Transactional Memory

SE205, 2017

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: Hybrid TM

Transactional memory

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

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

Invaluant:
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

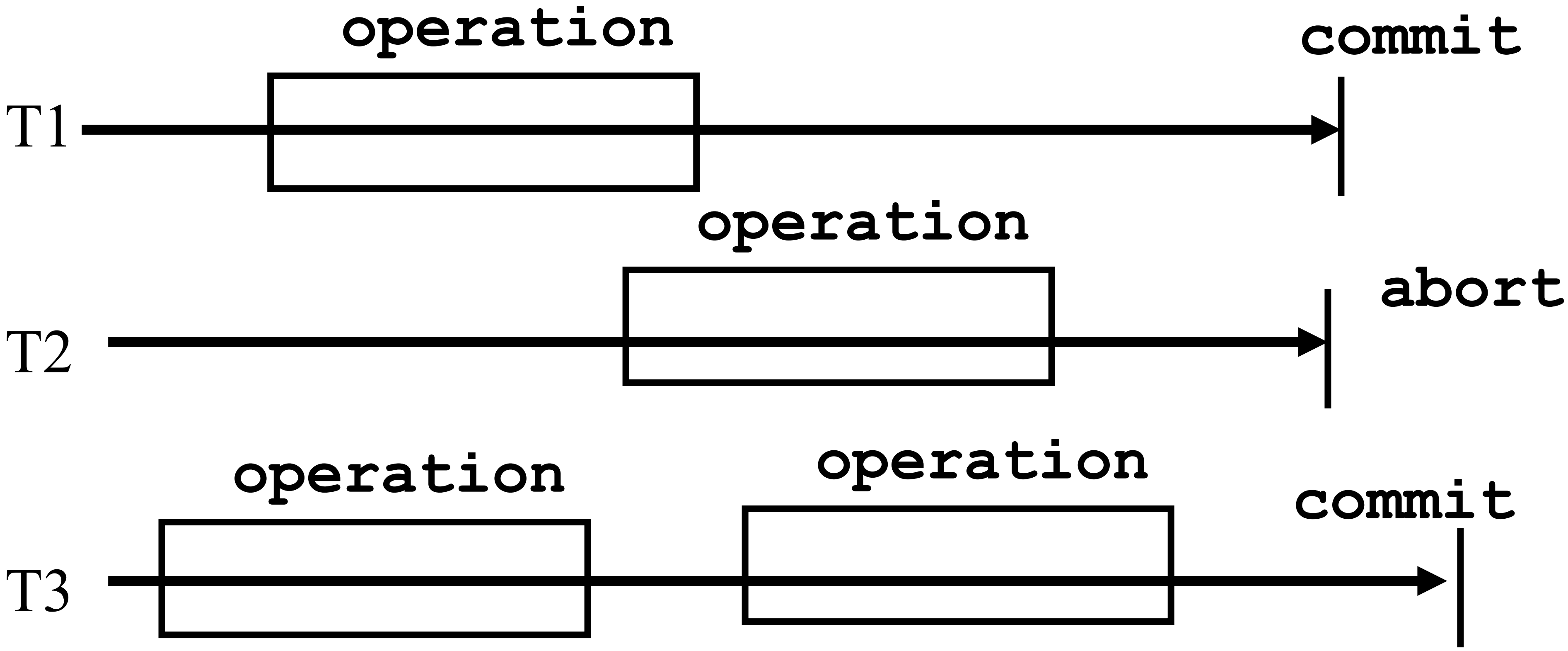
- How to say that a TM history is correct
  - ✓Equivalent to a legal sequential one

- What is a TM history?
- What is legal?
- What is sequential?
- What is equivalent

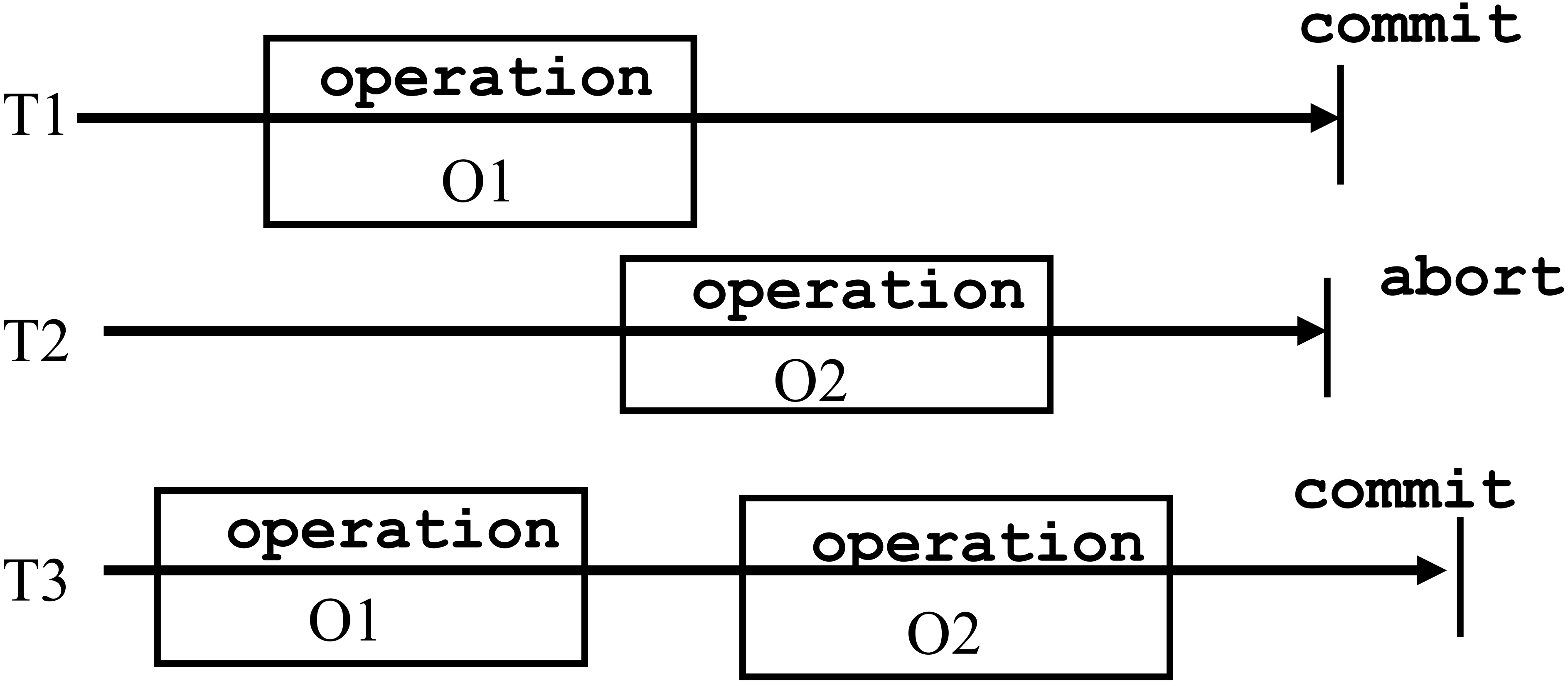
Transactions and objects

- 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)

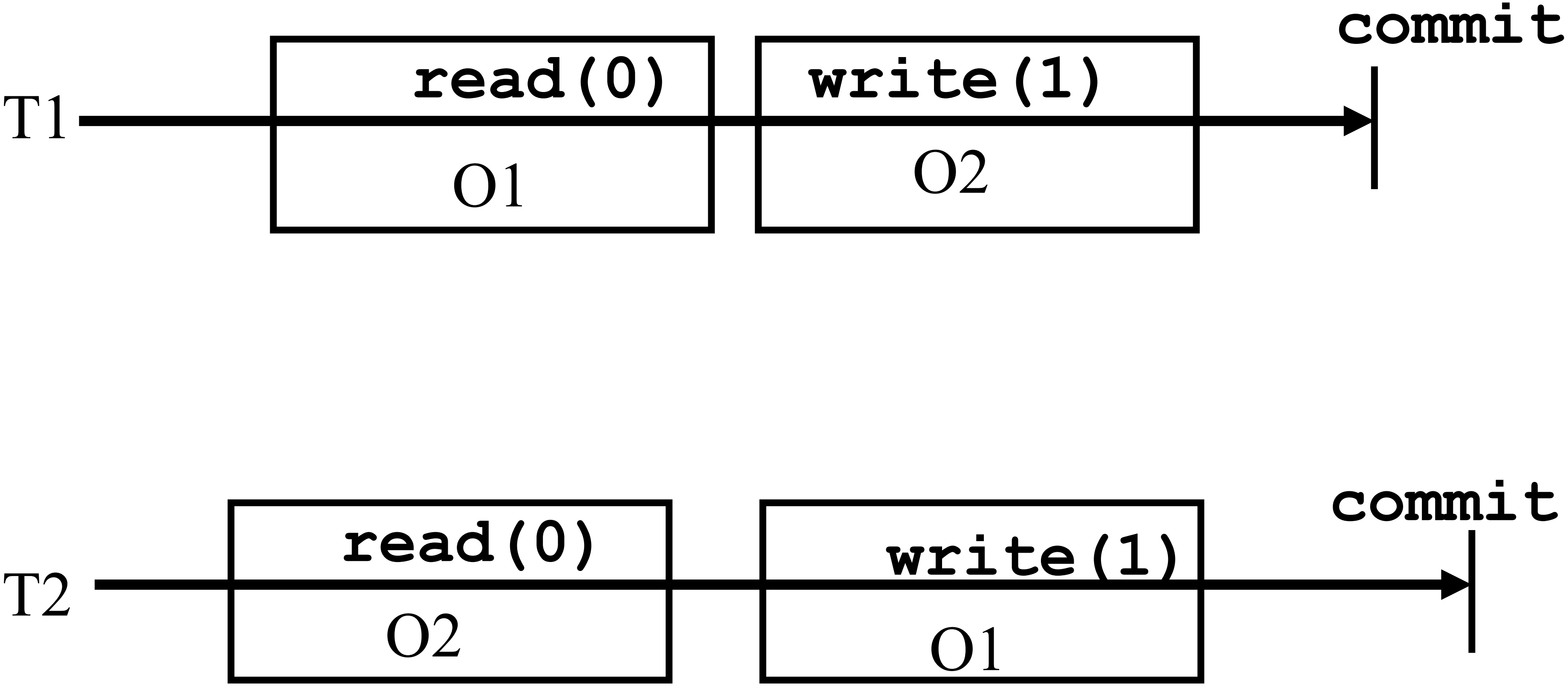
Histories

- 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



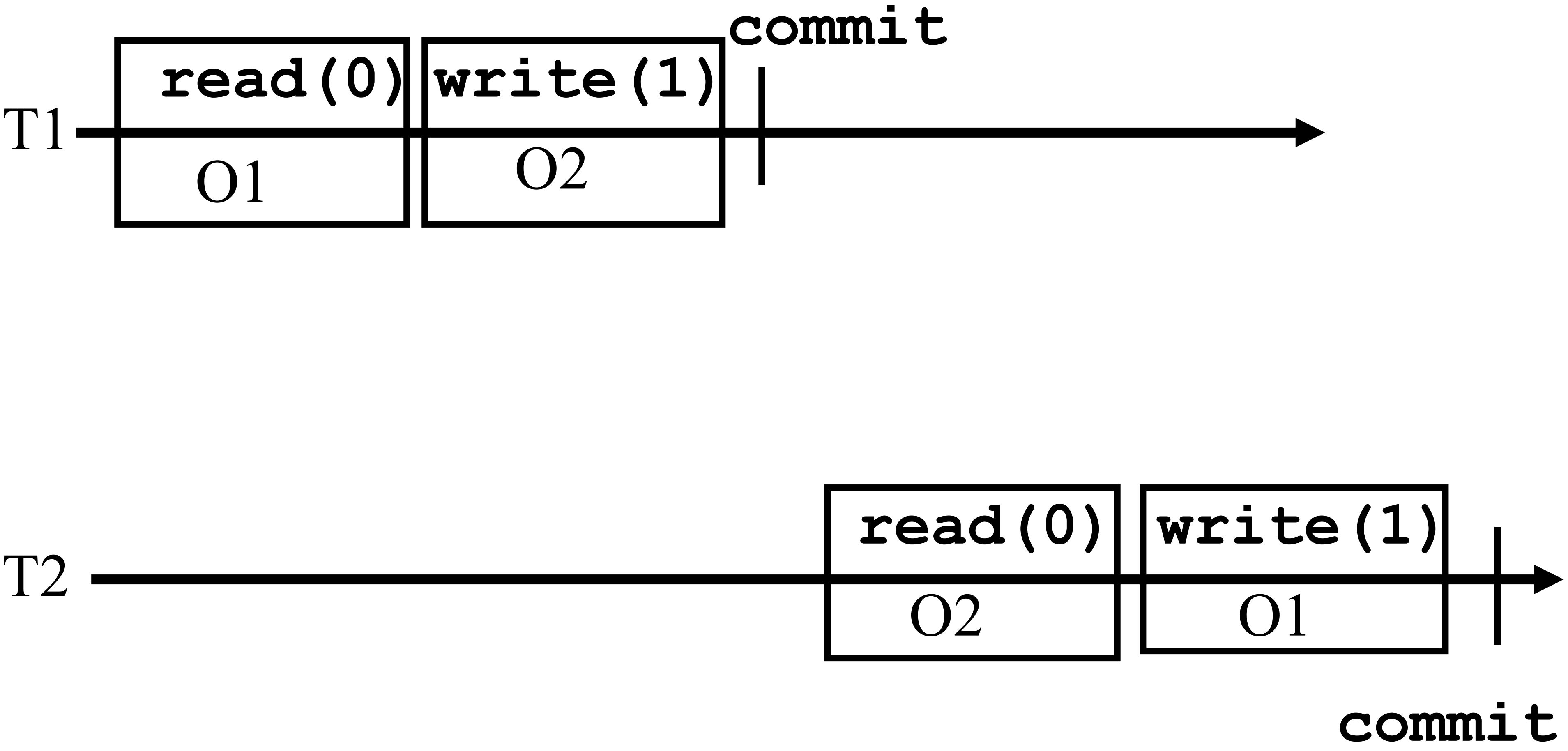
Histories

- 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

Sequential history H2 ≈ H1



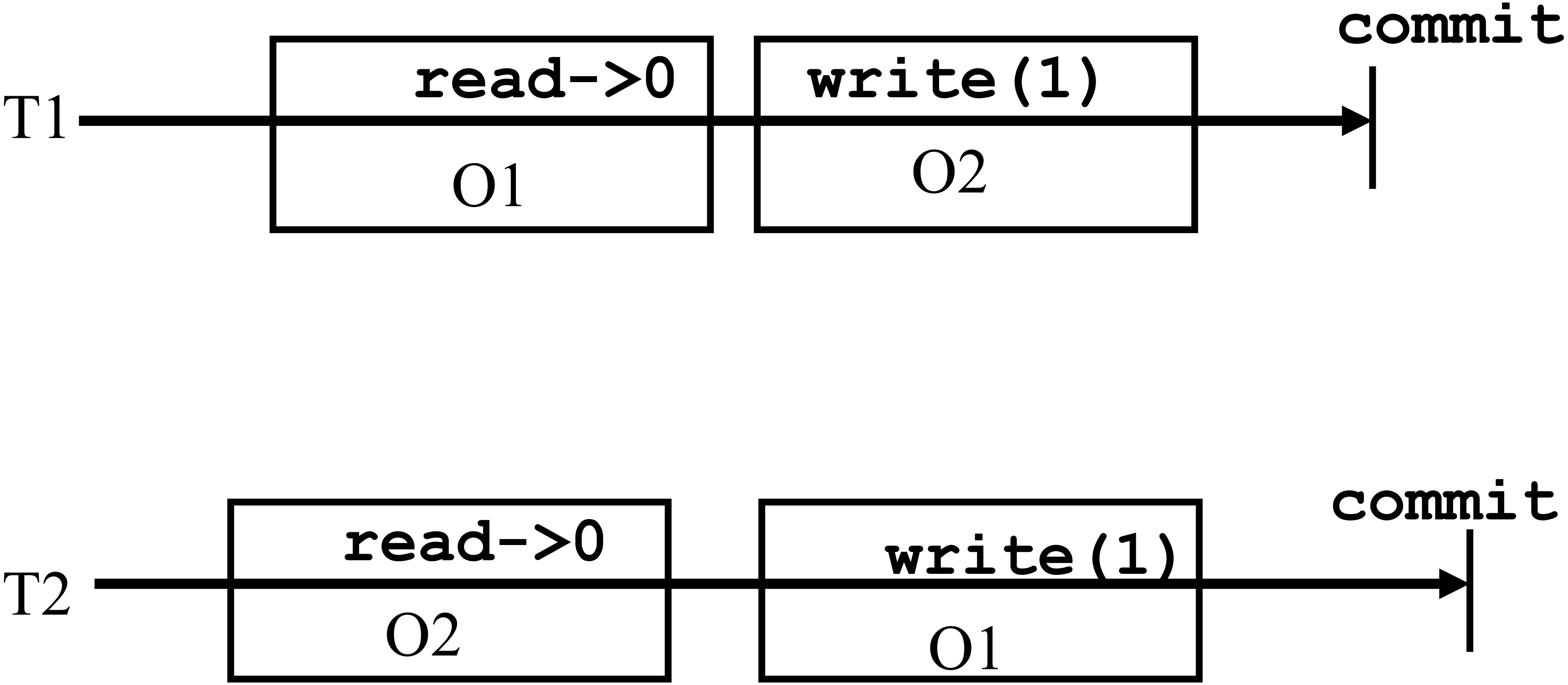
Classical transactional safety [Pap79]

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

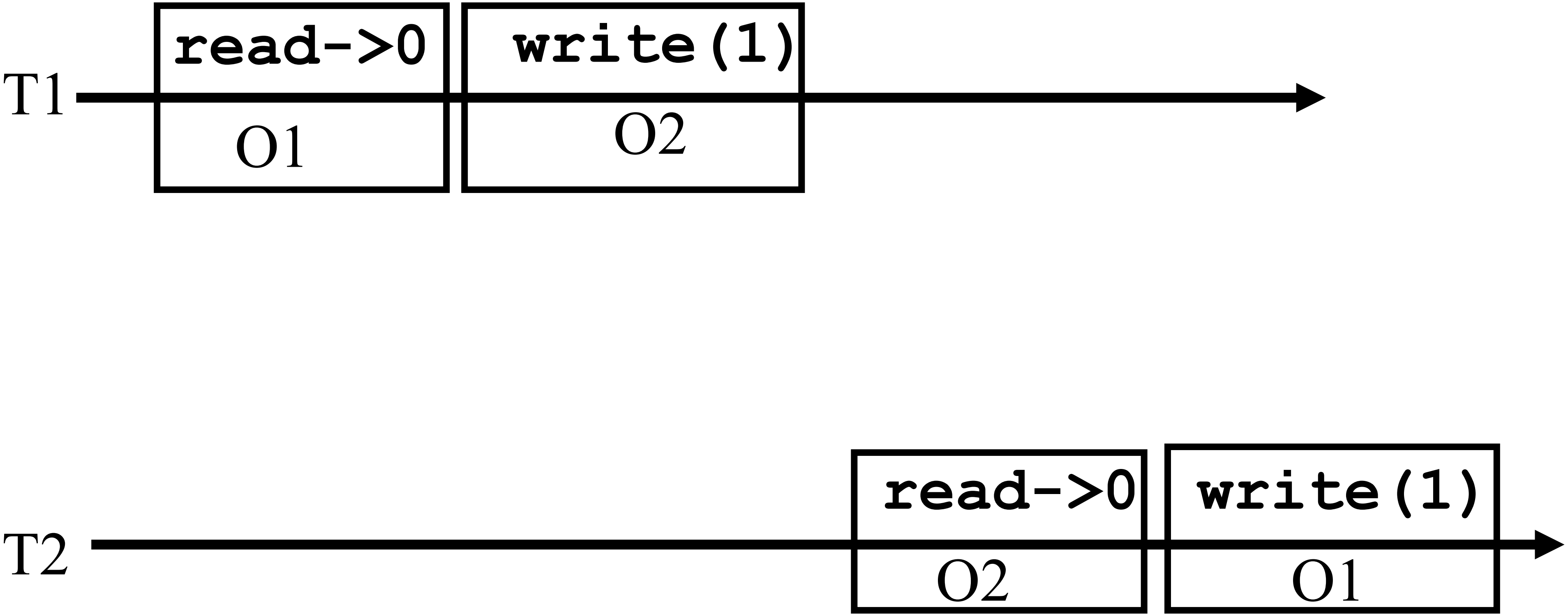
A history H of committed transactions is serializable if there is a history S(H) such that:

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

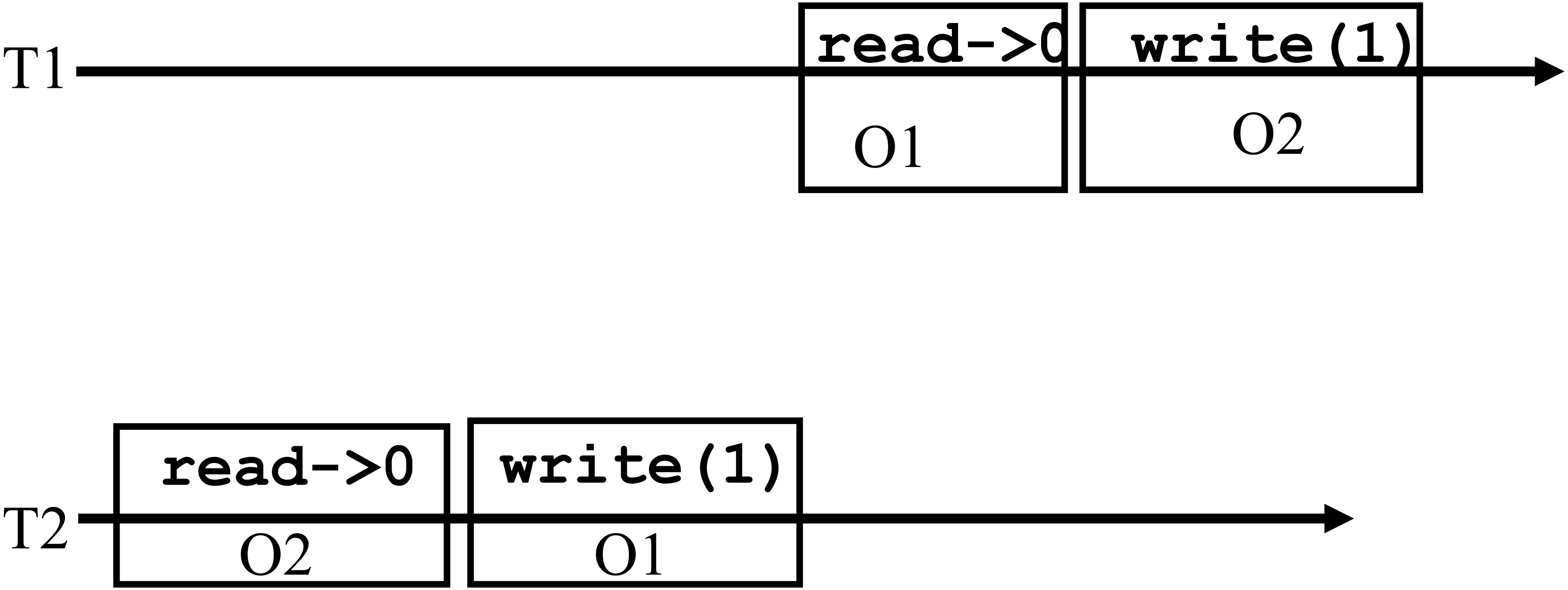
Atomic history?



Sequential history?



Sequential history?



T1

read->0

O1

write(1)

O2

T2

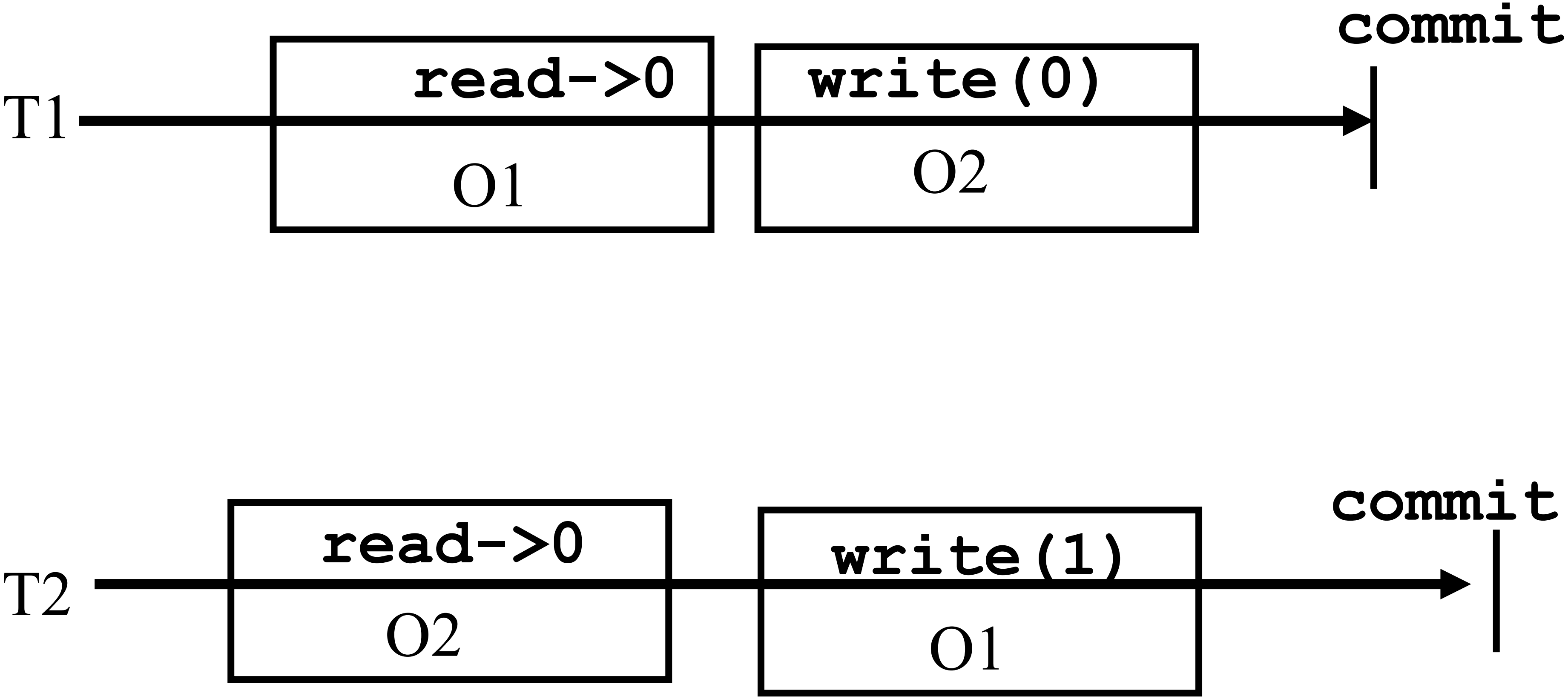
read->0

O2

write(1)

O1

Atomic history?



T1

`read->0`

O1

`write(0)`

O2

`commit`

T2

`read->0`

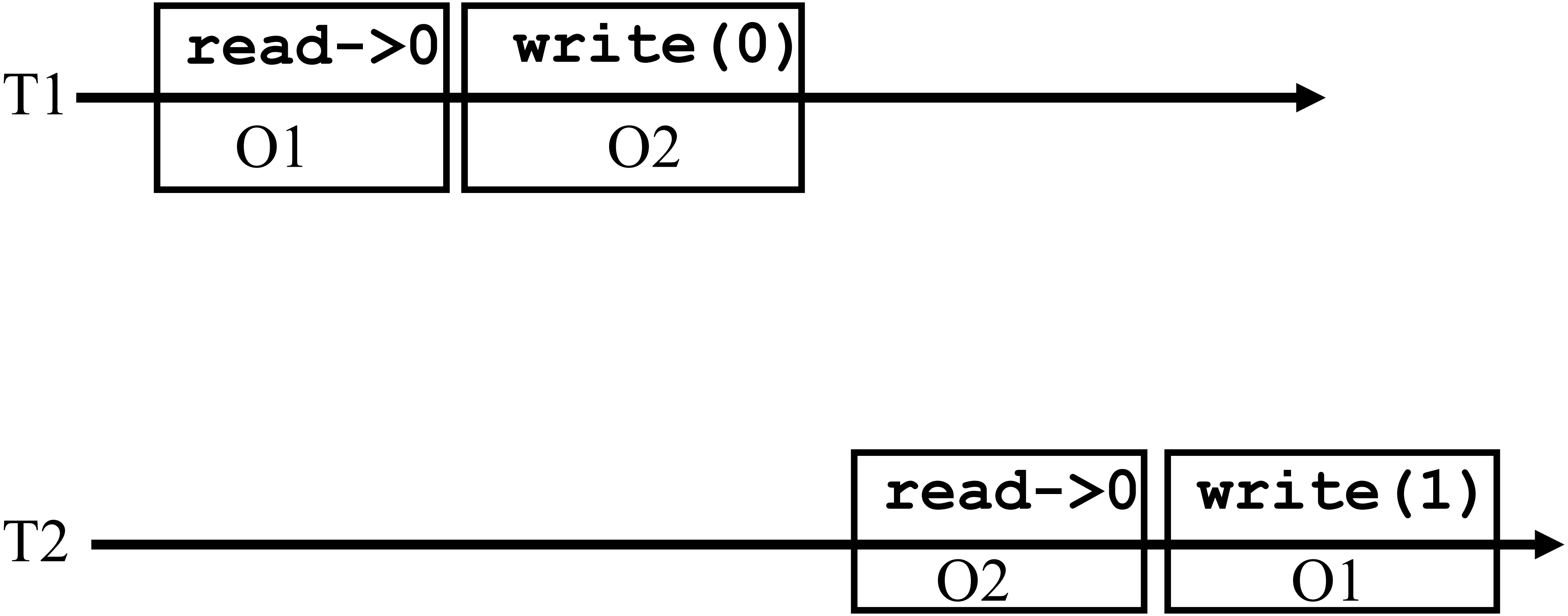
O2

`write(1)`

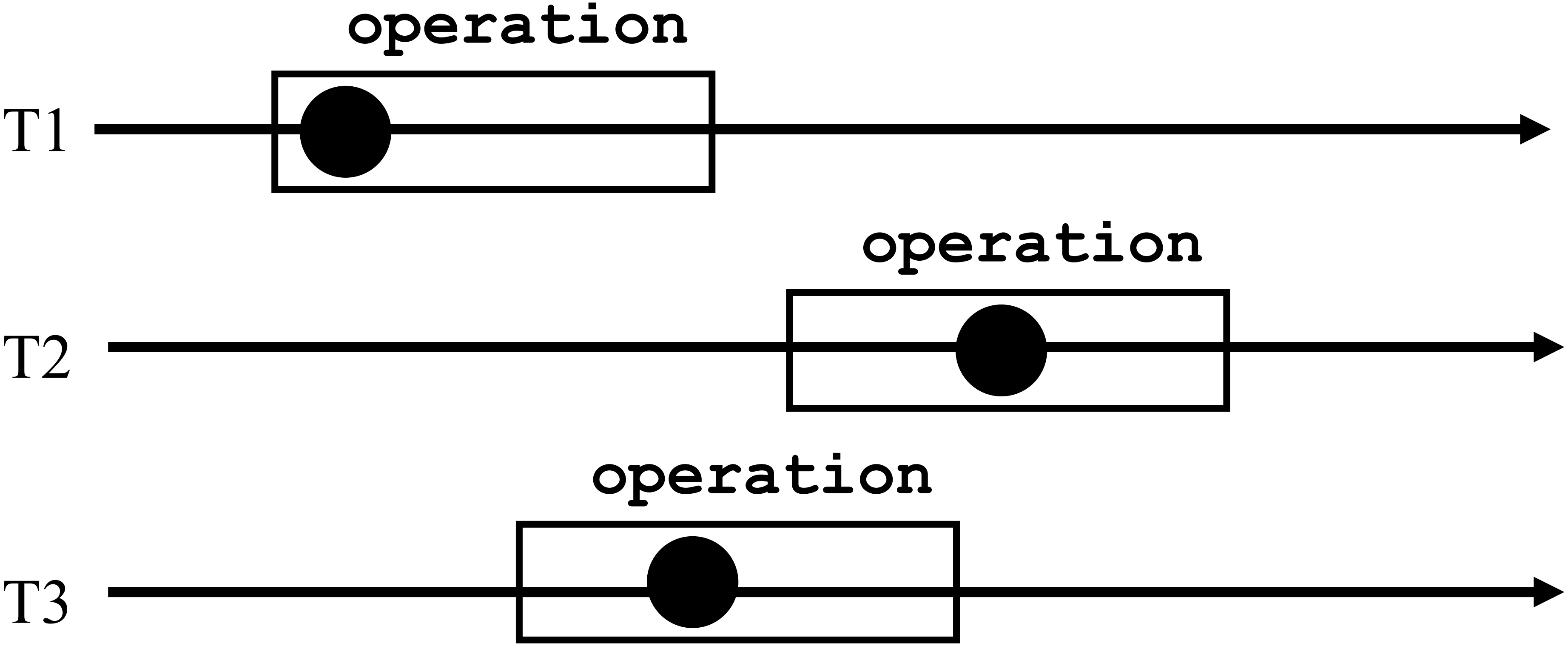
O1

`commit`

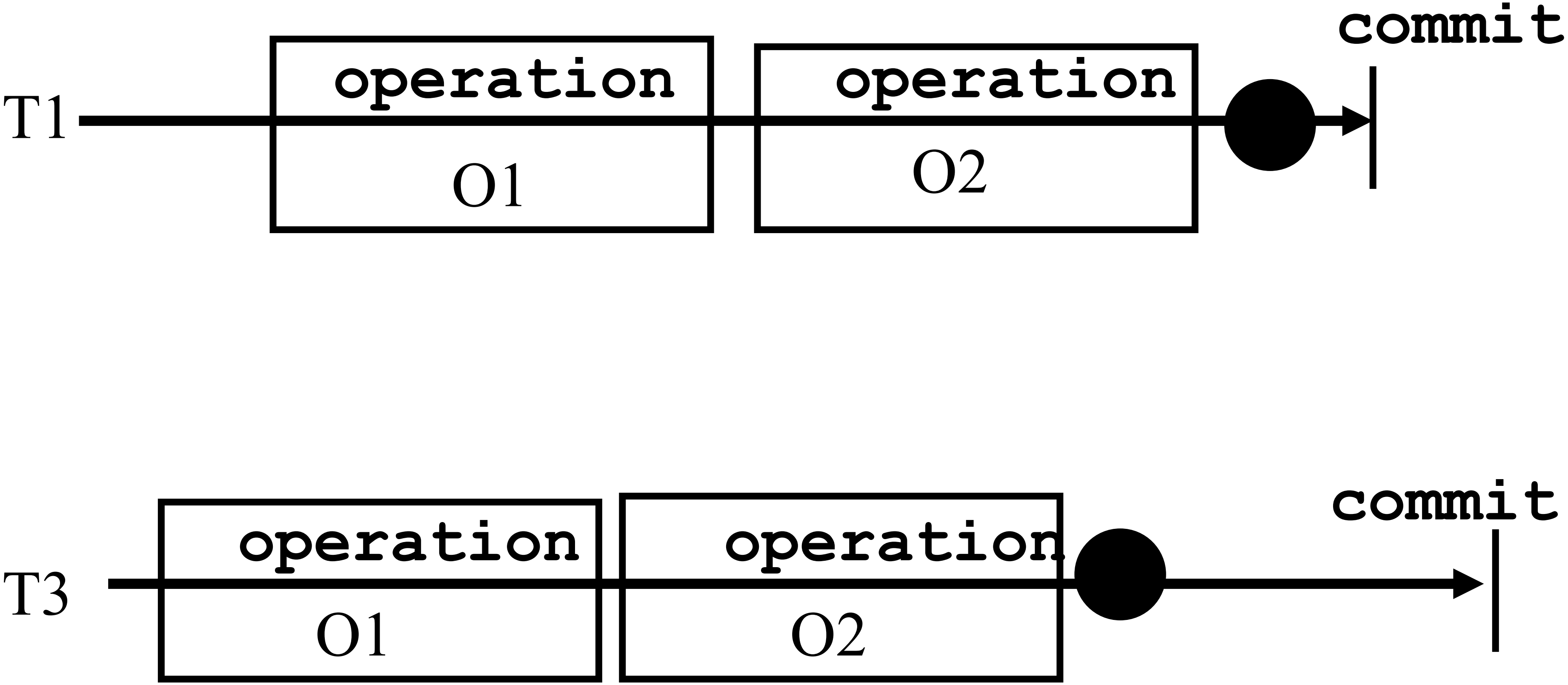
Sequential history



Operation atomicity (linearizability)



Transaction atomicity

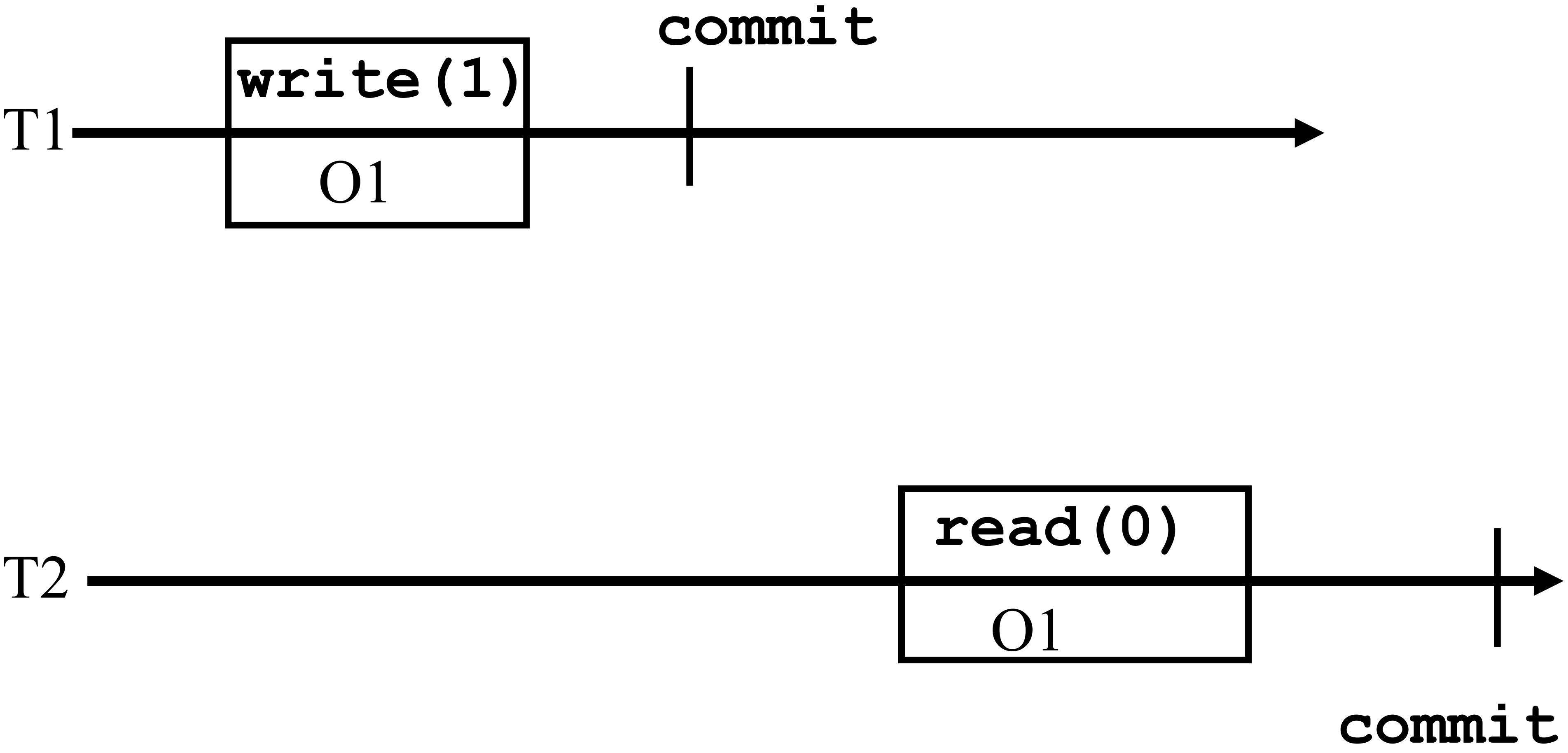


Serializability

- 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 written value

Real-time



commit

write(1)

T1

O1

read(0)

T2

O1

commit

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 S

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 tries to acquire ownership 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()
   (stat,curr)=getValue(owner,ov,nv)
   if stat=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]
   (stat,curr)=getValue(owner,ov,nv)
   if stat!= 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

try aborting the concurrent transaction

Grab the ownership on the object and set value v and old value curr

Check if all previously read objects keep the same values

Set status to committed

DSTM: getValue() and valid()

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

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

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

Example: run-time error

Initially: `x=1, y=2`
Invariant (sequential): `0<x<y`

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

But what if:

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

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

Example: infinite loop

```
T1:
      x := 3;
      y:= 6

T2:
      a := y;
      b:= x;
      repeat
        b:= b + 1;
      until a = b;
```

Quiz 1: unsafe transactions and ABA

- Sketch a simple strictly serializable TM implementation that exhibits histories with
  - ✓Division-by-zero exception
  - ✓Infinite loops
  - ✓Hint: take a “simplified” version of DSTM and run it with T1, T2 described in slides 34 and 35
- Is DSTM subject to the ABA problem?

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)

Histories

- Let H be any history (made of commited, aborted and *pending* transactions)

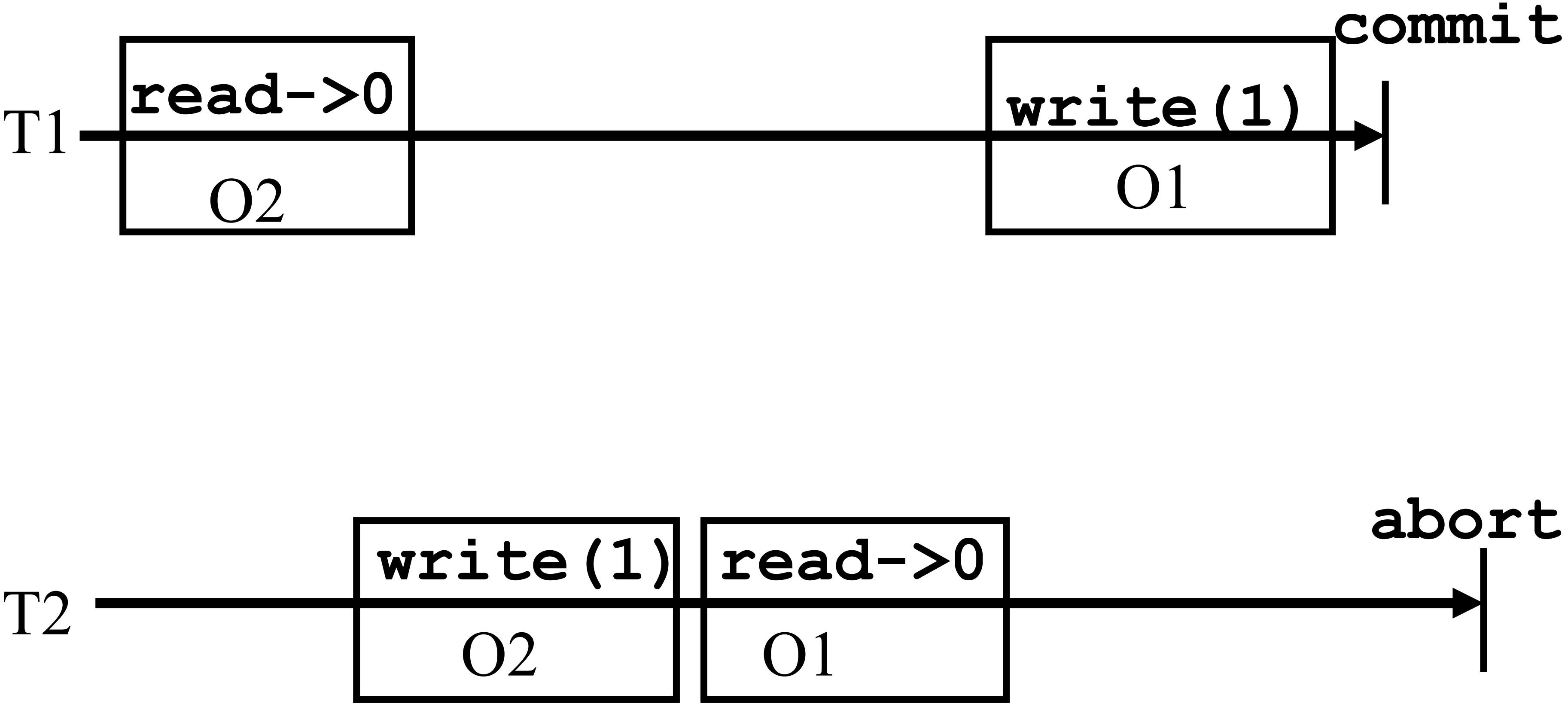
- Complete(H) is the history made of all transactions of H by completing pending ones with abort events
  - ✓And some of *pending commits* with commits

Opacity [GK'08]

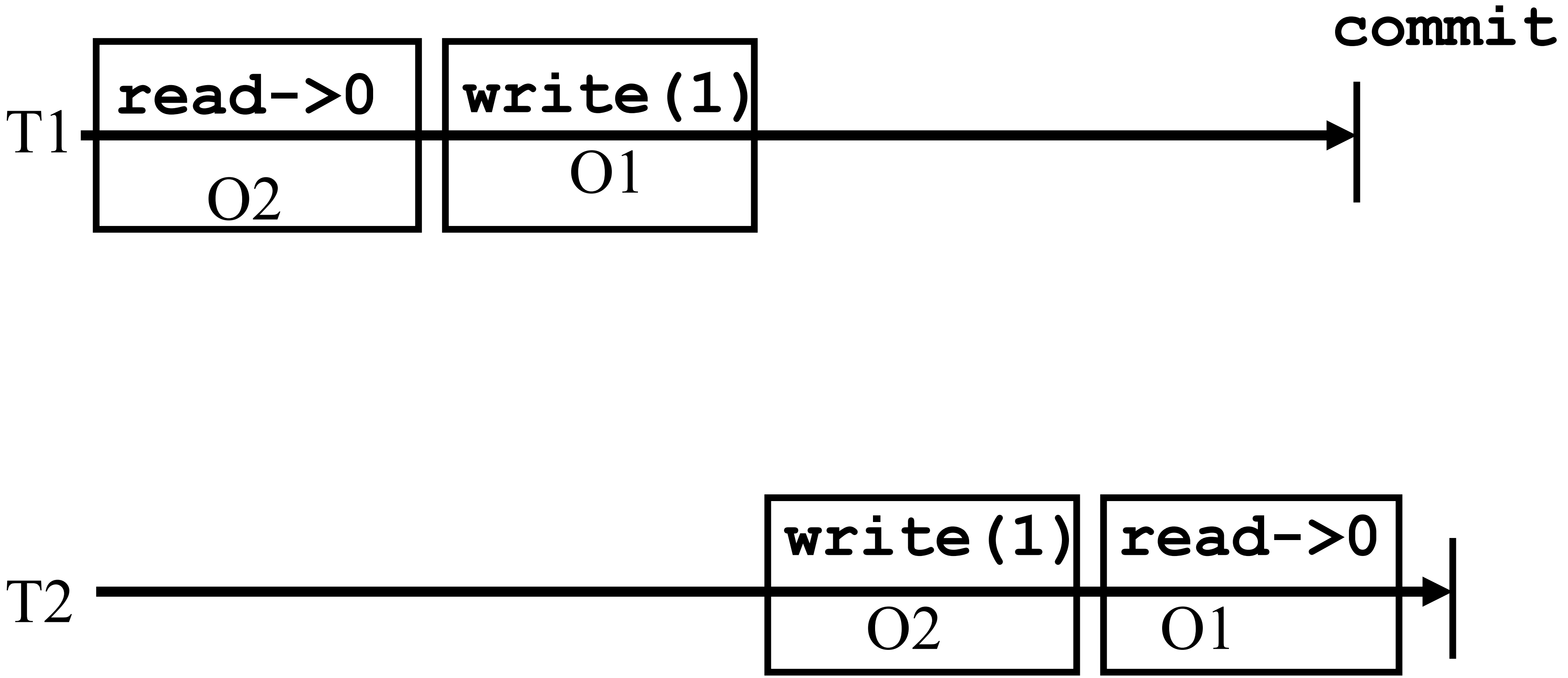
A history H of opaque if there is a history S such that:

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

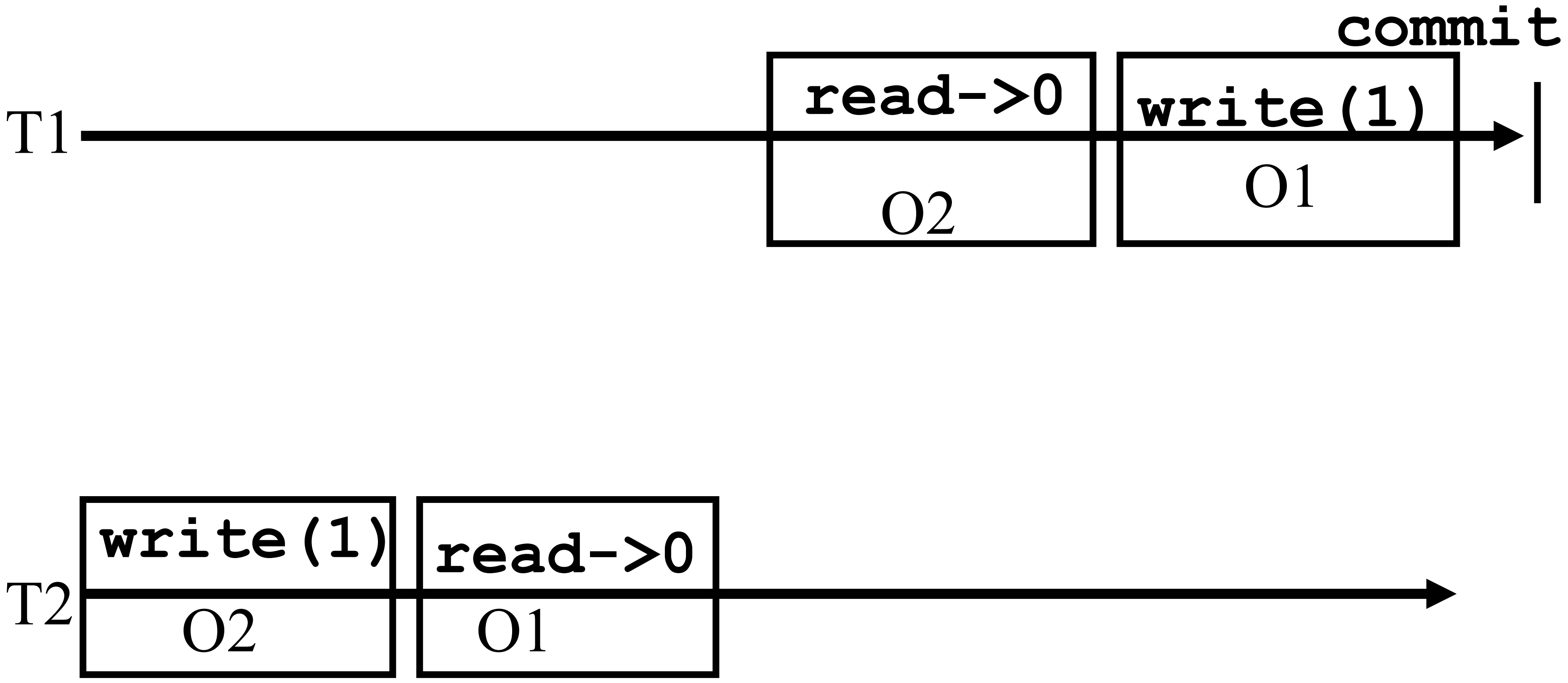
Opacity?



Not legal



Legal



Simple algorithm (DSTM)

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

Visible Read (SXM; RSTM)

- Write is mega killer: to write to an object O, a transaction aborts any live transaction which has read or written O

- Visible but not so careful read: when a transaction reads an object, it says so

Visible Read

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

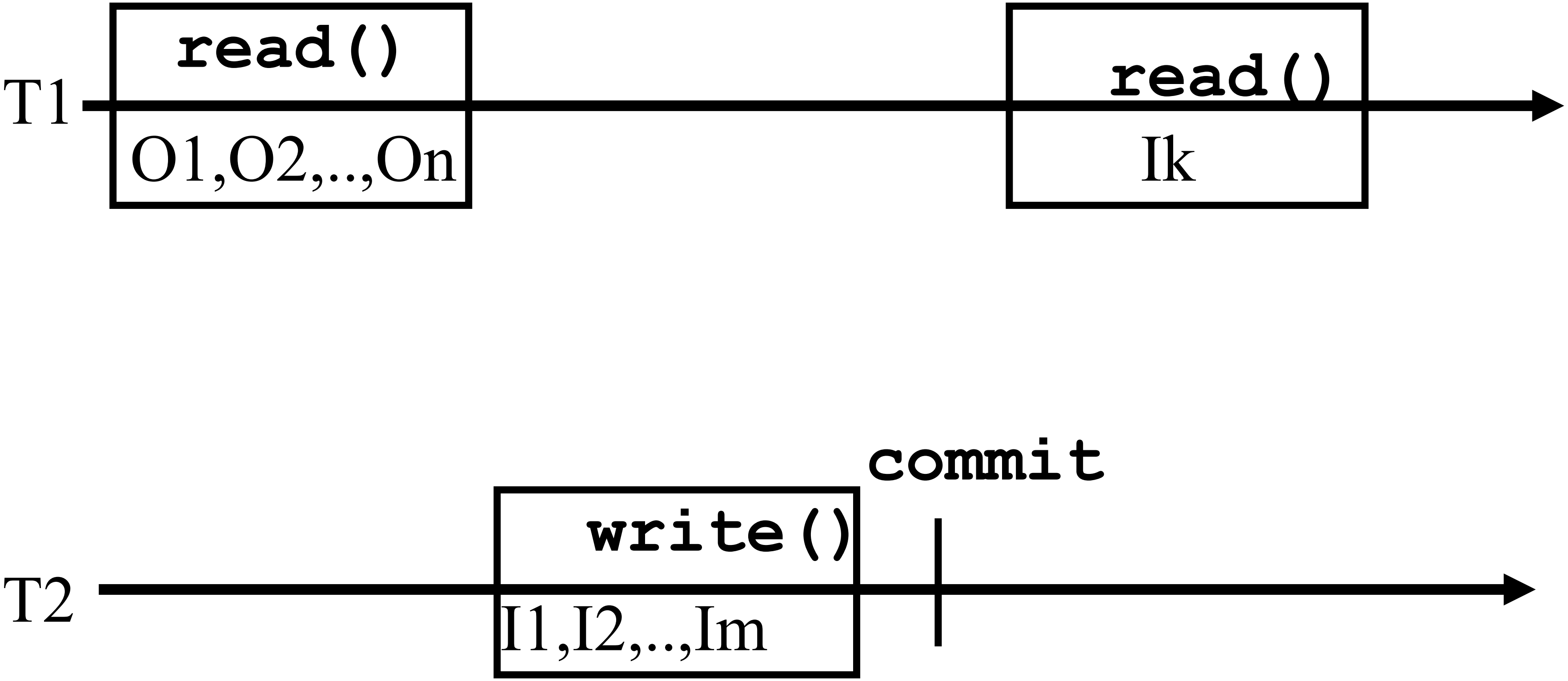
Unavoidable (in some sense)

Theorem [GK'08]

In an opaque TM, reads are either visible or careful

NB. Modulo the assumption of a single versions (at any moment, at most one value is stored for each object) and a weak progress property (progressiveness: commit if no read-write or write-write conflicts)

Intuition of the proof



T1

`read()`

O1,O2,..,On

`read()`

Ik

T2

`write()`

I1,I2,..,Im

`commit`

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!

Quiz 2: read visibility and validation

- Why does not the “visibility-validation” theorem hold for multi-versioned TMS maintaining multiple versions of each object

- Why does not the theorem hold for strictly serializable TMs?

Liveness and progress of a TM

What progress can we expect?

What is progress?

- Operations eventually return?
- Transactions eventually terminate?

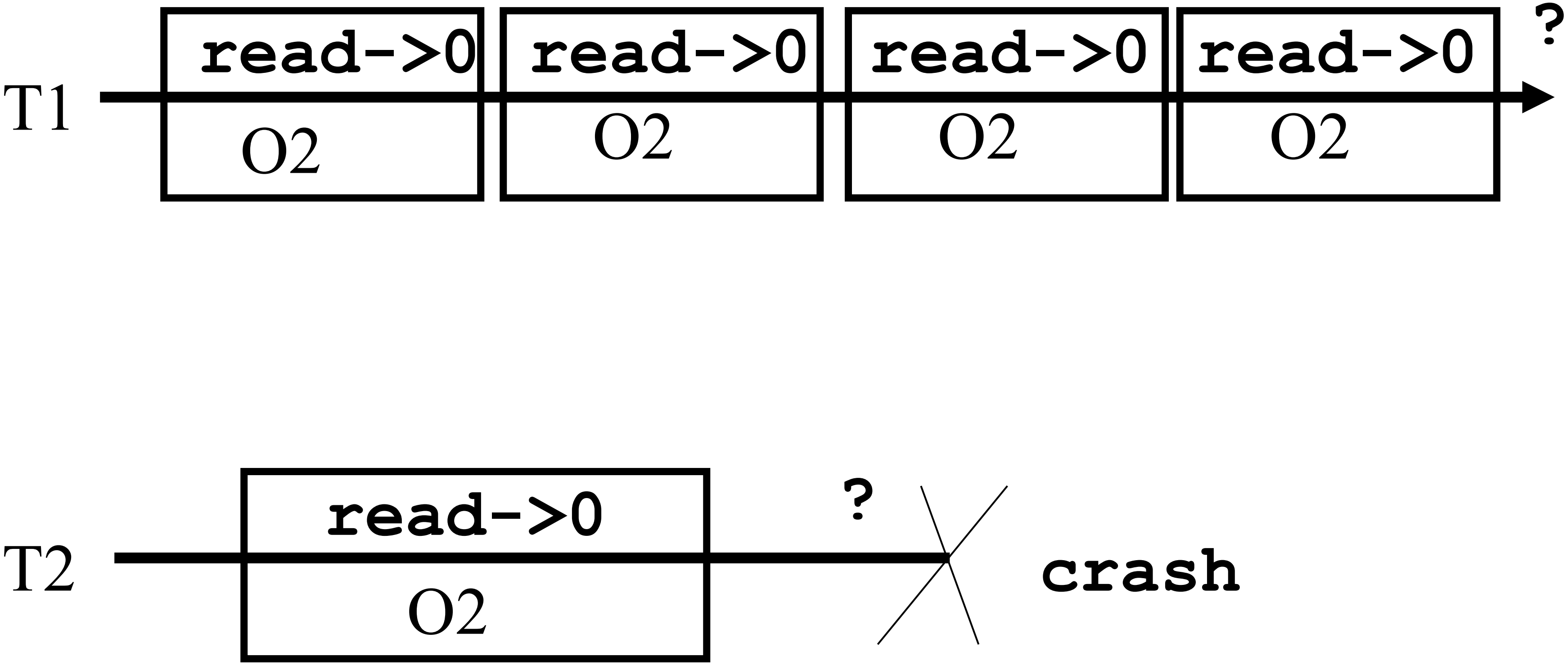
What is progress?

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

What is progress?

- We cannot require a TM to commits transactions:
  - ✓from a dead process, i.e., dead transactions
  - ✓that infinitely loop, i.e., never trying to commit

Progress?

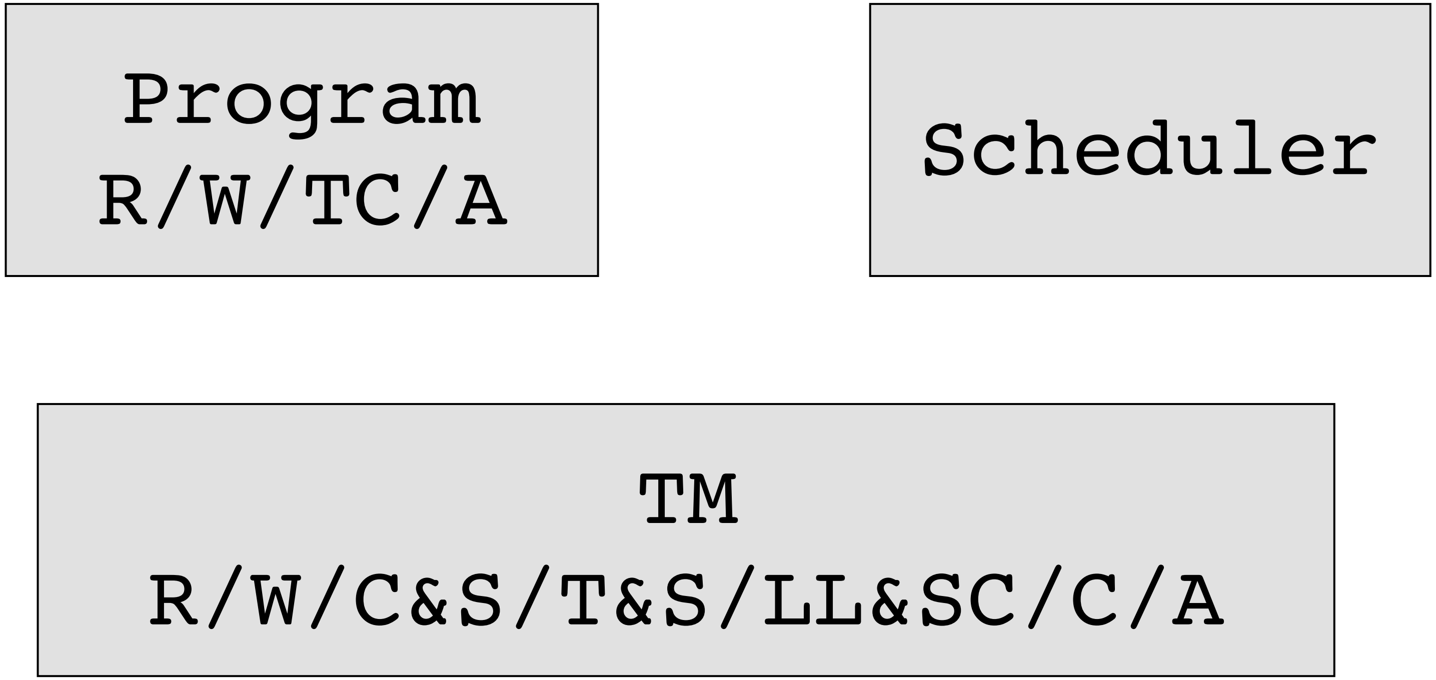


Progress

- We can only expect progress for correct transactions

- How to define a correct transaction?

Correctness depends on the scheduler and the program



Program
R/W/TC/A

Scheduler

TM
R/W/C&S/T&S/LL&SC/C/A

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

- Low-level history: a total order of invocation, response, try-commit, commit and abort events plus events of the implementation (steps)

Correct transactions in low-level histories

- 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

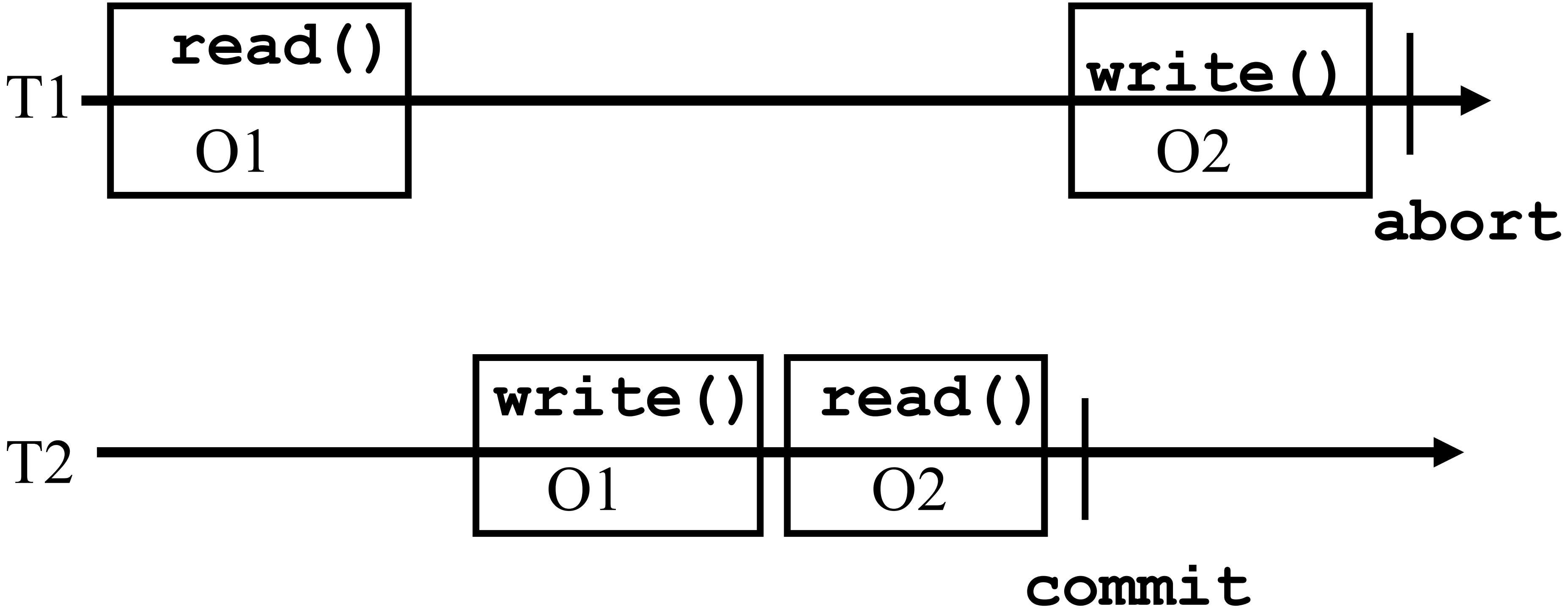
Ideal progress/liveness? Wait-freedom!

- No correct transaction ever aborts

- NB. This is not a liveness property, should be combined with
  - ✓Every operation executed by a correct transaction eventually returns

- Can we achieve this?
  - ✓No: even if we allow a correct transaction to abort finite number of times

Wait-free TM?



Wait-free TM?

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

Conditional progress/liveness? Obstruction-freedom

A correct transaction that not encounter step contention (no interleaving steps of other transactions) commits

- Obstruction-freedom: seems reasonable and indeed can be implemented

OF DSTM

- To write O, T tries to acquire ownership 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

DSTM: write, read, tryCommit

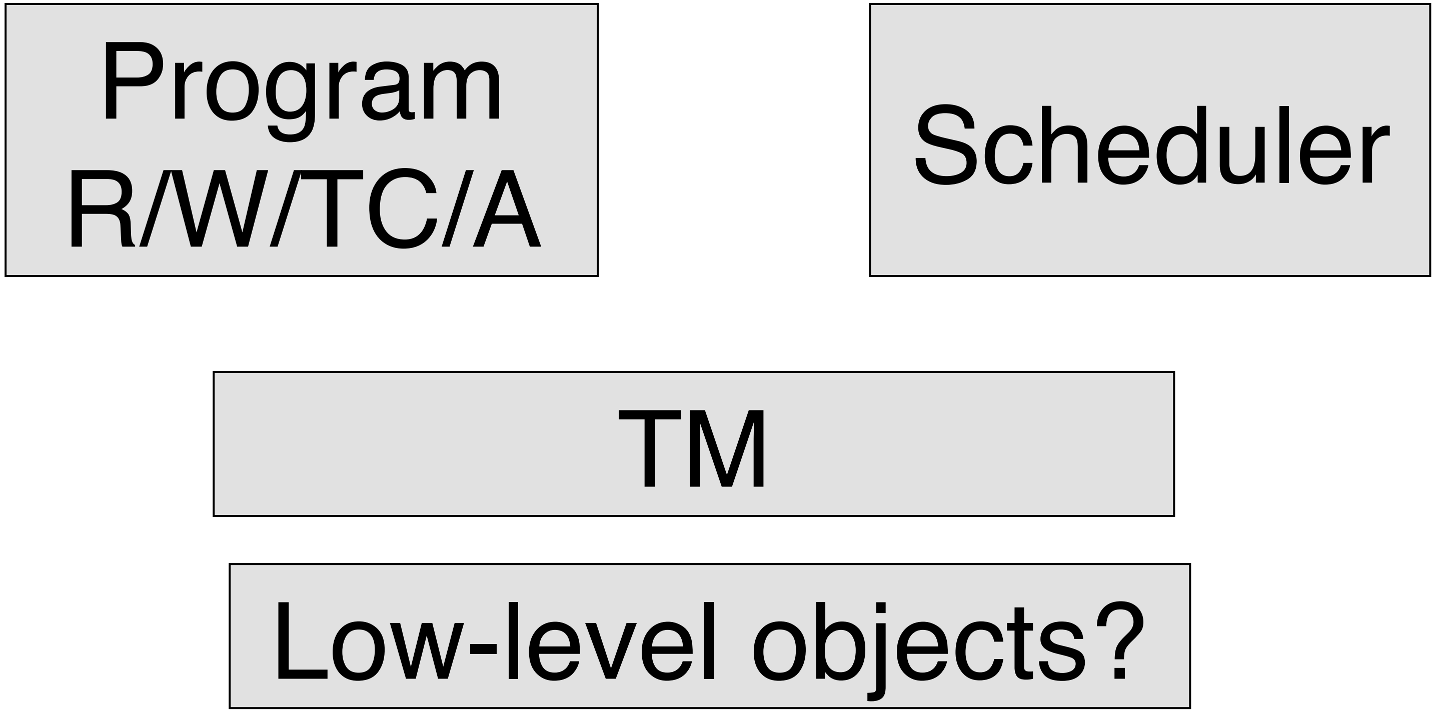
```
write(x,v)
   (owner,ov,nv)=tvar[x].read()
    curr=getValue(owner,ov,nv)
   if curr=live and !status[owner].cas(live,aborted) then return abort
   if tvar[x].cas([owner,ov,nv],[myself,curr,v]) then
         return ok
   else
         return abort


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

DSTM uses CAS

- CAS is the strongest synchronization primitive
  - Is OFTM possible with R/W objects?

OF-TM



Program
R/W/TC/A

Scheduler

TM

Low-level objects?

Consensus number of OF-TM?

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

FO-consensus

A process can decide or abort

- No two different values can be decided
- A value decided was proposed by a non-aborted process

- If abort is returned from *propose(v)* then there is step contention

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 CAS

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

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!

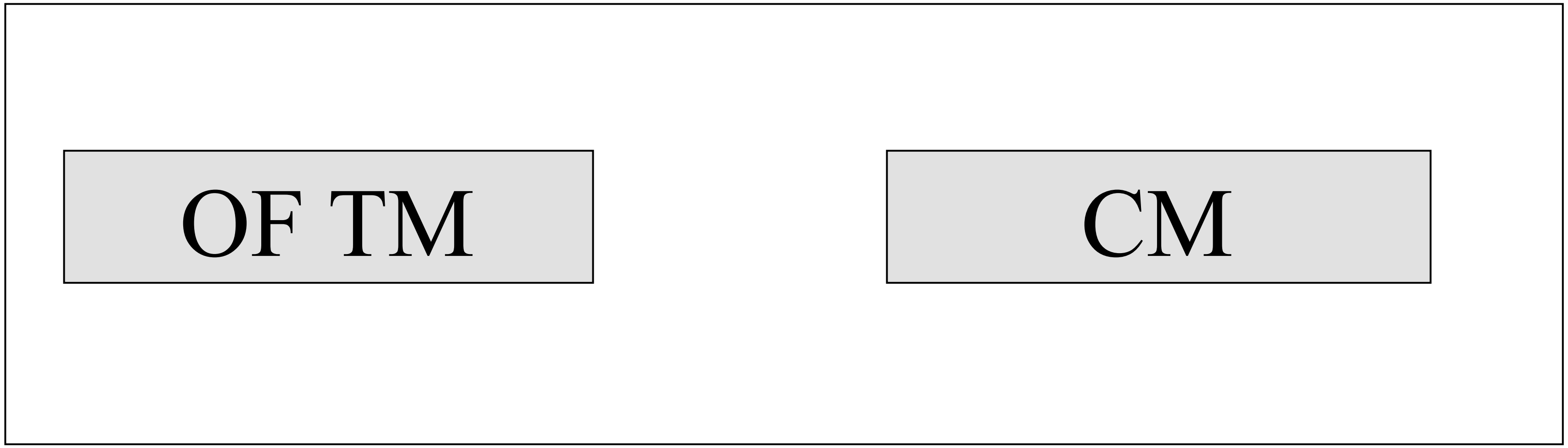
OF-TM vs. OF objects

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

TM Liveness

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

Boosting OF?



Contention management

- Conflict resolution delegated to a **contention manager**

- Responsible solely for progress (liveness)

(different from a DB concurrency control)

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’

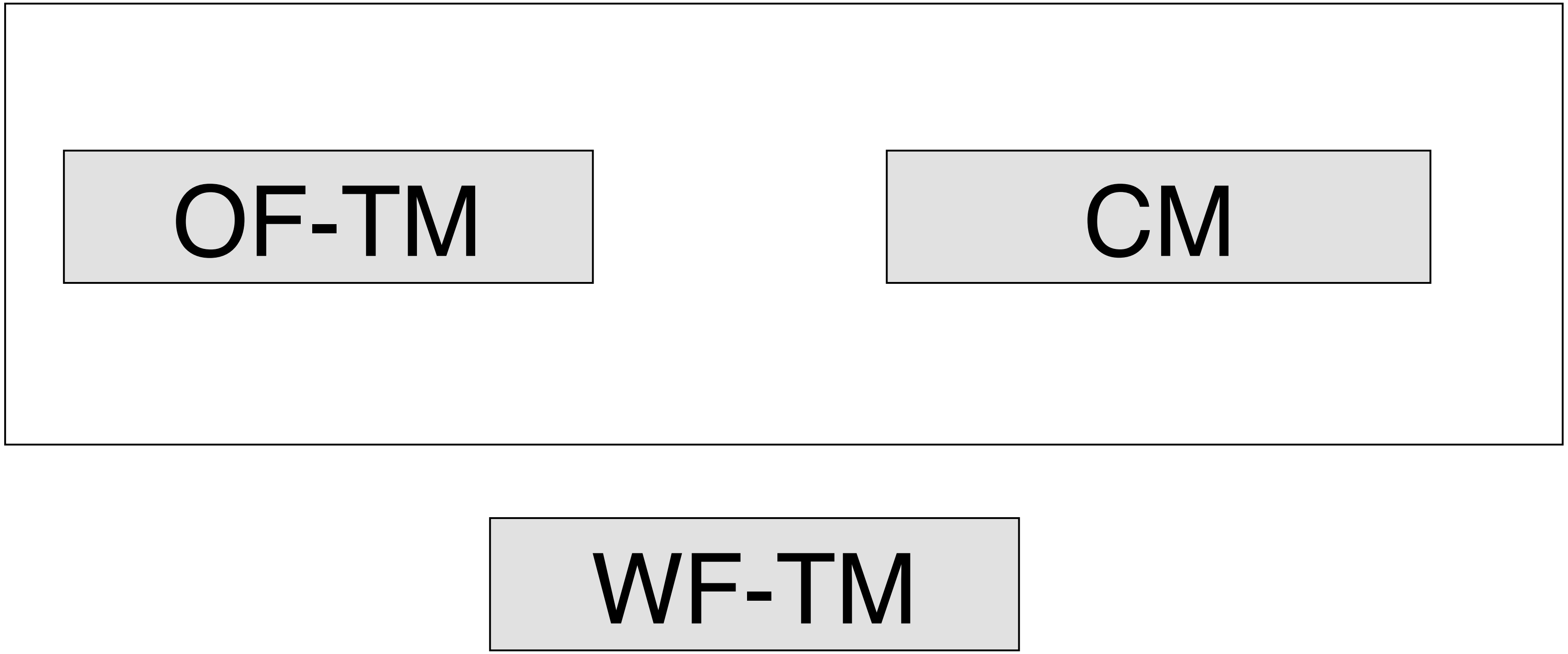
Contention managers

- **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

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

From OF to WF



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

Quiz 3: TM progress and liveness

- Why “no correct transaction ever aborts” is not a liveness property?
- Prove correctness of the consensus algorithm using OF-consensus

Why do we care?

- Modern computing is concurrent
- TM promises simplicity and efficiency

What is it?

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

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

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