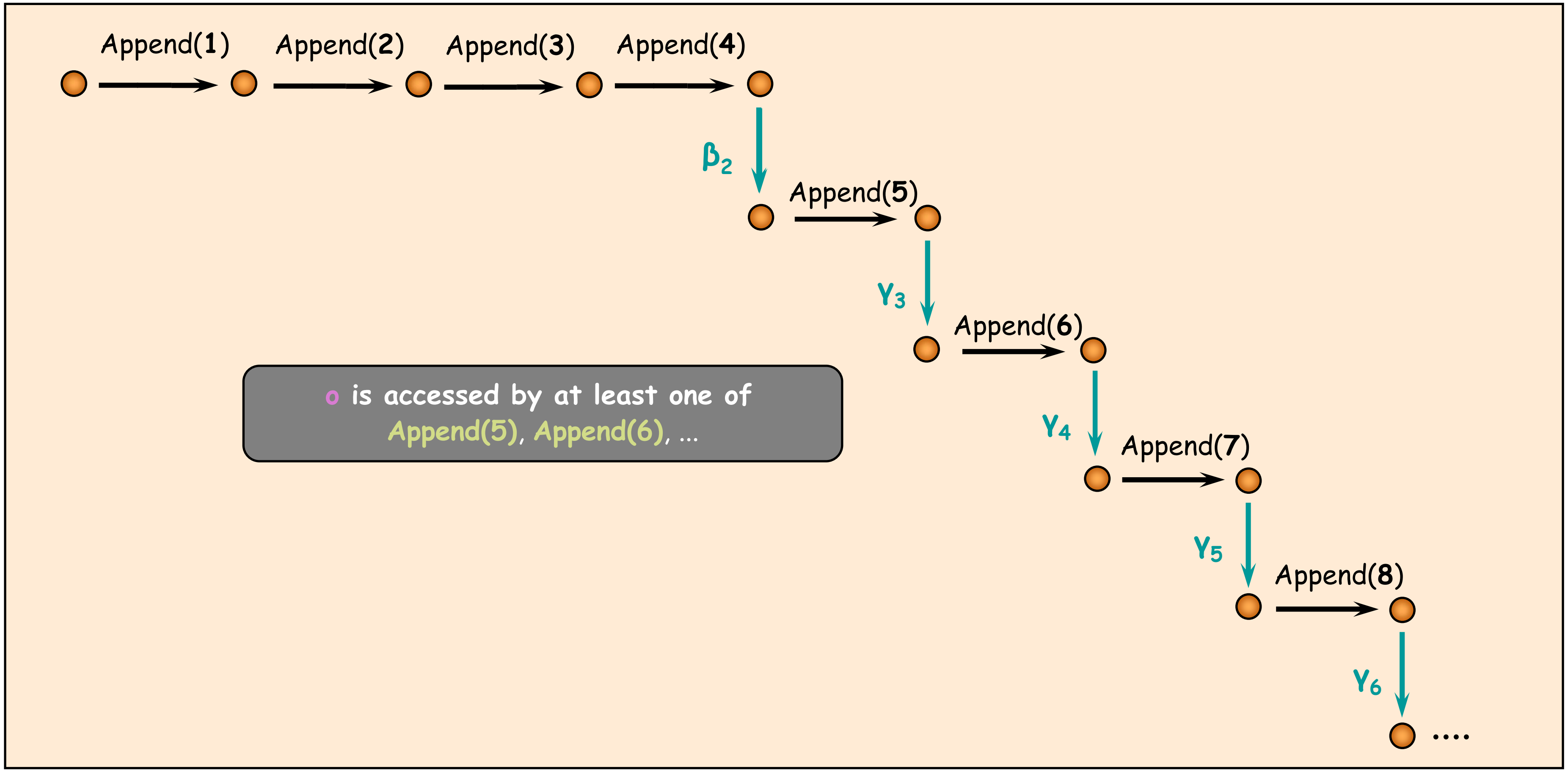


DAPism & wait-freedom: Impossible

Proof

3. As an example, we consider **i = 3** and by contradiction, we assume that γ_3, γ_4, and γ_5 are empty

3.1. p continues after β_2 (by accessing some base object o)



DAPism & wait-freedom: Impossible

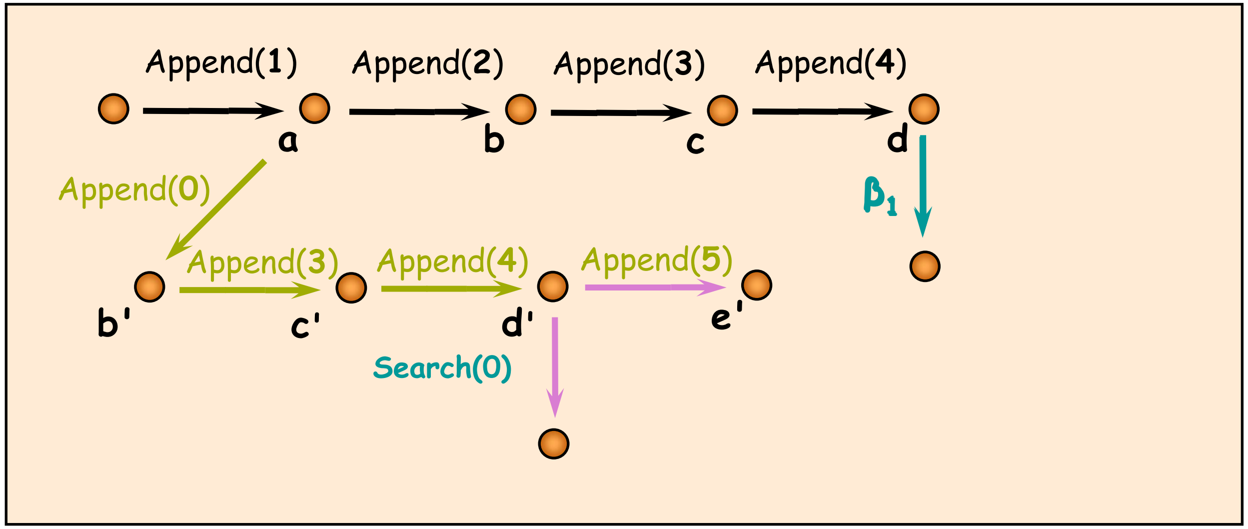
3.1. p continues after β_2 (by accessing some base object o that is ~~accessed~~ by at least one of

Contradiction!

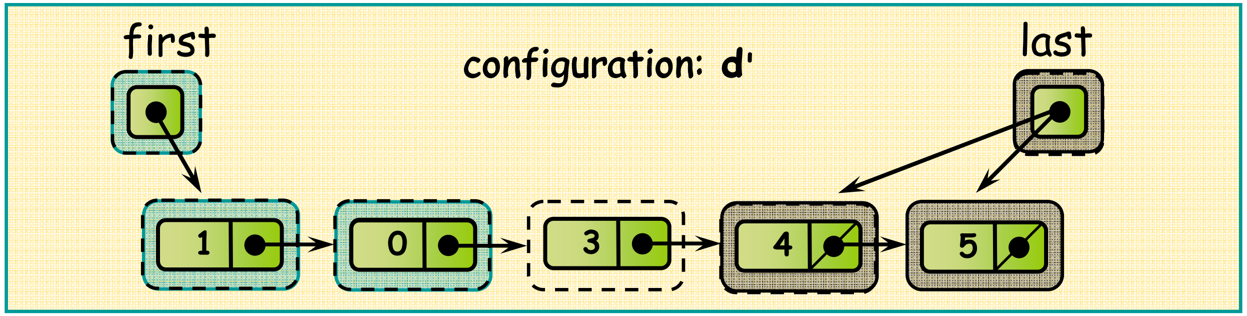
Append(5), Append(6), ...)

3.2. the solo executions of Append(5) and Search(0) starting from d' access o

3.3. the solo executions of Append(5) and Search(0) starting from d' access no common base object



Search(0) starting from d responds with false and Search(0) starting from d' responds with true



Possible/Impossible Combinations for strict serializable Implementations

strict dap + obstruction-freedom Guerraoui & Kapalka, SPAA 2008	Impossible
dap + nonblocking Fraser UCAM-CL-TR, 2003	Possible
dap + wait-freedom Ellen et al., PODC 2012	Impossible
dap + wait-freedom for operations with bounded data sets Ellen et al., PODC 2012	Possible
dap + nonblocking + read-only transactions never abort Attiya et al., SPAA 2009	Impossible
dap + read-only transactions never abort Avni and Shavit, SIROCCO 2008	Possible

Complexity

opacity + permissiveness:

- $\Omega(r)$ synchronization primitives required for a transaction that reads r data items*

opacity + progressiveness:

- $O(1)$ synchronization primitives suffice*

opacity + progressiveness + dap:

- $\Omega(w)$ synchronization primitives required for a transaction that writes w data items*

* Kuznetsov et al., OPODIS 2011

Complexity

- strict serializability + dap + wait-free read only transactions: reading r data items requires updating ≥ r −1 base objects

  Attiya et al., SPAA 2009

- strong atomicity + dap + invisible reads + progressiveness/obstruction freedom: the data set of a privitizing transaction must contain all privitized data items

  Attiya & Hillel, DISC 2010

- strong atomicity + dap + progressiveness/obstruction freedom: in the worst case, a transaction privitizing k data items must access at least k base objects

  Attiya & Hillel, DISC 2010

Part 5: TM Algorithms

Design of STM systems - Ownerships

- Recall that to ensure consistency, TM algorithms acquire **ownerships** on data items
- How long an ownership can be hold?
  - non-preemptive ownerships (e.g. locks)
    - ☞ lead to blocking STM algorithms
  - preemptive ownerships
    - a transaction may either **forcefully abort** some other transaction, or
    - **help** it so that the ownerships it holds are released
    - ☞ lead to non-blocking STM algorithms
- When ownerships are acquired?
  - upon first access to datum (eager acquisition)
  - after accessing datum, e.g. at commit time (lazy acquisition)

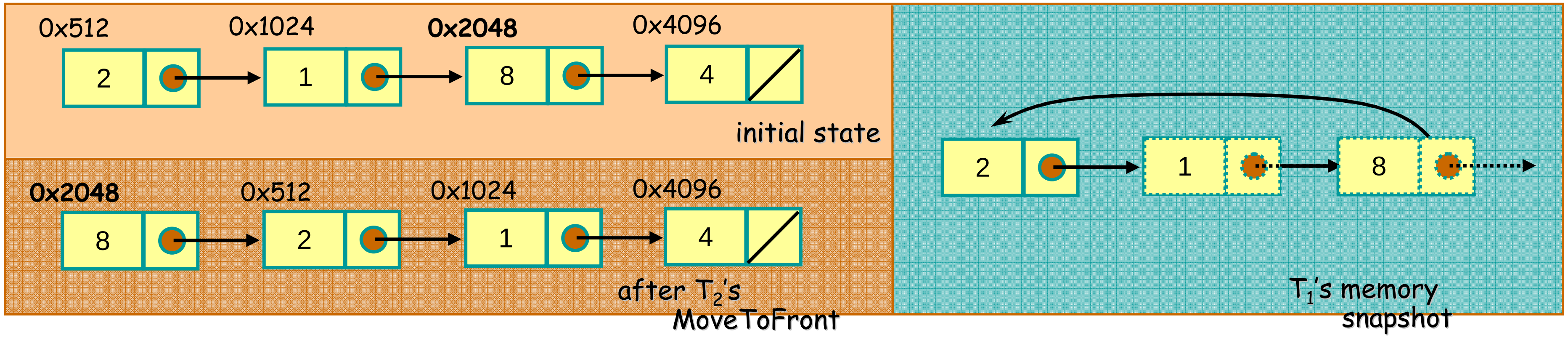
Design of STM systems

- **Writing** data items
  - When a transaction aborts, its modifications must be discarded and never become apparent
    - acquire the ownership before writing and maintain it until transaction's completion
      - with eager acquisition an **undo log** is maintained
      - with lazy acquisition a **redo log** is maintained

- **Reading** data items
  - it is possible some data item to be updated after reading it. Then, what was read may be **inconsistent** (in some executions)
    - the algorithm's correctness is violated
    - ☞ **zombie** or **doomed** transactions
  - other problems may be introduced due to inconsistencies
    - e.g. infinite loops, dereferencing null pointers, dividing by zero

Design of STM systems - Example of an indefinite loop

- Transactions T_1 and T_2 are concurrently accessing a linked list
  - T_1 searches the node containing number 4
  - T_2 executes a MoveToFront procedure
- Execution:
  - **T_1 begins and visits node with number 2**
  - **T_1 visits node with number 1**
  - **T_2 begins, executes MoveToFront(8) and commits**
  - **T_1 visits node with number 8**
  - **Infinite Loop!**



Design of STM systems - Reading Shared Data

- 1st Solution:
  - acquire the ownership of some data item before reading it
  - **visible reads** are used

- 2nd Solution: (when visible reads are not used)
  - introducing the notion of **version** for each data item
    - in each write, the version of the data item is also updated
    - maintained to data item's metadata
  - each transaction,
    - maintains the version for each data item it reads
    - checks if it has changed
      - a **validation mechanism** is used

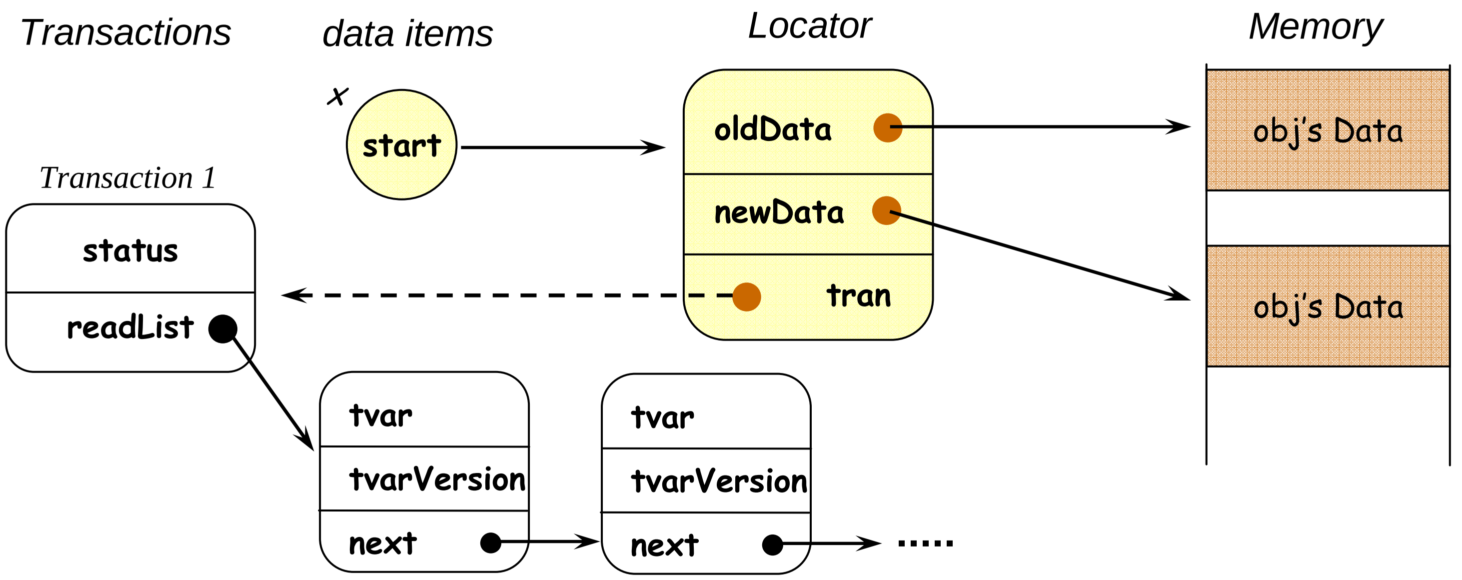
STM Algorithms

Several STM **algorithms** already exist

- non-blocking:
  - SSTM (Shavit & Touitou, PODC 1995)
  - DSTM (Herlihy et. Al, PODC 2003)
  - OSTM (Fraser 2003, PhD Thesis, Cambridge Un. Computer Lab.)
  - NBSTM (Kosmas, MSc Thesis, Un. of Ioannina, 2008)
  - ASTM (Marathe et. al, DISC 2005)
  - NZSTM (Tabba et. al, SPAA 2009)
  - this list is not exhaustive
- blocking:
  - TL (Dice et. al, TRANSACT 2006)
  - TLII (Dice et. al, DISC 2006)
  - RingSTM (Spear et. al, SPAA 2008)
  - TinySTM (Felber et. al, PPoPP 2008)
  - Norec (Dalessandro et. al, PPoPP 2010)
  - this list is not exhaustive

DSTM - Dynamic STM

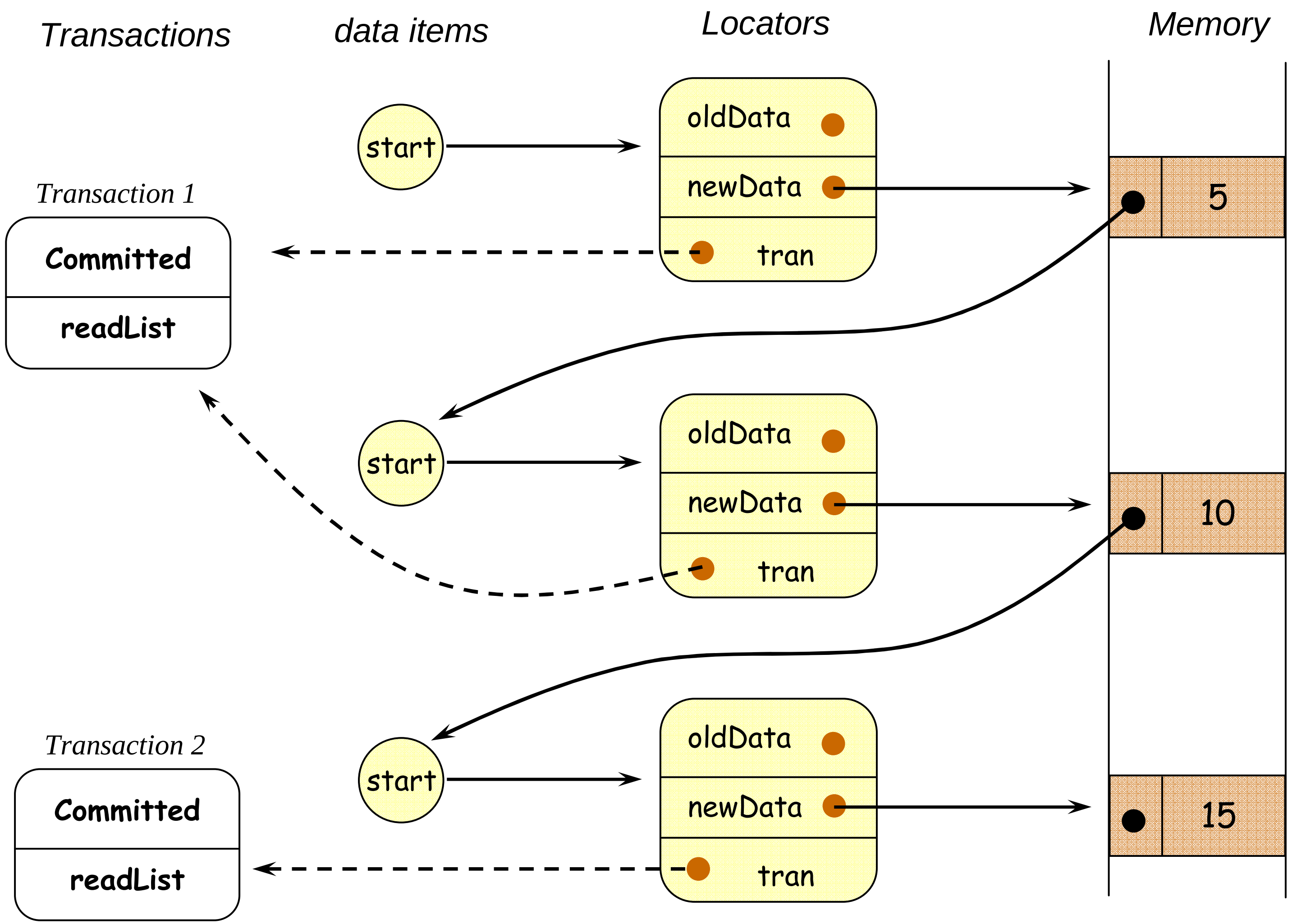
- Supports
  - **dynamic transactions**
  - the weakest non-blocking liveness property **obstruction-freedom**
- The representation of some data item *x* that describes an object *obj*:



- **status** $\in$ {Active, Committed, Aborted}
- **Read list** is used to implement validation
- Tm-variables are ***CAS registers***

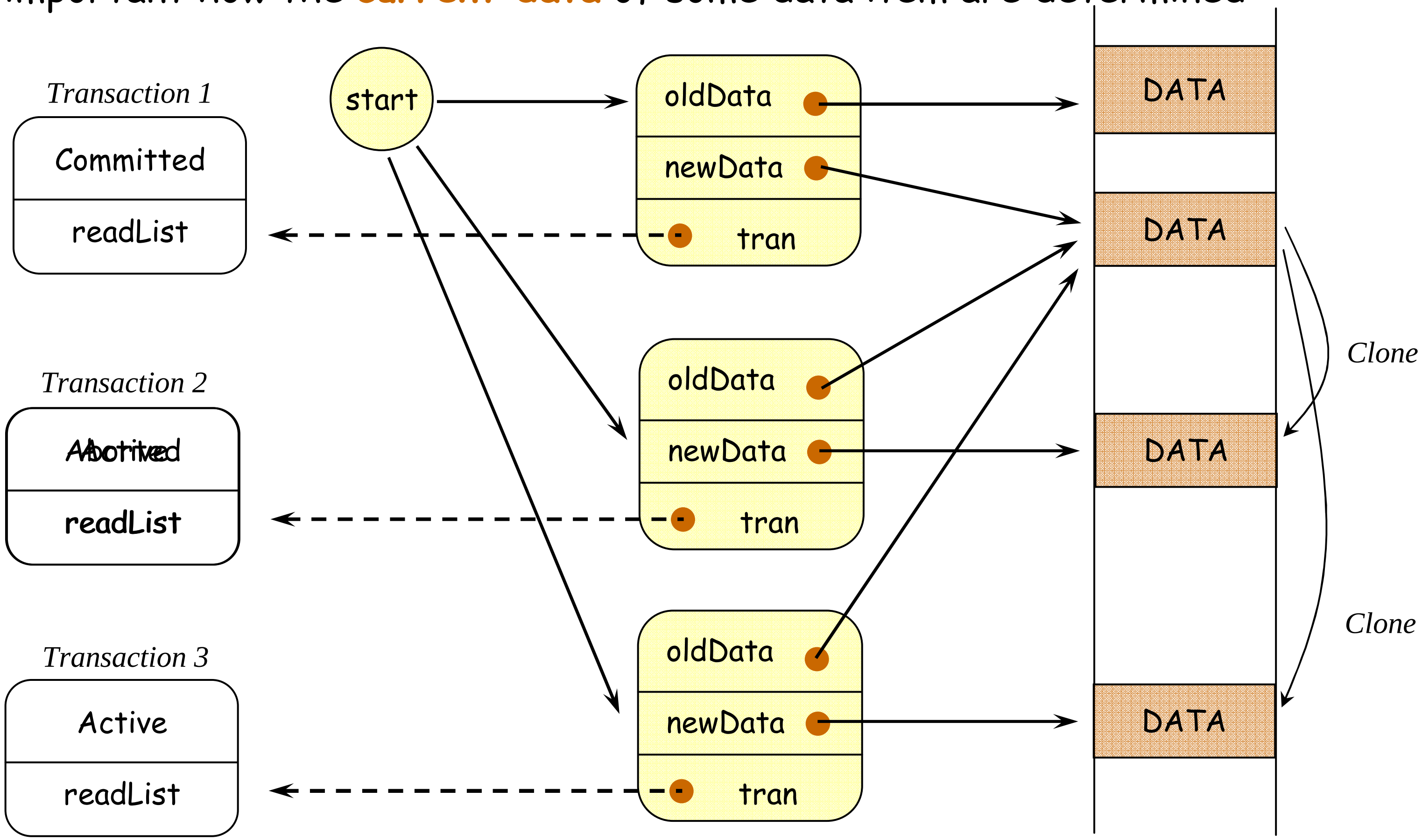
DSTM

Example: A sorted linked list in DSTM



DSTM - Ownership acquisition

- **Eager ownership acquisition**
- It is important how the **current data** of some data item are determined

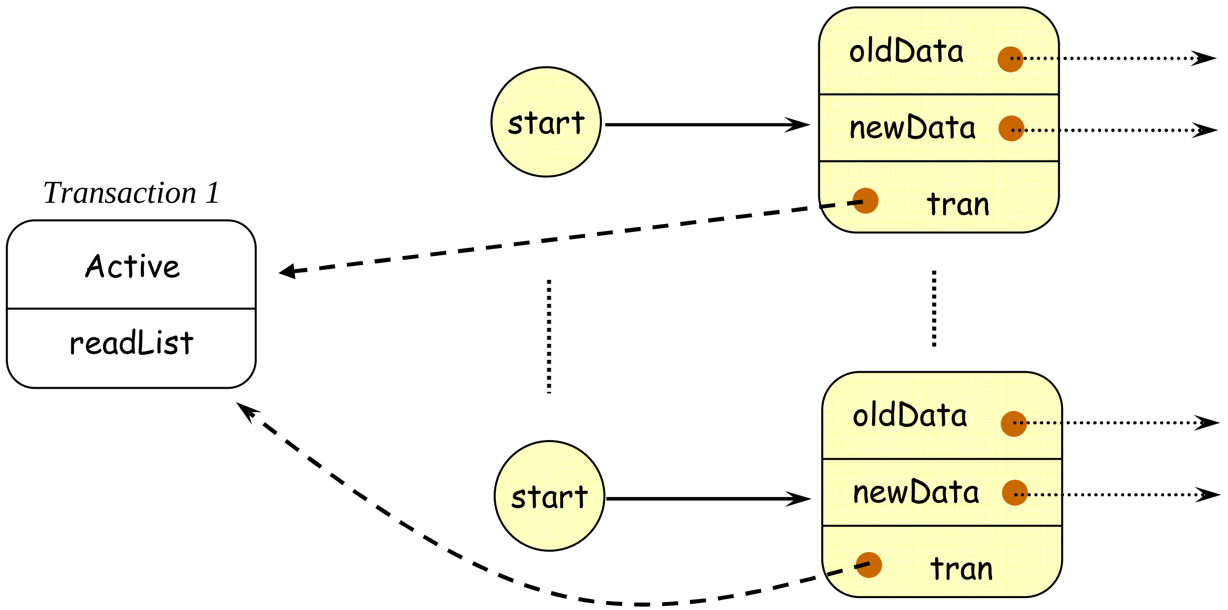


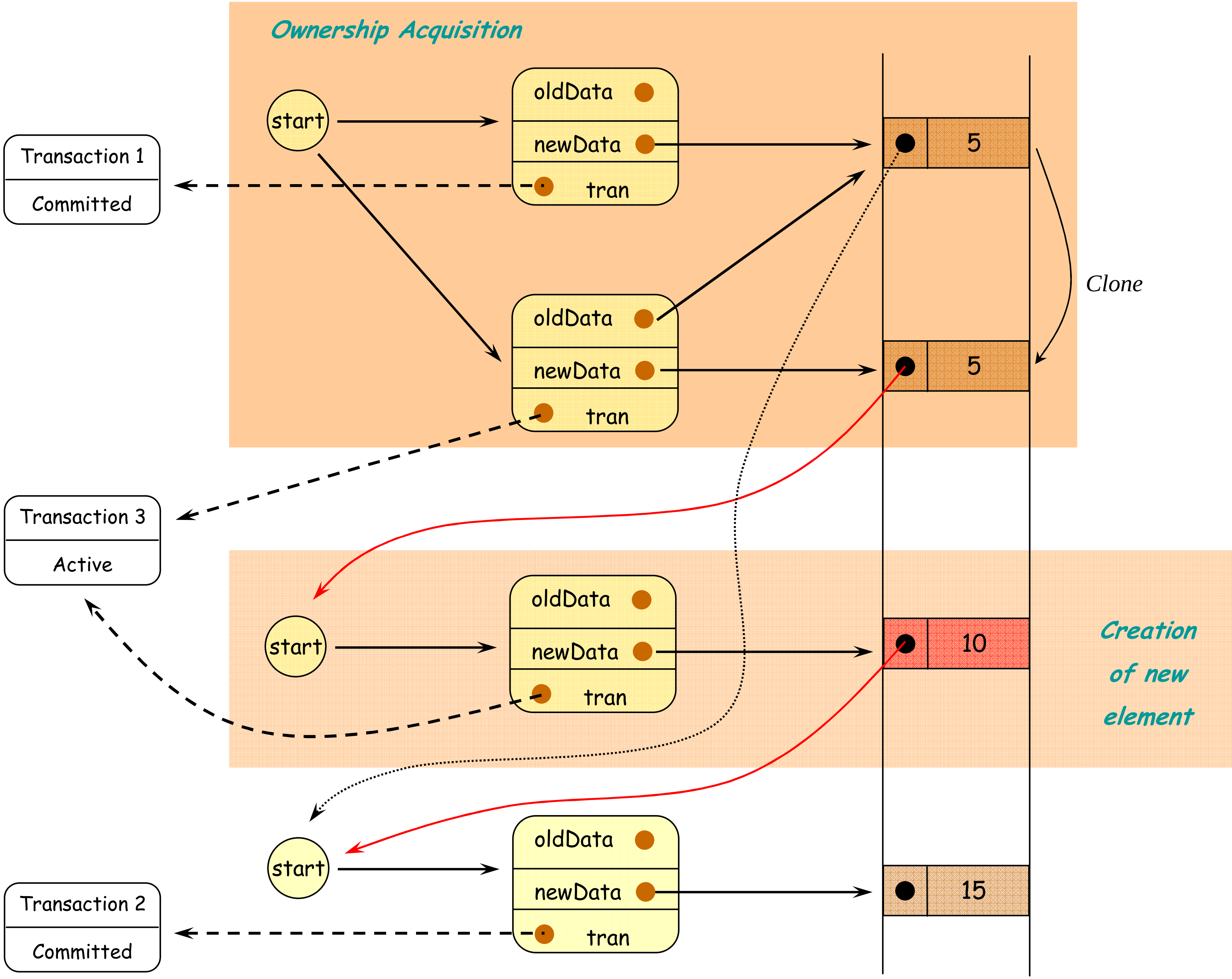
? When status is Active?

DSTM - Ownership release

- Some transaction **forcefully aborts** another by changing its status field to Aborted
  - Obstruction-freedom

- **Ownership release**: A transaction changes its status to either Committed or Aborted
  - execution of a **single** command **simultaneously** updates or restores the data of all data items




Ownership Acquisition
start
oldData
newData
tran
5
Clone
5
Transaction 1
Committed
Transaction 3
Active
start
oldData
newData
tran
10
Creation
of new
element
start
oldData
newData
tran
15
Transaction 2
Committed

DSTM - Validation mechanism

- Invisible reads

- **Version** of data item: The memory addresses in which the corresponding data item is stored

- **Read List**. For each data item read, is maintained:
  - its data item
  - its version

- **Validation Mechanism**
  - during execution of ReadTmVar operation all the elements of read list are checked
  - if **at least one of them** has been changed, the transaction is aborted

Thank you!

Questions?