Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Information Systems
Vienna University of Technology (TU Wien)

The power of registers (Chap. 4)

Hardware resource ("register" for some historical reasons; not every memory location might have the desired properties) or just memory location that can be read and written by different threads

Single/Multiple Readers/Writers

- SRSW
- MRSW
- SRMW
- MRMW

M-valued register: Can store values 0, … M-1. Special case: Boolean register, two values (**true/false**)

Mutual exclusion constructions have assumed "atomic" MRMW registers: Totally ordered, instantaneous read/write "events"

Question: Are "atomic" registers reasonable? Is it possible to construct stronger registers out of weaker ones? How weak can the weaker registers be?

Plan: Construct atomic registers out of weak ones such that operations (intervals) look like events

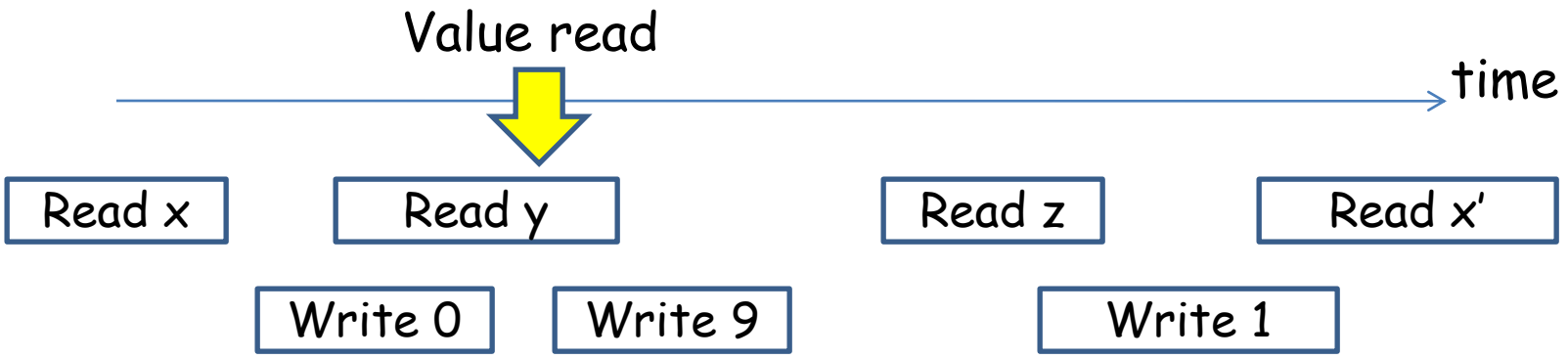
Constructions must be wait-free: Any thread can complete any read/write in a finite (bounded) number of own operations, independently of the operations of other threads

Sequential register:
A single thread performs read and write operations. There is trivially a total order on the operations: Program order. Each read operation returns the value written by the most recent write operation

Write 0 | Read x | Write 7 | Write 9 | Read y

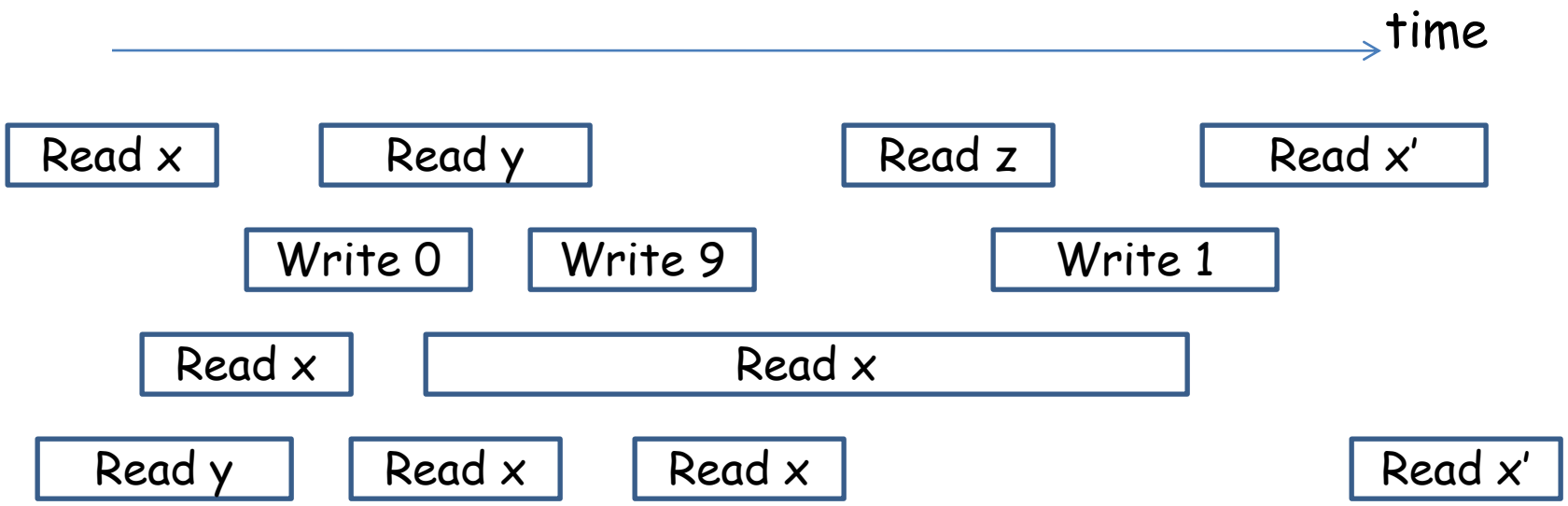
x=0, y=9 (not y=7, not y=0)

SRSW: A single thread reads, and a(nother) single thread writes; but reads and writes can overlap (partial order between read and write operations)



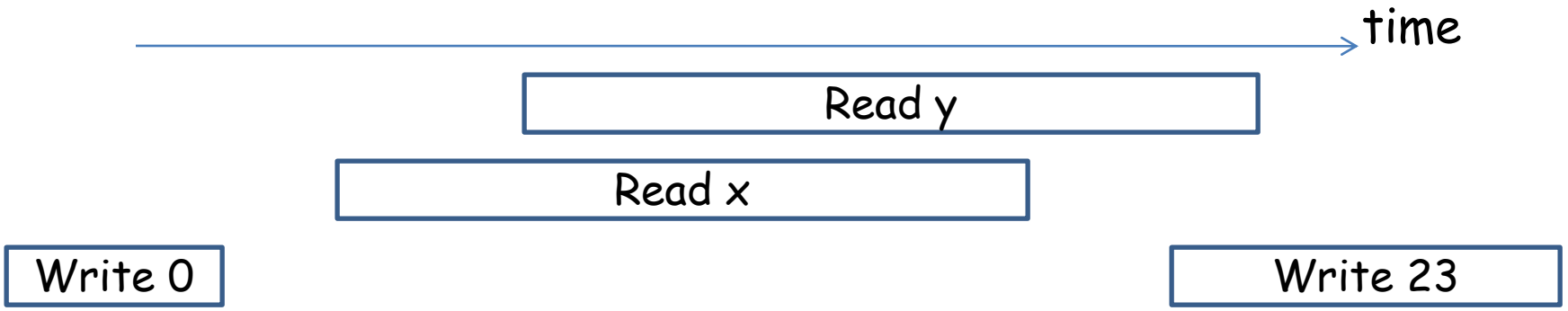
Read x does not overlap any write; read y etc. overlap write's

MRSW: Many threads read, and a single thread writes; and reads and writes can overlap (partial order on read and write operations)



Definition:

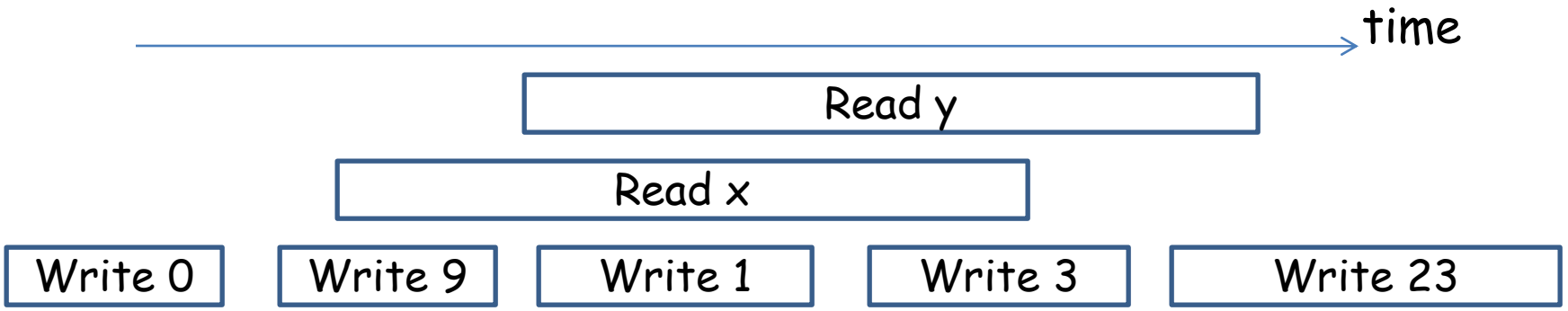
A MRSW is safe if a read interval that does not overlap a write interval returns the most recently written value; a read interval that is concurrent with a write interval may return any value in the register's range 0, …, M-1



x=0, y=?? because Read y overlaps Write 23

Definition:

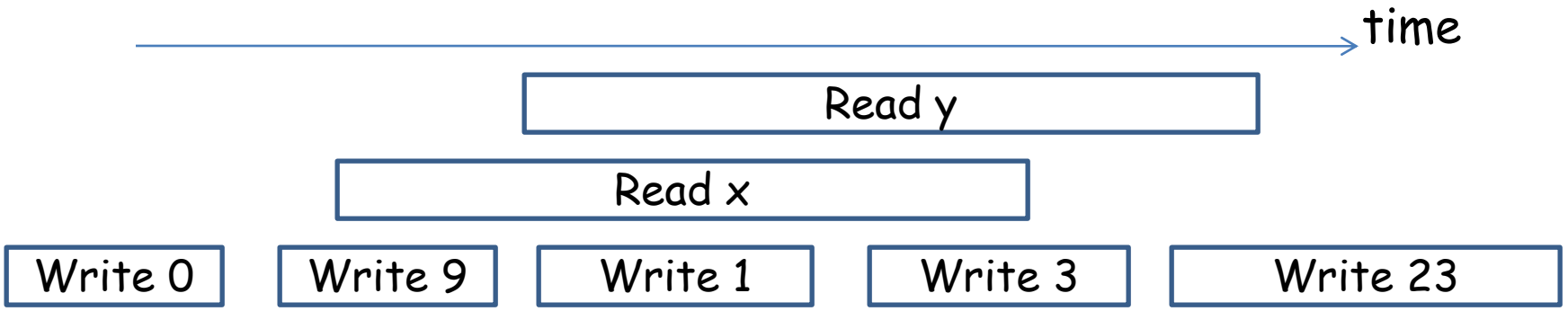
A MRSW is safe if a read interval that does not overlap a write interval returns the most recently written value; a read interval that is concurrent with a write interval may return any value in the register's range 0, …, M-1



x=??, y=?? (but ?? is in the range of the register)

Definition:

A MRSW is regular if a read interval that does not overlap a write interval returns the most recently written value; a read that overlaps one or more writes returns either the most recently written (preceding) value or any of the values being written by the concurrent (overlapping) writes



x=either of 0,9,1,3; y=either of 9,1,3,23

Note:
In these definitions, safe and regular registers are always SW

Definition:

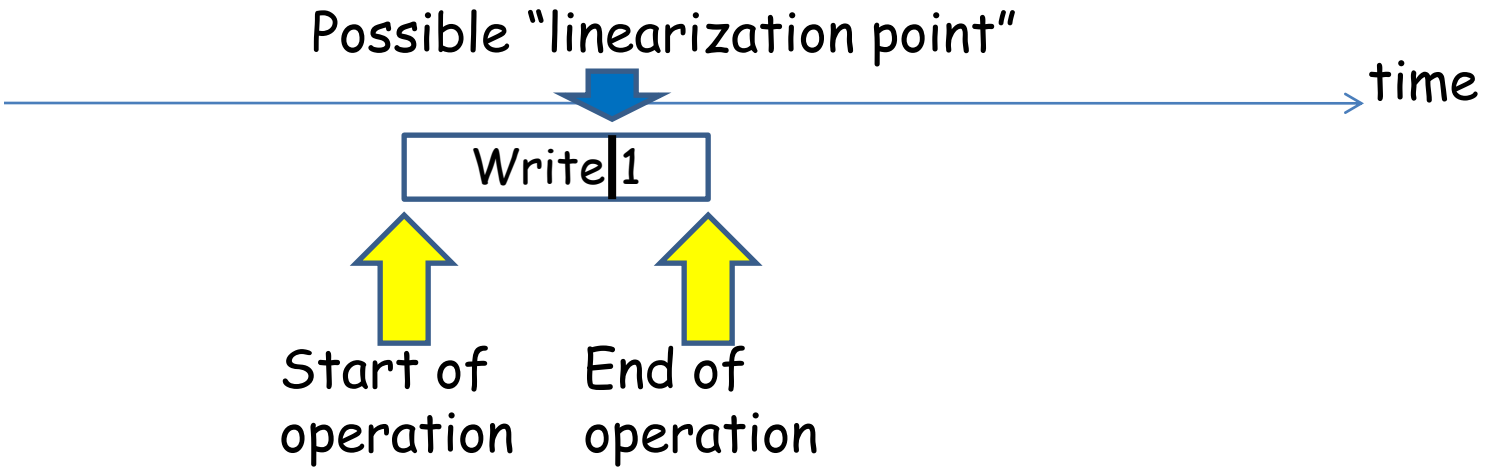
A MRMW register is atomic if reads and writes look as if they are totally ordered events where each read operation returns the value written by the most recent write operation:

- Each read and write operation appears to have taken effect at some instant (event) between the start and the end of the operation's interval
- The resulting sequence of read/write operations ordered by the instants in which they take effect could have been executed by a single thread on a sequential register, that is, each read returns the value of the most recent write

Notes:

The instants where an operation "takes effect" or "becomes visible" are called linearization points (next lecture)

An atomic register is such that any history (next lecture) generated with that register is linearizable, that is equivalent to a (correct) sequential execution



Write order:

Let W0, W1, …, Wi be the write operations on a register, with total write order => such that Wi => W(i+1)

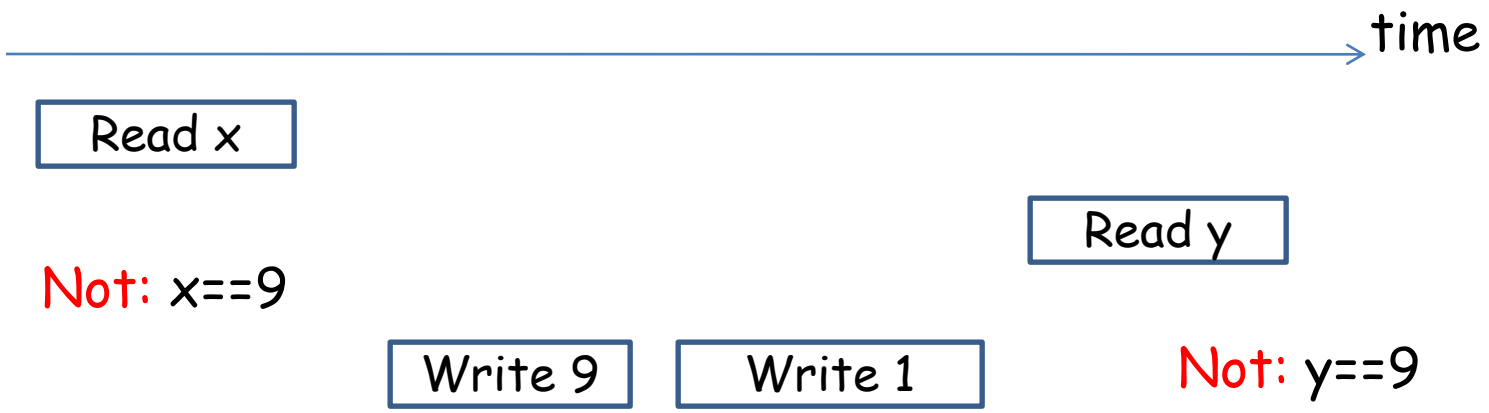
For SW registers, the write order is the interval order ->, since there are no overlapping writes (in particular, safe and regular registers). For MW registers, impose order by the time instant in which a write "takes effect"

Name the read operations, such that Ri is the read operation that returns value x==xi that was written by Wi

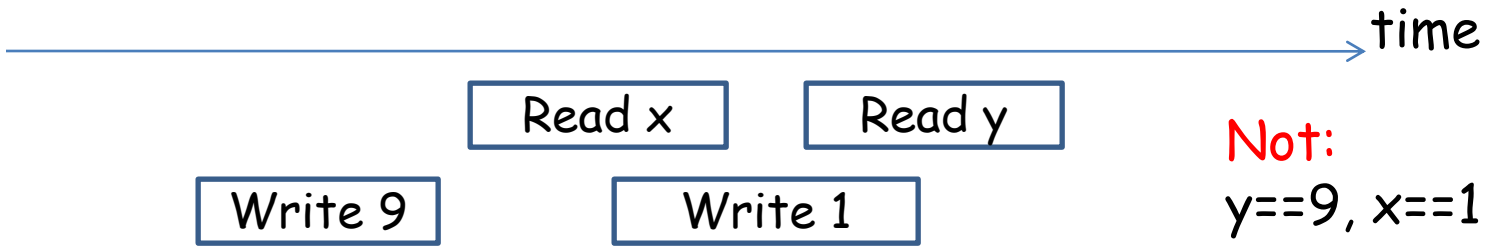
Note: It is assumed (wlog) that each Wi writes a unique value. Each Wi is unique, but there can be several read operations Ri, R'i, … that returns value xi written by Wi

A MRSW register is regular iff

- Not Ri -> Wi: A read never returns a value from the future
- Not Wi => Wj -> Ri: A read never returns a(n overwritten) value from the distant past



Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

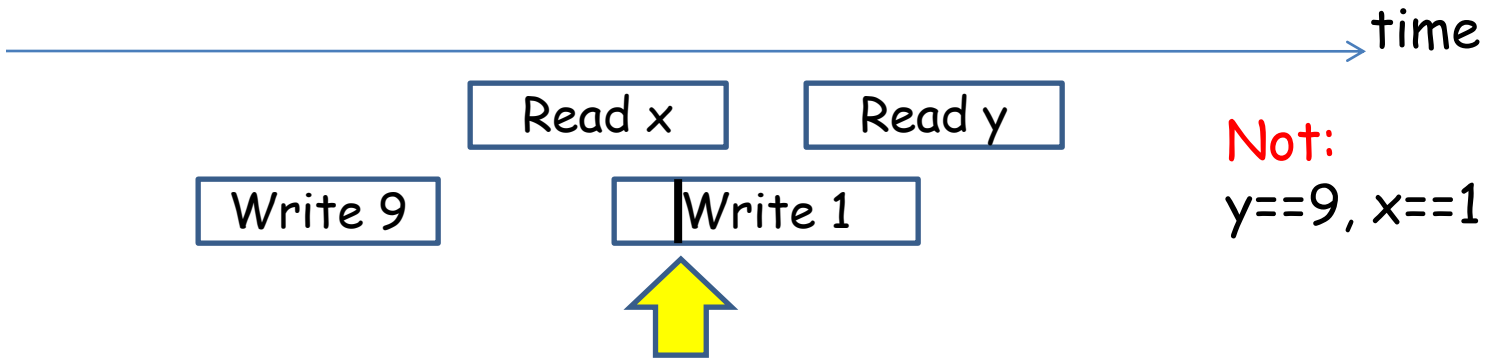


A register is atomic iff it is regular and additionally

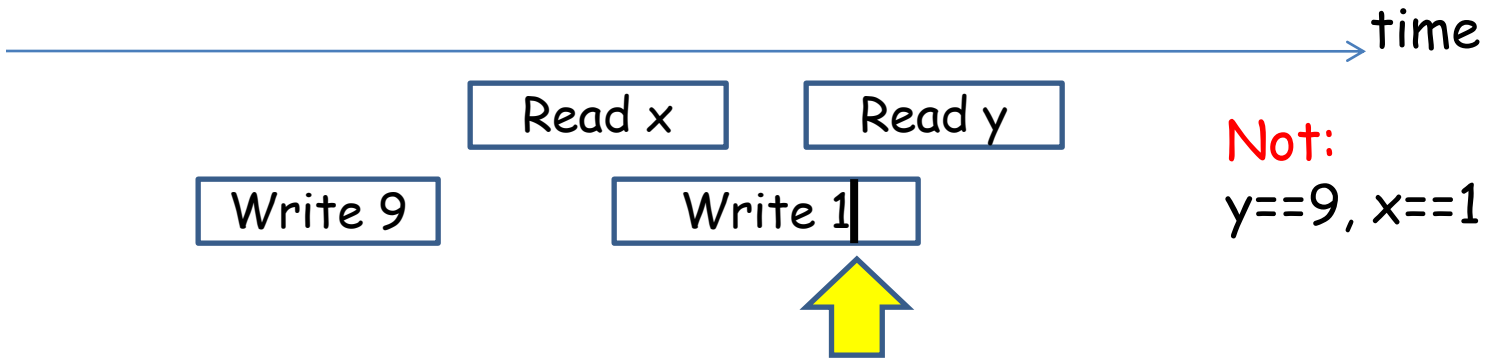
- If Ri -> Rj, then i≤j: An earlier read must return a value that was written no later than the value returned by a later read

To show that a register is atomic: determine the write order, and verify the atomic conditions

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value

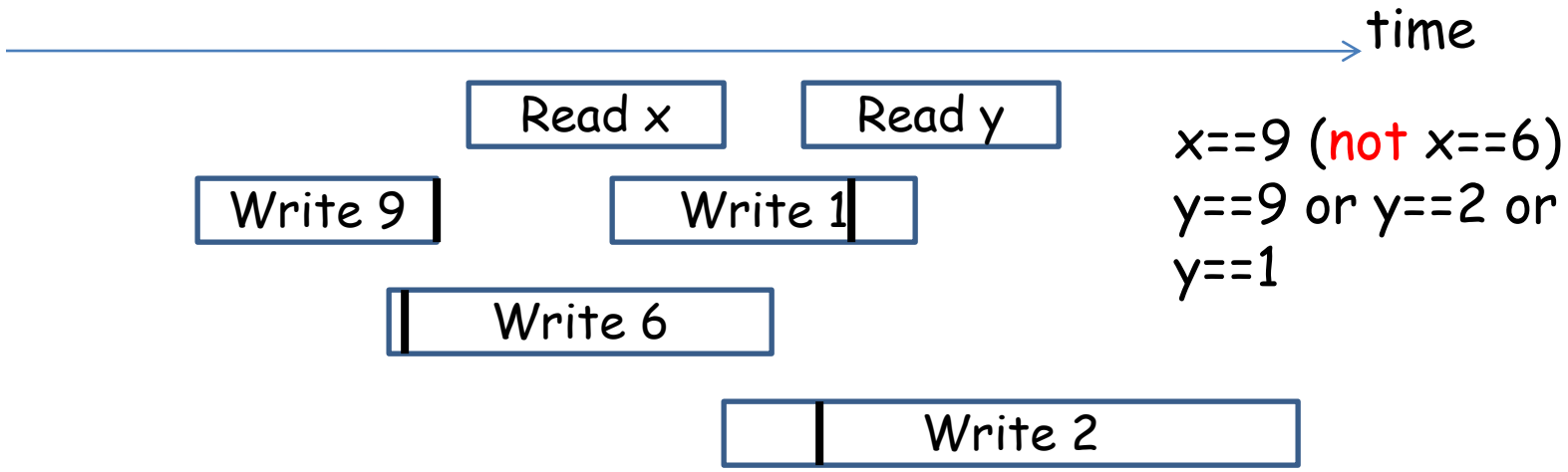


Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value



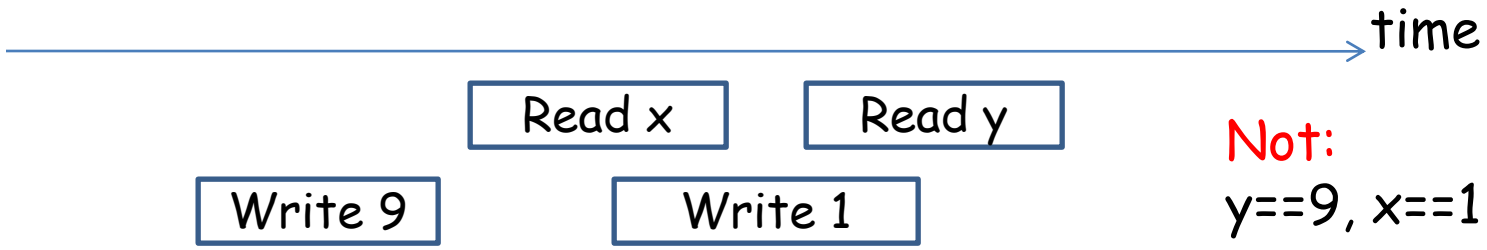
"linearization point": write becomes "visible" here, x==9, and y==1 or y==9

Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value



Linearization points determine total write order:
Write 6 => Write 9 => Write 2 => Write 1

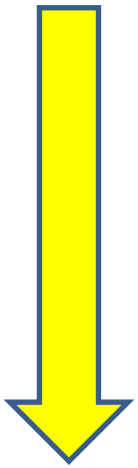
Definition:
A MRMW is atomic if it behaves as if reads and writes are totally ordered events such that each read returns the most recently written value



Theorem:
Atomic MRMW are no more powerful than safe SRSW

Proof: Construct atomic MRMW out of a number of safe SRSW as in the following slides (with a large number of registers)

The construction



1. SRSW safe
2. MRSW safe (Boolean)
3. MRSW Boolean regular
4. MRSW regular
5. SRSW atomic (from SRSW regular)
6. MRSW atomic
7. MRMW atomic

Safe SRSW to safe MRSW

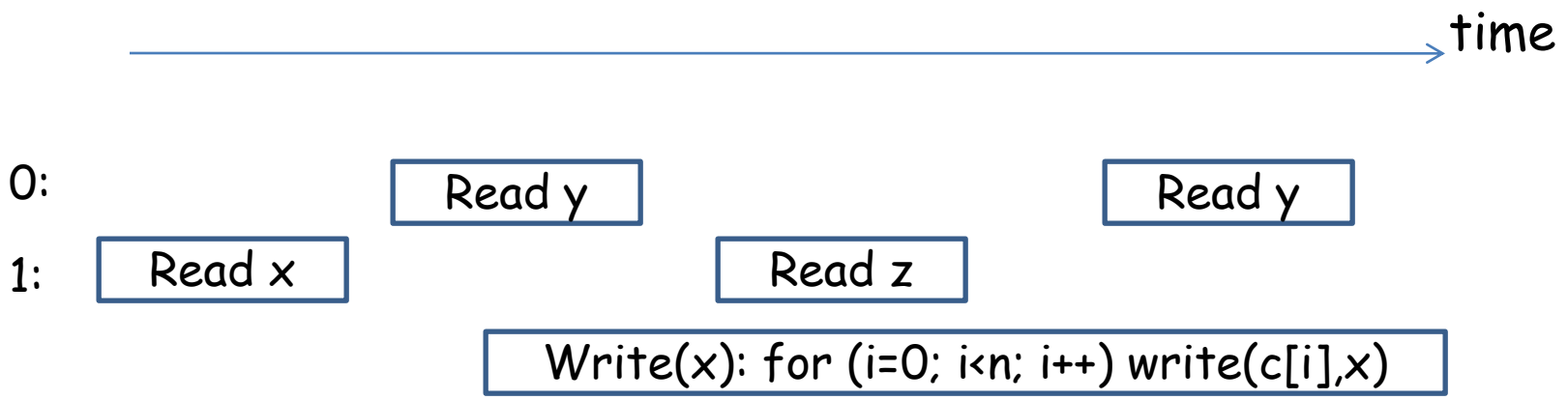
Construction:

```
Array c[n] of safe SRSW
read(thread i): return c[i]
write(x): for (i=0; i<n; i++) c[i] = x;
```

Lemma: The construction is a safe MRSW

Proof: Each thread reads only a single register c[i]. If a read does not overlap write, it returns the most recently written value; if a read overlaps a write it may return any value, either an old c[i], a new c[i] just written, or if the write to c[i] overlaps the read of c[i], any value since c[i] is safe.

Safe SRSW to safe MRSW



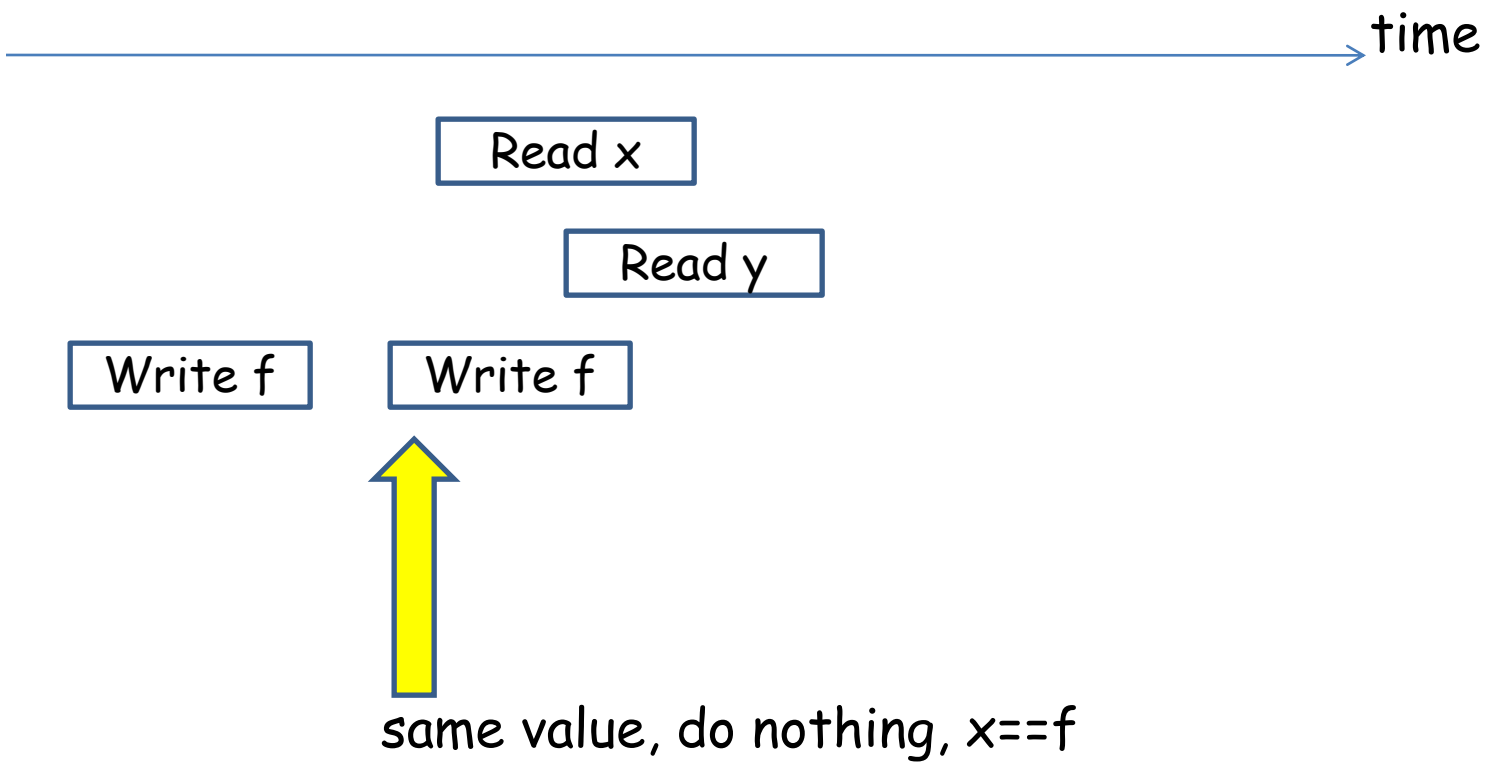
Safe Boolean MRSW to regular Boolean MRSW

Observation: A regular Boolean register differs from a safe Boolean register only when the value x written (during an overlapping read) is the same as the old value already in the register: a regular register can only return x, while a safe register can return either true/false.

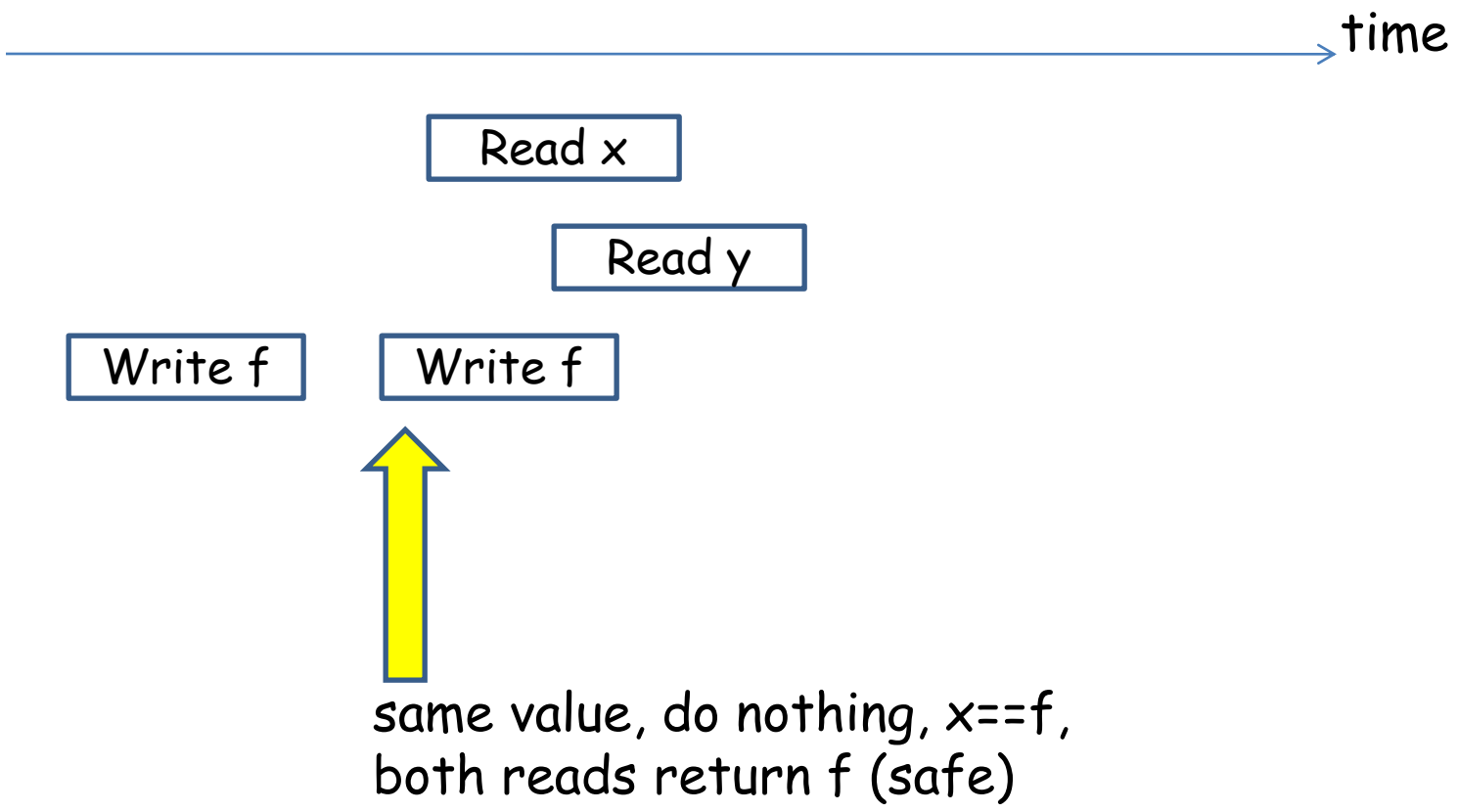
Construction: Associate an extra, local variable (old) with the register; write x only if x≠old (and update old)

Note: An M-valued register, M>2, can be turned into a Boolean register by reading and writing modulo 2

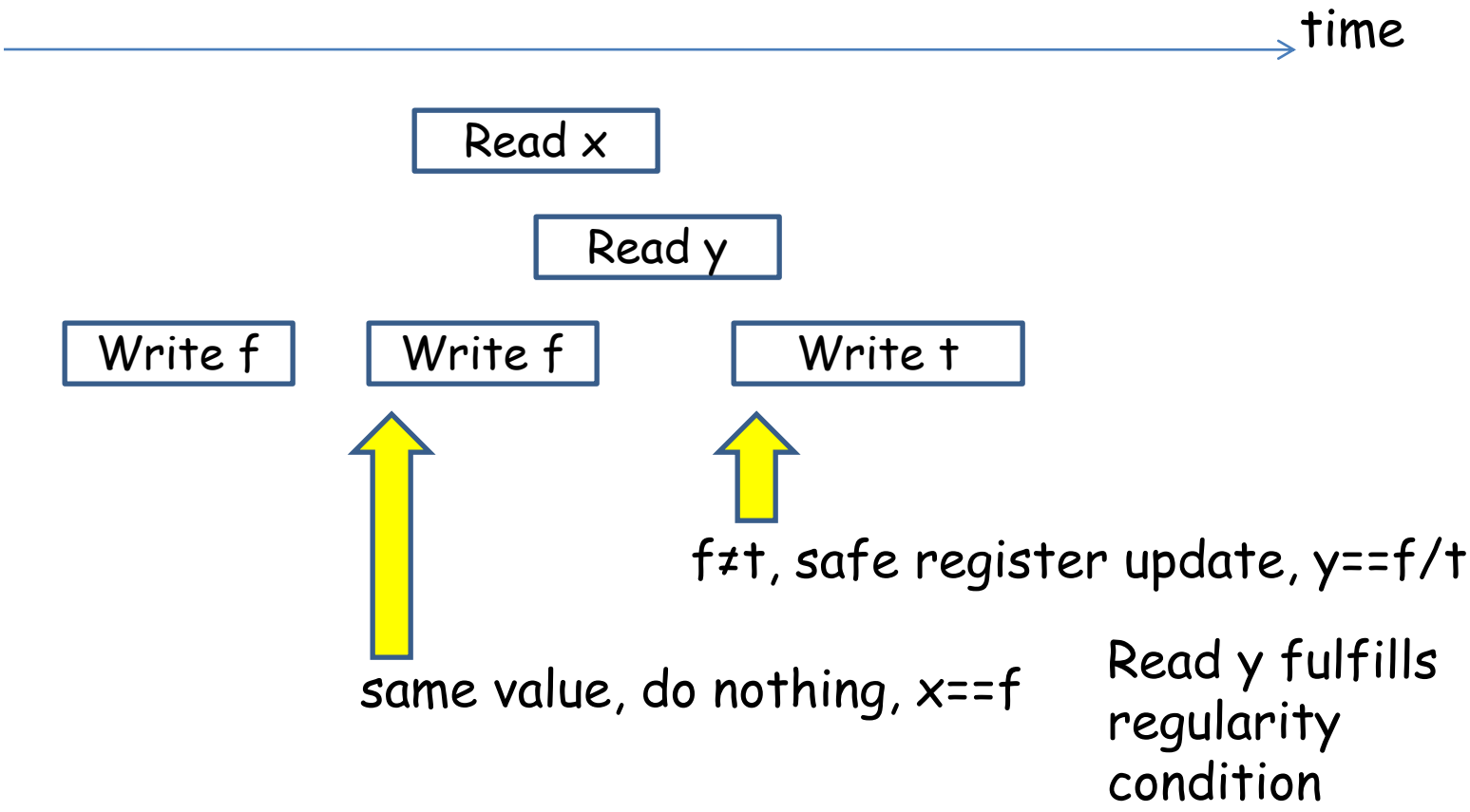
Lemma: The construction is a Boolean regular MRSW



Lemma: The construction is a Boolean regular MRSW



Lemma: The construction is a Boolean regular MRSW

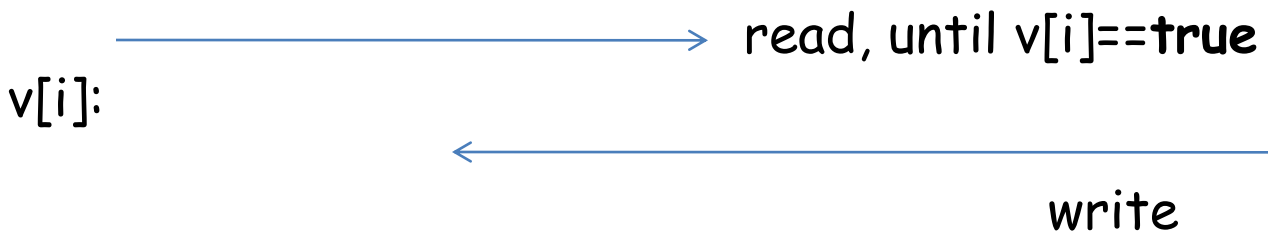


Regular Boolean MRSW to regular M-valued MRSW

Represent $0 \le x < M$ in unary, use M regular Boolean regular registers to represent the M-valued regular register

Construction:

```
Array v[M] of regular Boolean MRSW registers
Init: v[0] = true;
read(thread i): for (j=0; j<M; j++) if (v[j]==true) return j;
write(x): v[x] = true; for (i=x-1; i≥0; i--) v[i] = false;
```

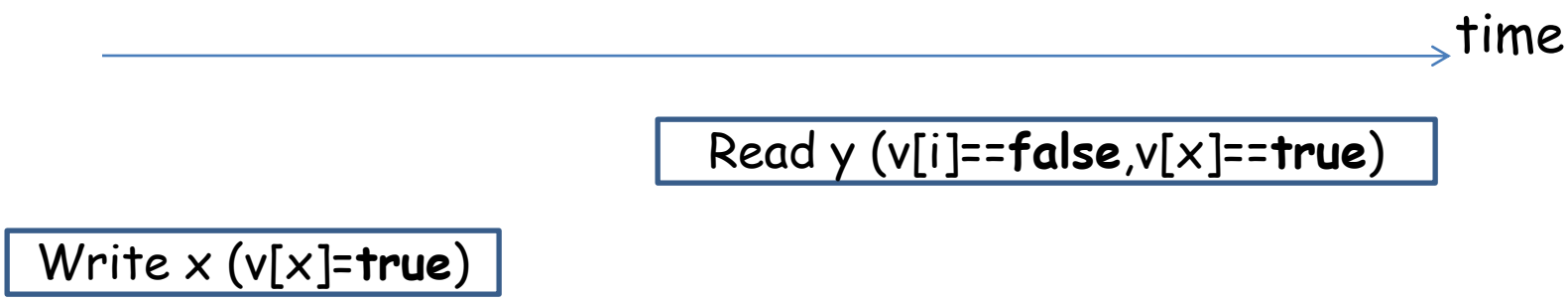


Lemma: The construction is a regular M-valued MRSW

Proof: When a write(x) completes, v[i]==**false** for all i<x; the smallest j with v[j]==**true** is the most recently written value.

If write(x) completes before read, read will eventually find v[x]==**true** and return x.

If reader reads v[j]==**false**, then most recent write(x) had x>j

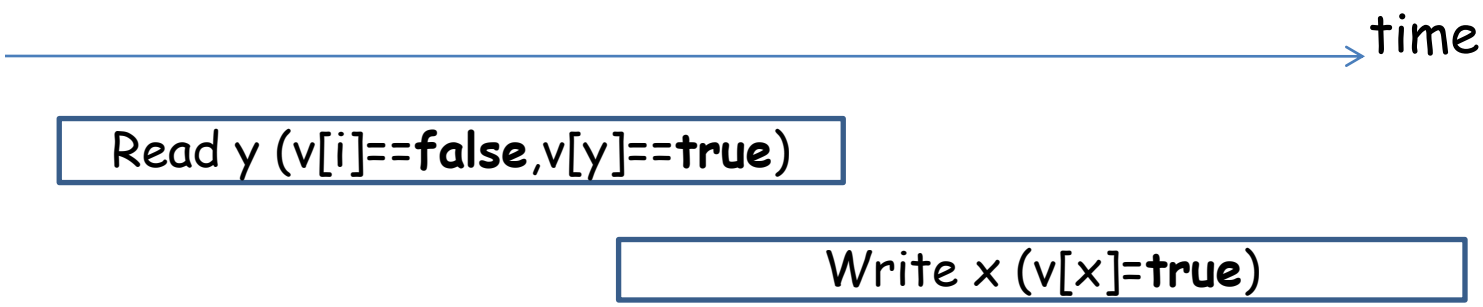


Lemma: The construction is a regular M-valued MRSW

Proof: Assume the register has been initialized, some v[i]=**true**.

When a write(x) completes, v[i]==**false** for all i<x; the smallest j with v[j]==**true** is the most recently written value.

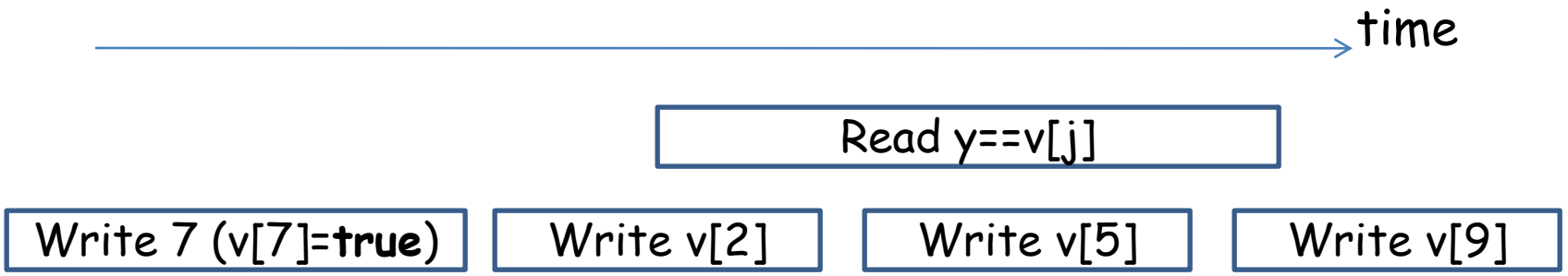
If read(y) completes before write(x) (v[x]=**true**), then y<x, and y is the most recently (previous) written value



Proof (con't):

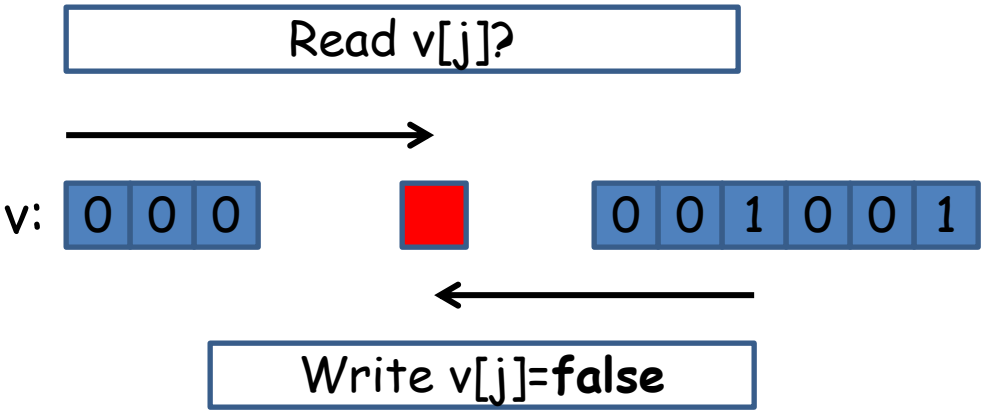
If read overlaps write, consider the cases where reader and writer access the same Boolean SRSW register v[j], writer coming from some index x≥j, reader from 0≤j

Example: y may be any of 7, 2, 5, 9, but nothing else



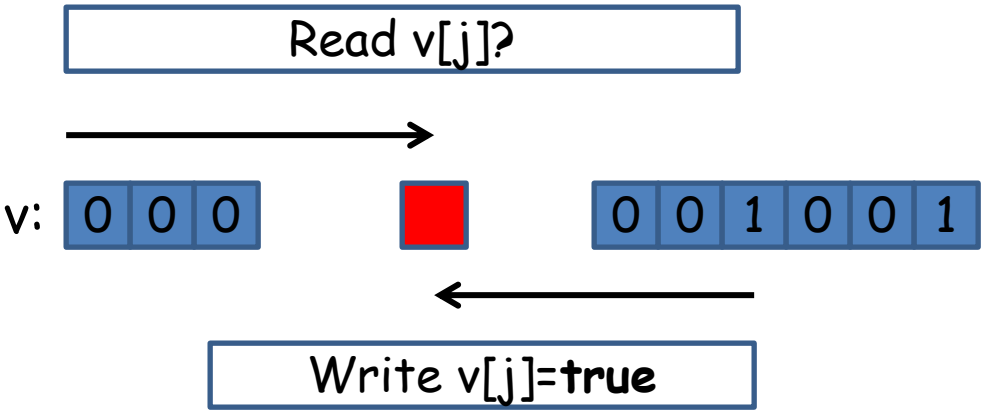
Proof (con't): 4 cases to consider

a) Reader reads v[j]==**false**, writer writes v[j]=**false**. Then the write operation is to an x>j, so ok for the reader to continue
b) Reader reads v[j]==**true**, writer writes v[j]=**false**. Then there must have been a previous (overlapping or most recent non-overlapping) write(j), since v[j] is a regular register. Ok for reader to return j

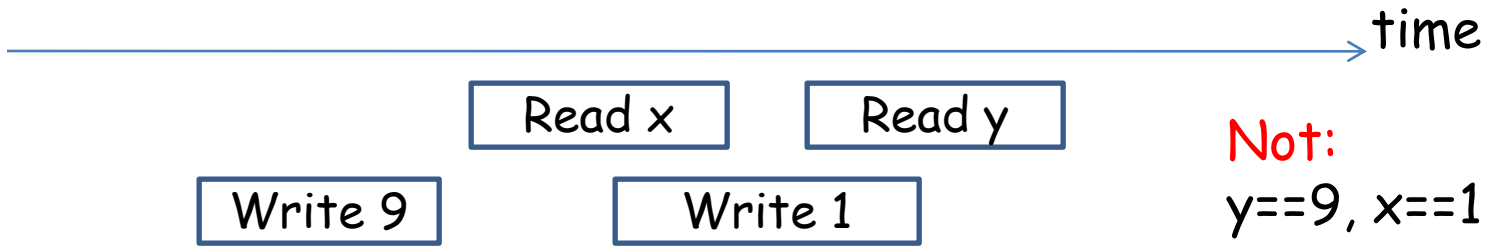


Proof (con't): 4 cases to consider

c) Reader reads v[j]==**true**, writer writes v[j]=**true**; correct for the reader to return j
d) Reader reads v[j]==**false**, but writer writes[j]=**true**. Then, since register v[j] is regular, there must have been a most recent previous write(k) with k>j, so the reader may continue



Regular SRSW to atomic SRSW



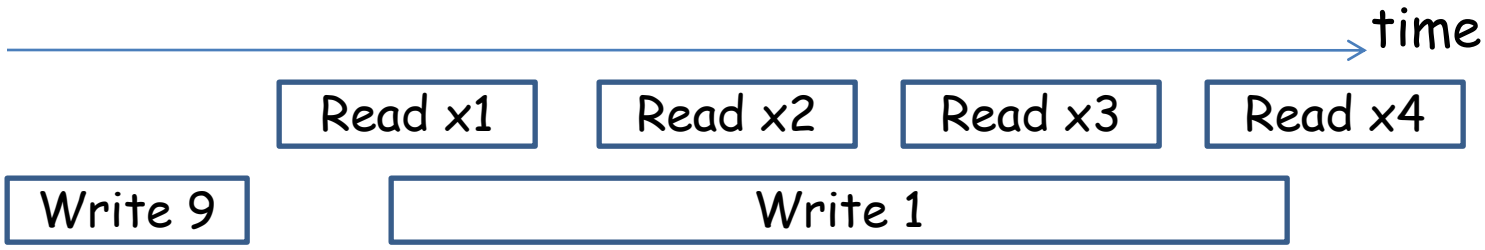
Idea:
Use time stamps to prevent that a later read gets an earlier value than a preceding read. Assumption: Registers can store time stamped values.

Construction: Maintain an old, time-stamped value; reader reads register and compares to old value, returns most recent value of either and updates; writer increments the time stamp

```
private:
unbounded timestamp stamp;   // incremented by writer
old: pair of <timestamp,value>; // updated by reader

Regular SRSW v of <timestamp,value> pairs; // register

Init: stamp = 0;

read():
w = v; // read regular SRSW register
if (old.timestamp<w.timestamp) old = w; // write takes effect
return old.value;

write(x):
stamp += 1; // increment time-stamp
v = <stamp,x>; // write regular SRSW register
```

Lemma: The construction is an atomic M-valued SRSW

Time stamps are strictly increasing. Reader cannot return an earlier value than a preceding read, since that would have a lower time stamp



xi==1 or 9 for regular registers. In the construction, the moment xi==1, also xj==1 for j≥i

Note:
This and the following constructions use unbounded time stamps, that cannot be represented with bounded, M-valued registers.
The proof-chain collapses…

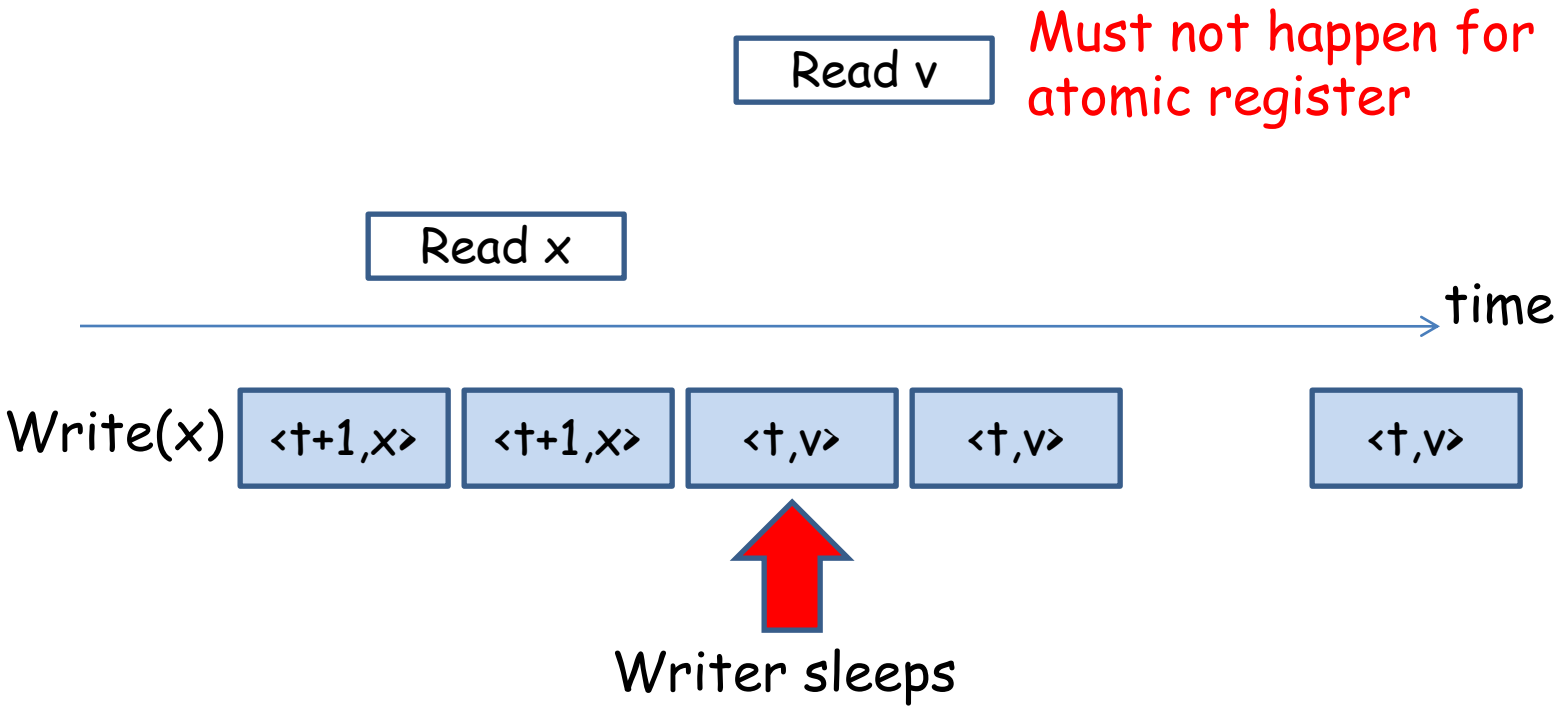
The problem has been addressed with other constructions (not in book), e.g.:

Haldar, Vidaysankar: Constructing 1-writer multireader multivalued atomic variables from regular variables. JACM, 42(1): 186-203, 1995.
Israeli, Li: Bounded time-stamps. Distributed Computing, 6(4): 205-209, 1993.
Singh, Anderson, Gouda: The elusive atomic register. JACM, 41(2): 311-339, 1994.
Li, Tromp, Vitanyi: How to share concurrent wait-free variables. JACM, 43(4): 723-746, 1996.

Atomic SRSW to atomic MRSW

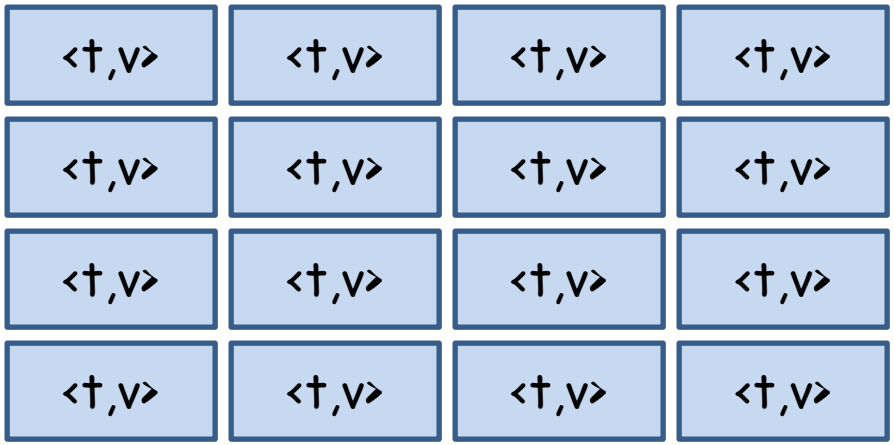
The single-array construction (safe SRSW to safe MRSW) does not work:

read(thread 0) returns c[0] and any later read(thread i) for i>0 could return an earlier value, if the write(x) was delayed when updating c[i] after having written c[0]



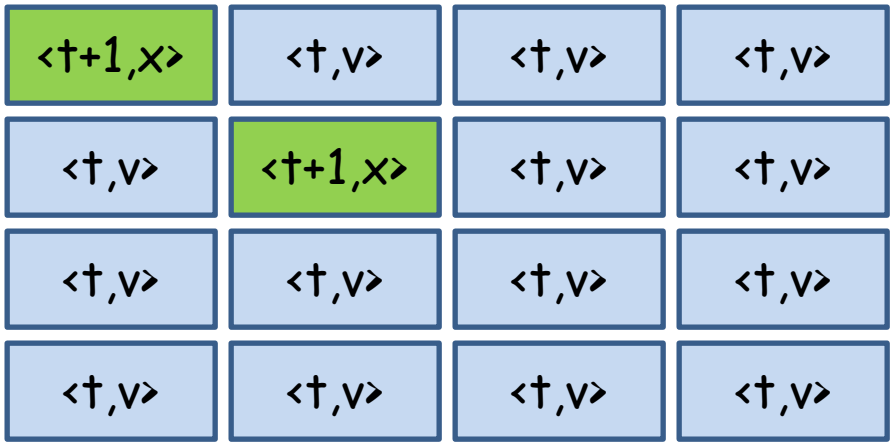
Working idea:
nxn matrix of atomic SRSW, reading thread helps later reading threads to get the correct value

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW



write(x): t = t+1; write <t,x> in diagonal c[i][i]

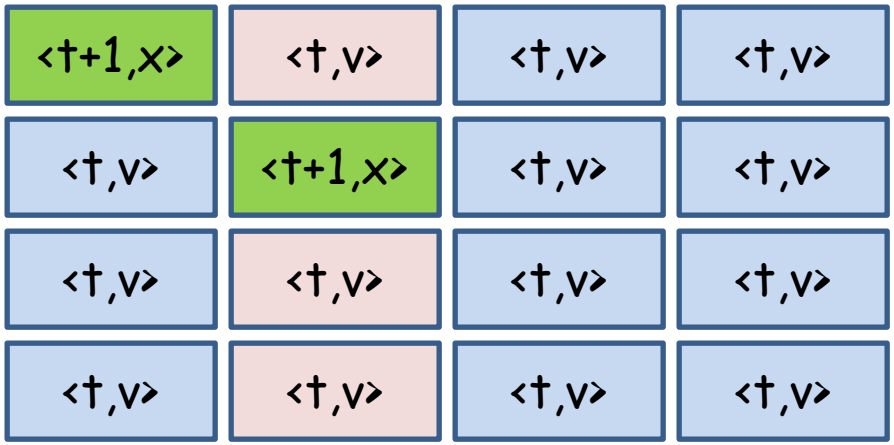
Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW



Example: writer is delayed after i=1,...

write(x): t = t+1; write <t,x> in diagonal c[i][i]

Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

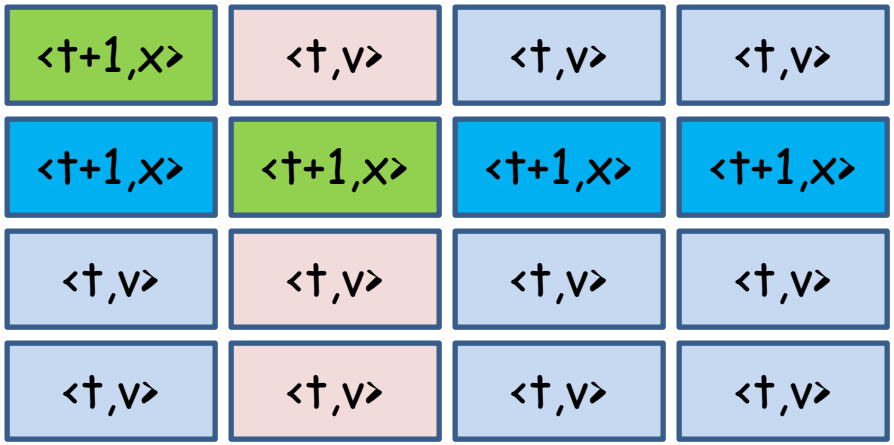


max = <t+1,x>

write(x): t = t+1; write <t,x> in diagonal c[i][i]

read(thread i): find value w with max timestamp in column i

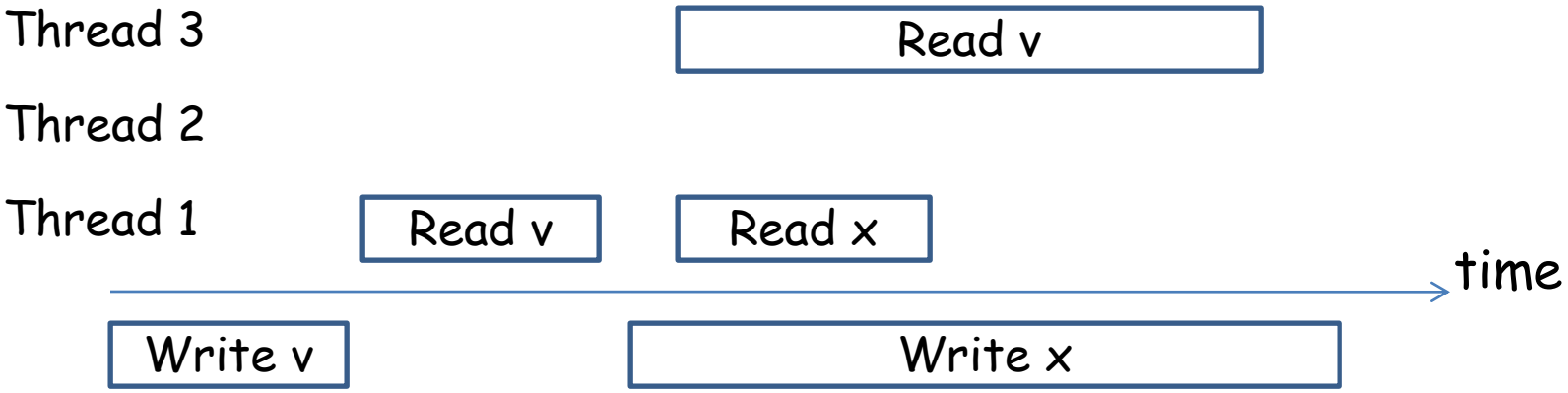
Each entry c[i][j] is a <stamp,value> pair of an atomic SRSW

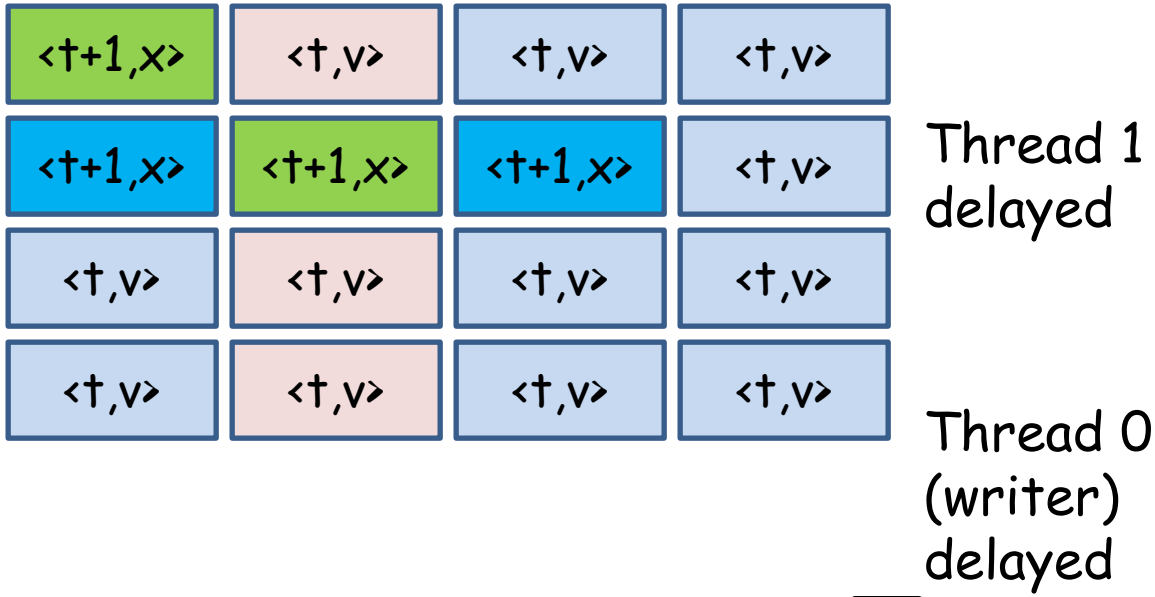


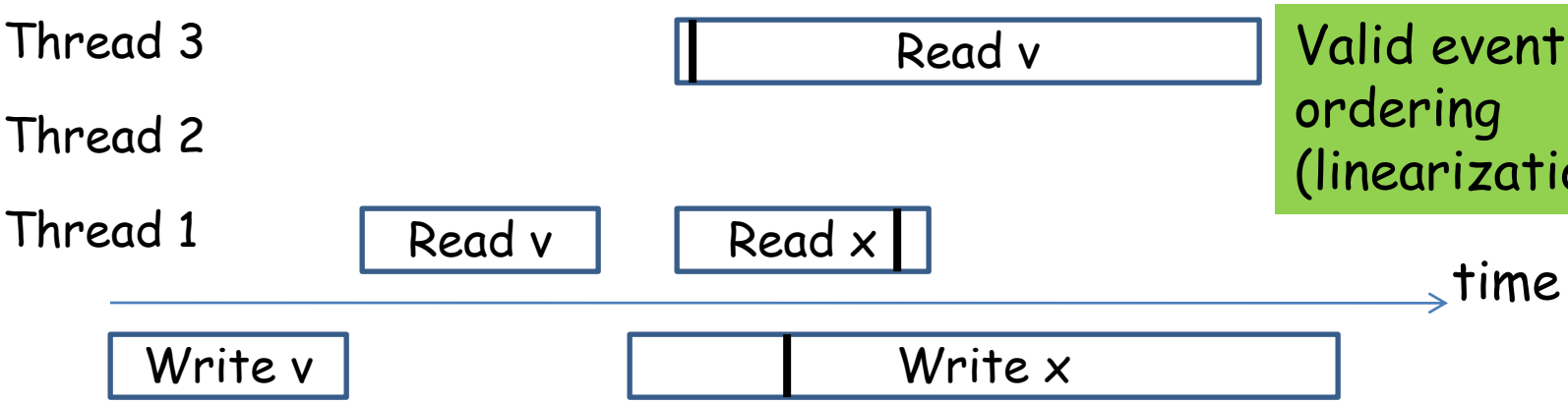
max = <t+1,x>

write(x): t = t+1; write <t,x> in diagonal c[i][i]

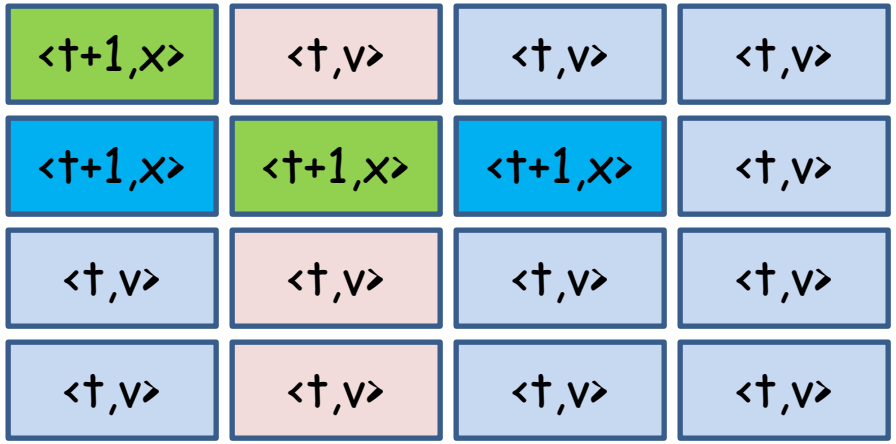
read(thread i): find value w with max timestamp t' in column i
write <t',w> in row i (except for c[i][i] since SW)
return w

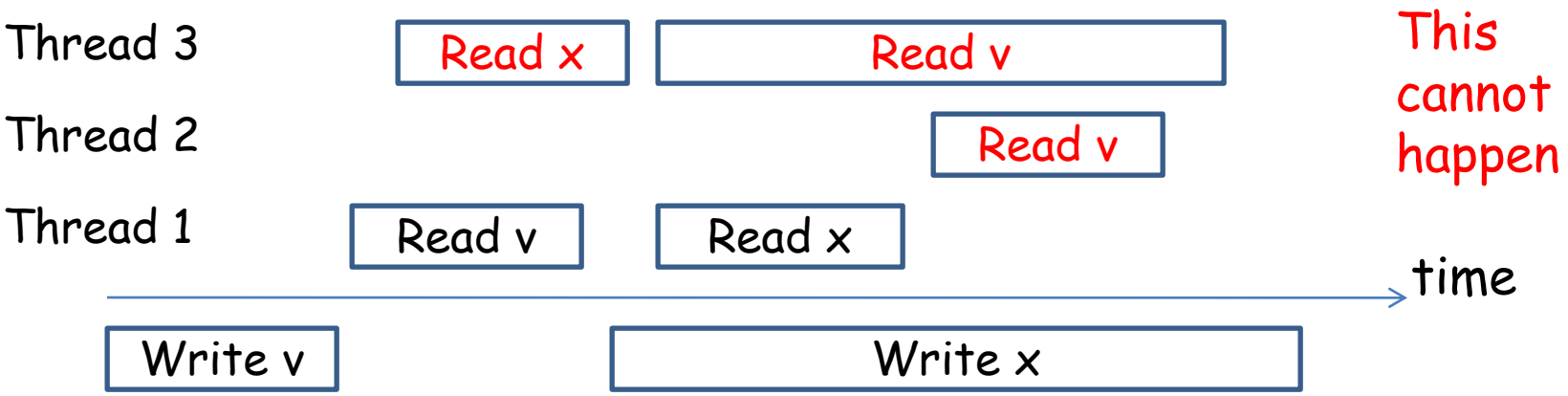

Thread 3
Read v
Thread 2
Thread 1
Read v
Read x
time
Write v
Write x

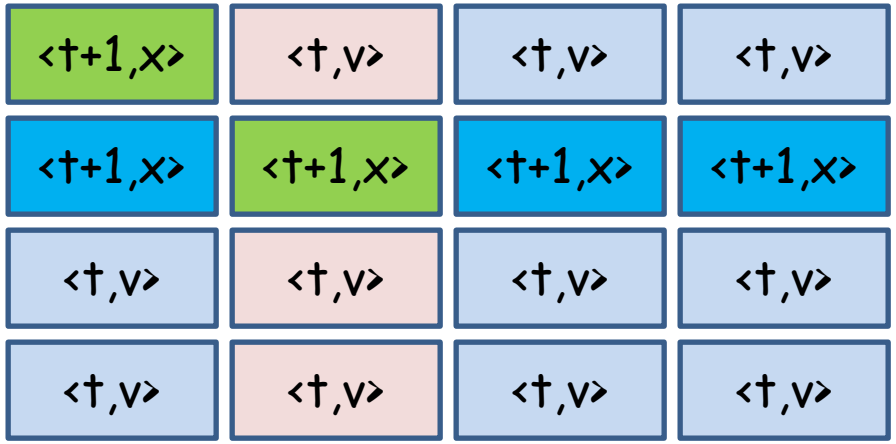

<t+1,x>
<t,v>
<t,v>
<t,v>
<t+1,x>
<t+1,x>
<t+1,x>
<t,v>
<t,v>
<t,v>
<t,v>
<t,v>
<t,v>
<t,v>
<t,v>
<t,v>
Thread 1 delayed
Thread 0 (writer) delayed



Valid event ordering (linearization)







Lemma: The construction is an atomic M-valued MRSW

Proof: Time-stamp t of writer is strictly increasing with each write. The maximum time stamp in any row or column is strictly increasing in time.

A reader clearly returns a value written by an overlapping or most recently completed write (no value from the future). If a write operation with timestamp t precedes a read operation, then the read will return this or some later value (no value from the distant past). If a read by thread A precedes a read by thread B, then A has written a time-stamped value in B's column, so the value returned cannot be a value written before the value read by A (atomicity).

Atomic MRSW to atomic MRMW

Idea: Array of atomic MRSWs with time stamps

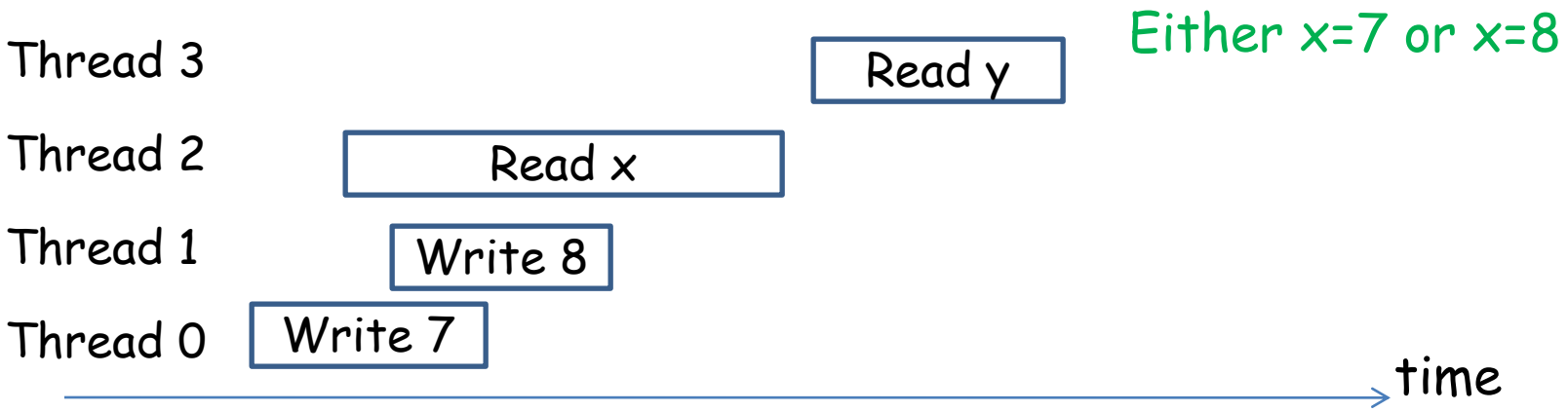
read(thread i):
Scan through array, return the value with the maximum time stamp in the array

write(thread i, value x):
Scan through array, find latest time stamp t, write <t+1,x> to register at index i

Lemma: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

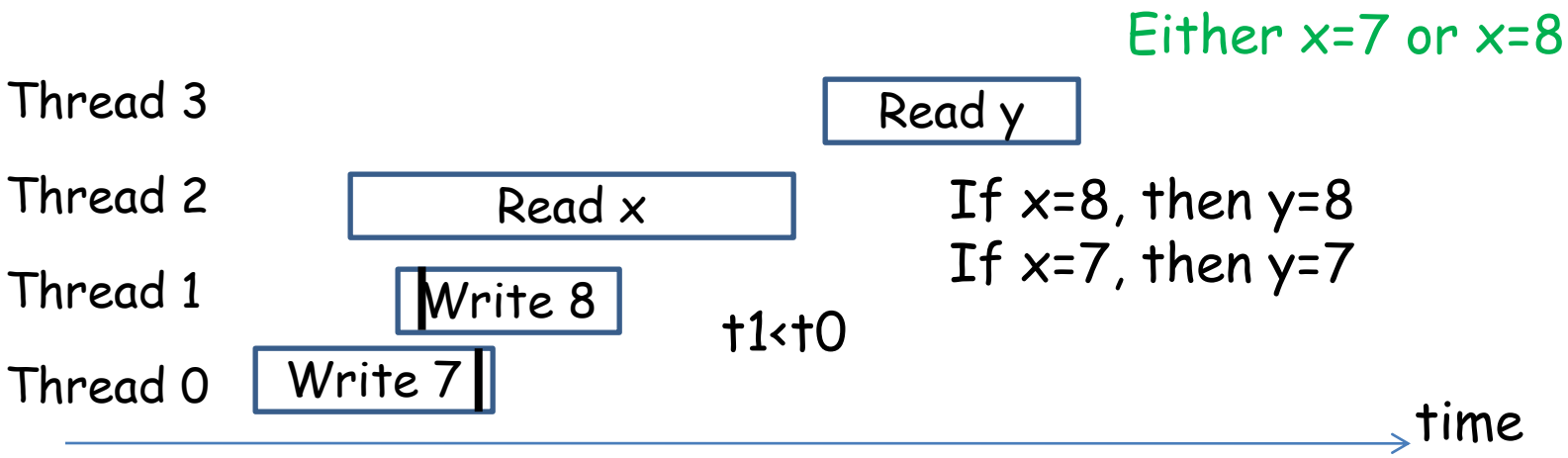
Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).



Lemma: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

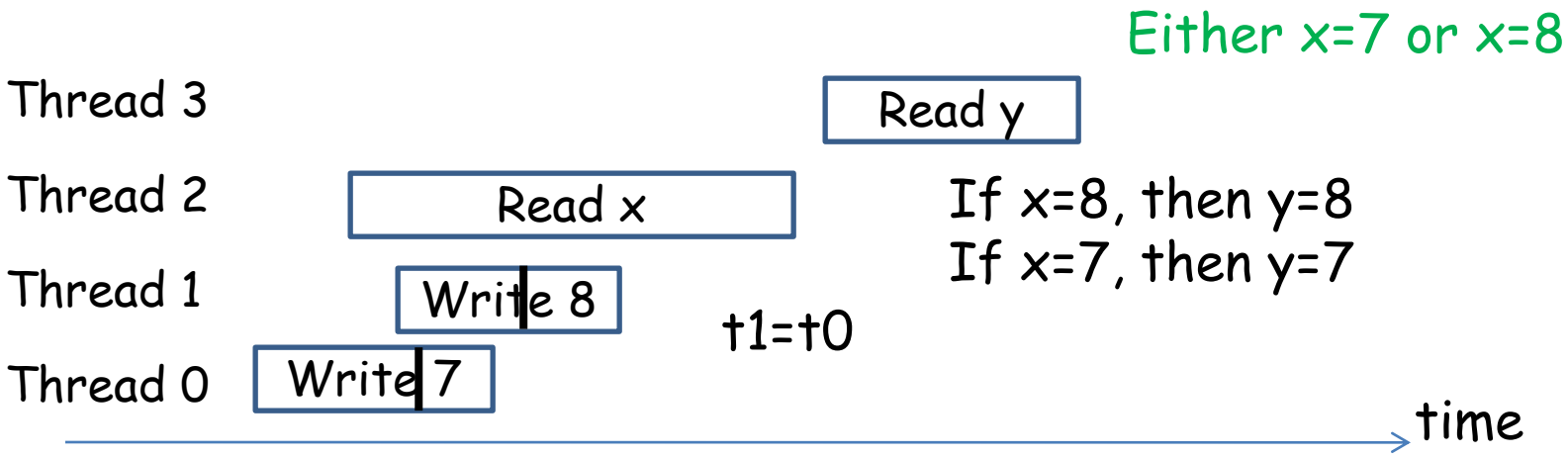
Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).



Lemma: The construction is an atomic M-valued MRMW

Time stamping as in Bakery algorithm, for each writing thread, time stamps are strictly increasing.

Write order: Define that a write by A with time stamp tA precedes a write by B, A => B, with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).



Lemma: The construction is an atomic M-valued MRMW

Write order: Define that a write by A with time stamp tA precedes a write by B with time stamp tB if tA<tB, or tA==tB and A<B (lexicographic order).

If write A => write B (temporal order) then tA < tB. If write A and write B overlap, one will be ordered before the other either by time-stamp or by thread id.

No value from the future: A writing thread B which is after a reading thread A, A -> B, writes a time-stamp that is at least one larger than the time-stamp read by A. Thus, the value read by A cannot be the value from the future written by B.

Lemma: The construction is an atomic M-valued MRMW

No value from the distant past: Assume write(thread A) => write(thread B) -> read(thread C). If A=B, then B has overwritten the time-stamped value written by A, so C will return B's value. If A≠B, since A's time-stamp is smaller than B's, C will return the value written by B (or some other later thread after A). Thus, no value from the distant past can be returned.

Lemma: The construction is an atomic M-valued MRMW

Atomicity, values read in write order: Assume read(thread A) -> read(thread B), and write(thread C) => write(thread D) in write order (the two writes may be concurrent). It must shown that if A returns the value written by D, then B cannot return the value written by C.

If tC<tD, then A read tD from index D, and since B also reads index D, it cannot return the value written by C (at index C). If tC=tD (concurrent writes) then C<D, and B must also return the value from index D (or a later one)

Theorem:
Atomic MRMW are no more powerful than safe SRSW

Proof (now completed):
Constructions implemented an atomic MRMW out of a (large) number of safe SRSW

BUT:
Constructions are not practical. High space requirements. The atomic constructions rely on unbounded time stamps

On the register constructions:

1. No RW operation by some thread blocks RW operations of other threads
2. RW operations by any thread can progress independently of actions by other threads
3. RW operations always complete in a finite (bounded) number of steps, $O(n)$, $O(n^2)$, $O(M)$, … independently of actions by other threads

Algorithms, object methods, … with property 3 are called (bounded) wait-free

Lock-free: Infinitely often, some method/execution will finish in a finite number of steps (or: when several threads invoke some method, some thread will finish)

Progress conditions

Lock-freeness and wait-freeness are non-blocking, independent progress conditions

Blocking (dependent) progress conditions (progress of thread i can be delayed/blocked arbitrarily dependent on actions or non-actions of other threads):

- Starvation freedom
- Deadlock freedom

Definitions (non-blocking, independent progress):

- A concurrent algorithm (object, method call) is wait-free, if in any concurrent execution, each thread can finish in a finite (bounded) number of steps, independent of the actions of other threads
- A concurrent algorithm (object, method call) is lock-free, if in any execution some thread can finish in a finite (bounded) number of steps (or, in long execution, equivalently, some thread can complete a method call infinitely often), independent of the actions of other threads

A wait-free algorithm whose bounds do not depend on the number of threads is called population oblivious

Definitions (blocking, dependent progress):

- A concurrent algorithm (object, method call) is starvation free, if in any execution, any thread will eventually be able to complete, provided that other threads take steps
- A concurrent algorithm (object method call) is deadlock free, if in any execution, some thread will eventually be able to complete, provided that other threads take steps

Degrees of starvation freedom: Fairness

Main examples: locks, monitors, …: progress possible if threads relinquish the shared resource (unlock), in particular do not stall (crash) indefinitely

Blocking	Non-blocking
	Obstruction-free
Deadlock-free	Lock-free
Starvation-free	Wait-free

Note:
- Starvation-free implies deadlock-free
- Wait-free implies lock-free

Question:
Are wait- or lock-free implementations of concurrent data structures/objects/methods/algorithms always possible? Needed?

Non-blocking but dependent progress condition:

Obstruction freedom: A thread that executes in isolation (no other threads taking steps), can finish in a finite number of steps

Remark:
An obstruction free algorithm can sometimes be turned into a useful, "wait-free" algorithm by introducing back-offs: A thread that detects some kind of conflict with other threads, will step back and yield (processor) to another thread

Atomic snapshots

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e. take a consistent snapshot(*)

Simplified version (MRSW):

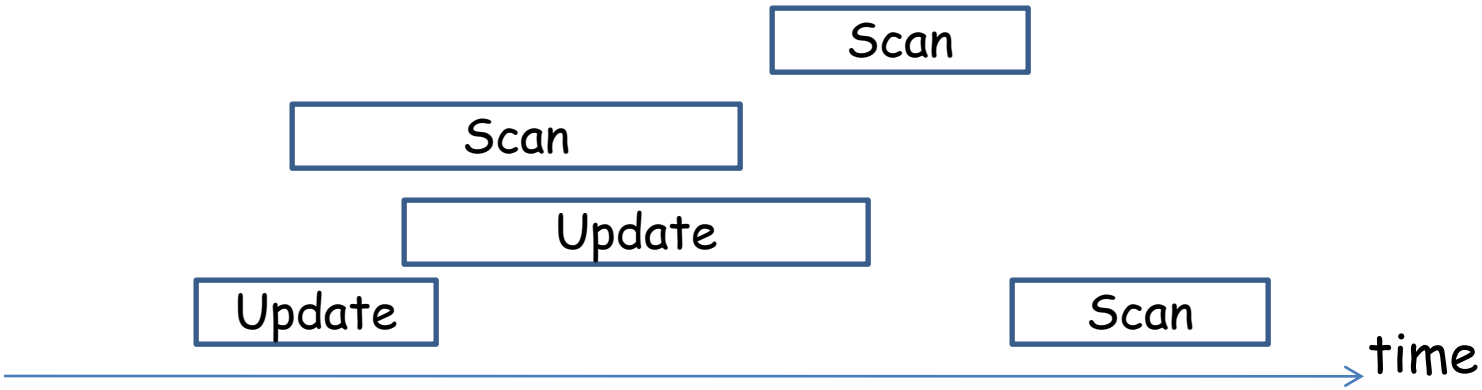
- Each thread has a register, update(value) operation writes value to threads register (SW)
- scan() operation takes snapshot of all thread's registers (MR) and copies into local array

Is this possible in a wait-free manner?

(*) state that actually exist(ed) at some point in the global time

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

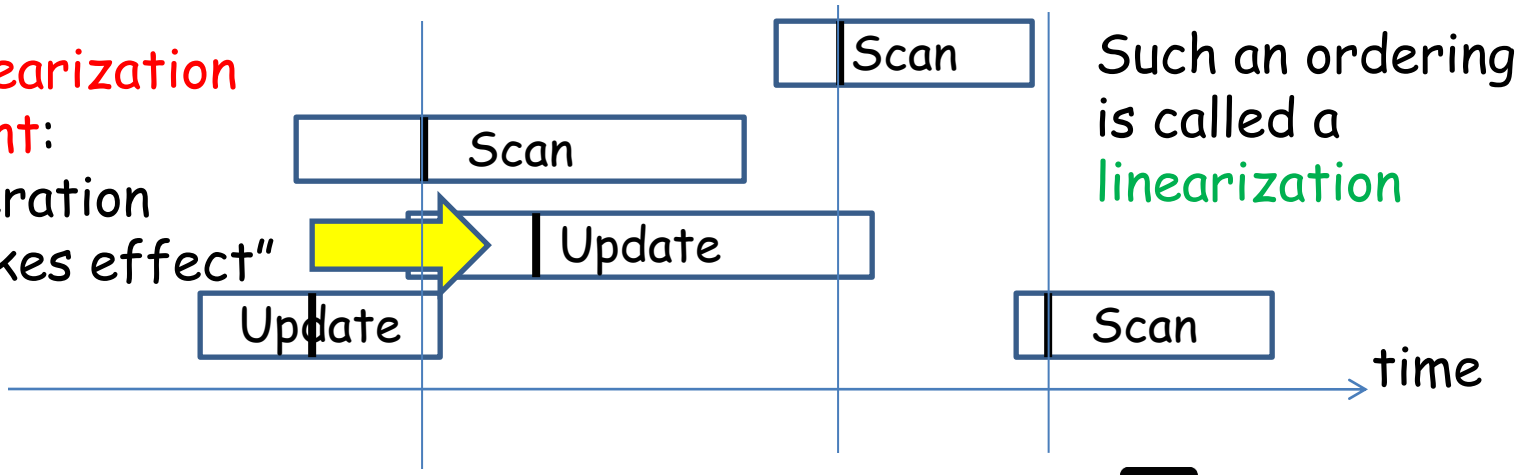
Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update



Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update

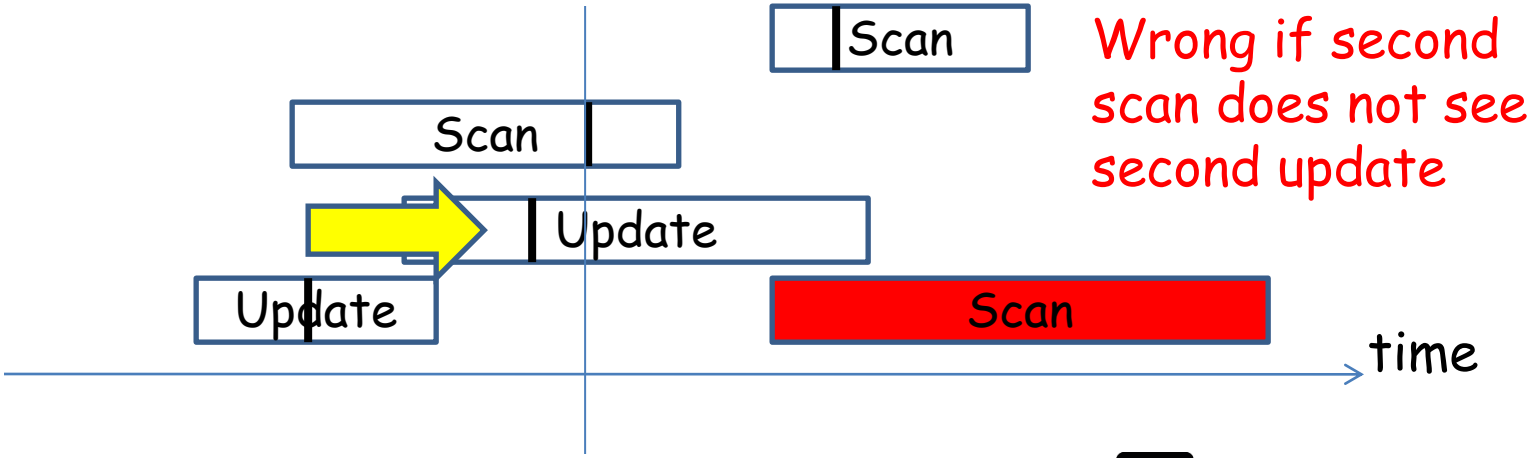
Linearization point: operation "takes effect"



Such an ordering is called a linearization

Assume a whole set of atomic registers describing the state of some algorithm…
Might be beneficial for any thread to be able to read this state atomically, i.e., take a consistent snapshot

Correct: Possible to find a total order on the operations, such that sequential specification is satisfied, and such that the snapshot was valid in the interval of the operation, that is, it reflects the state after the most recent preceding or overlapping update



Blocking (lock-based, sequential) implementation:

```
class LockSnapshot<T> implements Snapshot<T> {
public synchronized void update(T v) {
    regarray[ThreadID.get()] = v; // set my state value
}
public synchronized void T[] scan() {
  T[] snapshot = (T[]) new atomic reg[regarray.length];
  for (int i=0; i<regarray.length; i++) {
    snapshot[i] = regarray[i];
  }
  return snapshot;
}
}
```

An obstruction-free snapshot

Each register is MRSW atomic. Each thread maintains a local time-stamp that is incremented at each update operation, <timestamp,value> pairs are written in register

update(value):
increment local timestamp and write <timestamp,value>

Lemma (observation): The update operation is wait-free

Note:
Timestamps here just ensure that written values are unique

A collect operation simply copies the thread registers into a local array (in some order)

Observation: If two successive collects return the same value (array contents), then this interval can have had no update, so the collected value (array) is a correct snapshot. Call this a clean double collect.

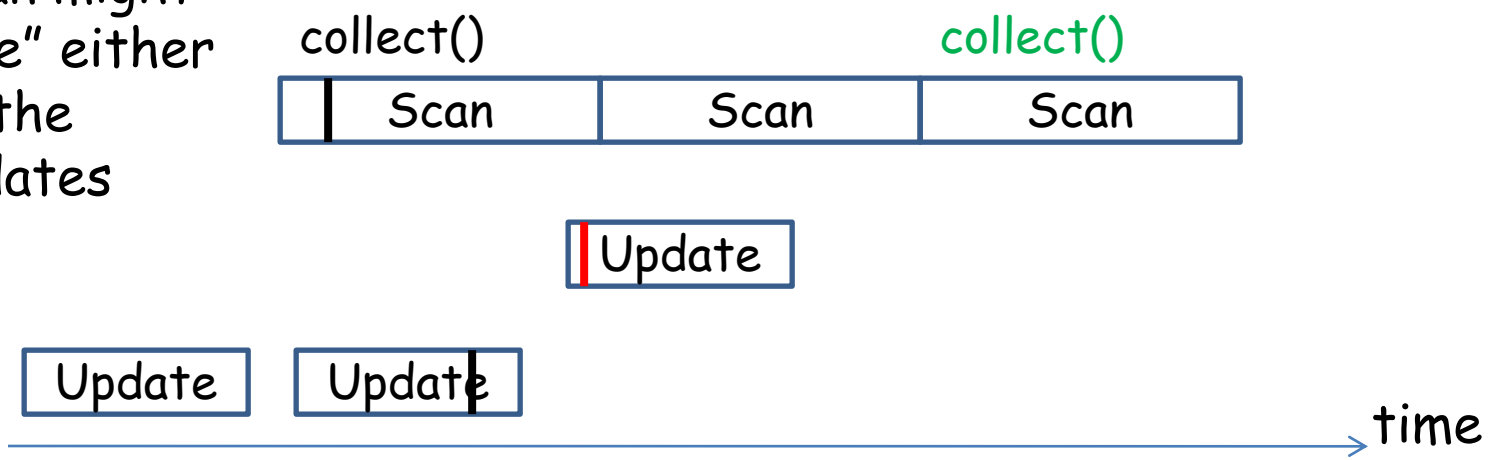
scan():
Perform collect until the last double collect is clean

Lemma: The scan operation is obstruction-free, but not wait-free. A scan can always be disturbed by an overlapping update. But as soon as there are no concurrent updates (scanning thread executes in isolation), it can complete correctly.

Not possible to linearize scan with update if only a single collect is done (could try to linearize scan either before or after update, however, some later update may or may not be seen; or another concurrent scan could be incompatible with this order)

Clean double collect implies no concurrent update

Scan might "see" either of the updates



A wait-free snapshot

Previous solution is dependent on other threads (to not make updates). For the wait-free solution: helper scheme. Updating threads shall help concurrent or next scan

An update takes a snapshot by calling scan, before it updates its register

Idea:
If a scanning thread fails to make a clean double collect, use the snapshot taken by the thread that performed overlapping update

…does not quite work

regarray: the registers constituting the state, one register per thread

```
public void update(T value) {
  int i = TheadID.get();
  T[] snap = scan(); // help by taking snapshot
  int t = regarray[i].stamp;
  Stamped<T> newval = new Stamped(t+1,value,snap);
  regarray[i] = newval;
}
```

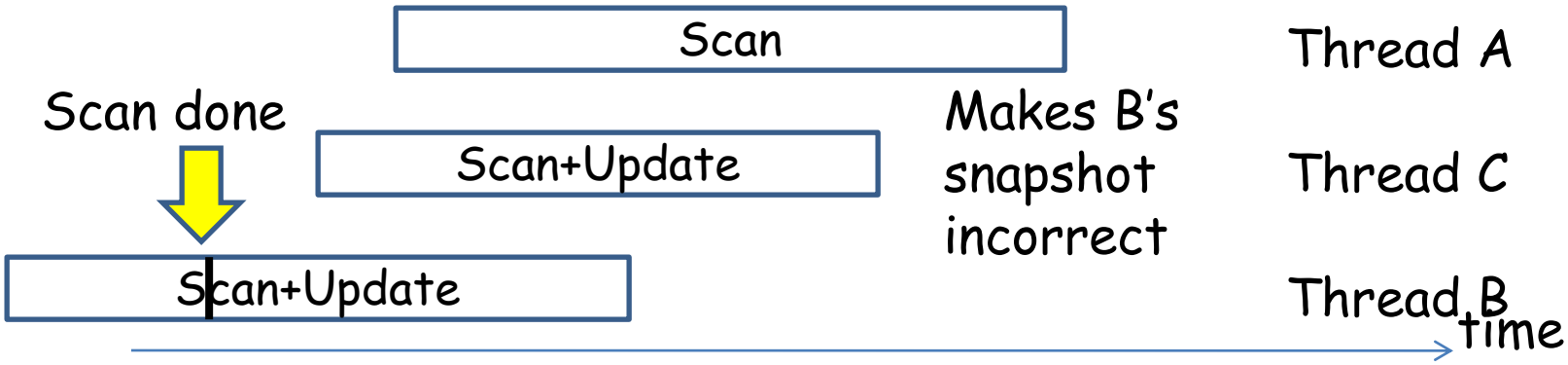
Registers now store a triplet: <timestamp,value,reference to snapshot>

Java note: memory management (free'ing allocated memory) is left to garbage collection system (must then be wait-free etc.!!)

A scan that repeatedly fails uses the snapshot taken by one of the concurrent updates

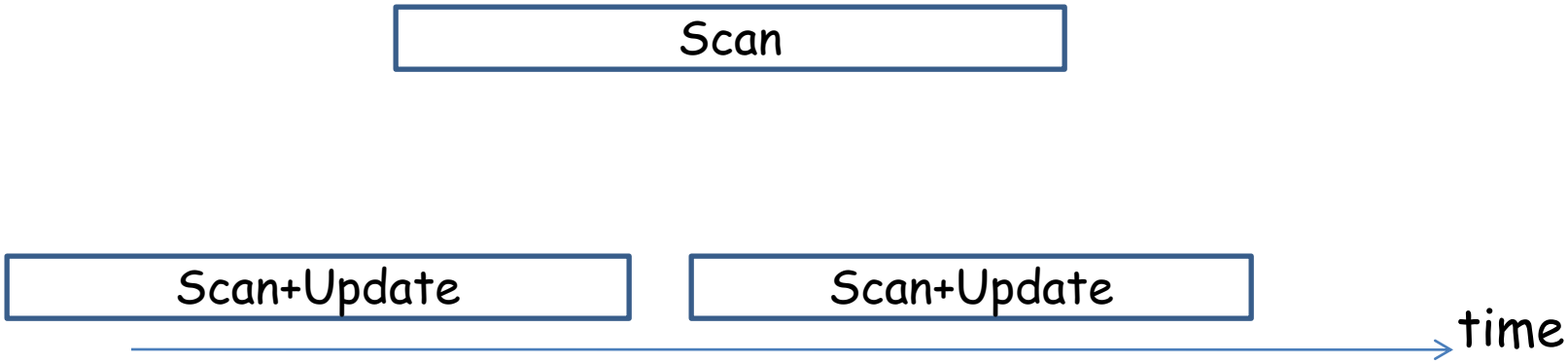
A thread is said to move if it completes an update() operation.

Idea: When a scan by A fails, because B has moved, take B's snapshot? Does not work since B could have completed its scan before A started scanning, so the snapshot was outside A's interval



Repair: If A sees B move twice while performing collects, then B's most recent snapshot is within A's interval

Scan: Perform double collect, if not clean, let some concurrent thread move twice and use this thread's snapshot



```
public T[] scan() {
  Stamped<T> oldsnap, newsnap;
  boolean[] moved = new boolean[threads]; // all false

  oldsnap = collect();
  collect: while (true) {
    newsnap = collect(); // double collect
    for (j=0; j<threads; j++) { // clean?
      if (oldsnap[j].stamp!=newsnap[j].stamp) {
        if (moved[j]) return newsnap[j].snap;
        else {
          moved[j] = true;
          oldsnap = newsnap;
          continue collect;
        }
      }
    } // clean: allocate and copy into local array, done
} }
```

Observations:

- If a thread makes a clean double collect, the snapshot is correct (a state that existed in the registers in the interval of the scan)
- If A sees a change in B's register (time-stamp) during two different collects, then the value in B's register read during the last collect was written by an update() that began some time after the first collect started.
- It follows that the snapshot returned by a scan() corresponds to a state in the interval of the scan (clear if A made a clean double collect; otherwise, B's scan(), which is inside A's interval was from a clean double collect, or some other embedded scan by C made a clean double collect… But there can be at most n-1 embedded updates, n being the number of threads)

Lemma: Every scan() or update() operation completes after at most $O(n^2)$ and at least $\Omega(n)$ register reads or writes, where n is the number of threads

Proof: For any scan() after at most n+1 double collects, either one is clean, or some thread is observed to move twice (in which case its snapshot is returned); each collect takes n reads

Corollary: The snapshot implementation is wait-free

Extension and simplification

Problems:
- MRSW registers
- Triple <stamp,value,reference> in registers

The idea can easily be extended to MRMW registers: Keep track of which tread updated a register.

Let `regarray[]` be an array of m MRMW registers

Each thread has a reference to a snapshot (array of m values)

```
public void update(int i, T value) {
  int w = TheadID.get();
  int t = regarray[i].stamp;
  Stamped<T> newval = new Stamped(t+1,w,value);
  regarray[i] = newval;
  snap[w] = scan(); // help after update
}
```

The register array now stores triples <stamp,thread id,value> which might fit better in register with finite number of bits (next stamp could also be maintained thread locally).

The helper snapshot is taken after the register update (a reference to the array of values computed by scan())

```
public T[] scan() {
  Stamped<T> oldsnap, newsnap;
  boolean[] moved = new boolean[threads]; // all false

  oldsnap = collect();
  collect: while (true) {
    newsnap = collect(); // double collect
    for (j=0; j<m; j++) { // all m registers clean?
      if (oldsnap[j]!=newsnap[j]) {
        int w = newsnap[j].w; // which thread moved?
        if (moved[w]) return snap[w];
        else {
          moved[w] = true;
          oldsnap = newsnap;
          continue collect;
        }
      }
    } // clean: allocate and copy into local array, done
} }
```

The correctness proof is a bit more involved, but establishes strong "freshness" properties of the snapshot. Also extends to "partial snapshots" (a subset of the registers is read)

> Damien Imbs, Michel Raynal: Help when needed, but no more: Efficient read/write partial snapshot. J. Parallel Distrib. Comput. 72(1): 1-12 (2012)

Issue: Memory management, when can a snapshot be thrown away?

Other snapshot results and algorithms

Yehuda Afek, Gideon Stupp, Dan Touitou: Long-lived and adaptive atomic snapshot and immediate snapshot (extended abstract). PODC 2000: 71-80
James H. Anderson: Composite Registers. Distributed Computing 6(3): 141-154 (1993)
James H. Anderson: Multi-Writer Composite Registers. Distributed Computing 7(4): 175-195 (1994)
Hagit Attiya, Ophir Rachman: Atomic Snapshots in O(n log n) Operations. SIAM J. Comput. 27(2): 319-340 (1998)
Elizabeth Borowsky, Eli Gafni: Immediate Atomic Snapshots and Fast Renaming (Extended Abstract). PODC 1993: 41-51
Prasad Jayanti: An optimal multi-writer snapshot algorithm. STOC 2005: 723-732