TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Replication and Consistency

04 Foundations of Shared Memory

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Winter Term 2019

Thank you!

These slides are based on companion material of the following books:

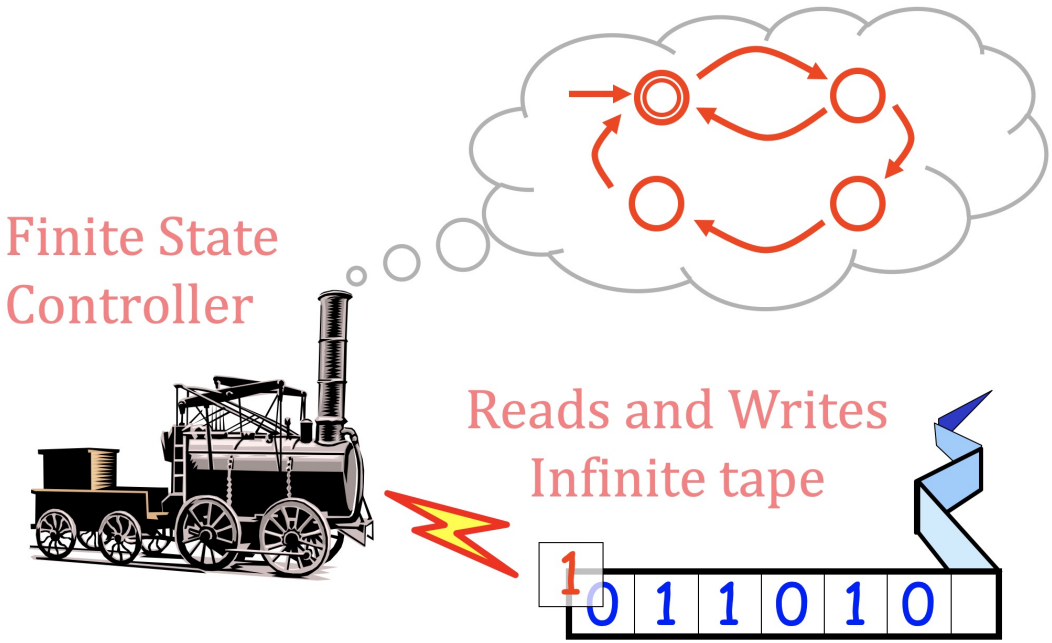
- **The Art of Multiprocessor Programming** by Maurice Herlihy and Nir Shavit
- **Synchronization Algorithms and Concurrent Programming** by Gadi Taubenfeld

Last time

- Defined concurrent objects using linearizability and sequential consistency
- *Fact:* Implemented linearizable objects (Two-thread FIFO Queue) in read-write memory without mutual exclusion
- *But:* Hardware does not provide linearizable read-write memory

What is the weakest form of communication that supports mutual exclusion?
What is the weakest shared object that allows shared-memory computation?

The Turing Machine



- Mathematical model of computation
- Helped us understand what is and is not computable on a sequential machine
- Efficiency (mostly) irrelevant

Shared Memory Computability?

- Mathematical model of **concurrent** computation
- What is (and is not) concurrently computable
- Efficiency (mostly) irrelevant

Foundations of Shared Memory

To understand modern multiprocessors we need to ask some basic questions:

What is the **weakest** useful form of shared memory?

What **can** it do?

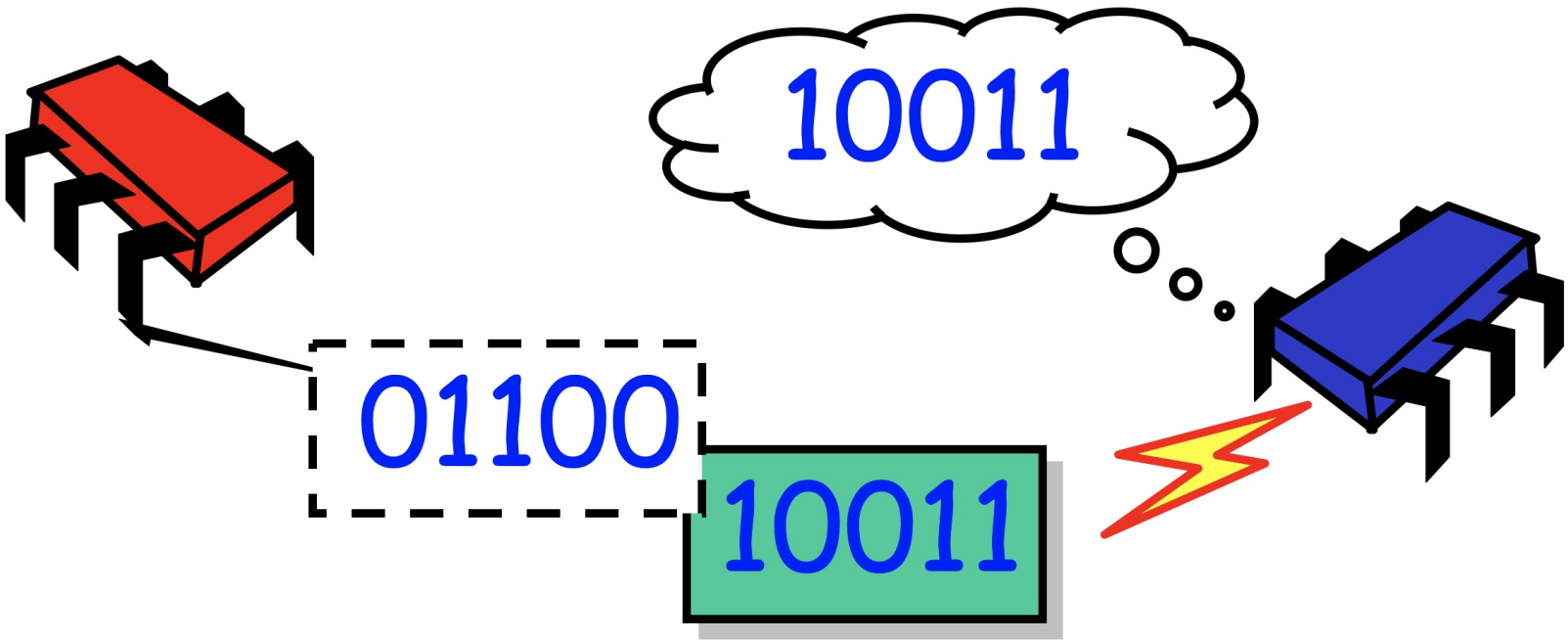
What **can't** it do?

Register(Lamport 1986)

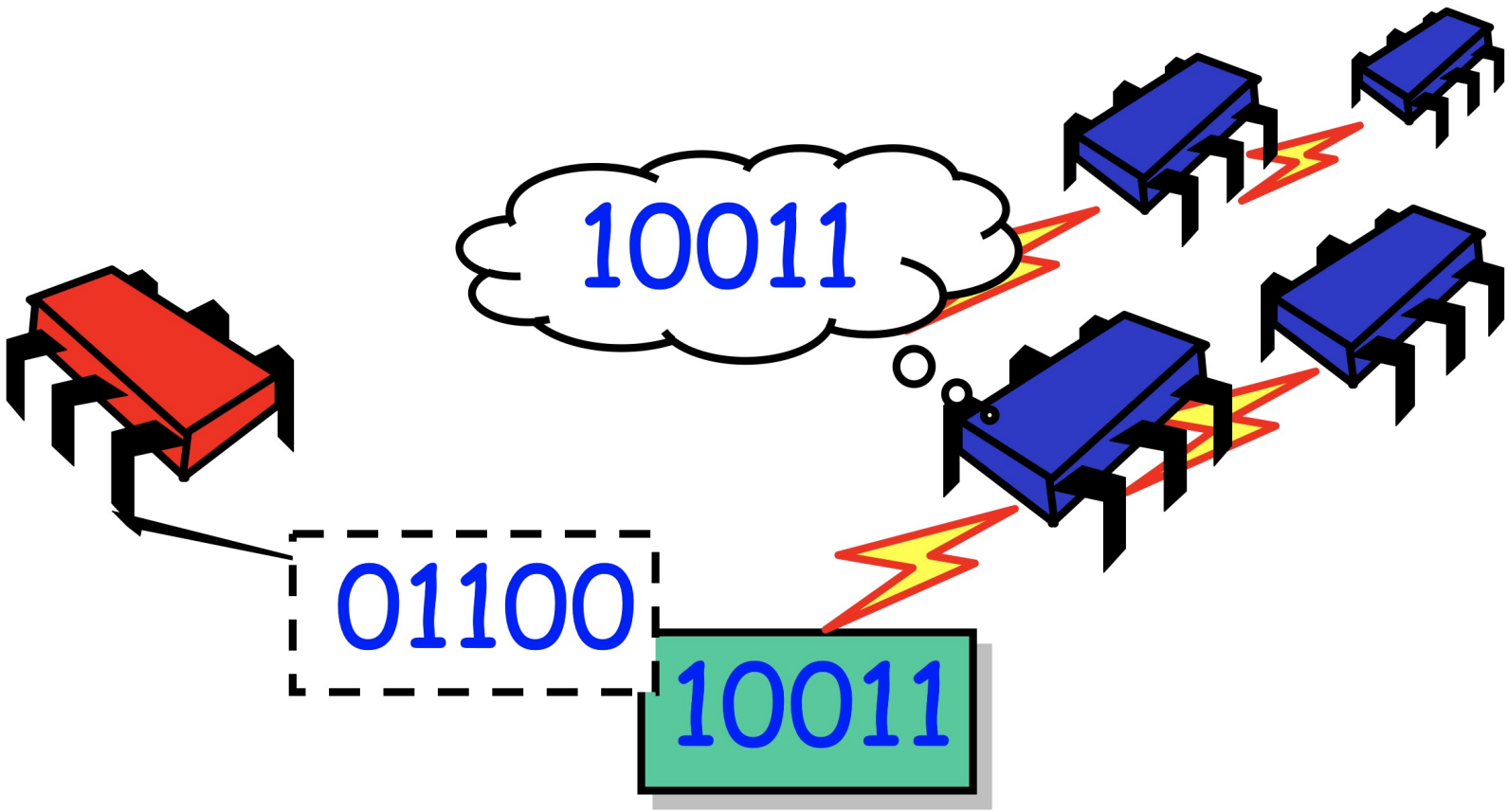
```
interface Register<T> {
  public T read();
  public void write(T v);
}
```

- A single memory location is (historically) named **register**.
- Holds a (binary) value
- Can be read and written
- In the following, type `T` is Boolean or m-bit Integer

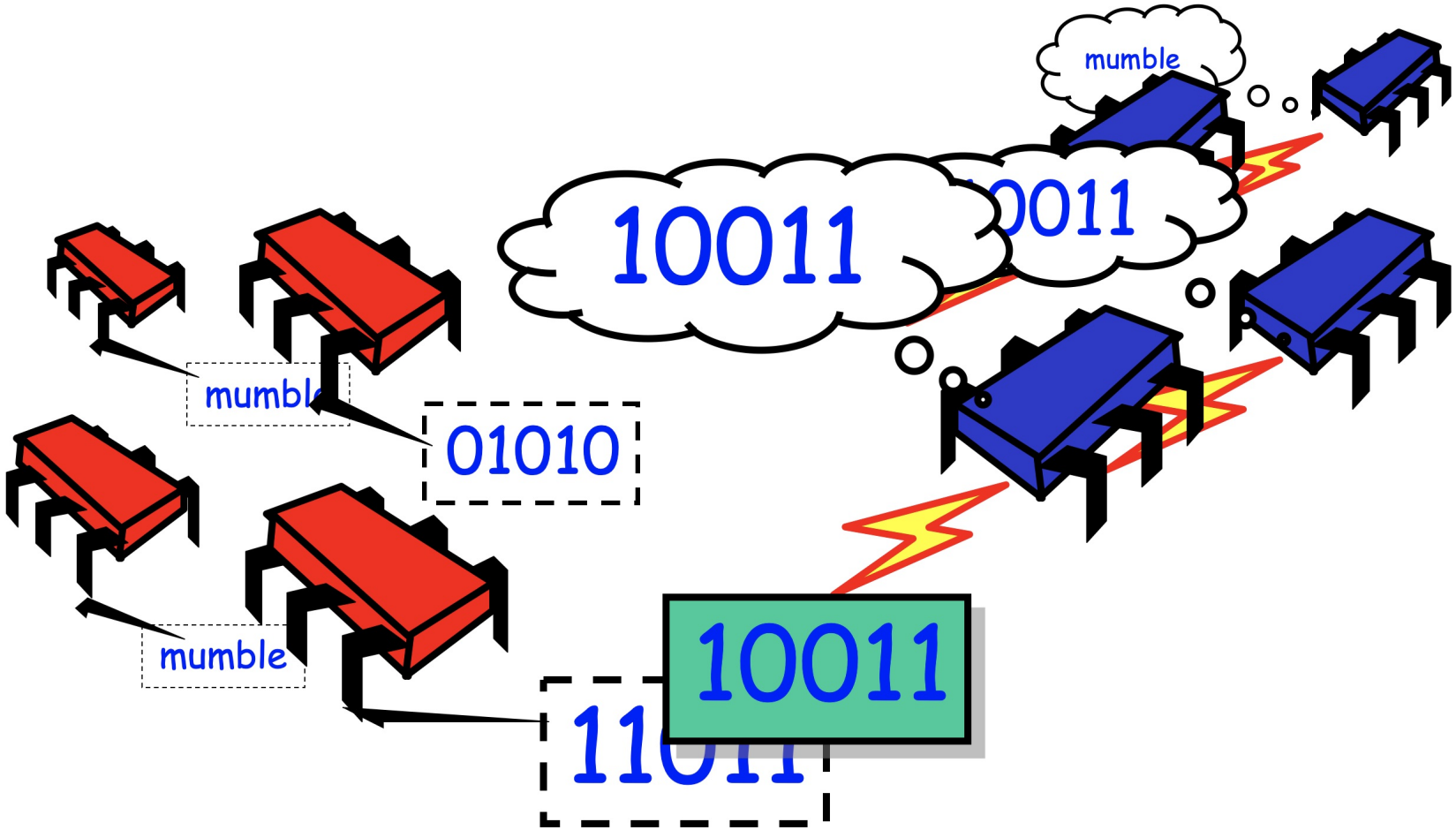
Single-reader/Single-writer Register (SRSW)



Multiple-reader/Single-writer Register (MRSW)



Multiple-reader/Multiple-writer Register (MRMW)



Safe Register

A single-writer, multi-reader register implementation is **safe** if a read that does not overlap a write returns the value written by the most recent write.

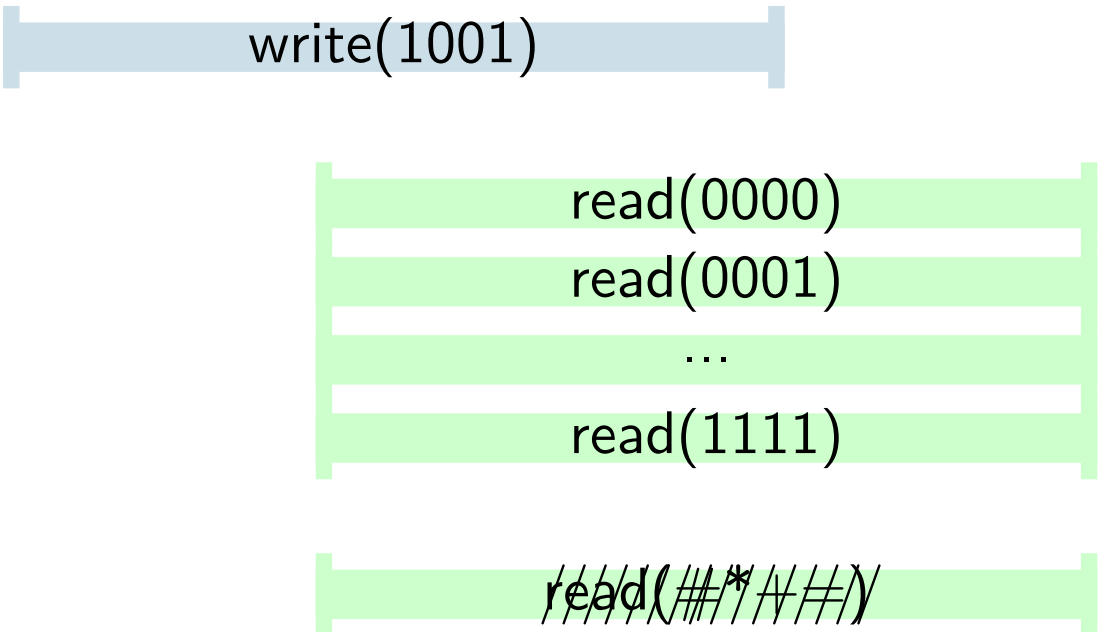
When reads and writes don't overlap:

write(1001)

read(1001)

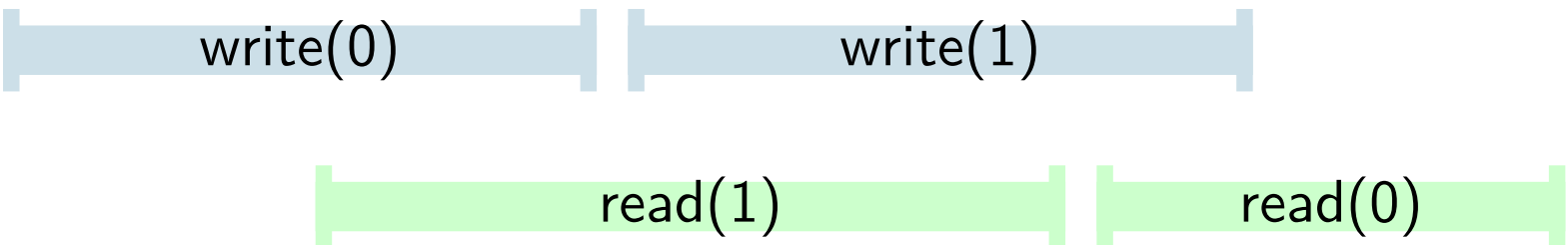
Safe Register

When reads and writes do overlap:



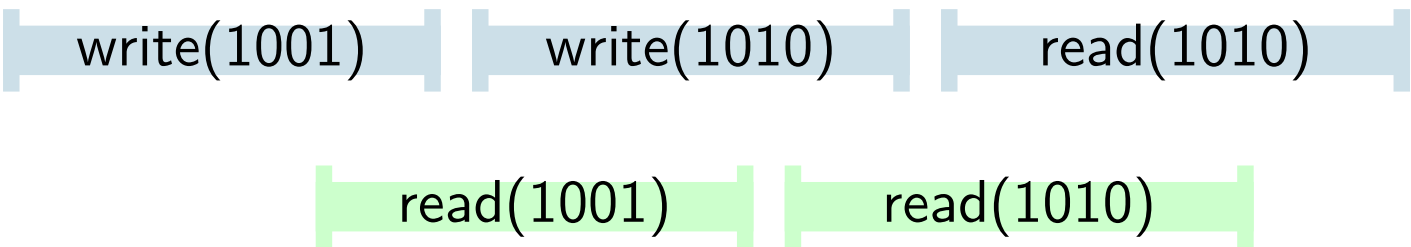
Regular Register

A single-writer, multi-reader register implementation is **regular** if it is safe and a read that overlaps with the i-th write call returns either the i^{th} or $(i-1)^{th}$ value.

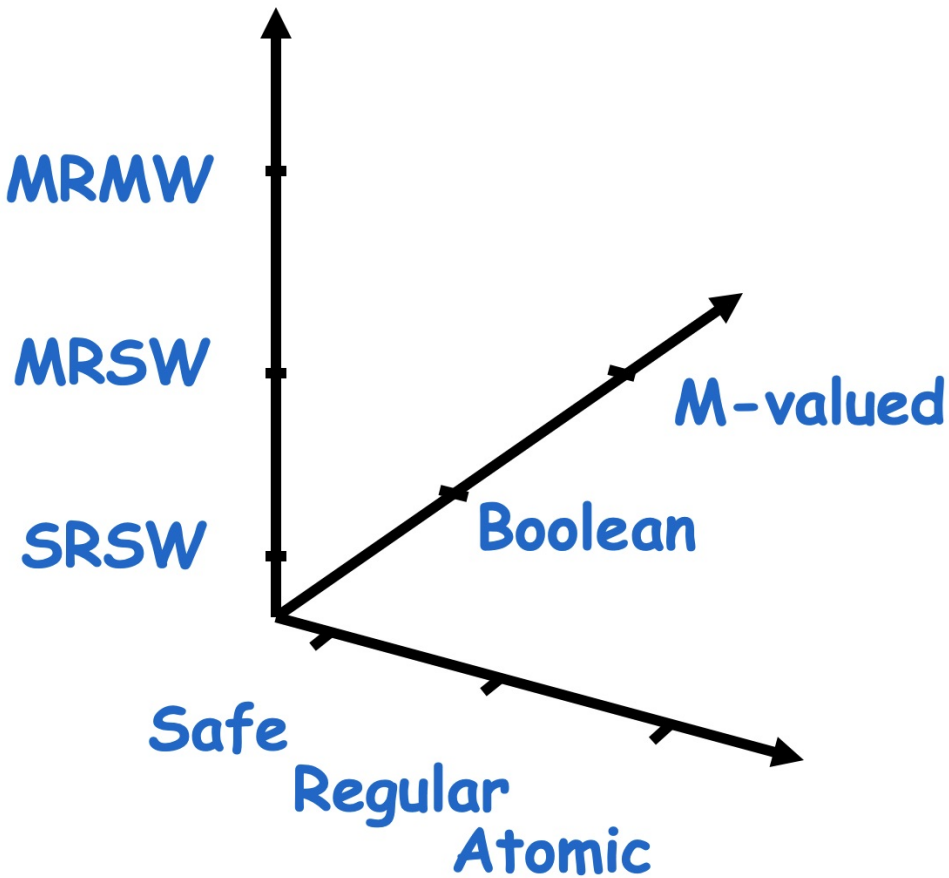


Atomic Register

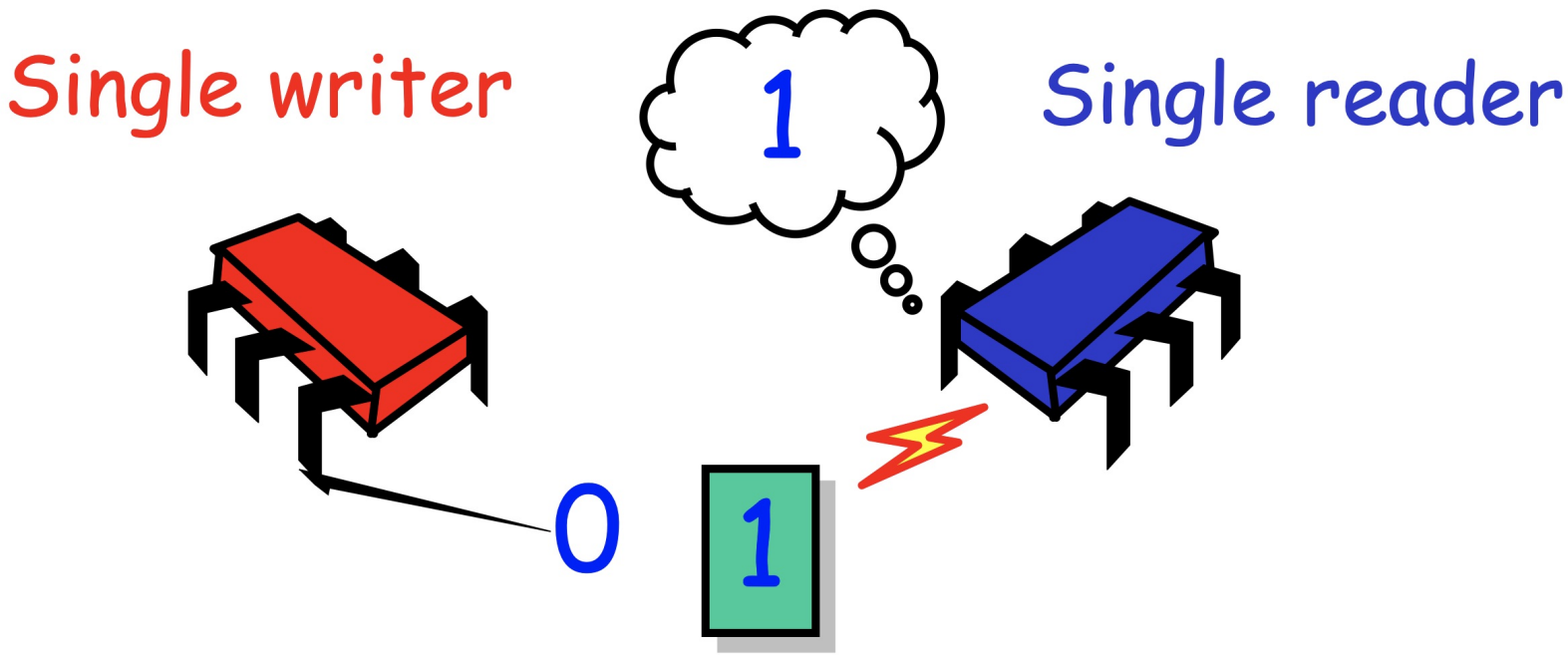
A single-writer, multi-reader register implementation is **atomic** if it is linearizable to a sequential safe register.



Summary: Classification of Registers



Weakest Register



Safe Boolean SRSW Register

The weakest Register is quite powerful!

From the SRSW safe Boolean register, we can construct:

- All the other registers
- Mutual exclusion

But not everything!

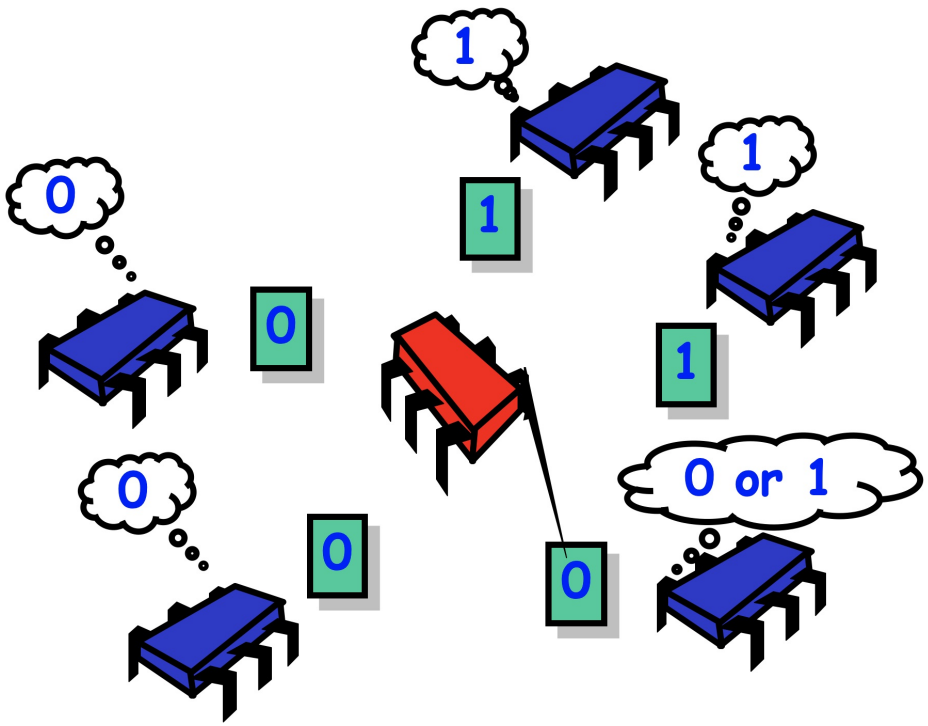
- Consensus hierarchy ($\Rightarrow$ next lecture!)

Roadmap

- **SRSW safe Boolean**
- **MRSW safe Boolean**
- MRSW regular Boolean
- MRSW regular M-valued
- MRSW atomic M-valued
- MRMW atomic M-valued
- Atomic snapshot

Safe Boolean MRSW from Safe Boolean SRSW

- Each thread has own safe SRSW register
- When writing, update (= write) each thread's register one at a time
- Read own register only



Safe Boolean MRSW from Safe Boolean SRSW

```
class SafeBoolMRSWRegister implements Register<Boolean> {

  private SafeBoolSRSWRegister[] r = new SafeBoolSRSWRegister[N];

  public void write(boolean x) {
   for (int j = 0; j < N; j++)
    r[j].write(x);
  }

  public boolean read() {
   int i = ThreadID.get();
   return r[i].read();
  }
}
```

Roadmap

- SRSW safe Boolean
- **MRSW safe Boolean**
- **MRSW regular Boolean**
- MRSW regular M-valued
- MRSW atomic M-valued
- MRMW atomic M-valued
- Atomic snapshot

Regular Boolean MRSW from Safe Boolean MRSW

- OK to read 0 **or** 1 if read is concurrent with a write that changed the value
- Must return the old if the value written is the same as the old one
- *Trick:* Remember the old value!

Regular Boolean MRSW from Safe Boolean MRSW

- OK to read 0 **or** 1 if read is concurrent with a write that changed the value
- Must return the old if the value written is the same as the old one
- *Trick:* Remember the old value!

Question

Does the construction also work for a safe **multi-valued** MRSW?

Regular Boolean MRSW from Safe Boolean MRSW

- OK to read 0 **or** 1 if read is concurrent with a write that changed the value
- Must return the old if the value written is the same as the old one
- *Trick:* Remember the old value!

Question

Does the construction also work for a safe **multi-valued** MRSW?
⇒ No! Boolean registers return 0 or 1 even if register value changes.
But safe m-valued register can return value in range other than old or new when value changes.

Regular Boolean MRSW from Safe Boolean MRSW

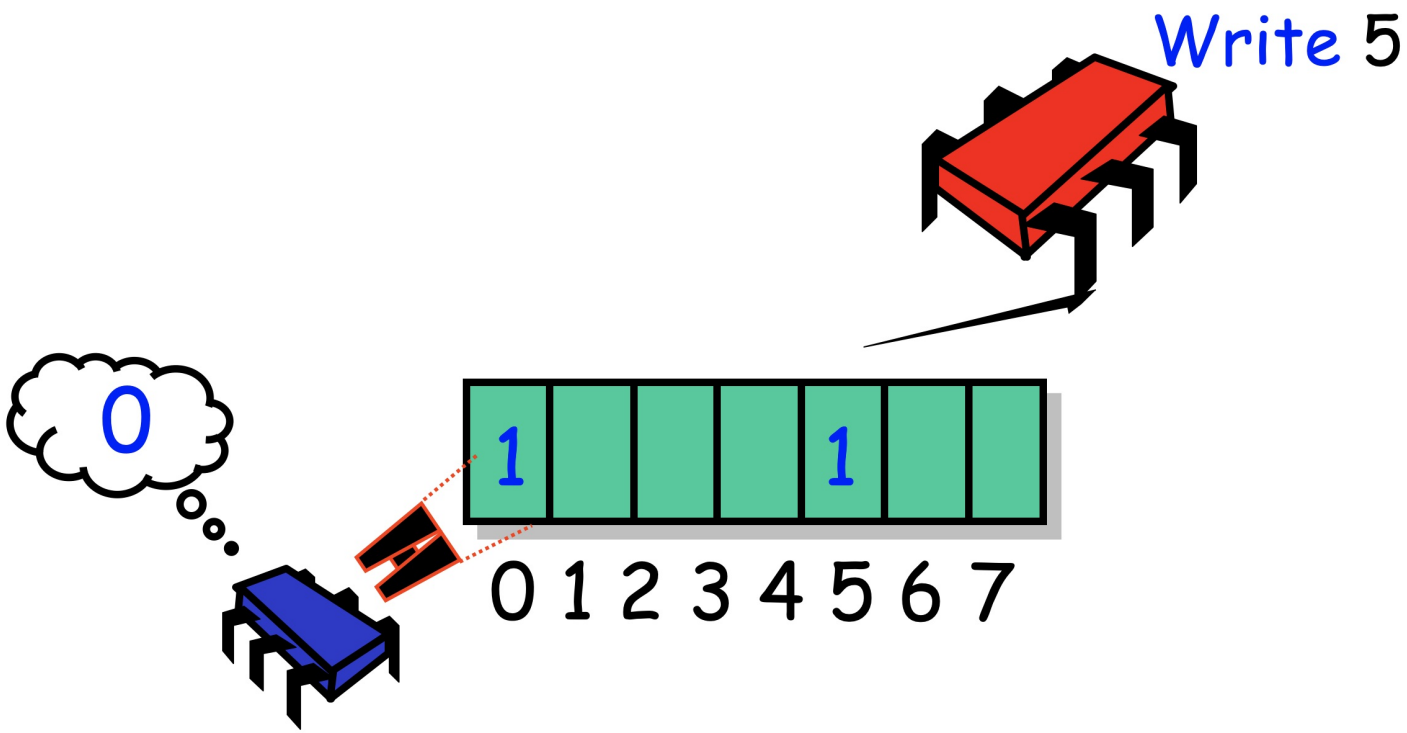
```
class RegBoolMRSWRegister implements Register<Boolean> {

  private ThreadLocal<Boolean> old; // local to the writer,
    cheating here on Java ...
  private SafeBoolMRSWRegister value; // actual value

  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }
  }

  public boolean read() {
   return value.read();
  }
}
```

Roadmap

- SRSW safe Boolean
- MRSW safe Boolean
- **MRSW regular Boolean**
- **MRSW regular M-valued**
- MRSW atomic M-valued
- MRMW atomic M-valued
- Atomic snapshot

MRSW Regular M-valued from MRSW Regular Boolean

- Unary representation of value: `bit[i]` means value `i`
- When writing, set bit `x` and clear bits from higher to lower
- When reading, scan from lower to higher and return first bit set



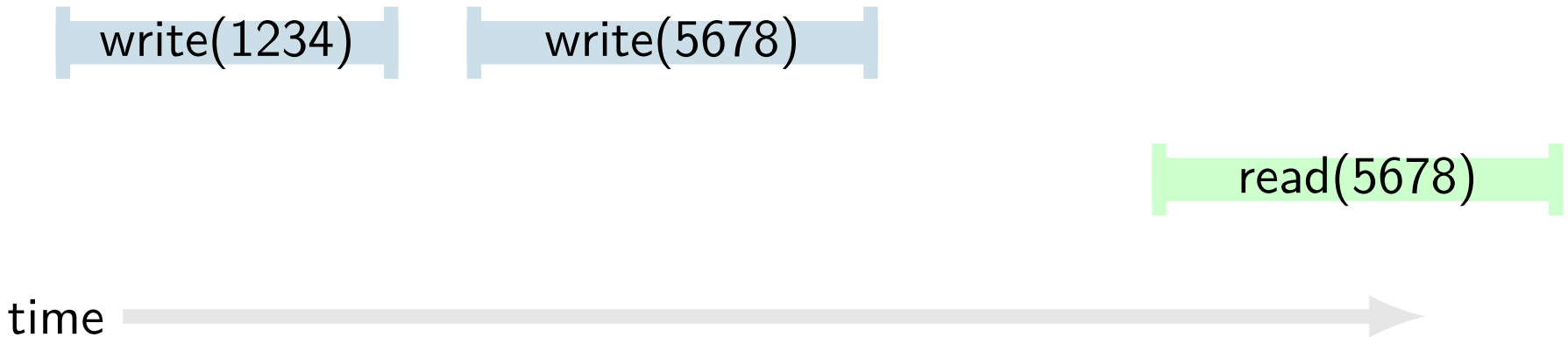
MRSW Regular M-valued from MRSW Regular Boolean

```
class RegMRSWRegister<T> implements Register<T> {
  private RegBoolMRSWRegister[M] bit;

  public void write(int x) {
    this.bit[x].write(true);
    for (int i=x-1; i>=0; i--)
      this.bit[i].write(false);
  }

  public int read() {
    for (int i=0; i < M; i++)
      if (this.bit[i].read())
        return i;
    }
}
```

Roadmap

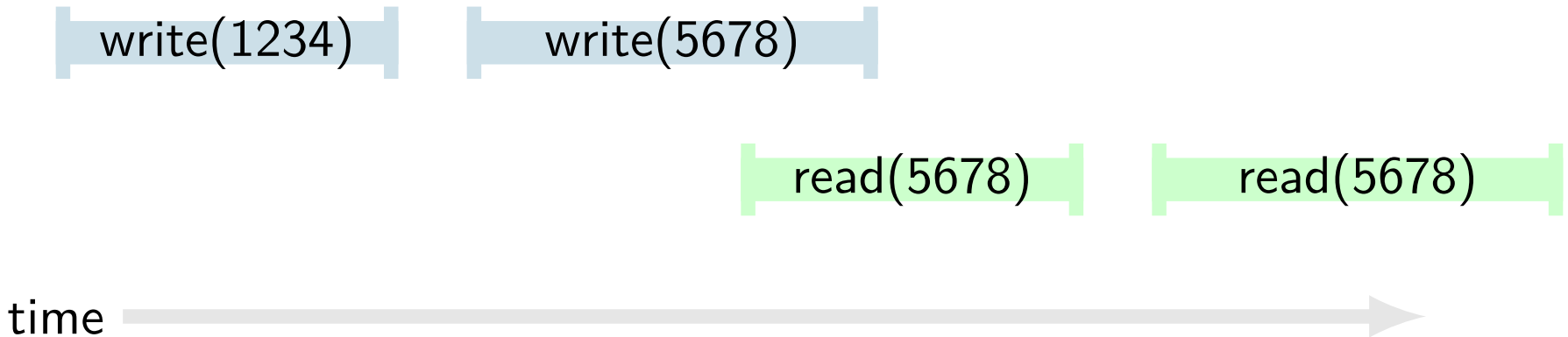
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- **MRSW regular M-valued**
- **MRSW atomic M-valued**
- MRMW atomic M-valued
- Atomic snapshot

Detour: SRSW Atomic from SRSW Regular



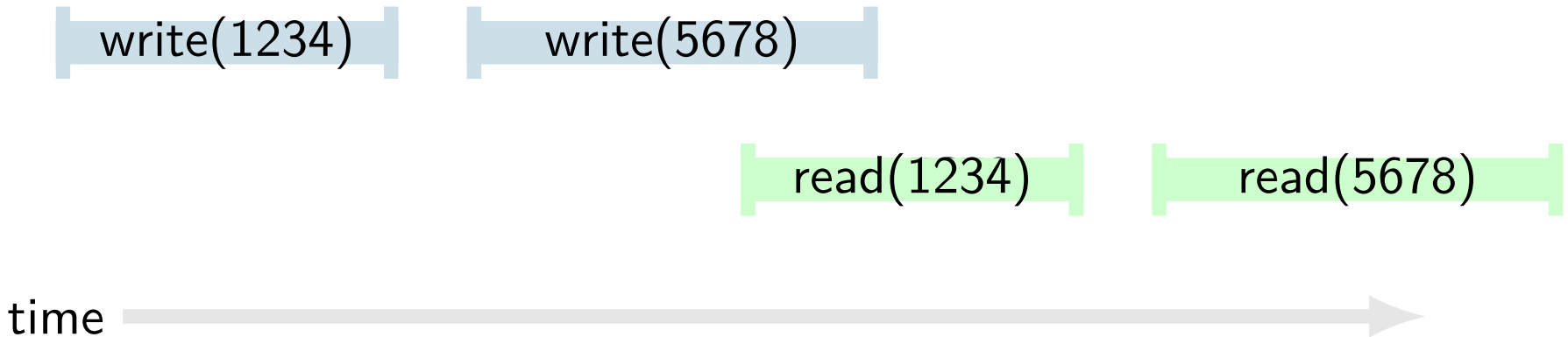
When does the behavior of atomic and regular differ?

Detour: SRSW Atomic from SRSW Regular



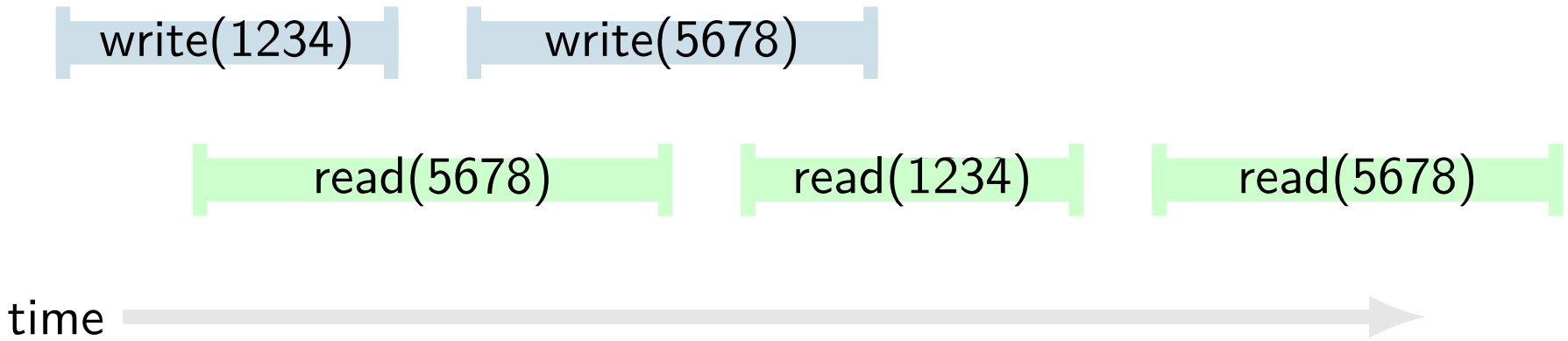
When does the behavior of atomic and regular differ?

Detour: SRSW Atomic from SRSW Regular



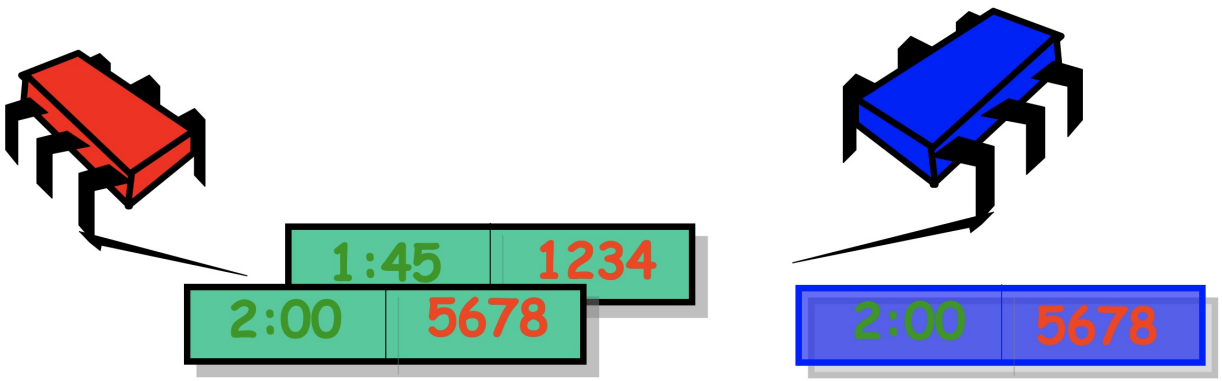
When does the behavior of atomic and regular differ?

Detour: SRSW Atomic from SRSW Regular



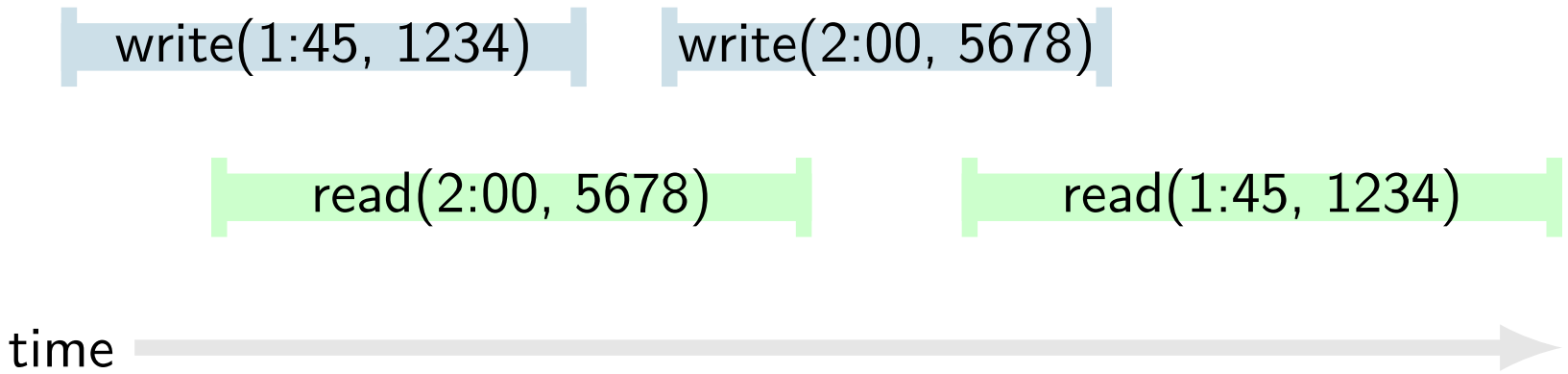
When does the behavior of atomic and regular differ?

Trick: Timestamped Values



- Writer writes value and stamp together
- Reader saves last read (value + stamp) and returns new value only if new value has a higher stamp

SRSW Atomic from SRSW Regular



Since `(1:45, 1234)` is more recent than `(2:00, 5678)`, the thread returns its prior read value `5678`.

Towards Atomic Single Reader to Atomic Multi-Reader

thread	(stamp, value)
T_0	(2:00, 5678)
T_1	(2:00, 5678)
T_2	(1:45, 1234)

- Writer starts writing at `2:00`
- After updating the entry for reader thread T_1, the writer falls asleep. . . .
- When T_1 reads its entry, it sees the updated value
- When later T_2 reads its entry, it sees the old value

$\Rightarrow$ Not atomic!

Towards Atomic Single Reader to Atomic Multi-Reader

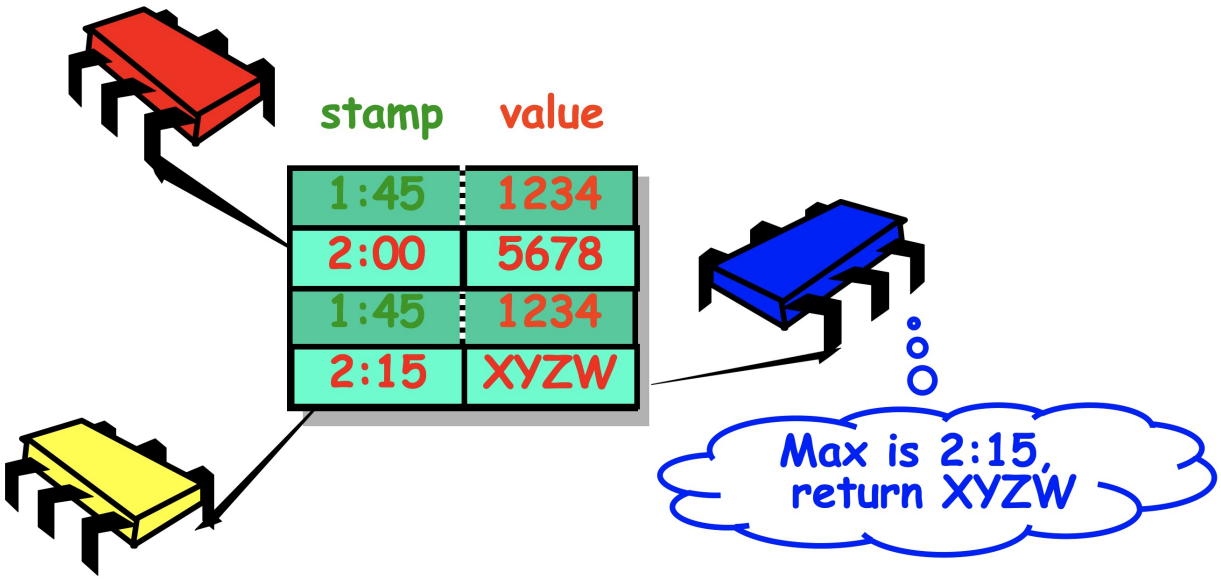
thread	T_0	T_1	T_2
T_0	(2:00, 5678)	(1:45, 1234)	(1:45, 1234
T_1	(2:00, 5678)	(1:45, 1234)	(1:45, 1234
T_2	(1:45, 1234)	(1:45, 1234)	(1:45, 1234

- Writer writes its own column
- Reader reads its own row and updates all entries in its own column to notify others about updates
- In the example, T_0 starts updating its column
- Now, T_2 reads the stamp+value updated by T_1 and returns the new value
- The "bad" case only happens if one read happens before the other; for concurrent reads, both old and new value are ok

Roadmap

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular M-valued
- **MRSW atomic M-valued**
- **MRMW atomic M-valued**
- Atomic snapshot

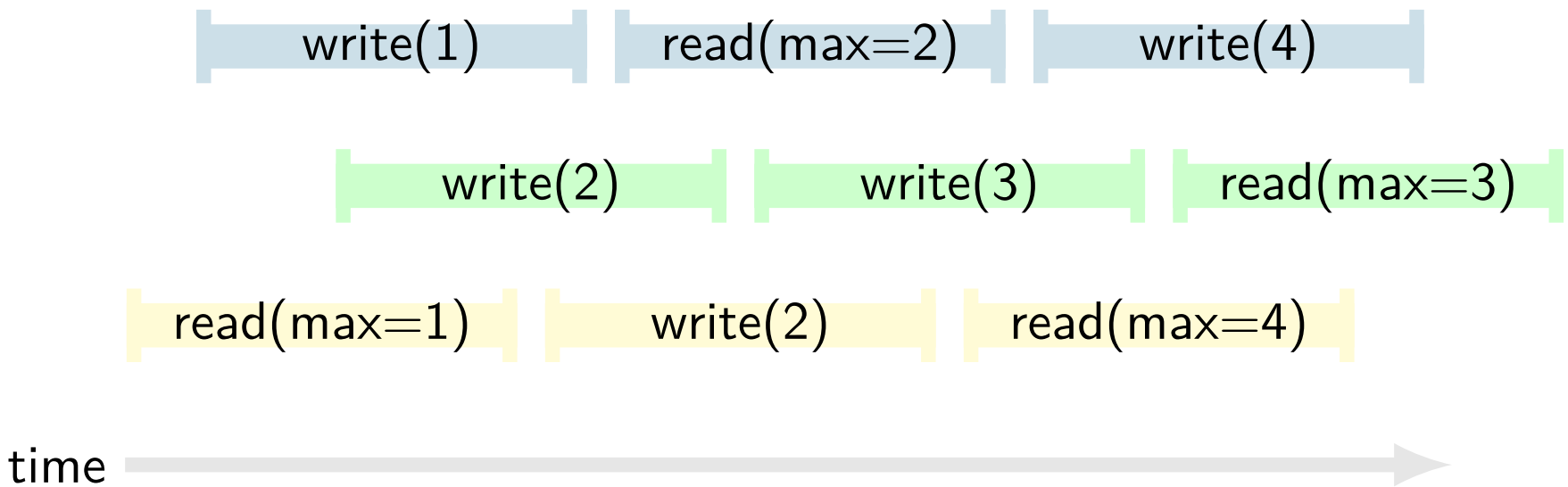
Multi-Writer Atomic from Multi-Reader



- Each writer reads all entries, then writes entry with strictly larger timestamp to its own register
- Readers read all entries and return maximum wrt lexicographic order (like Bakery Algorithm)

Quizz: Is this execution atomic (= linearizable)?

Here: Only timestamps!



Remarks

- First, order write by time stamp
  - Later writes must have strictly greater stamps
  - Concurrent writes can have the same stamps
- Then, order reads such that each read's linearization point is right after the timestamp that it read for `max`
  - Later reads must read the same or a greater stamp

Roadmap

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular M-valued
- MRSW atomic M-valued
- MRMW atomic M-valued
- **Atomic snapshot**

Atomic Snapshots

Problem description:

- Assume you have an array of SWMR atomic registers
- Take instantaneous snapshot of all registers without "stopping the world" (i.e. wait-free)

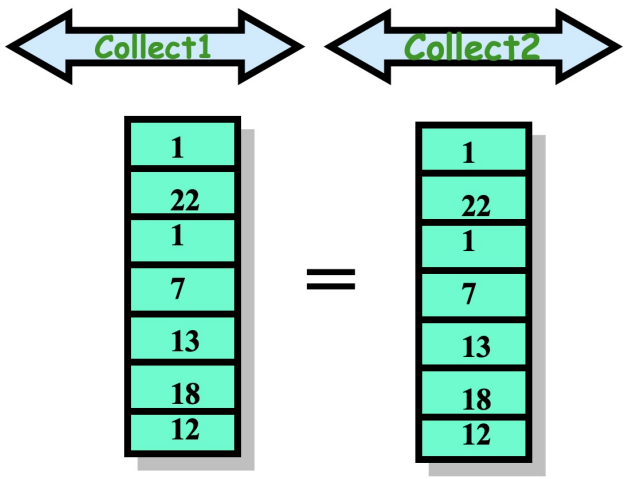
```
interface Snapshot {
  public int update(int v);   // Thread i writes v to its register
  public int[] scan();        // Instantaneous snapshot of all
    threads' registers
}
```

Problem

- Incompatible concurrent collects
- Result not linearizable

Simple Snapshot

- *Idea:* Use **clean collect** (i.e. collect during which nothing changed)
- Put increasing labels (= timestamps) on each entry
- Collect twice
  - If both results are identical, we are done
  - Otherwise, try again



Simple Snapshot: Update

```
class SimpleSnapshot implements Snapshot {
  private AtomicMRSWRegister[] register;

  public void update(int value) {
    int i = Thread.myIndex();
    LabeledValue oldValue = register[i].read();
    LabeledValue newValue =
        new LabeledValue(oldValue.label+1, value);
    register[i].write(newValue);
  }
...
}
```

Simple Snapshot: Scan

```
public int[] scan() {
  LabeledValue[] oldCopy, newCopy;
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    if (!equals(oldCopy, newCopy)) {
      oldCopy = newCopy;
      continue collect;
    }
  }
  return getValues(newCopy);
}
```

Simple Snapshot: Collect

```
private LabeledValue[] collect() {
  LabeledValue[] copy = new LabeledValue[n];
  for (int j = 0; j < n; j++)
    copy[j] = this.register[j].read();
  return copy;
}
```

Properties of Simple Snapshot

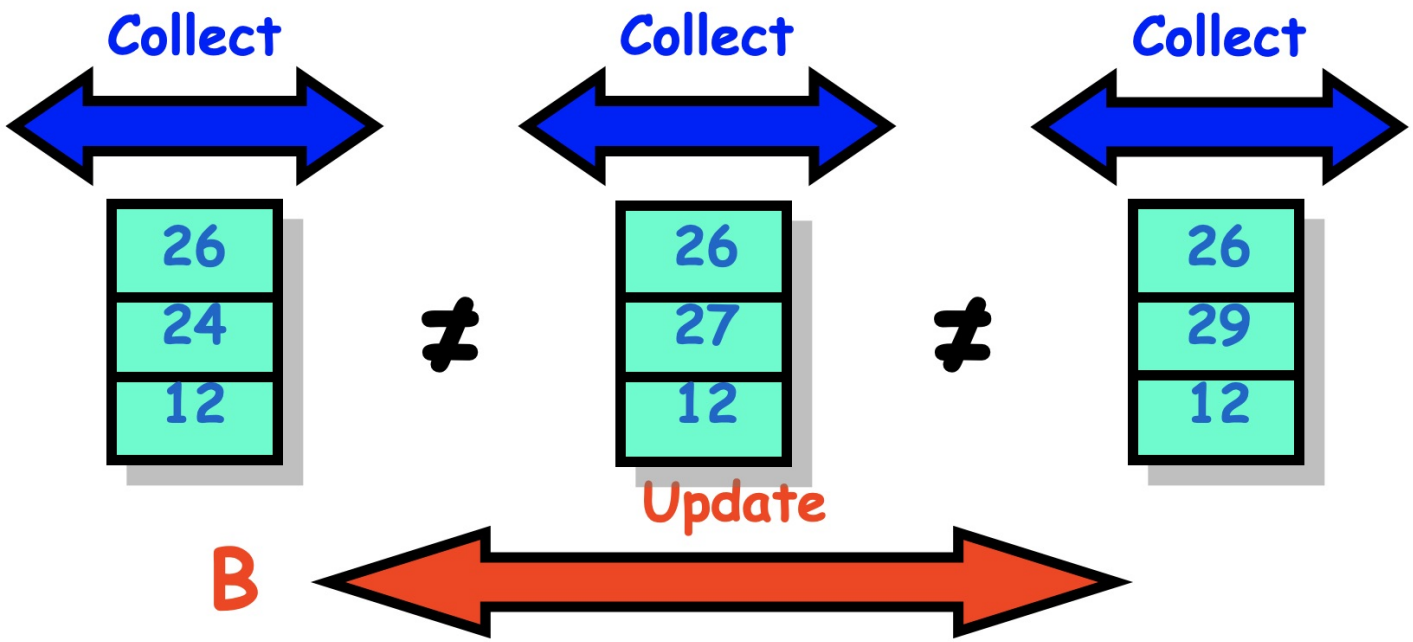
- Linearizable
- `update` is wait-free
  - No unbounded loops
- But `scan` can starve if interrupted by concurrent update

Wait-free Snapshot

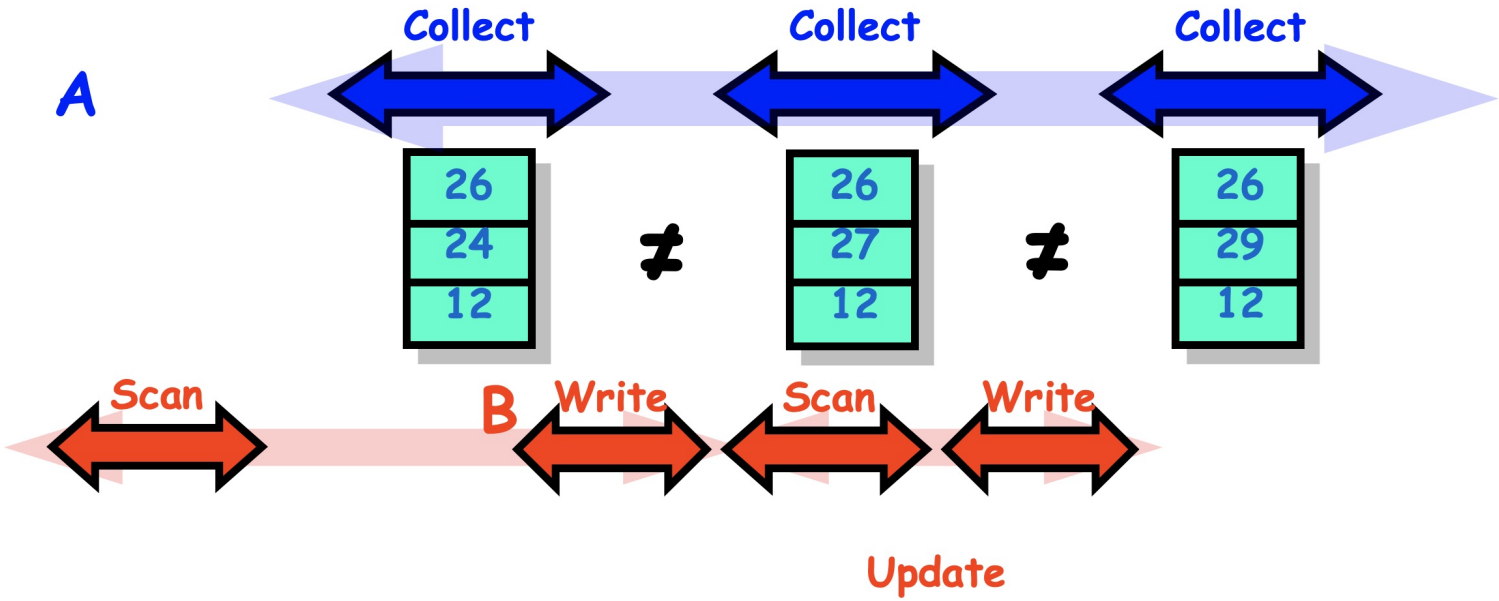
- Add a scan before every update
- Write resulting snapshot together with update value
- If scan is continuously interrupted by updates, scan can take the update's snapshot

Idea

If A's scan observes that B moved twice, then B completed a full update while A's scan was in progress

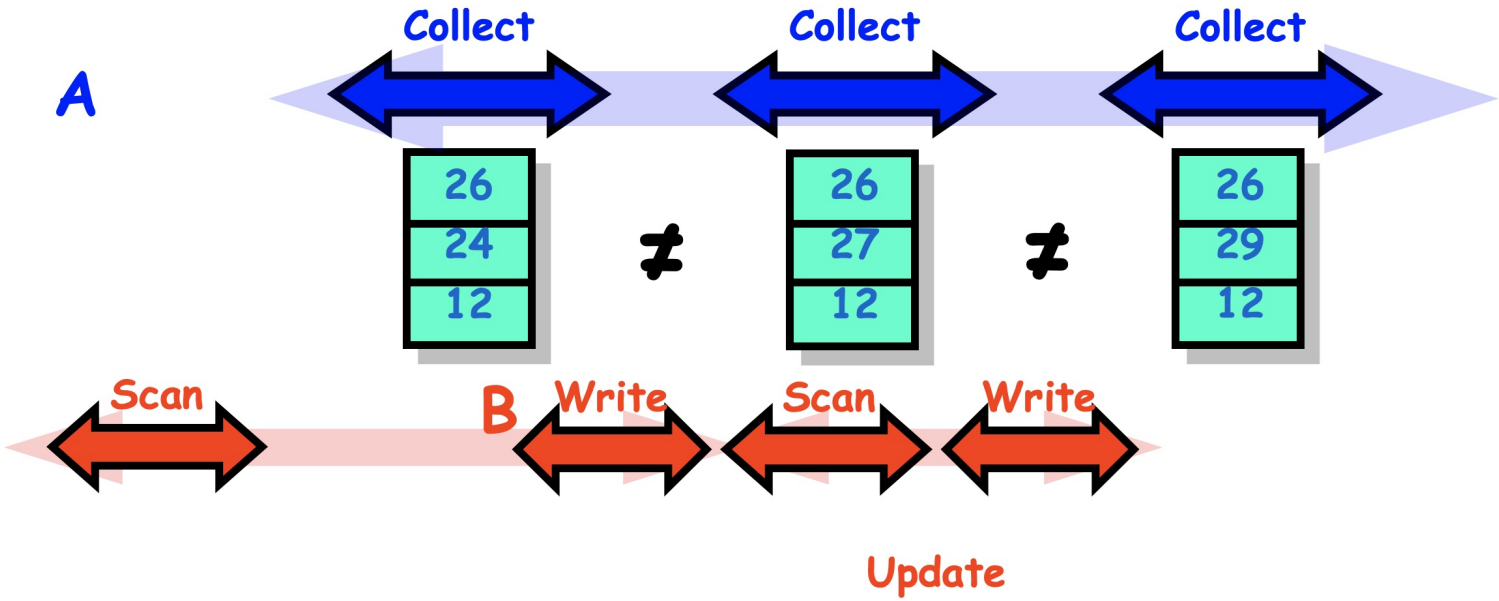


Idea



- B's first update must have been written during the first collect
- Scan of B's second update must be within the interval of A's scan
- So A can steal result of B's scan!

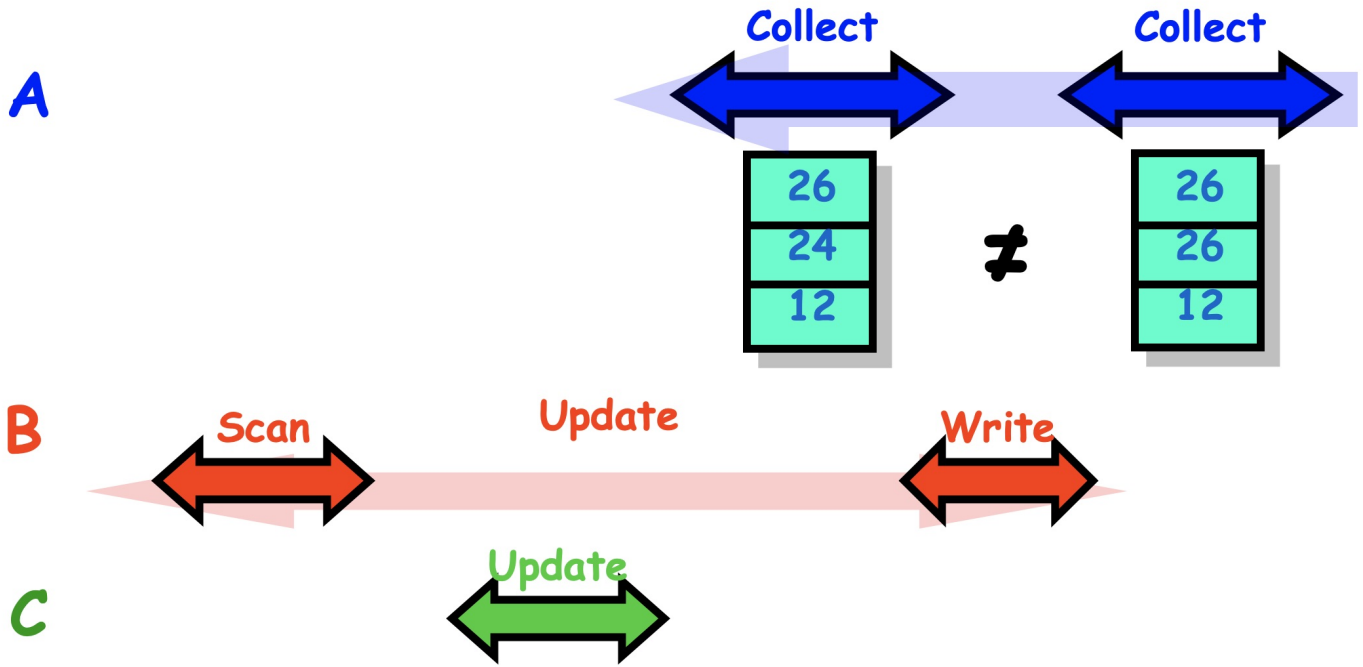
Idea



- B's first update must have been written during the first collect
- Scan of B's second update must be within the interval of A's scan
- So A can steal result of B's scan!

- Why can't we use the scan of B's first update?

Once is not enough



- Another thread might have interfered before A's scan started!
- *Insight:* If we collect n times, some thread must have obtained a clean scan (pigeon-hole principle)

Wait-Free Scan

- Each scan gets a clean collect or is interrupted while taking its scan
- Scan of the interrupting thread could also have been interrupted etc.
- But this can however happen only $n-1$ times after which the are no more threads that can inturrupt

$\Rightarrow$ Scan is wait-free

Wait-free Snapshot: Label

```
class SnapValue {
  int   label; // counter as timestamp, incremented with each
    snapshot
  int   value; // actual value
  int[] snap;  // most recent snapshot
}
```

Wait-free Snapshot: Update

```
public void update(int value) {
  int i = Thread.myIndex();
  // take scan
  int[] snap = this.scan();
  SnapValue oldValue = r[i].read();
  // label value with scanned snap
  SnapValue newValue =
        new SnapValue(oldValue.label+1, value, snap);
  r[i].write(newValue);
}
```

Wait-free Snapshot: Scan

```
public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  // keep track of who moved
  boolean[] moved = new boolean[n];
  // repeat double collect
  oldCopy = collect();
  collect: while (true) {
    newCopy = collect();
    // if missmatch detected... (next slide)
    for (int j = 0; j < n; j++) {
      if (oldCopy[j].label != newCopy[j].label) {
      ... // see next slides!
      }
    }
    return getValues(newCopy);
  }
}
```

Wait-free Snapshot: Mismatch detected

```
if (oldCopy[j].label != newCopy[j].label) {
  if (moved[j]) { // if second move
    return newCopy[j].snap; // steal its second snapshot!
  } else {
    moved[j] = true;  // remember move
    oldCopy = newCopy;
    continue collect;
  }
}
```

Wait-free Snapshot: Properties

- Uses unbounded counters
  - Can be replaced with 2 bits
- Assumes SWMR registers for labels
  - Can be extended to MRMW

Summary

- Construction of MRMW M-valued snapshot objects form SRSW binary safe registers
- What are open research directions here?
  - Atomic writes to multiple locations!
  - Transactional memory (TM)

Copyright

Further reading

Lamport, Leslie. 1986. "On Interprocess Communication. Part II: Algorithms." *Distributed Computing* 1 (2): 86–101. https://doi.org/10.1007/BF01786228.