

# ***Transactional Memory***

***R. Guerraoui, EPFL***

***Locking is "history"***

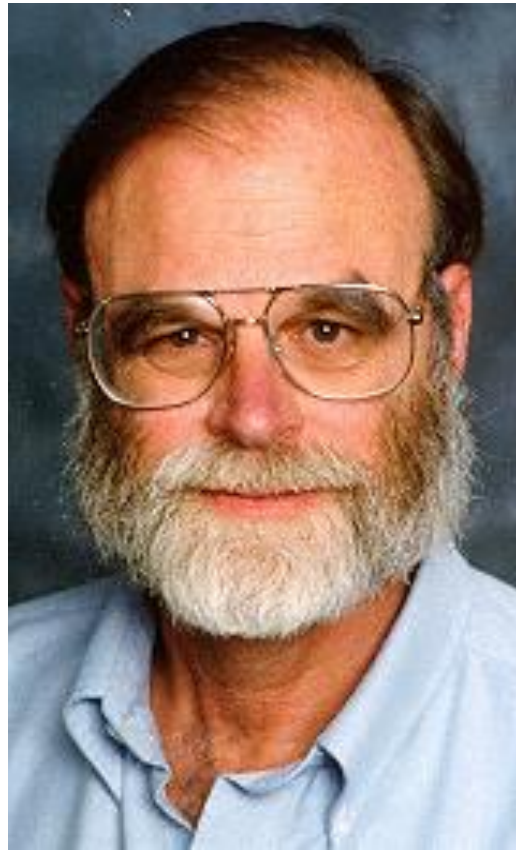
***Lock-freedom is "difficult"***

***Wanted***



***A synchronisation abstraction that  
is simple, robust and efficient***

# *Transactions*



# *Historical perspective*

- ☞ Eswaran et al (CACM'76) Databases
- ☞ Papadimitriou (JACM'79) Theory
- ☞ Liskov/Sheifler (TOPLAS'82) Language
- ☞ Knight (ICFP'86) Architecture
- ☞ Herlihy/Moss (ISCA'93) Hardware
- ☞ Shavit/Touitou (PODC'95) Software
- ☞ Herlihy et al (PODC'03) Software – Dynamic

## ***Back to the sequential level***

- ☞ accessing object 1;
- ☞ accessing object 2;

# ***Back to the sequential level***

*atomic* {

☛ accessing object 1;

☛ accessing object 2;

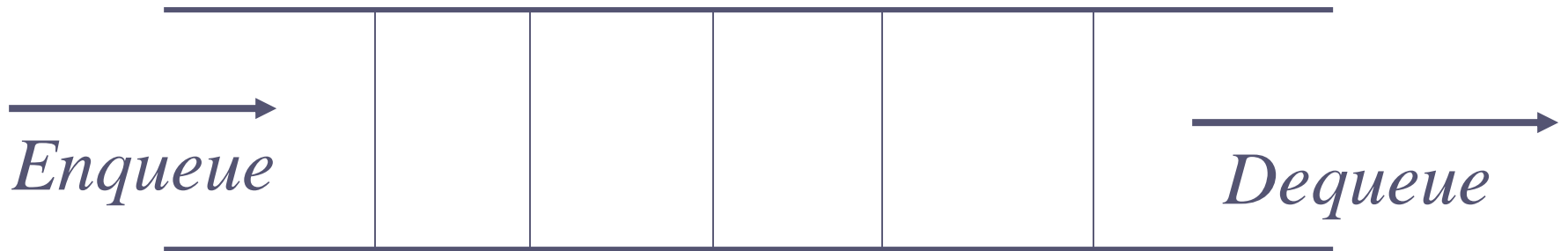
}

# ***Semantics (serialisability)***

Every transaction appears to execute  
at an indivisible point in time  
(linearizability of transactions)

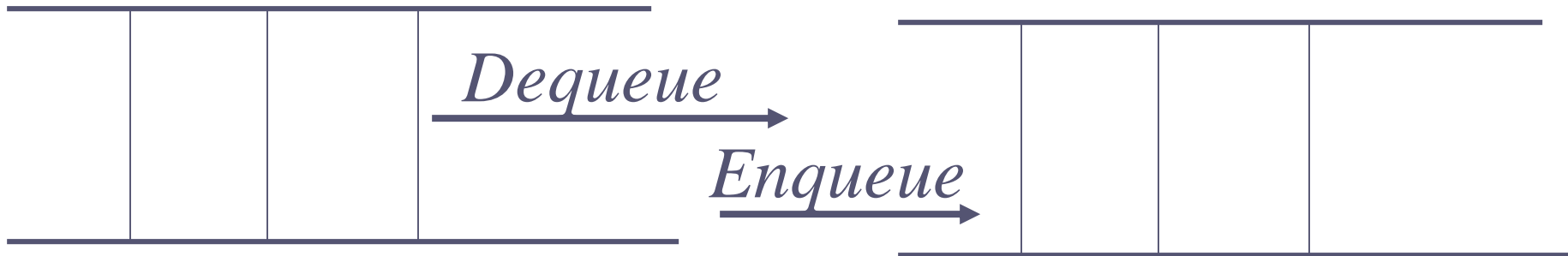


# ***Double-ended queue***



```
class Queue {
    QNode head;
    QNode tail;
    public enq(Object x) {
        atomic {
            QNode q = new QNode(x);
            q.next = head;
            head = q;
        }
    }
    ... }
```

# *Queue composition*



```
class Queue {  
    ...  
    public transfer(Queue q) {  
        atomic {  
            Qnode n = this.dequeue();  
            q.enqueue(n) }  
        }  
        ... }  
}
```

***Simple example***  
(consistency invariant)

$$0 < x < y$$

# *Simple example* (transaction)

☞ T:  $x := x+1 ; y := y+1$

# *The illusion of a critical section*

*atomic {*

☛ accessing object 1;

☛ accessing object 2;

*}*

☛ *“It is better for Intel to get involved in this [Transactional Memory] now so when we get to the point of having ...tons... of cores we will have the answers”*

☛ *Justin Rattner, Intel Chief Technology Officer*



☛ “...we need to explore new techniques like **transactional memory** that will allow us to get the full benefit of all those transistors and map that into higher and higher performance.”

● **Bill Gates**

☞ “...manual synchronization is intractable...**transactions** are the only plausible solution....”

● **Tim Sweeney, Epic Games**

# ***The TM Topic has been a VERY HOT topic***

- ☛ Sun/Oracle, Intel, AMD, IBM, MSR
- ☛ Fortress (Sun); X10 (IBM); Chapel (Cray)

# ***The TM API***

## ***(a simple view)***

- ***begin()*** returns *ok*
- ***read()*** returns a value or *abort*
- ***write()*** returns an *ok* or *abort*
- ***commit()*** returns *ok* or *abort*
- ***abort()*** returns *ok*

# Two-phase locking

- To *write* or *read*  $O$ ,  $T$  requires a *lock* on  $O$ ;  $T$  *waits* if some  $T'$  acquired a *lock* on  $O$
- At the end,  $T$  *releases* all its locks

# Two-phase locking (more details)

- Every object  $O$ , with state  $s(O)$  (a *register*), is protected by a lock  $l(O)$  (a **c&s**)
- Every transaction has local variables  $wSet$  and  $wLog$
- Initially:  $l(O) = \text{unlocked}$ ,  $wSet = wLog = \emptyset$

# Two-phase locking

Upon op = *read()* or *write(v)* on object O

if  $O \notin wSet$  then

wait until unlocked = l(O).c&s(unlocked,locked)

wSet = wSet U O

wLog = wLog U S(O).read()

if op = read() then return S(O).read()

S(O).write(v)

return ok

# Two-phase locking (cont'd)

Upon *commit()*

cleanup()

return ok

Upon *abort()*

rollback()

cleanup()

return ok



# Two-phase locking (cont'd)

Upon *rollback()*

for all  $O \in wSet$  do  $S(O).write(wLog(O))$

$wLog = \emptyset$

Upon *cleanup()*

for all  $O \in wSet$  do  $I(O).write(unlocked)$

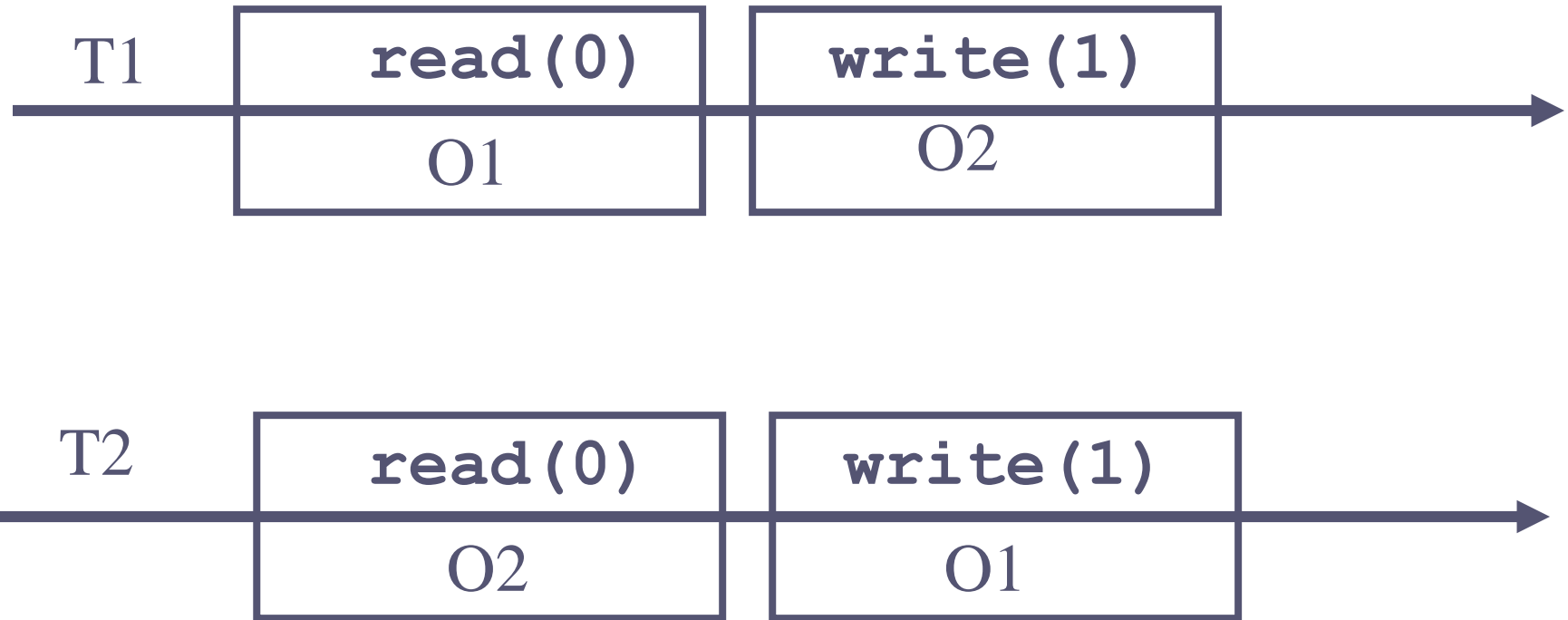
$wSet = \emptyset$

# Why two phases? (what if?)

• To *write* or *read* O, T requires a *lock* on O;  
T *waits* if some T' acquired a *lock* on O

• T *releases* the lock on O when T is done with O

# *Why two phases?*



# Two-phase locking (read-write lock)

- ☛ To *write*  $O$ ,  $T$  requires a *write-lock* on  $O$ ;  
 $T$  *waits* if some  $T'$  acquired a *lock* on  $O$
- ☛ To *read*  $O$ ,  $T$  requires a *read-lock* on  $O$ ;  
 $T$  *waits* if some  $T'$  acquired a *write-lock* on  $O$
- ☛ Before committing,  $T$  *releases* all its locks

# ***Two-phase locking*** ***- better dead than wait -***

- ☛ To ***write*** O, T requires a ***write-lock on*** O;  
T ***aborts*** if some T' acquired a ***lock*** on O
- ☛ To ***read*** O, T requires a ***read-lock*** on O;  
T ***aborts*** if some T' acquired a ***write-lock*** on O
- ☛ Before committing, T releases all its locks
- ☛ A transaction that aborts restarts again

# ***Two-phase locking***

## ***- better kill than wait -***

- ☛ To ***write*** O, T requires a ***write-lock on O***;  
T ***aborts T'*** if some T' acquired a ***lock*** on O
- ☛ To ***read*** O, T requires a ***read-lock*** on O;  
T ***aborts T'*** if some T' acquired a ***write-lock*** on O
- ☛ Before committing, T releases all its locks
- ☛ A transaction that is aborted restarts again

# ***Two-phase locking*** ***- better kill than wait -***

- ☛ To ***write*** O, T requires a ***write-lock on O***;  
T ***aborts T'*** if some T' acquired a ***lock*** on O
- ☛ To ***read*** O, T requires a ***read-lock*** on O;  
T ***waits*** if some T' acquired a ***write-lock*** on O
- ☛ Before committing, T releases all its locks
- ☛ A transaction that is aborted restarts again

# ***Visible Read***

(SXM, RSTM, TLRW)

- ☛ ***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



# *Visible Read*

- ☛ A visible read invalidates cache lines
- ☛ For read-dominated workloads, this means a lot of traffic on the bus between processors
  - This reduces the throughput
  - Not a big deal with single-CPU, but with many core machines (e.g. SPART T2 Niagara)

# ***Two-phase locking with invisible reads***

- ☛ To ***write*** O, T requires a ***write-lock on*** O;  
T **waits** if some T' acquired a ***write-lock*** on O
- ☛ To ***read*** O, T checks if ***all objects read remain valid*** - else T **aborts**
- ☛ Before committing, T checks if all objects read remain valid and releases all its locks

# Invisible reads (more details)

- Every object  $O$ , with state  $S(O)$  (register), is protected by a lock  $l(O)$  (c&s)
- Every transaction maintains, besides  $wSet$  and  $wLog$ :
  - a local variable  $rset(O)$  for every object

# Invisible reads

Upon *write(v)* on object O

if  $O \notin wSet$  then

    wait until unlocked = I(O).c&s(unlocked,locked)

    wSet = wSet U O

    wLog = wLog U S(O).read()

    (\*,ts) = S(O).read()

    S(O).write(v,ts)

    return ok

# Invisible reads

Upon *read()* on object  $O$

$(v, ts) = S(O).read()$

if  $O \in wSet$  then return  $v$

if  $l(O) = \text{locked}$  or  $\text{not validate}()$  then  $\text{abort}()$

if  $rset(O) = 0$  then  $rset(O) = ts$

return  $v$

# Invisible reads

Upon *validate()*

for all  $O$  s.t  $rset(O) > 0$  do

$(v, ts) = S(O).read()$

    if  $ts \neq rset(O)$  or

$(O \notin wset \text{ and } l(O) = \text{locked})$

then return false

else return true

# Invisible reads

Upon *commit()*  
if not validate() then abort()  
for all  $O \in wset$  do  
     $(v, ts) = S(O).read()$   
     $S(O).write(v, ts+1)$   
cleanup()

# Invisible reads

Upon *rollback()*

for all  $O \in wSet$  do  $S(O).write(wLog(O))$

$wLog = \emptyset$

Upon *cleanup()*

for all  $O \in wset$  do  $I(O).write(unlocked)$

$wset = \emptyset$

$rset(O) = 0$  for all  $O$



# ***DSTM (SUN)***

- ☛ To ***write*** O, T requires a ***write-lock on O***; T aborts T' if some T' acquired a ***write-lock*** on O
- ☛ To ***read*** O, T checks if all objects read remain valid – else T **abort**
- ☛ Before committing, T releases all its locks

# ***DSTM***

☛ ***Killer write*** (ownership)

☛ ***Careful read*** (validation)

# *More efficient algorithm?*

Apologizing versus asking permission

- *Killer write*

- *Optimistic read*

  - validity check only at commit time

# *Example*

Invariant:  $0 < x < y$

Initially:  $x := 1; y := 2$

# *Division by zero*

☞ T1:  $x := x+1 ; y := y+1$

☞ T2:  $z := 1 / (y - x)$

# *Infinite loop*

☛ T1:  $x := 3; y := 6$

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

# Opacity

- Serializability

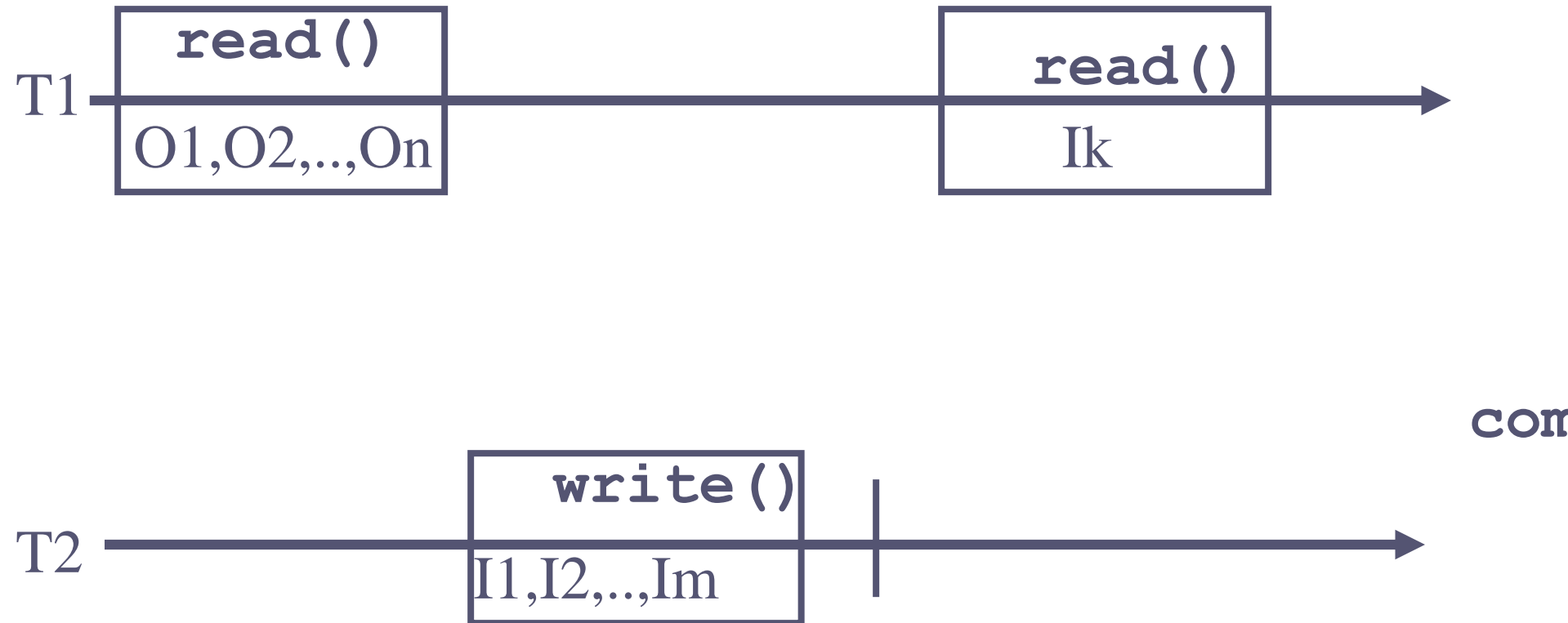
- Consistent memory view

# ***Trade-off***

The read is either  
***visible*** or ***careful***



# *Intuition*



# *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 multiversions: T2 would not have written "on" T1

# ***Conditional progress*** ***- obstruction-freedom -***

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

# ***DSTM***

- ☛ To ***write*** O, T requires a ***write-lock on*** O (use C&S); T aborts T' if some T' acquired a ***write-lock*** on O (use C&S)
- ☛ To ***read*** O, T checks if all objects read remain valid - else abort (use C&S)
- ☛ Before committing, T releases all its locks (use C&S)

# *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)

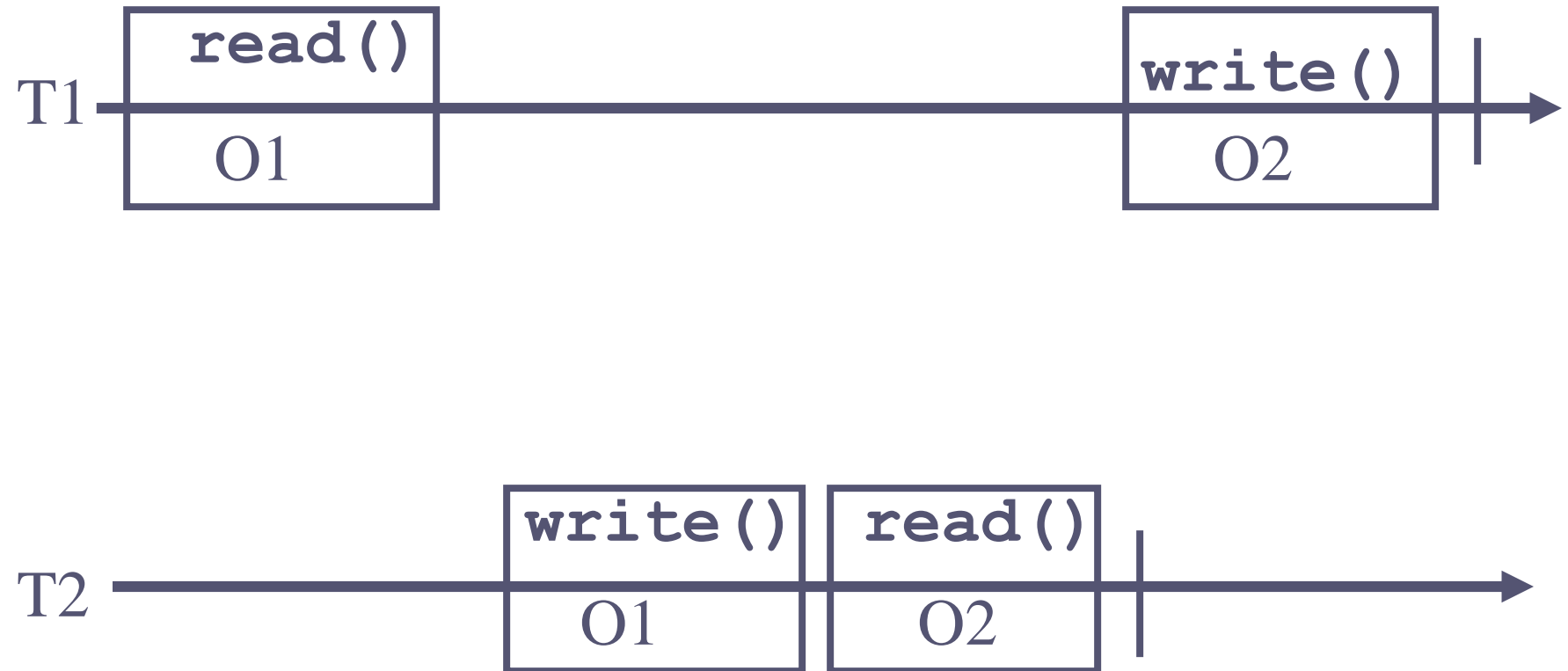
## ☞ **Wait** if other has

- Higher priority AND not waiting

## ☞ **Abort** other if

- Lower priority OR waiting

# *Aborting is a fatality*





## *Concluding remarks*

TM does not always replace locks:  
it hides them

Memory transactions look like db  
transactions but are different

# ***The garbage-collection analogy***

- ☛ In the early times, the programmers had to take care of allocating and de-allocating memory
- ☛ Garbage collectors do it for you: they are now incorporated in Java and other languages
- ☛ Hardware support was initially expected, but now software solutions are very effective



MORGAN & CLAYPOOL PUBLISHERS

# Principles of Transactional Memory

Rachid Guerraoui  
Michał Kapalka

*SYNTHESIS LECTURES ON  
DISTRIBUTED COMPUTING THEORY*

*Nancy Lynch, Series Editor*