Transactional Memory

INF346, 2014

Dealing with concurrency

- Locks:
  - ✓Coarse-grained: inefficient
  - ✓Fine-grained: deadlock-prone
  - ✓Do not compose
- Non-blocking:
  - ✓Difficult
  - ✓Inefficient?
  - ✓Still an active research area
- Experts are needed!
  - ✓(took 2 years to include a non-blocking queue to java.until.concurrency)
- Needed: efficient and simple concurrency control

Historical perspective

- Eswaran et al (CACM'76) Databases
- Papadimitriou (JACM'79) Theory
- Liskov/Sheifler (TOPLAS'83) Language
- Knight (ICFP'86) Architecture
- Herlihy/Moss (ISCA'93) Hardware
- Shavit/Touitou (PODC'95) Software
- Herlihy et al (PODC'03) Software – Dynamic
- Intel, AMD, … (2012) – hardware TM
- Now: PODC/POPL/PLDI/OOPSLA…CAV

Transactional memory

- Mark sequences of instructions as an **atomic transaction**:

```
atomic {
        if (tail-head == MAX){
        return full;
        }
        items[tail%MAX]=item;
        tail++;
}
return ok;
```

Invariant:
every item consumed,
no item consumed twice

- A transaction can be either **committed** or **aborted**
  - ✓ Committed transactions are **appear sequential**
  - ✓ Transactional memory (TM) resolves conflicts by aborting transactions
  - ✓ Easy to use: think sequential and program concurrent

What do we expect from TM?

- Safety:
  - ✓ Committed transactions make sense
- Liveness/progress
  - ✓A transaction eventually commits or aborts
  - ✓Some transactions commit
- Performance
  - ✓Enough transactions commit
  - ✓Underlying concurrency exploited

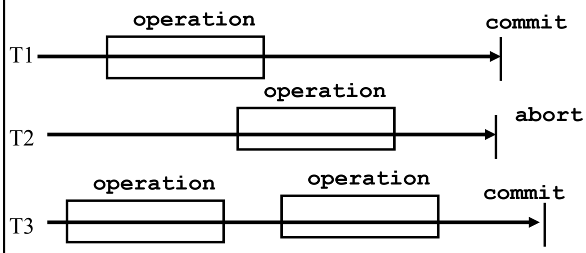
Safety of TM

- How to say that a TM history is correct
  - ✓Equivalent to a legal sequential obe
- What is a TM history?
- What is legal?
- What is sequential?
- What is equivalent

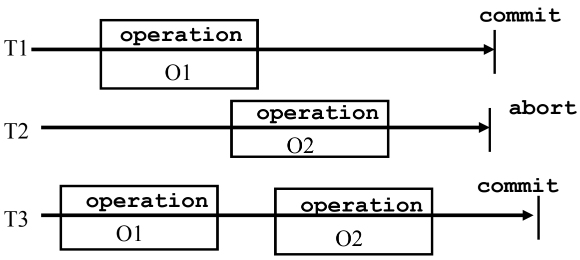
Transactions and objects

- **Transactions** invoke operations on shared **objects**
- Every operation **invocation** is expected to return a **reply**
- Every transaction is expected either to **abort** or **commit** (disclaimer for liveness)

Transactions and objects



Transactions and shared objects



Transactions

- Transactions are **sequential** units of computations
- Transactions are **asynchronous**

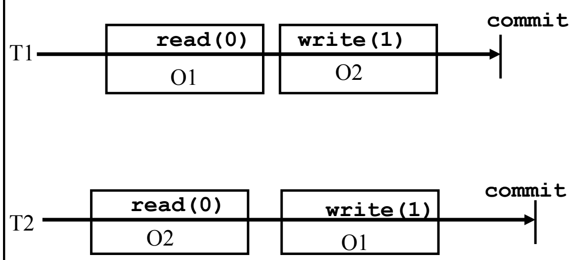
  (pre-emption, page faults, crashes)

Histories

- The execution of a set of transactions on a set of objects is modeled by a **history**
- A history is a **total order** of invocation and responses of operations, commit and abort **events**
  - ✓H = (E,<)

The history depicts what the user sees

History H1

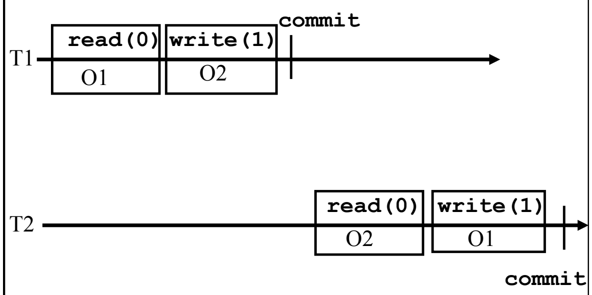


Histories

- Two **transactions** are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise
- A **history** is **sequential** if it has only sequential transactions; it is **concurrent** otherwise
- Two histories are **equivalent** if they **agree** on the the set of transactions

13

Sequential history H2 ≈ H1



14

Classical trensactional safety [Pap79]

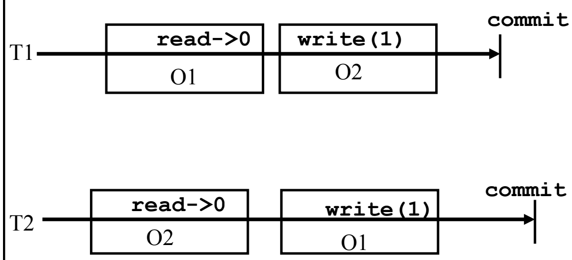
A history is **atomic** if its restriction to **committed** transactions is **serializable**

A history H of **committed** transactions is **serializable** if there is a history S(H) such that:

1. S is **equivalent** to H
2. S is **sequential**
3. in S, every read returns the **last value written**

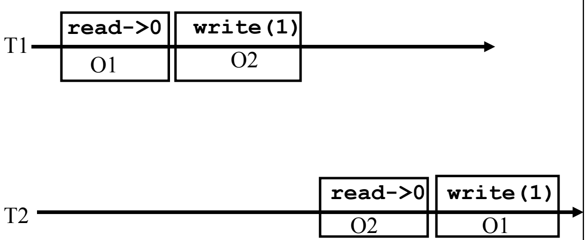
15

Atomic history?



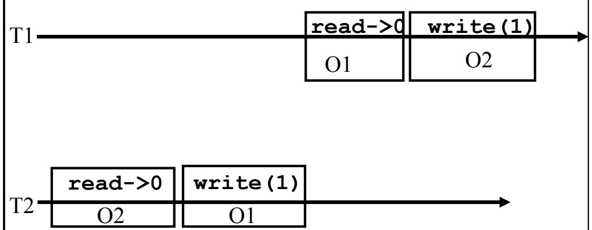
16

Sequential history?



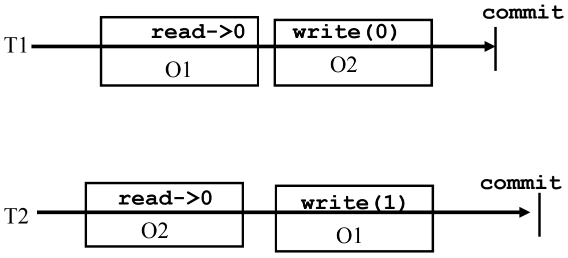
17

Sequential history?



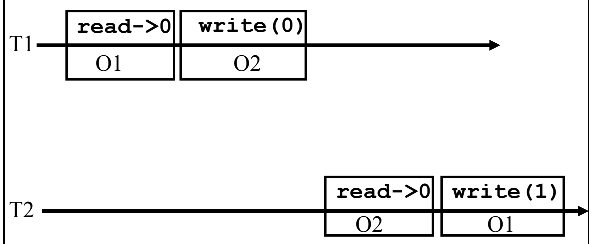
18

Atomic history?



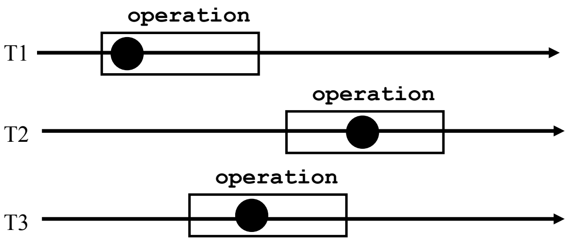
19

Sequential history



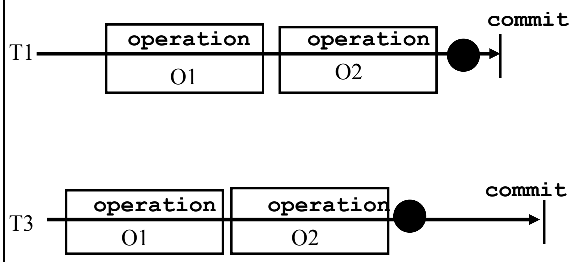
20

Operation atomicity (linearizability)



21

Transaction atomicity



22

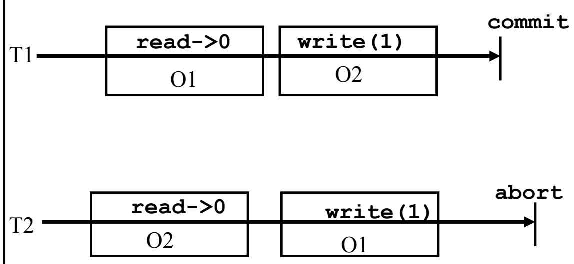
Serializability

- A history H of **committed** transactions is **serializable** if there is a history S(H) such that:

1. S is **equivalent** to H
2. S is **sequential**
3. in S, every read returns the **last value written**

23

Atomic history



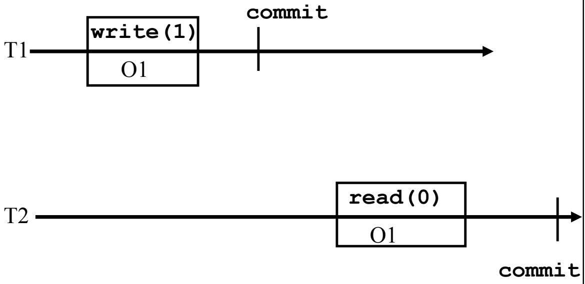
24

Serializability

- A history H of **committed** transactions is **serializable** if there is a history S(H) such that:

1. S is equivalent to H
2. S is sequential
3. in S, every **read returns the last value written**

Real-time



Preserving real-time order

- (T,T') is in H_{RT} if T terminates before T' begins
- S preserves the real-time order of H if
  - ✓H_{RT} is a subset of S_{RT}
    - If T precedes T' in H, T precedes T' in H

Strict serializability

A history H of **committed** transactions is **strictly serializable** if there is a history S such that:

1. S is equivalent to H
2. S is sequential
3. S is **legal** (with respect to each object)
4. S preserves the real-time order of H

Is it enough?

- Committed transactions stricly serializable
- Aborted transactions ignored

Is it safe?

(in a practical sense)

Simple algorithm (a la DSTM [Herlihy et al. 2003])

- To write O, T requires a **write-lock** on O;
T aborts T' if some T' holds ownership on O (using CAS)

- To read O, T checks if all objects read remain valid (keep the value read)- else abort

- Before committing, T checks if all objects read remain valid and changes its status to committed

Aggressive write, careful read
(obstruction-free writes, *progressive* progress)

DSTM: write, read, tryCommit

```
write(x,v)
   (owner,ov,nv)=tvar[x].read()
    curr=getValue(owner,ov,nv)
   if curr=live and !status[owner].cas(live,aborted) then return abort
   if tvar[x].cas([owner,ov,nv],[myself,curr,v]) then
         return ok
   else
         return abort

read(x)
   (owner,ov,nv)=tvar[x]
   curr=getValue(owner,ov,nv)
   if curr != live and valid() then
       rset = rset U {(x,[owner,ov,nv])}
       return curr
   else
         return abort

tryCommit()
   if valid() and status[myself].cas(live,committed) then
       return commit
   else
       return abort
```

New value of x, if the owner committed, old value of x if aborted or live, abort if live

try aborting the concurrent transaction

Grab the ownership on the object and set value v

Check if all previously read objects keep the same values

Set status to committed

31

DSTM: getValue() and valid()

```
getValue(owner,ov,nv)
   if status[owner]=committed
        return nv
   else if status[owner]=committed
        return ov
   else
        return live

valid()
   for each (x,[owner,ov,nv]) in rset do
       (owner',ov',nv')= tvar[x].read()
       if (owner',ov',nv')!=(owner,ov,nv) then
         return false
   return true
```

The value of x is not known (a concurrent transaction is writing to it)

Check every object in the "read set"

x has been overwritten

32

More efficient?

- Why validating all the time?
  - ✓"Apologizing vs. asking permission"
- Only validate at commit time
  - ✓Abort if did not succeed

Aggressive write, optimistic read

33

Example: run-time error

Initially: x=1, y=2

Invariants: 0<x<y

1/(y-x) is not supposed to give division-by-zero

But:

T1: x := x+1; y:= y+1

T2: z := 1 / (y - x)

34

Example: infinite loop

T1: x := 3; y:= 6

T2: a := y; b:= x;

repeat b:= b + 1 until a = b

35

More refined safety needed

We need a theory that restricts *all* transactions: this is what critical sections give us

Every transaction sees a consistent state

- sees?
- consistent?

A la critical sections (locks)

36

Histories

- Let H be any history (made of commited, aborted and pending transactions)
- Complete(H) is the history made of all transactions of H by completing pending ones with abort events
  - ✓And some of *pending commits* with commits

37

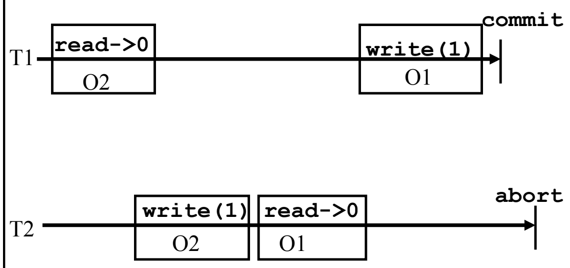
Opacity [GK'08]

A history H of **opaque** if there is a history S such that:

1. S is equivalent to (some history in) complete(H)
2. S is sequential
3. S is **legal** wrt committed transactions
4. S preserves the real-time order of H

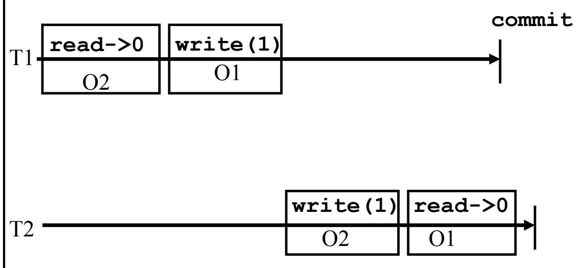
38

Opacity?



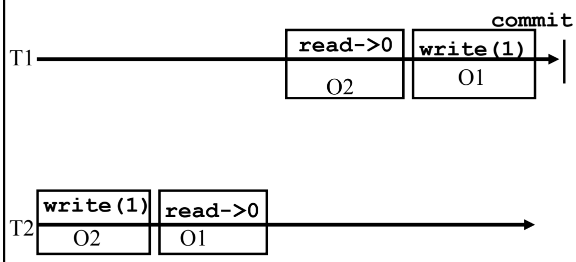
39

Not legal



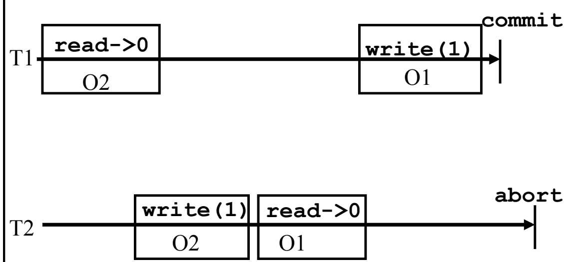
40

Legal



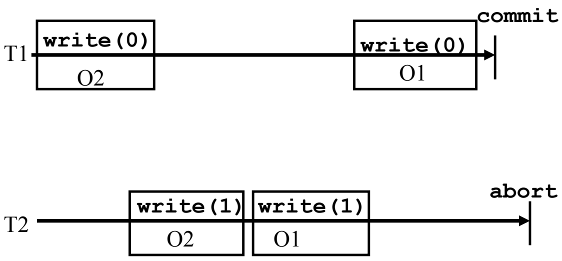
41

Recoverable (no dirty reads)



42

Opacity < rigorous scheduling



43

Simple algorithm (DSTM)

- Aggressive write (ownership)
- Careful read (validation)

44

Visible Read (SXM; RSTM)

- Write is **mega killer**: to write an object, a transaction aborts any live one which has read or written the object
- Visible but not so careful read: when a transaction reads an object, it says so

45

Visible Read

- A visible read invalidates cache lines
- For read-dominated workloads, this means a lot of traffic on the bus between processors
- This would reduce the throughput

46

Unavoidable (in some sense)

- **Theorem** **[GK'08]**

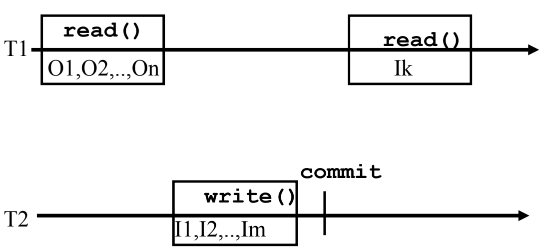
In an opaque TM, reads are either visible or careful

NB. Modulo a weak progress property (progressiveness) and the assumption of a single versions

Progressiveness: commit if no read-write or write-write conflicts

47

Intuition of the proof



48

Read invisibility

- The fact that the read is invisible means T1 cannot inform T2, which would in turn abort T1 if it accessed similar objects (SXM, RSTM)
- NB. Another way out is the use of multi-versions (maintain multiple copies of each object)
- The theorem does not hold for database (strictly serializable) transactions! Why?

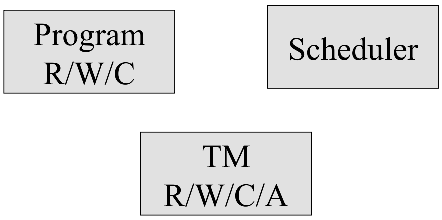
49

Verifying Opacity

- How to tell that a given history is opaque?
- Check that the conflict poly-graph is *acyclic*
  - ✓NP-Complete problem (equivalent to SAT)
  - ✓[Pap 79] for SR (serializability), holds for Opacity too. Why?
- But the space of verification can be reduced

50

Abstracting the problem



51

Reduce the space of verification

- Symmetric system

(all transactions are treated equally)

- ✓ Transaction names does not matter
- ✓ Variable names does not matter

52

TM verification theorem (GHS'08)

- A TM either violates opacity with **2 transactions and 3 variables** or satisfies it with **any** number of variables and transactions

53

Reference implementation

- A finite-state transition system (12.500 states) which generates all possible TM safe histories for 2 transactions and 3 variables

54

Model checking TM

- A TM is correct if the histories it generates could also be generated by the reference implementation
- Simulation relation between the TM (e.g., TL2 4500 states) and the reference implementation

55

Examples

- It takes 15mn to check the correctness of TL2 and DSTM
- Reverse two lines in TL2: bug found in 10mn - a history not permitted by the reference implementation

56

1. Safety of a TM

A. Do we need a new correctness criteria? Yes: opacity

B. How can we check it? Reduction

57

Why do we care?

- Modern computing is concurrent
- TM promises simplicity and efficiency

What should we expect?

- Safety: opacity (can be checked)

58

2. Liveness of a TM

What progress can we expect?

59

What is progress?

- Operations eventually return?
- Transactions eventually terminate?

60

What is progress?

- We want transactions to ***commit***, including long ones:
  - ✓rehashing the table,
  - ✓rebalancing the tree

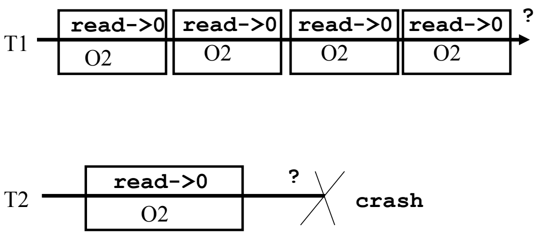
61

What is progress?

- We cannot require a TM to commits transactions:
  - ✓from a **dead** process; i.e., dead transactions
  - ✓that infinitely **loop**

62

Progress?



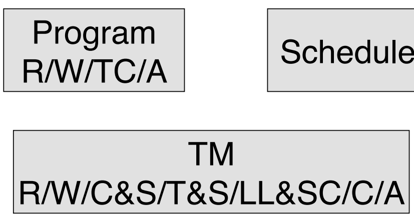
63

Progress

- We can only expect progress for ***correct*** transactions
- How to define a ***correct*** transaction?

64

Correctness depends on the scheduler and the program



65

History

- A history (as seen by the user) does not say what the ***scheduler*** does and whether the ***program*** behaves ***correctly***
- We need a ***refined*** notion of history

66

Low-level history

- A low-level history depicts the events of the ***implementation***
- A history is a ***total order*** of invocation, reply, try-commit, commit and abort events
  - ✓H = (S,<)

67

Low-level history

- The invocations and replies include also ***low-level*** objects used in the implementation
- The low-level history is a ***refinement*** of the high-level one (seen by the user)

68

Low-level history

- ***Well-formed*** (low-level) history:
  - ✓Every transaction that aborts is immediately repeated until it commits, i.e., :

  Every process executes:

  T1:op1; T1.op2; ..; T1:tryCommit; T1:abort; T1:op1;..; T1:commit; T1:op3…

69

Low-level history

- A transaction T is correct if
  - ✓ (a) try-commit is invoked after a finite number of invocation/reply events of T and
  - ✓ (b) either T commits or T performs an infinite number of steps
- (a) depends on the program
- (b) depends on the scheduler

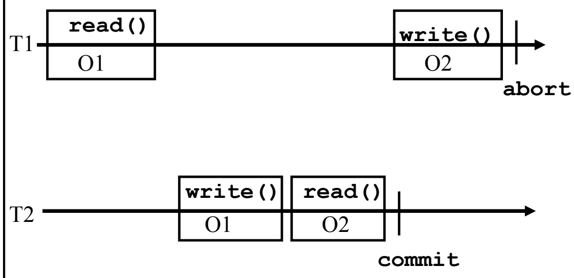
70

Ideal progress?

- No correct transaction aborts
- NB. This is not a liveness property
- Can we achieve this?

71

Aborting is a fatality



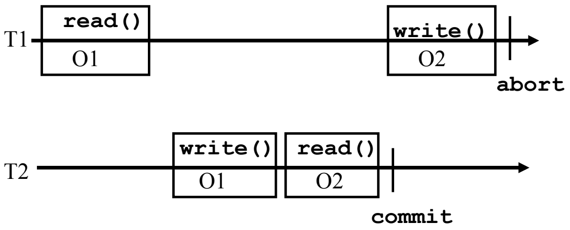
72

Global progress - wait-freedom -

- Every ***correct*** transaction ***eventually commits***
- NB. We allow the possibility for a transaction to abort a finite number of times as long as it eventually commits

73

Global progress - wait-freedom -



74

Impossible global progress - wait-freedom -

- Wait-freedom is **impossible** in an asynchronous system

- NB. This impossibility is fundamentally different from the impossibility of (wait-free) consensus [FLP85]: It holds for any underlying objects

75

Conditional progress - obstruction-freedom -

- A correct transaction that eventually does not encounter ***contention*** eventually commits
- ***Obstruction-freedom*** seem reasonable and is indeed possible

76

OF DSTM

- To ***write*** O, T requires a ***write-lock on*** O (use CAS);

T aborts T' if some T' acquired a ***write-lock*** on O (use CAS)

- To ***read*** O, T checks if all objects read remain valid - else abort (use CAS to abort a process holding locks on O)
- Before committing, T releases all its locks (use CAS)

77

DSTM: write, read, tryCommit

```
write(x,v)
   (owner,ov,nv)=tvar[x].read()
    curr=getValue(owner,ov,nv)
   if curr=live and !status[owner].cas(live,aborted) then return abort
   if tvar[x].cas([owner,ov,nv],[myself,curr,v]) then
         return ok
   else
         return abort

read(x)
   (owner,ov,nv)=tvar[x]
   curr=getValue(owner,ov,nv)
   if curr=live and !status[owner].cas(live,aborted) then return abort
   if curr != live and valid() then
       rset = rset U {(x,[owner,ov,nv])}
       return curr
   else
         return abort

tryCommit()
   if valid() and status[myself].cas(live,committed) then
       return commit
   else
       return abort
```

Read aborts the concurrent transaction

78

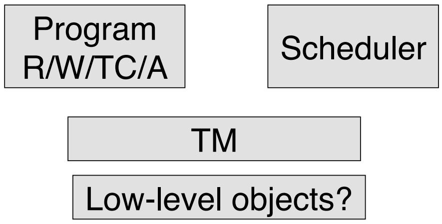
DSTM uses CAS

- ***CAS*** is the strongest synchronization primitive

  - Is OFTM possible with R/W objects?

79

OF-TM



80

Consensus number of OF-TM?

(∞)	Compare&Swap		
(..)	…		
(2)	Queue	Test&Set	Fetch&Add
(1)	Register	Snapshot	

81

FO-consensus

A process can decide or ***abort***

- No two different values can be decided
- A value decided was proposed

- If ***abort*** is returned from propose(v) then (1) there is contention and (2) v cannot be returned

82

OF-TM <=> FO-consensus

- From OF-TM to FO-consensus: ***propose()*** is performed within a transaction

- From FO-consensus to OF-TM: slightly more tricky - as for DSTM but using a one shot object instead of C&S

83

OF-consensus vs consensus

- OF-consensus can implement consensus among exactly 2 processes

- **Algorithm**
- P1 writes its value and keeps proposing until it decides a value
- P2 either decides or reads the value

84

The consensus number of OF-TM is 2

- OF-TM cannot be implemented with R/W objects only

But OF-TM does not need CAS!

85

OF-TM vs. OF objects

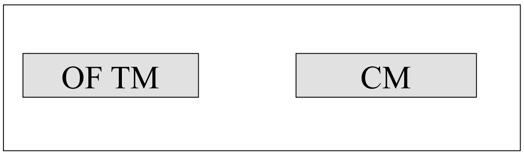
- Every OF object can be implemented with R/W objects
- Where is the bug?
- Abort really means the operation did not take place [AGHK'07]

86

TM Liveness

- Global progress (wait-freedom) is impossible
- Conditional progress (obstruction-freedom) is not trivial

Boosting OF?



87

Contention management

- Conflict resolution delegated to a **contention manager**
- Responsible solely for progress (liveness)

(different from a DB concurrency control)

88

Progress

- If a transaction T wants to write an object O owned by another transaction T', T calls a contention manager
- The contention manager can decide to wait, retry or abort T'

89

Contention managers

- **Aggressive**: always aborts the victim
- **Backoff**: wait for some time (exponential backoff) and then abort the victim
- **Karma**: priority = cumulative number of shared objects accessed – work estimate. Abort the victim when number of retries exceeds difference in priorities.
- **Polka**: Karma + backoff waiting

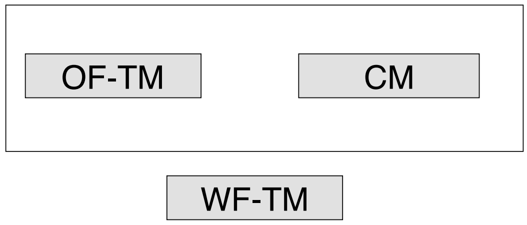
90

Greedy contention manager

- State
  - ✓Priority (based on start time)
  - ✓Waiting flag (set while waiting)
- Wait if other has
  - ✓Higher priority AND not waiting
- Abort other if
  - ✓Lower priority OR waiting

91

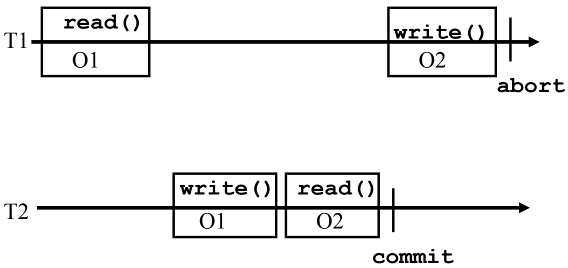
From OF to WF



Every correct transaction eventually commits,
(after finitely many aborts)

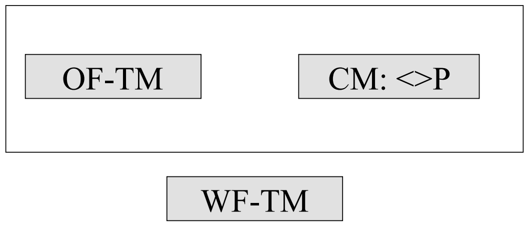
92

From OF-TM to WF-TM



93

The weakest synchrony assumption to implement WF-TM [GKK'06]



94

Why do we care?

- Modern computing is concurrent
- TM promises simplicity and efficiency

What is it?

-Safety: opacity, …
-Liveness: progressiveness, obstruction-freedom,…

95

Concluding

- TM does not replace locks: it *hides* them
  - ✓Can also be non-blocking
- TM only *looks* like db transactions and memory objects, but is quite different
  - ✓Safety, Liveness, Progress, …
- TM is another proof of the irrelevance of the notion of *relevance* …
  - ✓Like garbage collection in the old days

96

Take-aways

- Transactions (software and hardware) conquer concurrent computing
  - ✓Programmers are happy
- Making TM efficient is in fact tricky, there are inherent costs and trade-offs