Concurrent programming

Foundations of Shared Memory

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy, Nir Shavit, Victor Luchangco,
and Michael Spear

Modified by Piotr Witkowski

Last Lecture

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

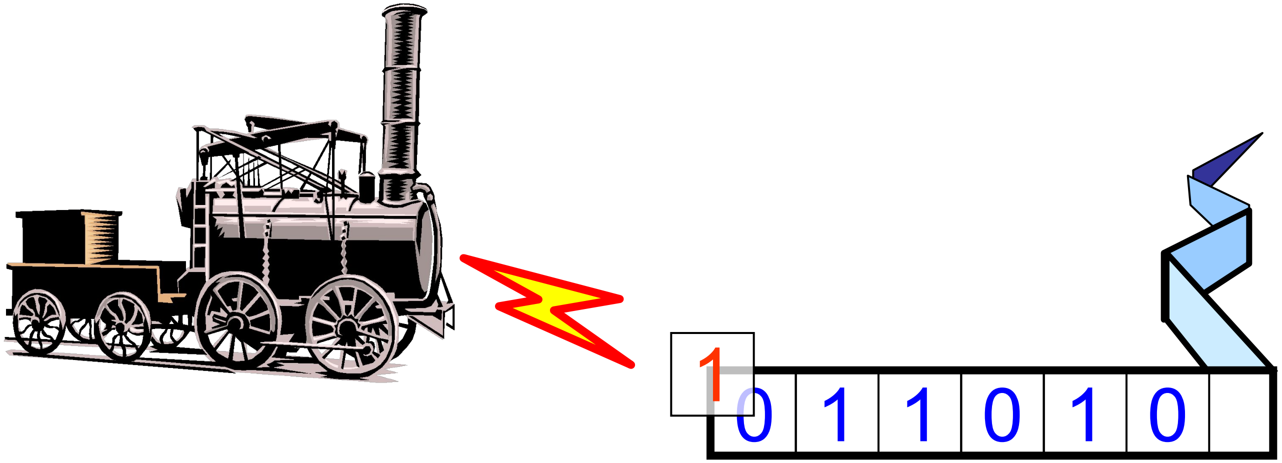
Fundamentals

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

Alan Turing

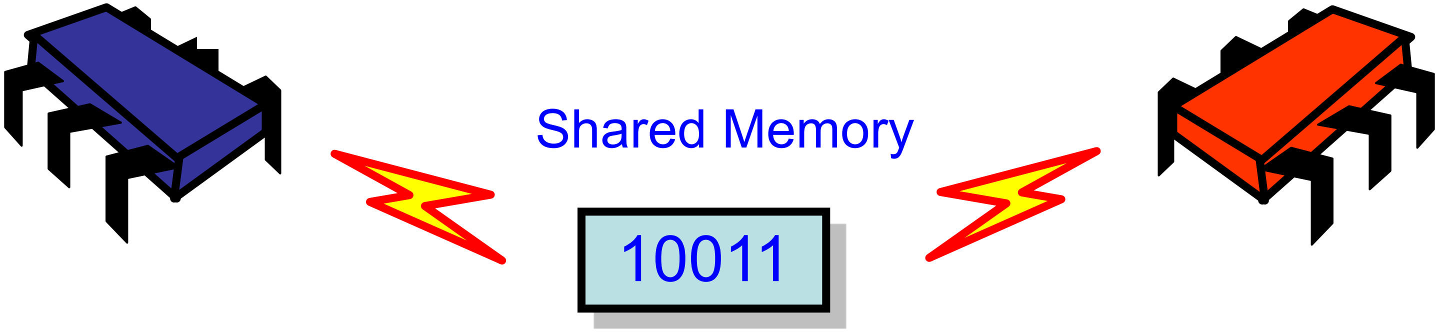
- Showed what is and is not computable on a sequential machine.
- Still best model there is.

Turing Computability



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

Shared-Memory Computability?



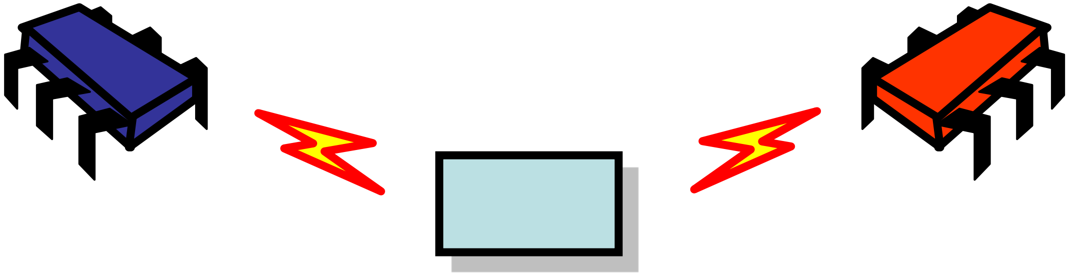
Shared Memory

10011

- 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 …



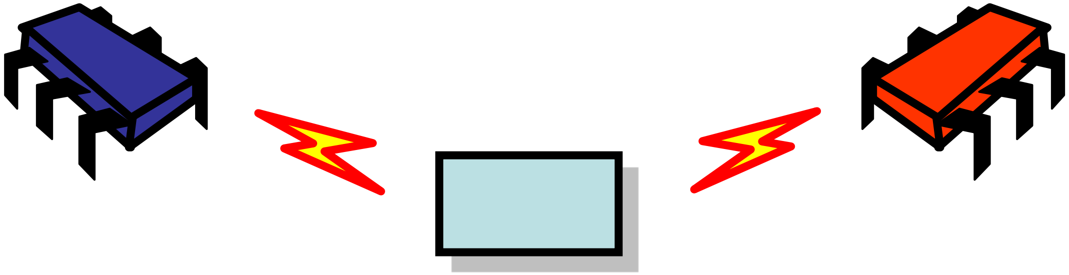
Foundations of Shared Memory

To understand modern

r

basic questions …

What is the weakest useful form of shared memory?



Foundations of Shared Memory

To understand modern

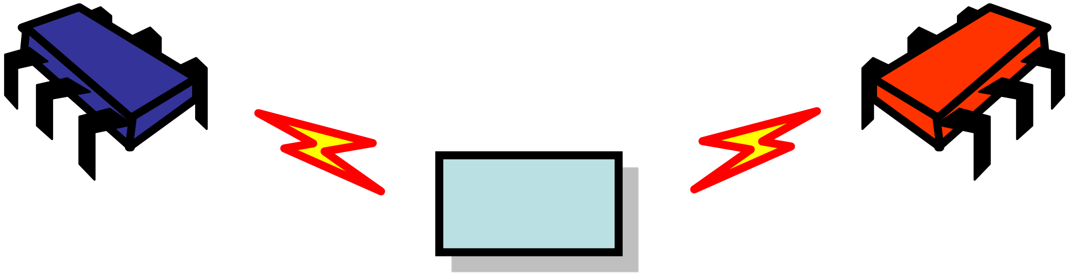
r

basic questions …

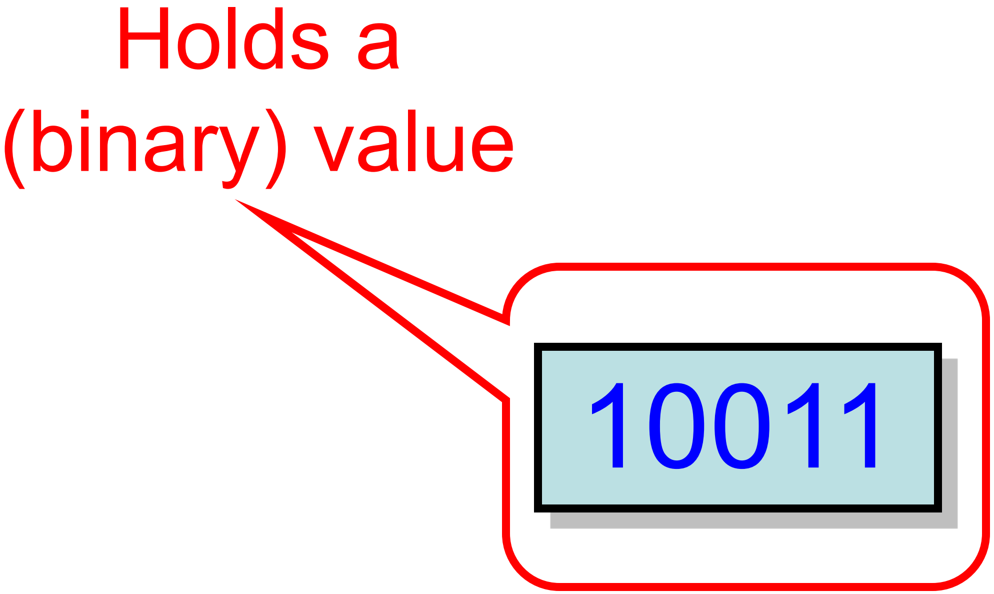
What is the weakest useful form of

shared memory?

What can it do?

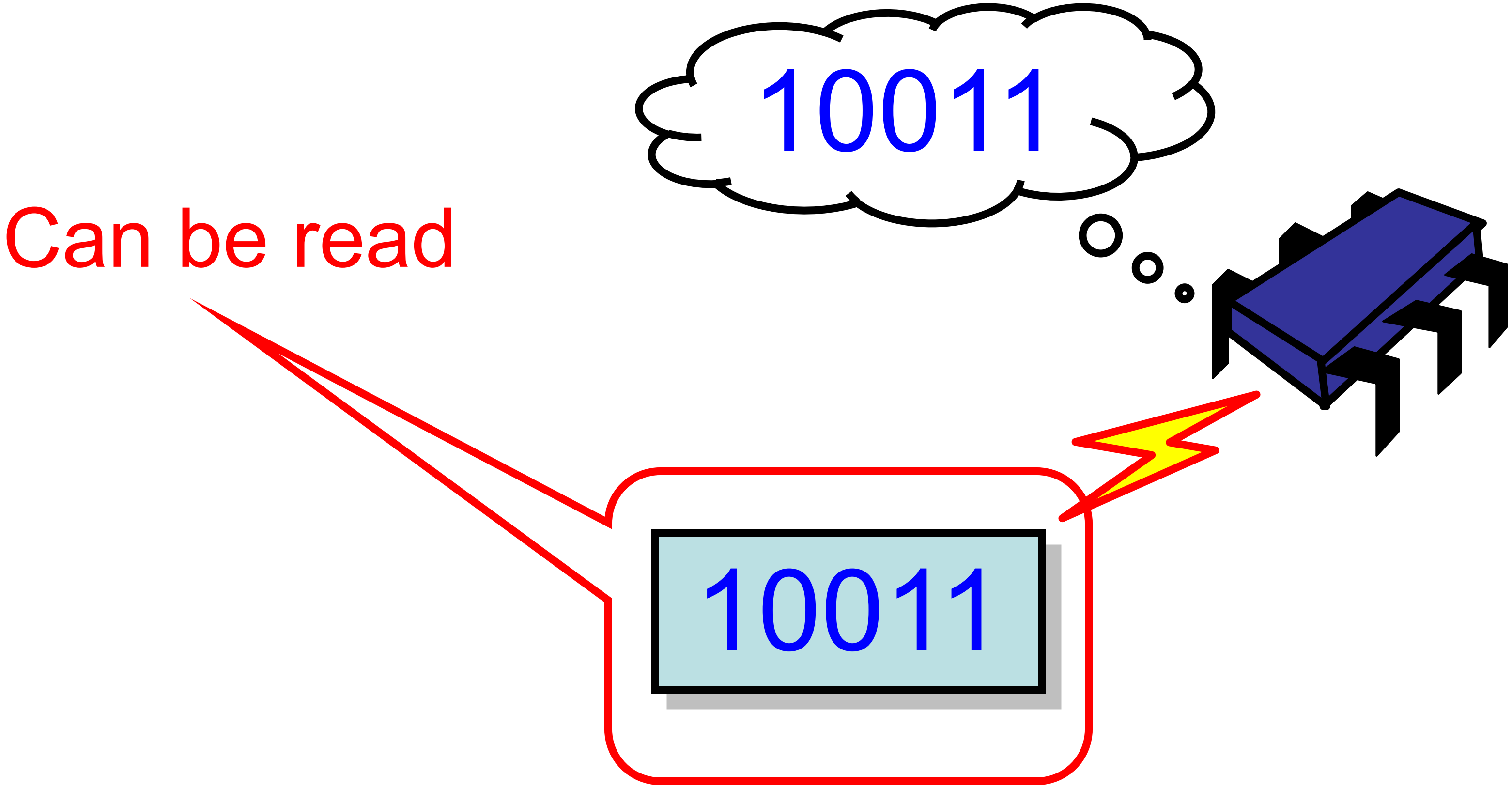


Register*



*** A memory location: name is historical**

Register

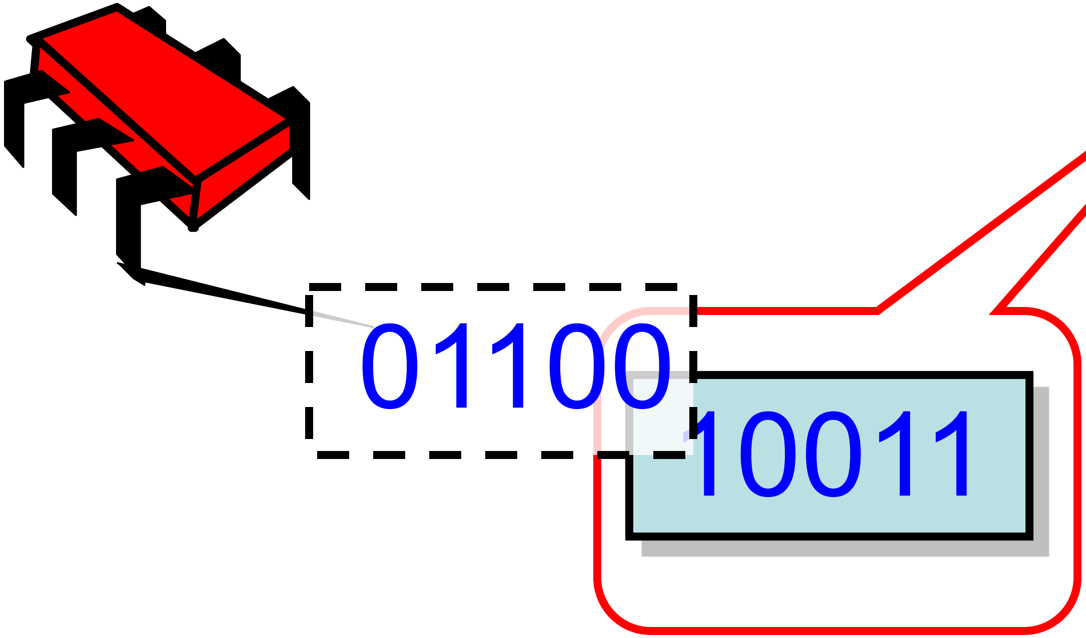


10011

Can be read

10011

Register



Can be written

Registers

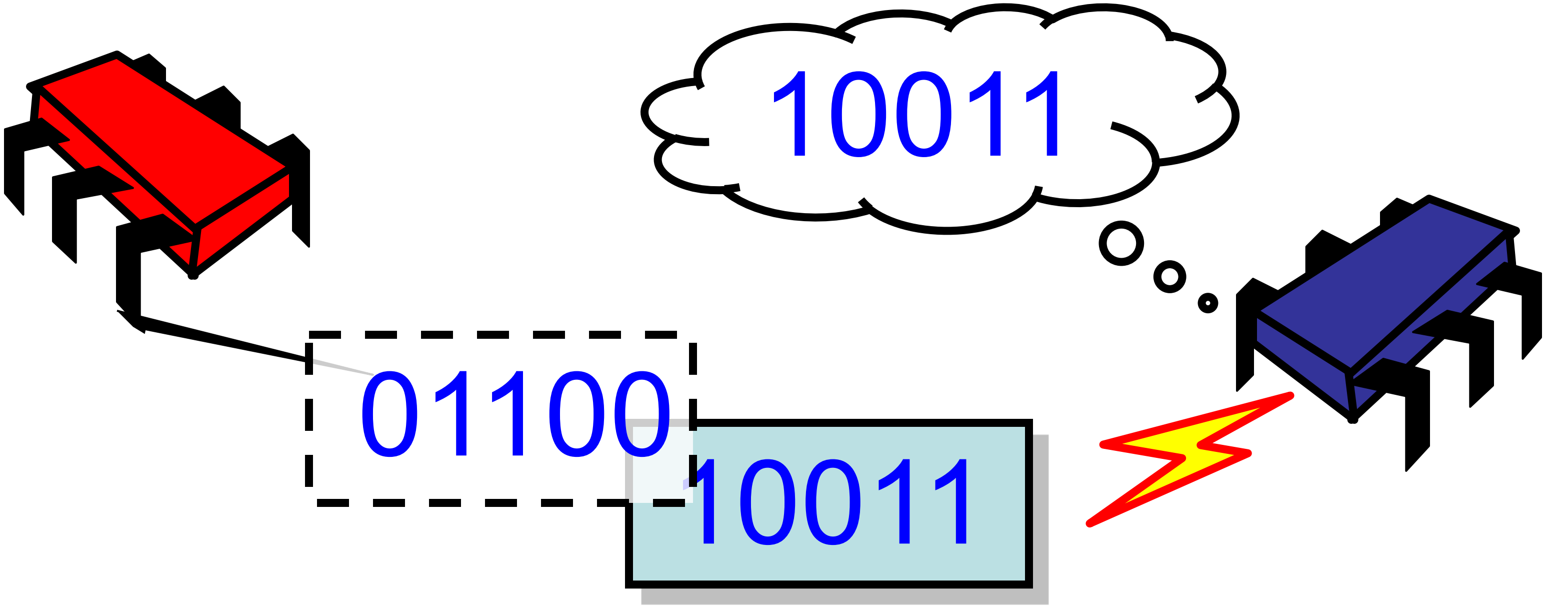
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

Registers

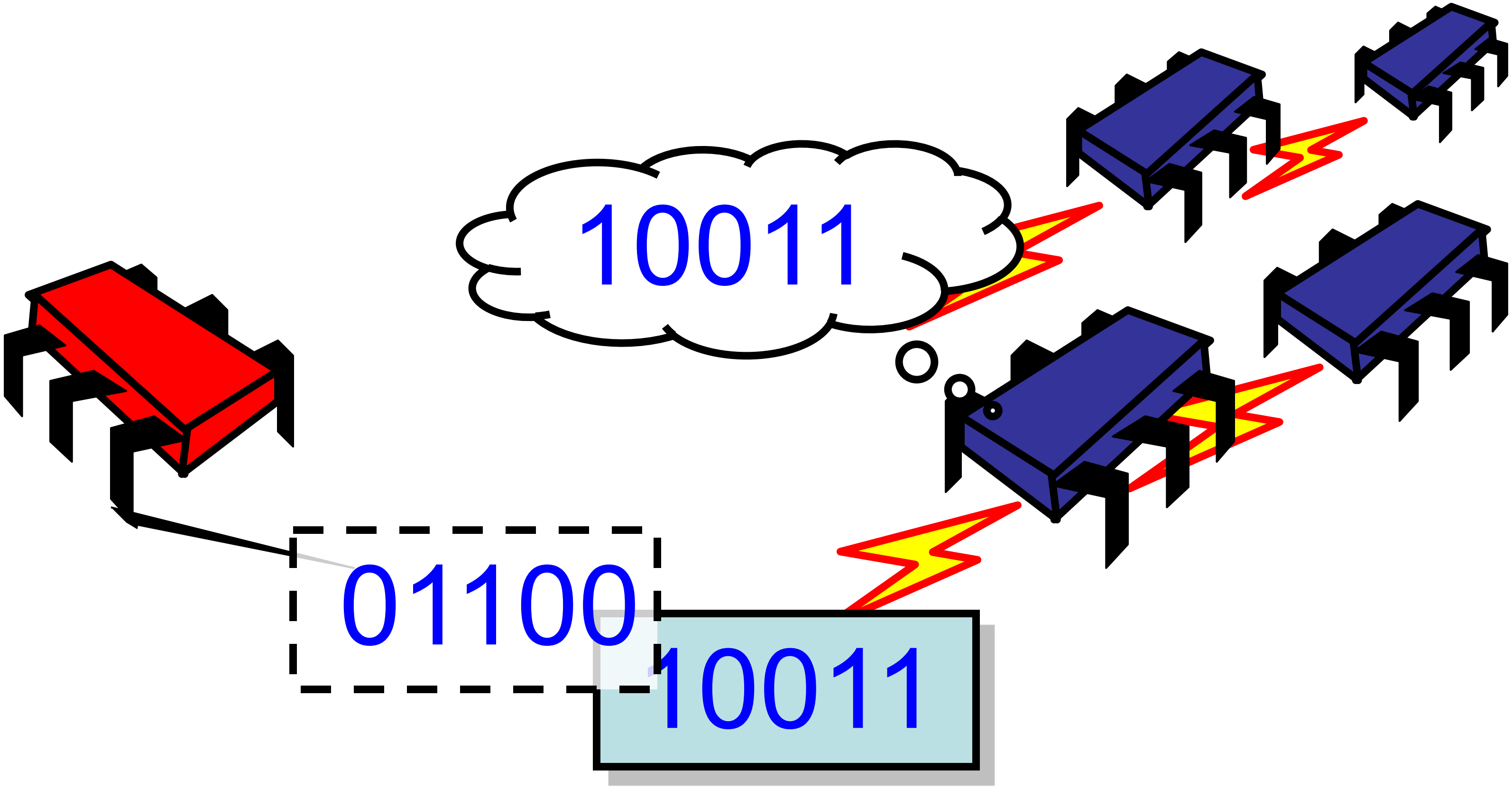
```
public interface Register<T> {
  public T read();
  public void write(T v);
}
```

Type of register
(usually Boolean or *m*-bit Integer)

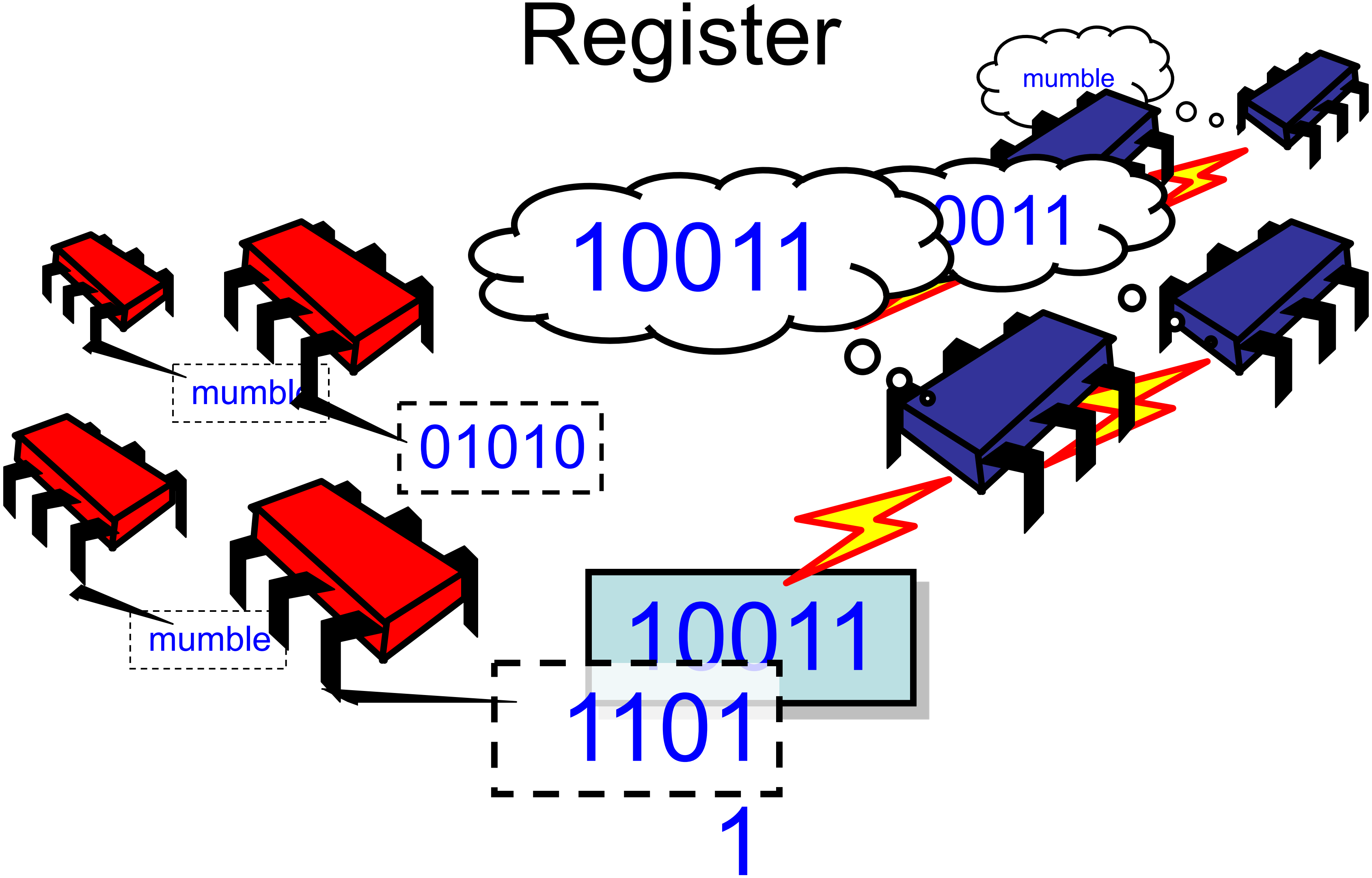
Single-Reader/Single-Writer Register



Multi-Reader/Single-Writer Register



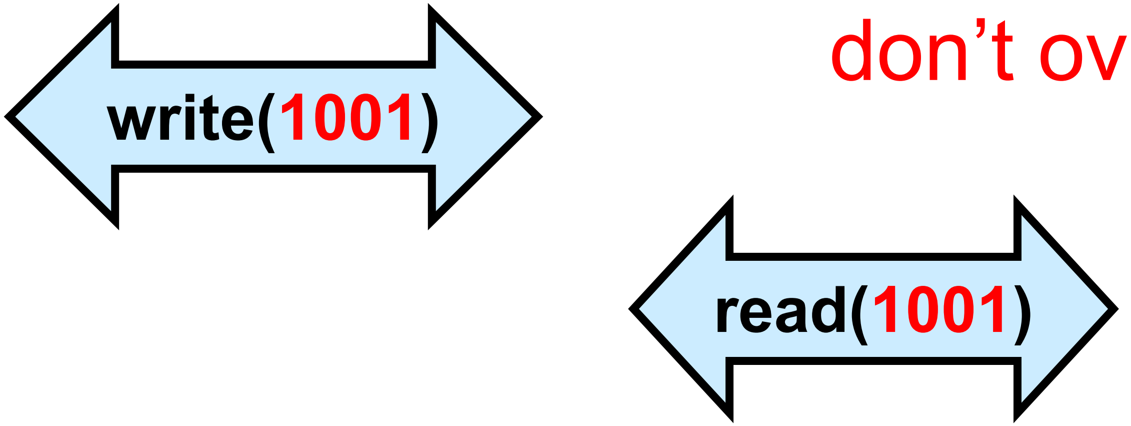
Multi-Reader/Multi-Writer Register



Jargon Watch

- SRSW
  - Single-reader single-writer
- MRSW
  - Multi-reader single-writer
- MRMW
  - Multi-reader multi-writer

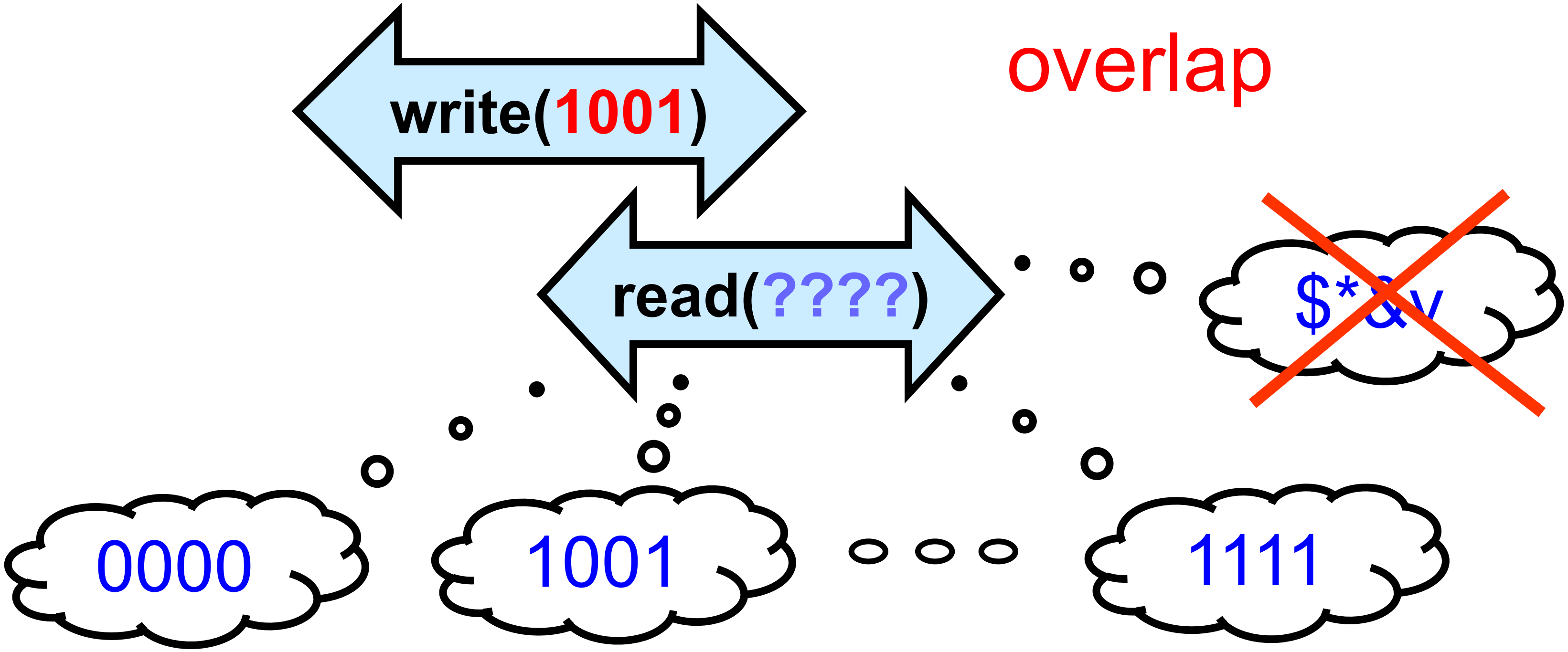
Safe Register



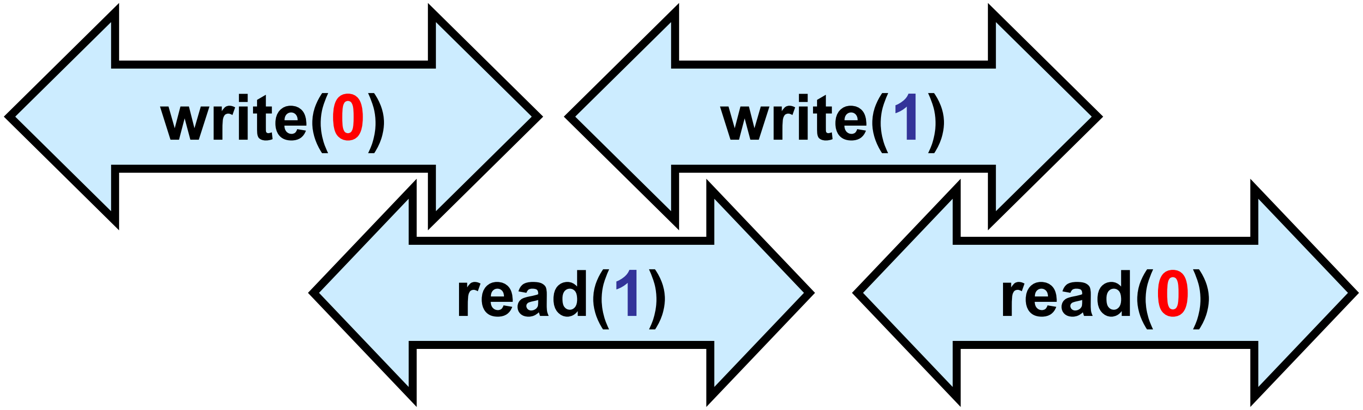
OK if reads and writes don’t overlap

Safe Register

Some valid value if reads and writes do overlap

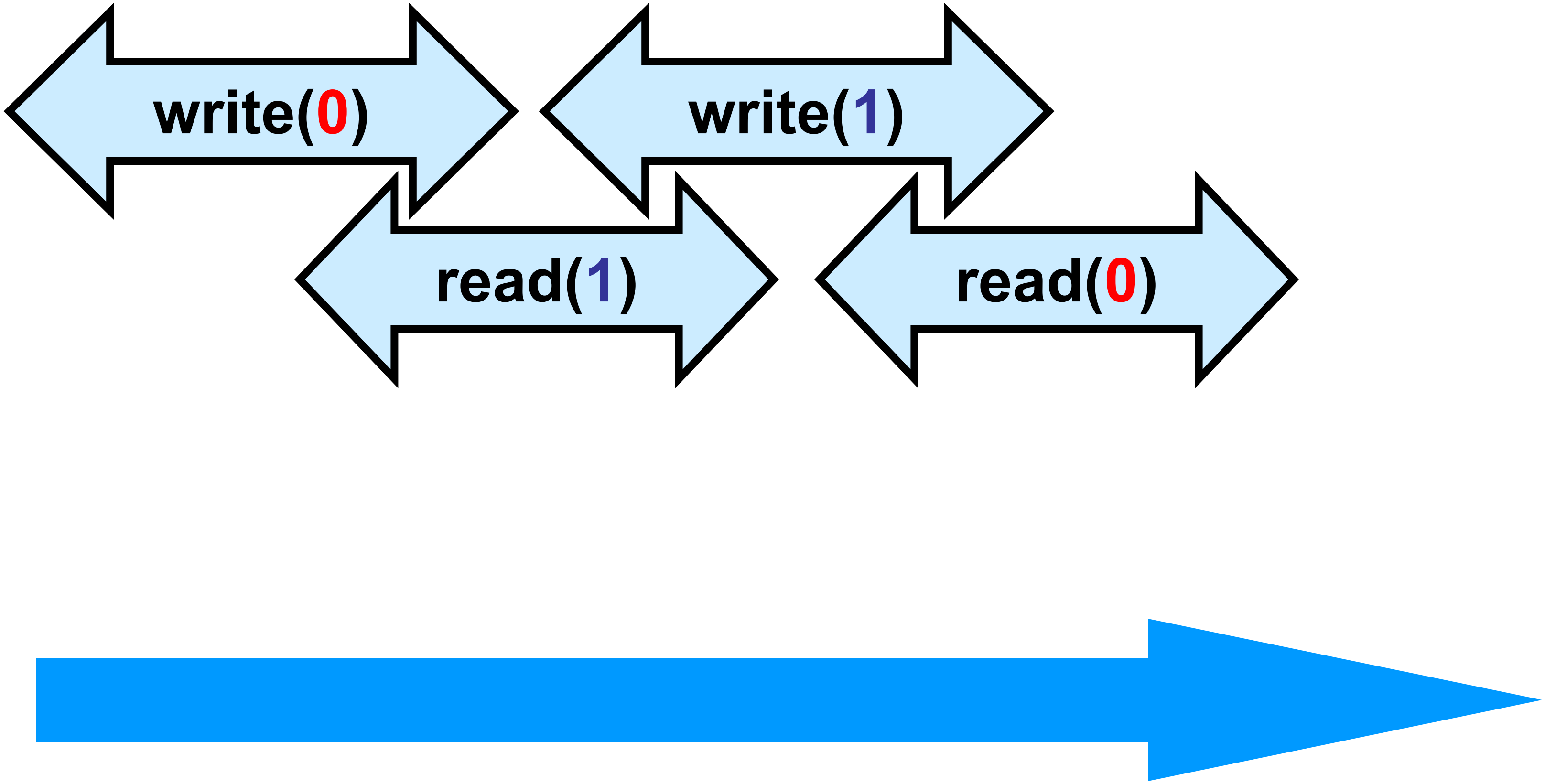


Regular Register

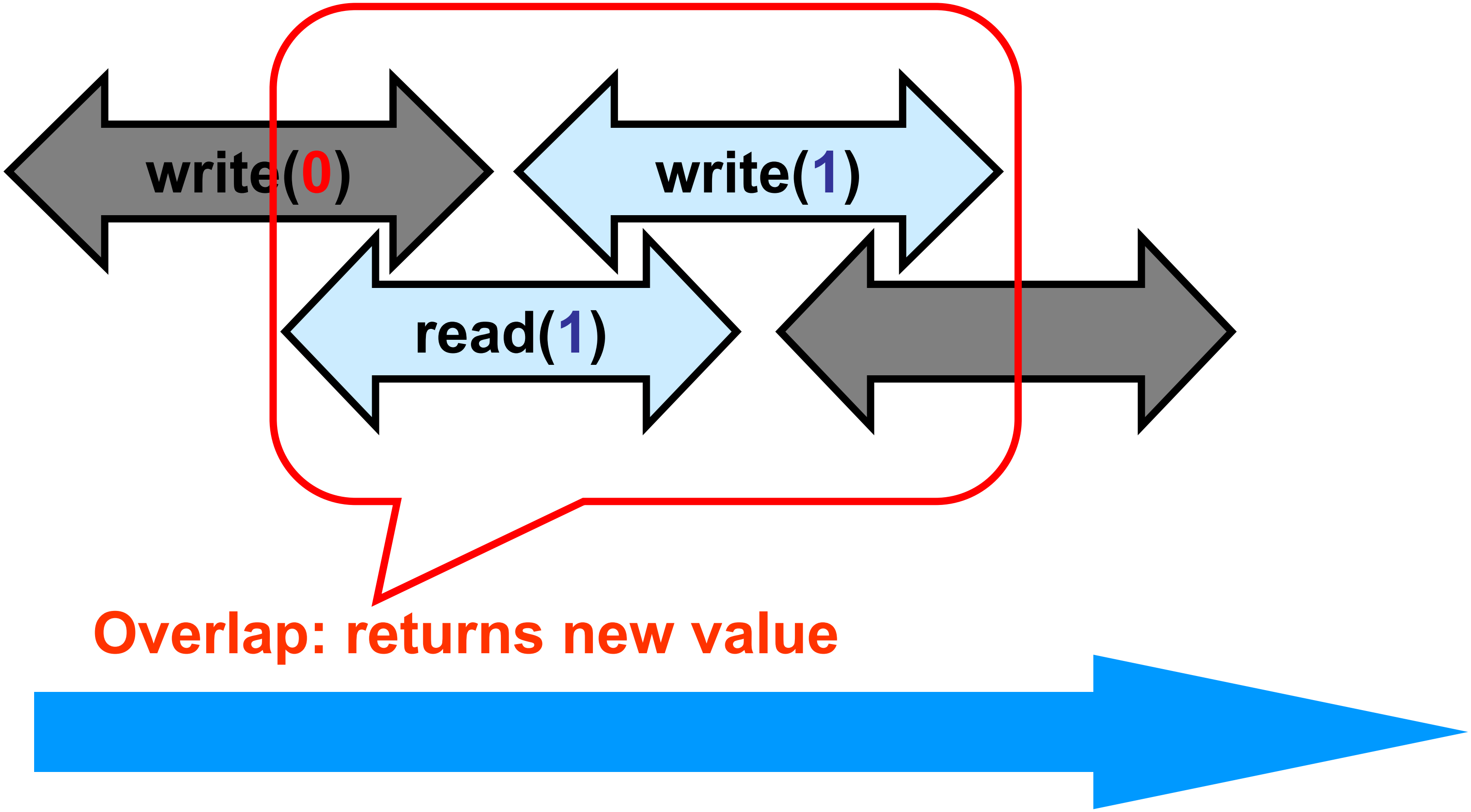


- Single Writer
- Readers return:
  - Old value if no overlap (safe)
  - Old or one of new values if overlap

Regular or Not?

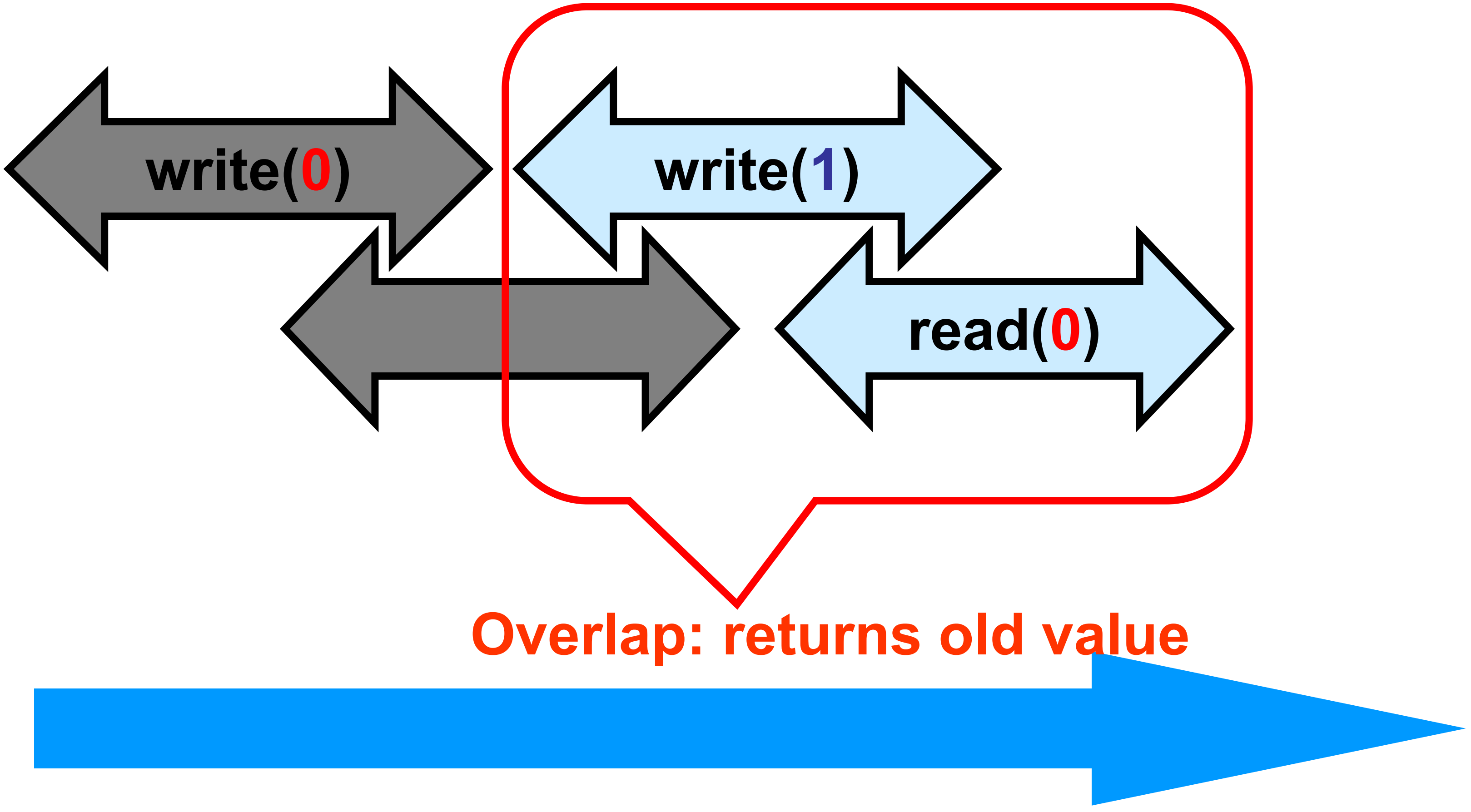


Regular or Not?

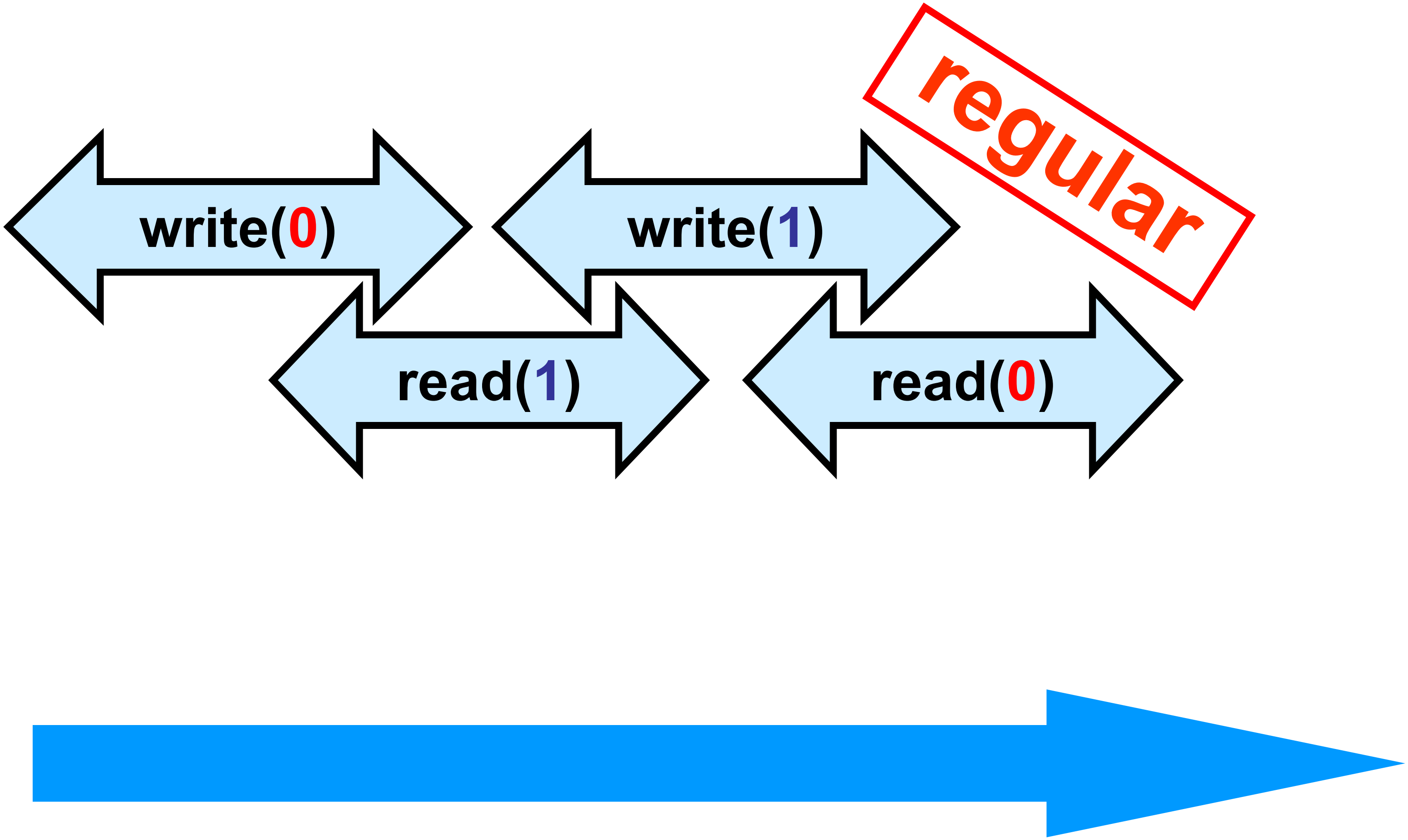


Overlap: returns new value

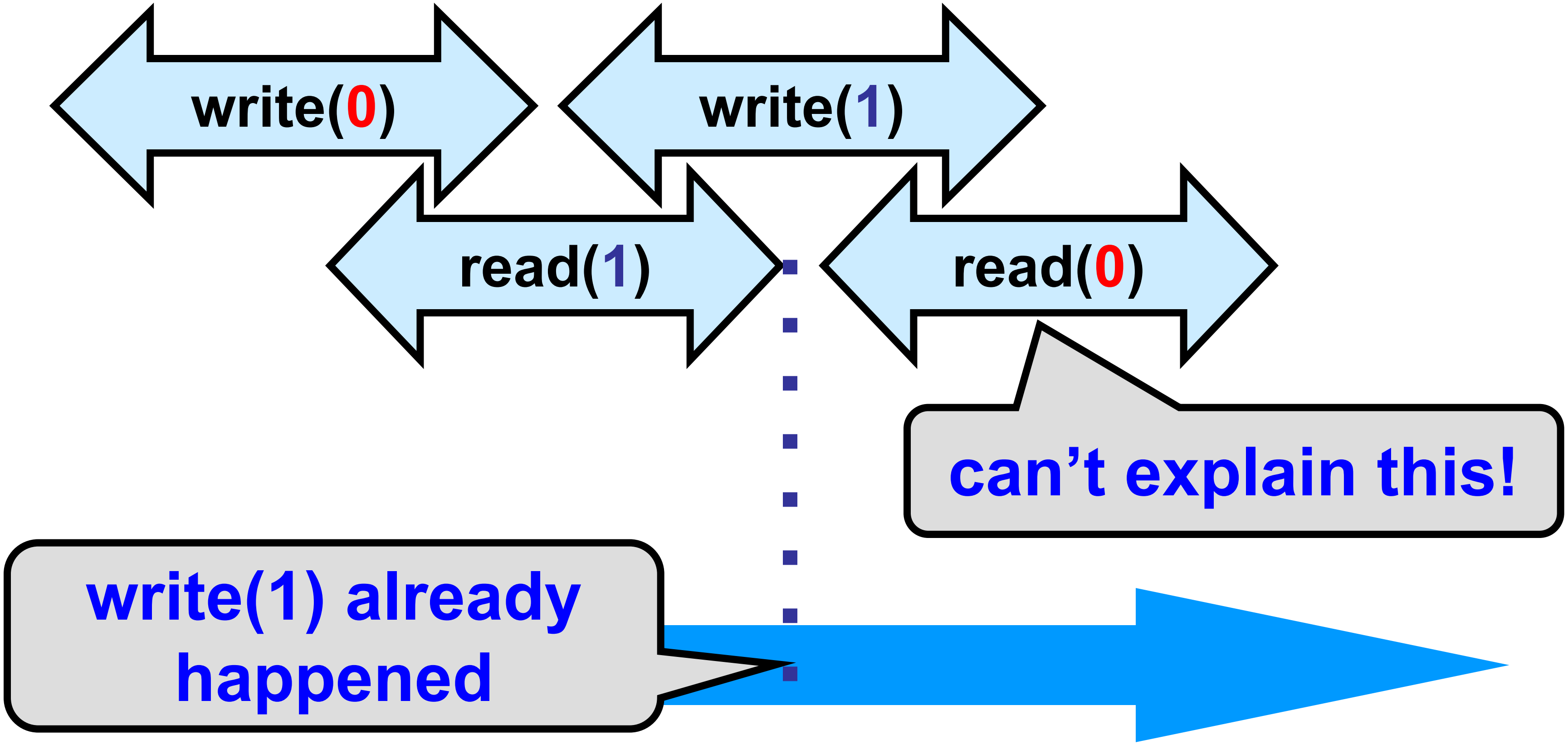
Regular or Not?



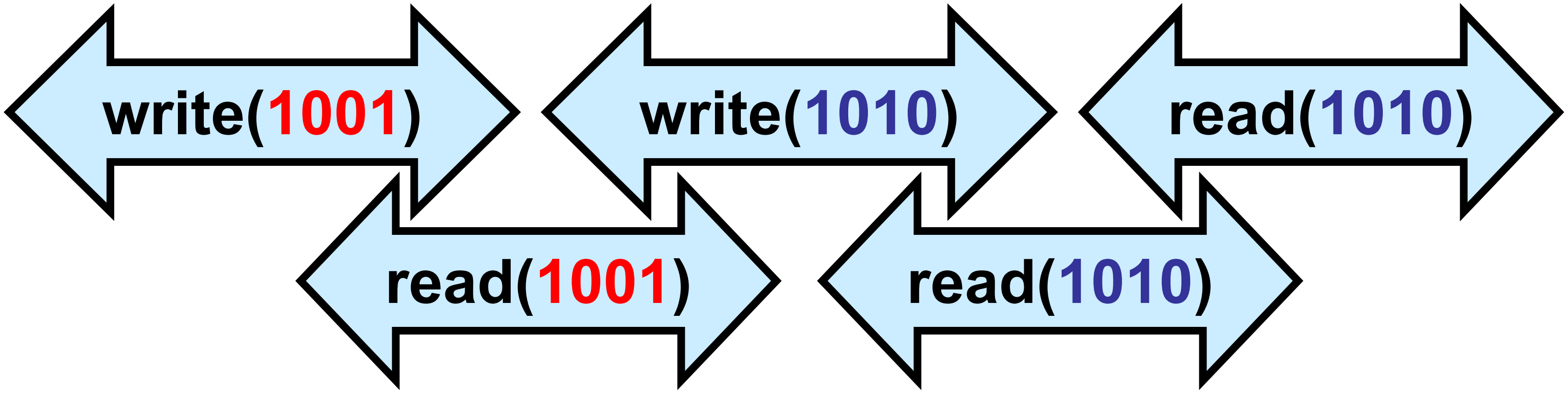
Regular or Not?



Regular ≠ Linearizable

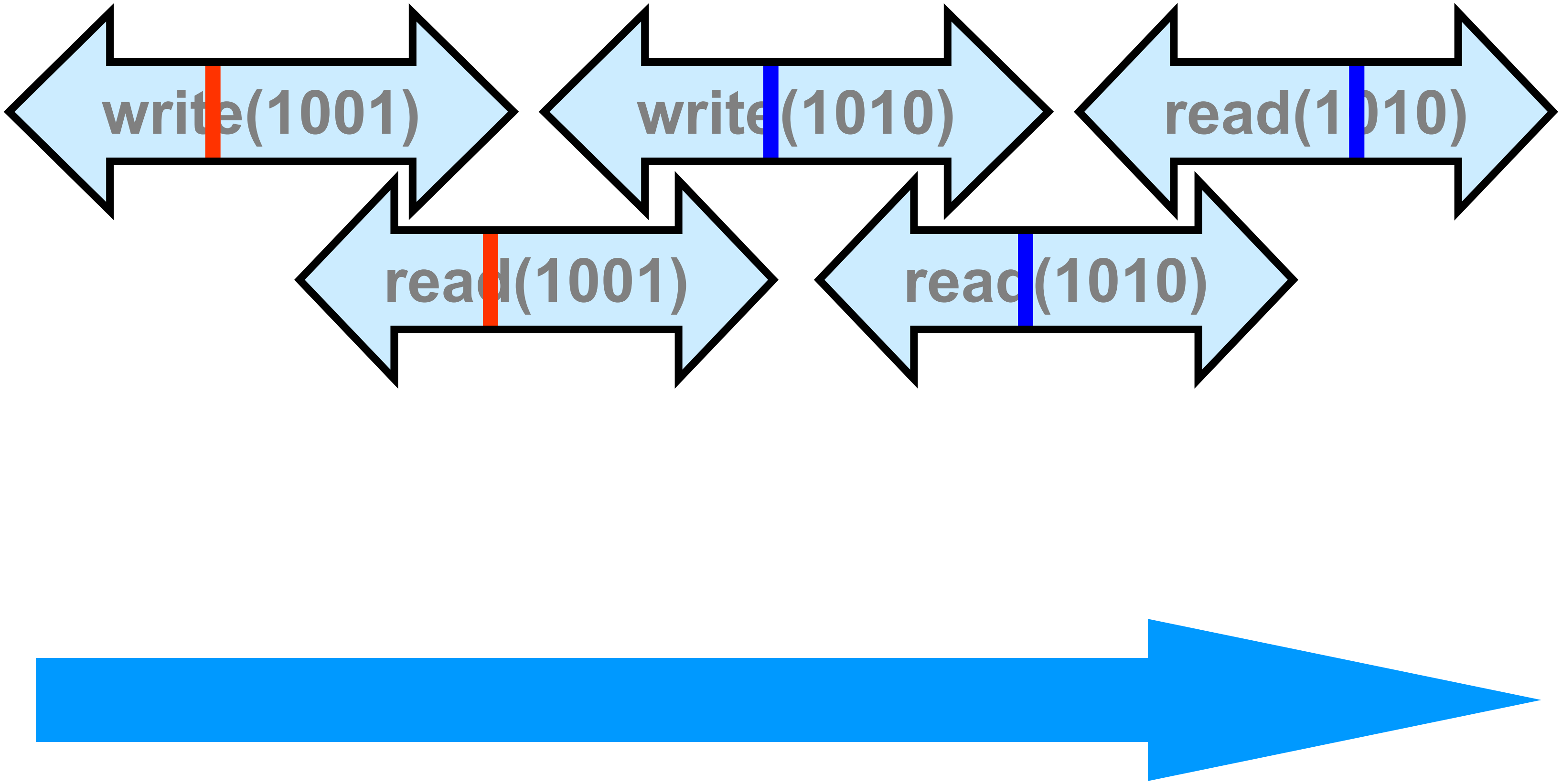


Atomic Register

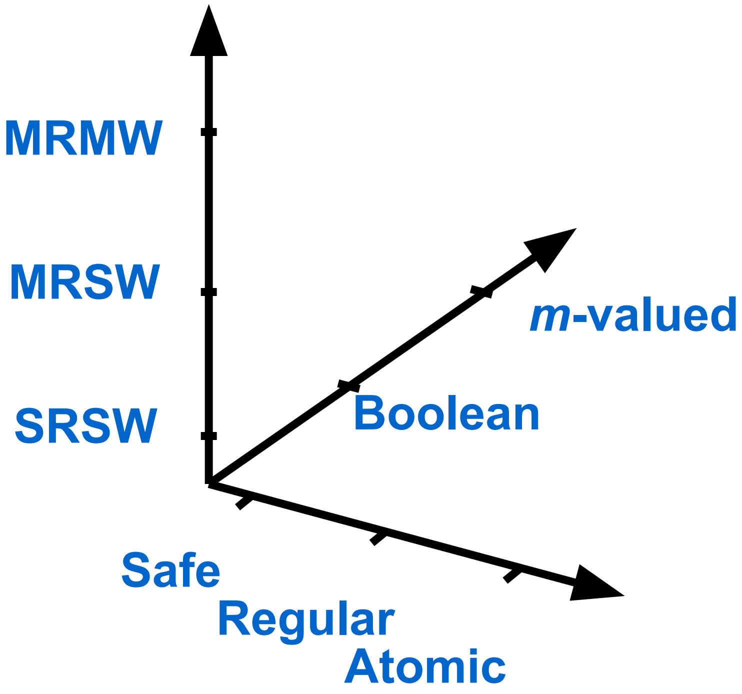


Linearizable to sequential safe register

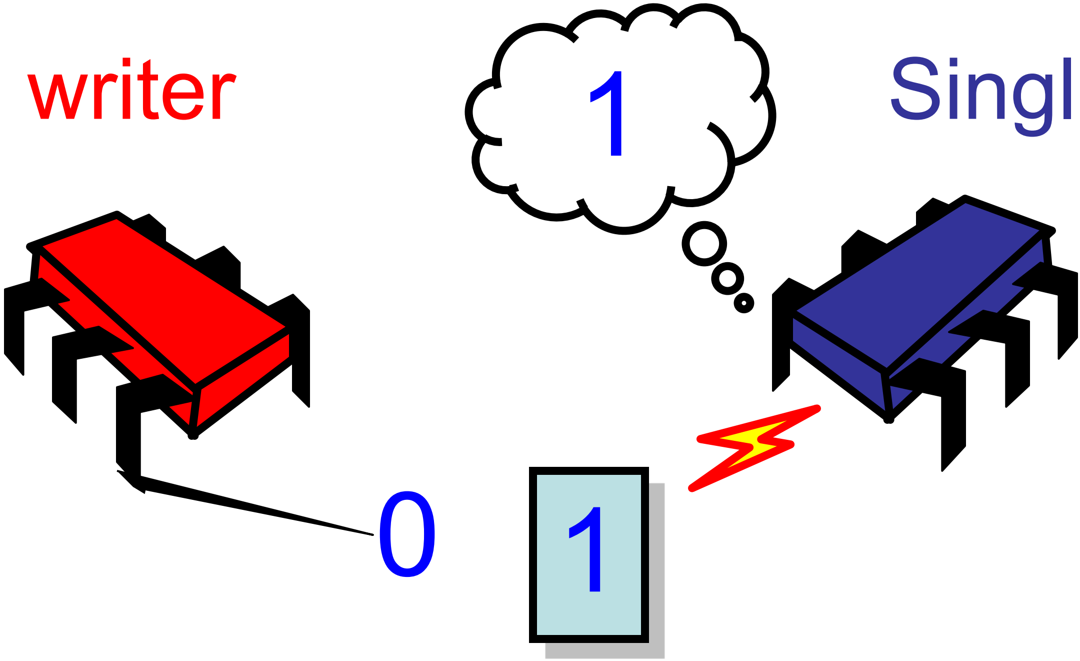
Atomic Register



Register Space



Weakest Register



Single writer

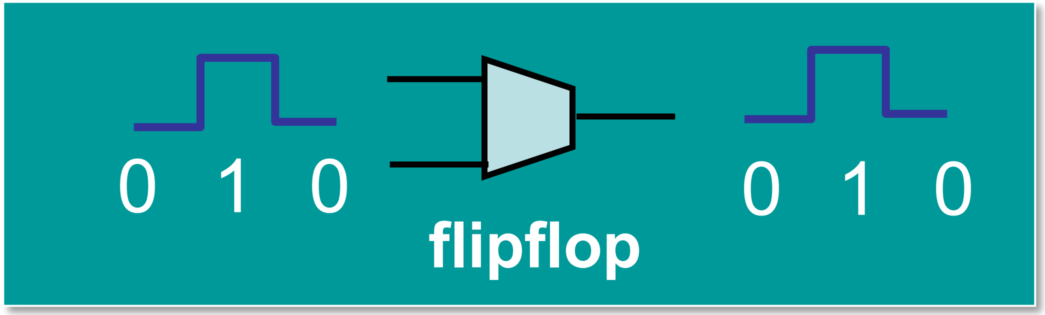
Single reader

Safe Boolean register

Weakest Register

Single writer

Single reader



Get correct reading if not during state transition

Results

- From SRSW safe Boolean register
  - All the other registers
  - Mutual exclusion

Foundations of the field

- But not everything!
  - Consensus hierarchy

The really cool stuff …

Locking within Registers

- Not interesting to rely on mutual exclusion in register constructions
- We want registers to implement mutual exclusion!
- It’s cheating to use mutual exclusion to implement itself!

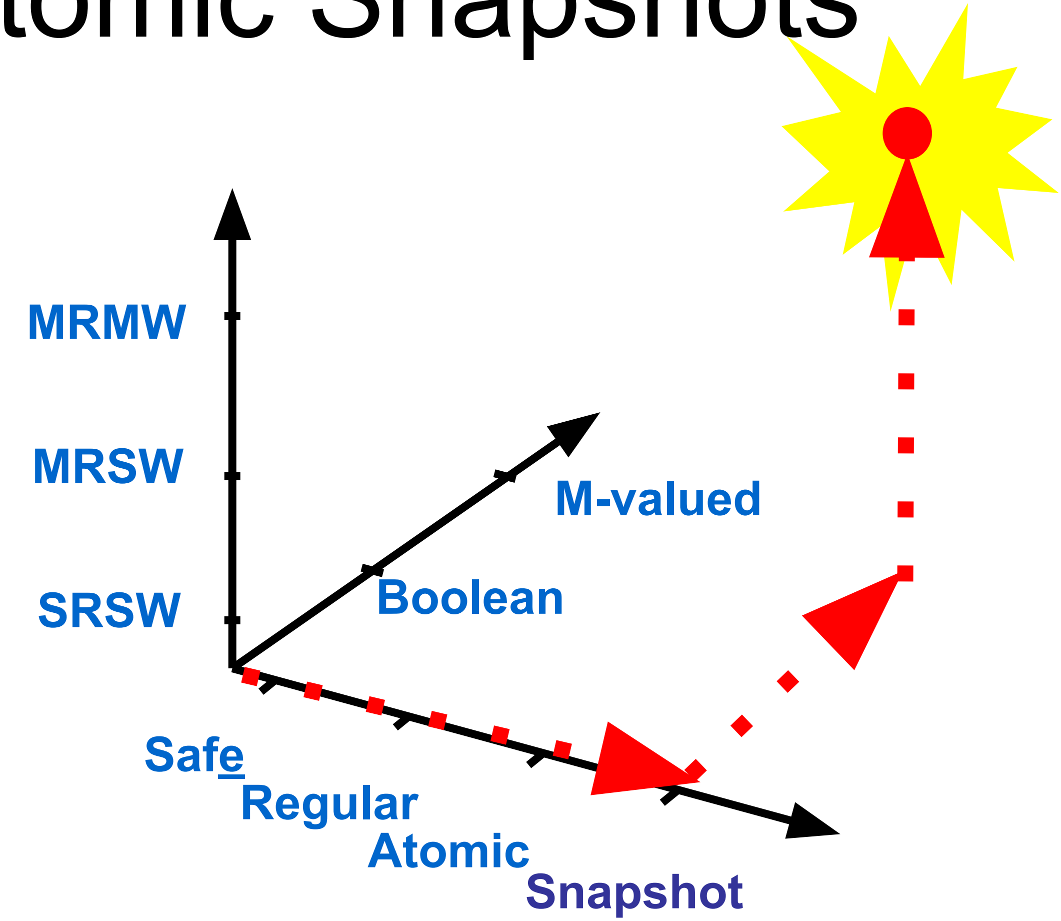
Definition

An object implementation is ***wait-free*** if every method call completes in a finite number of steps

No mutual exclusion

- Thread could halt in critical section
- Build mutual exclusion from registers

From Safe SRSW Boolean to Atomic Snapshots



Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Register Names

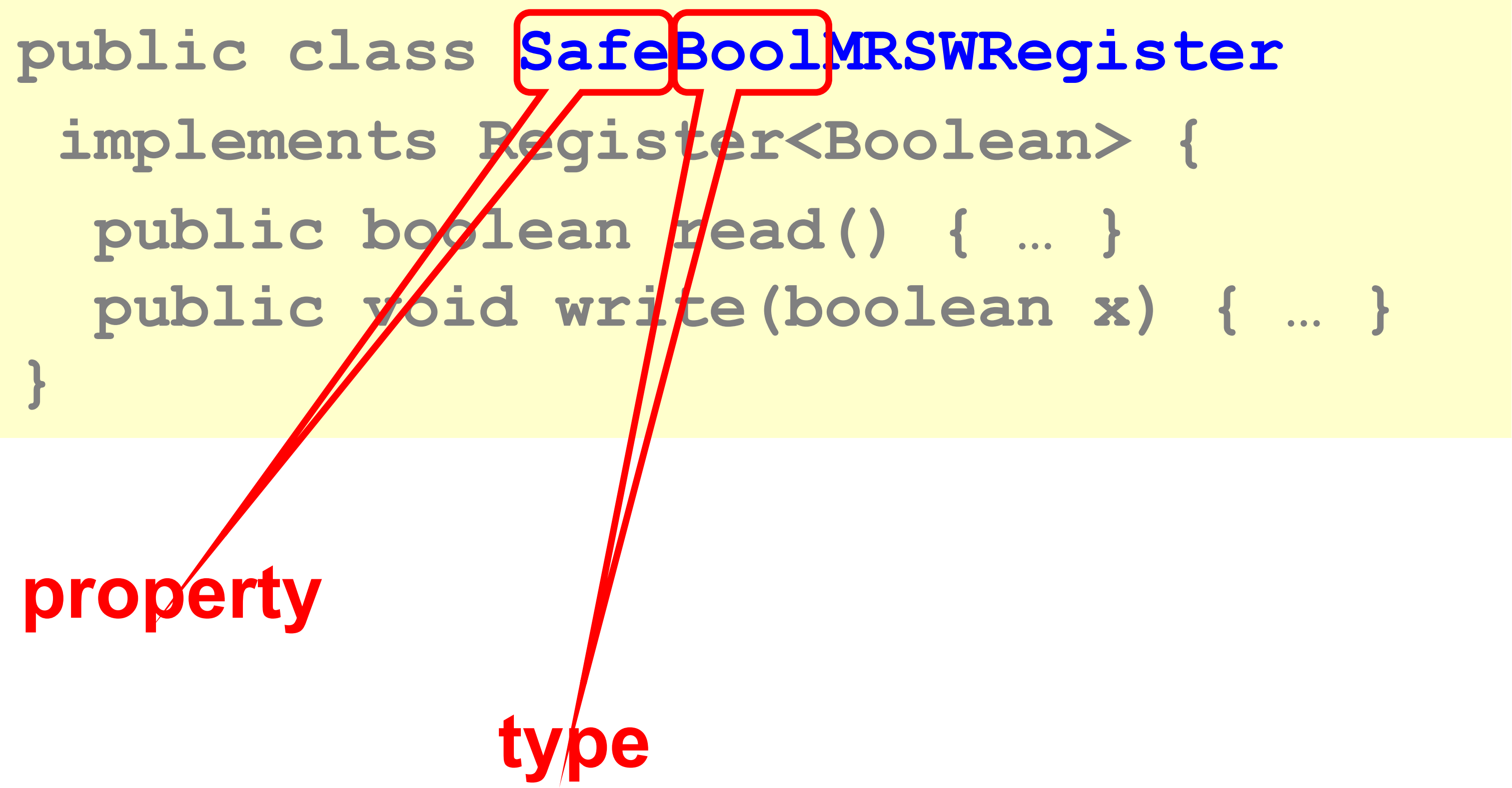
```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

Register Names

```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

property

Register Names



```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

property

type

Register Names

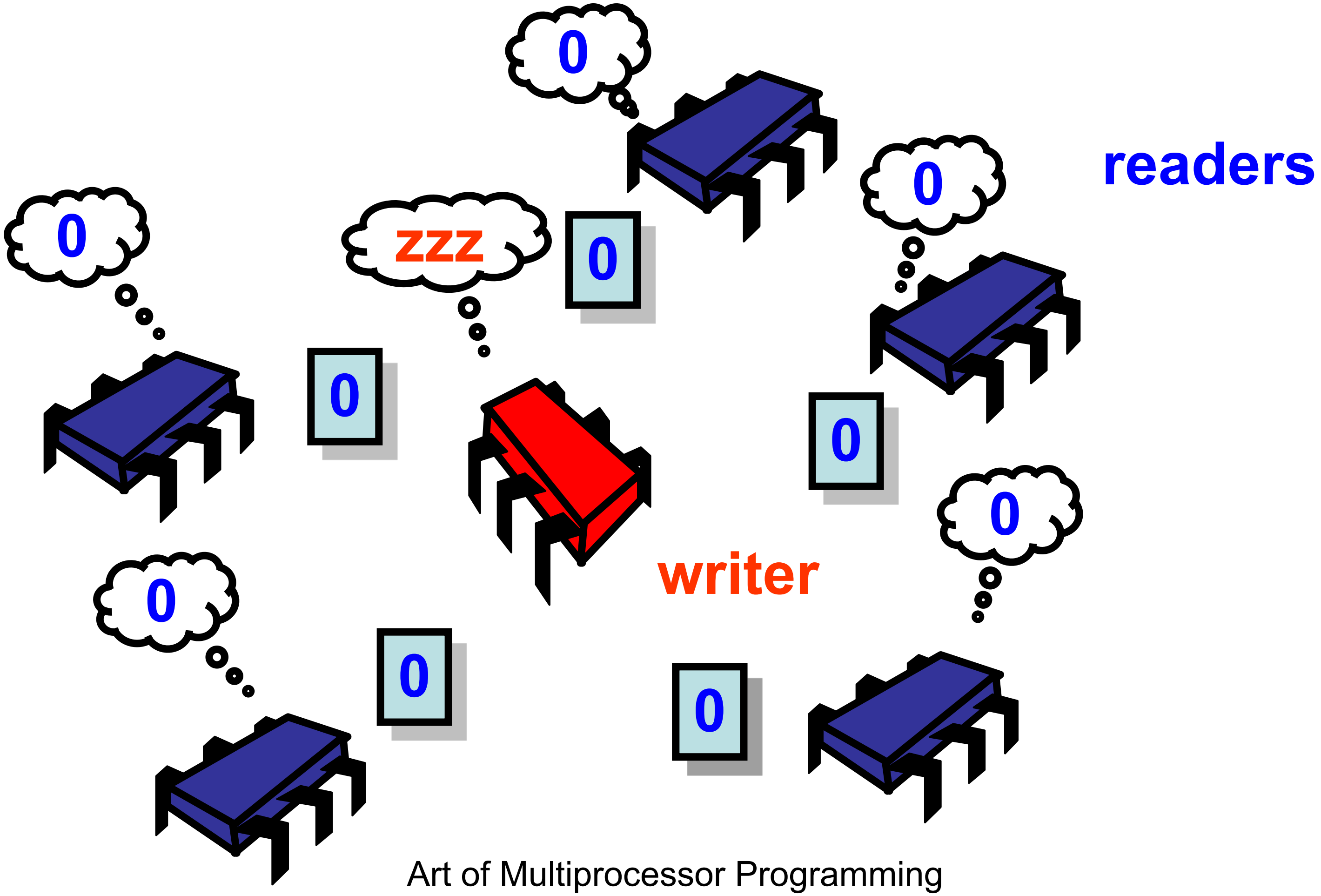
```
public class SafeBoolMRSWRegister
 implements Register<Boolean> {
  public boolean read() { … }
  public void write(boolean x) { … }
}
```

property

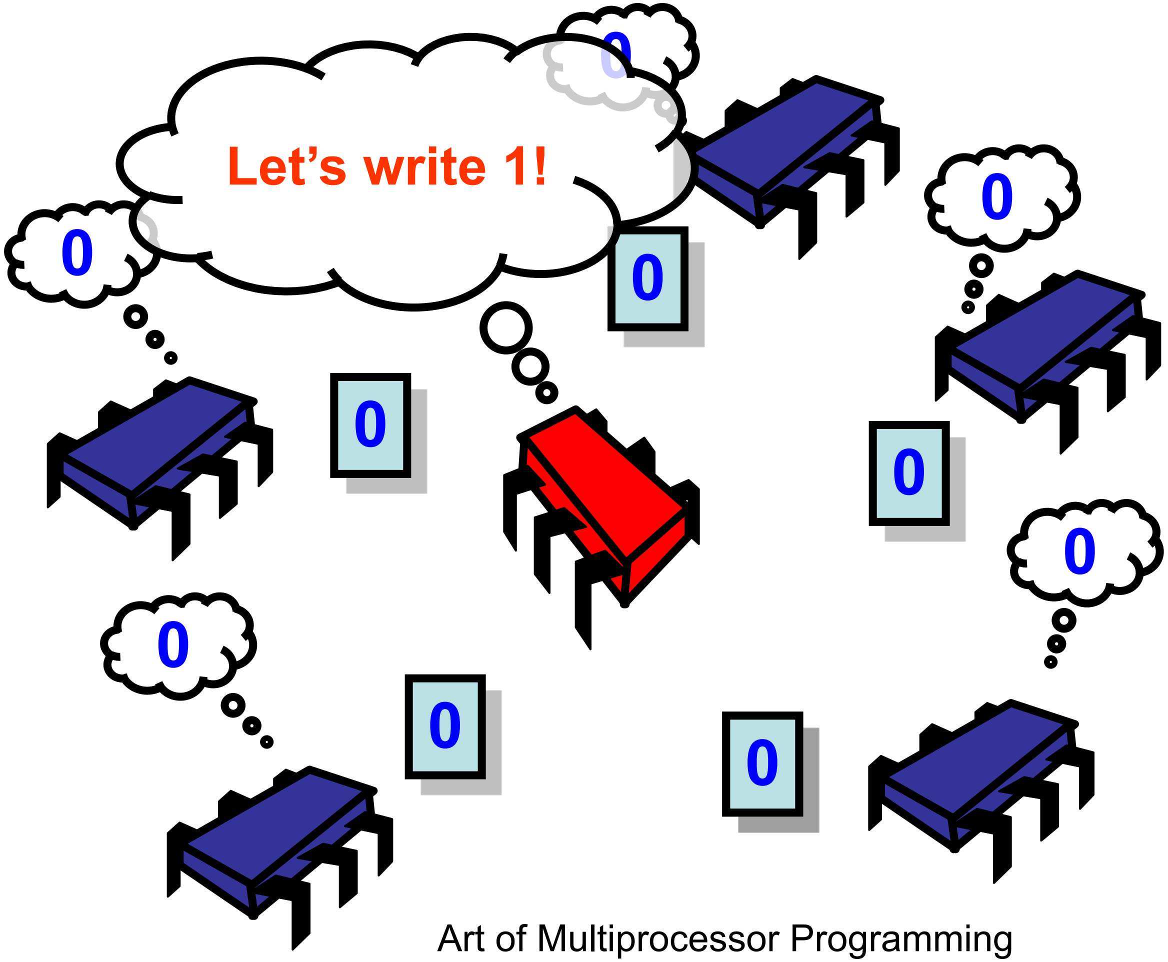
type

how many readers & writers?

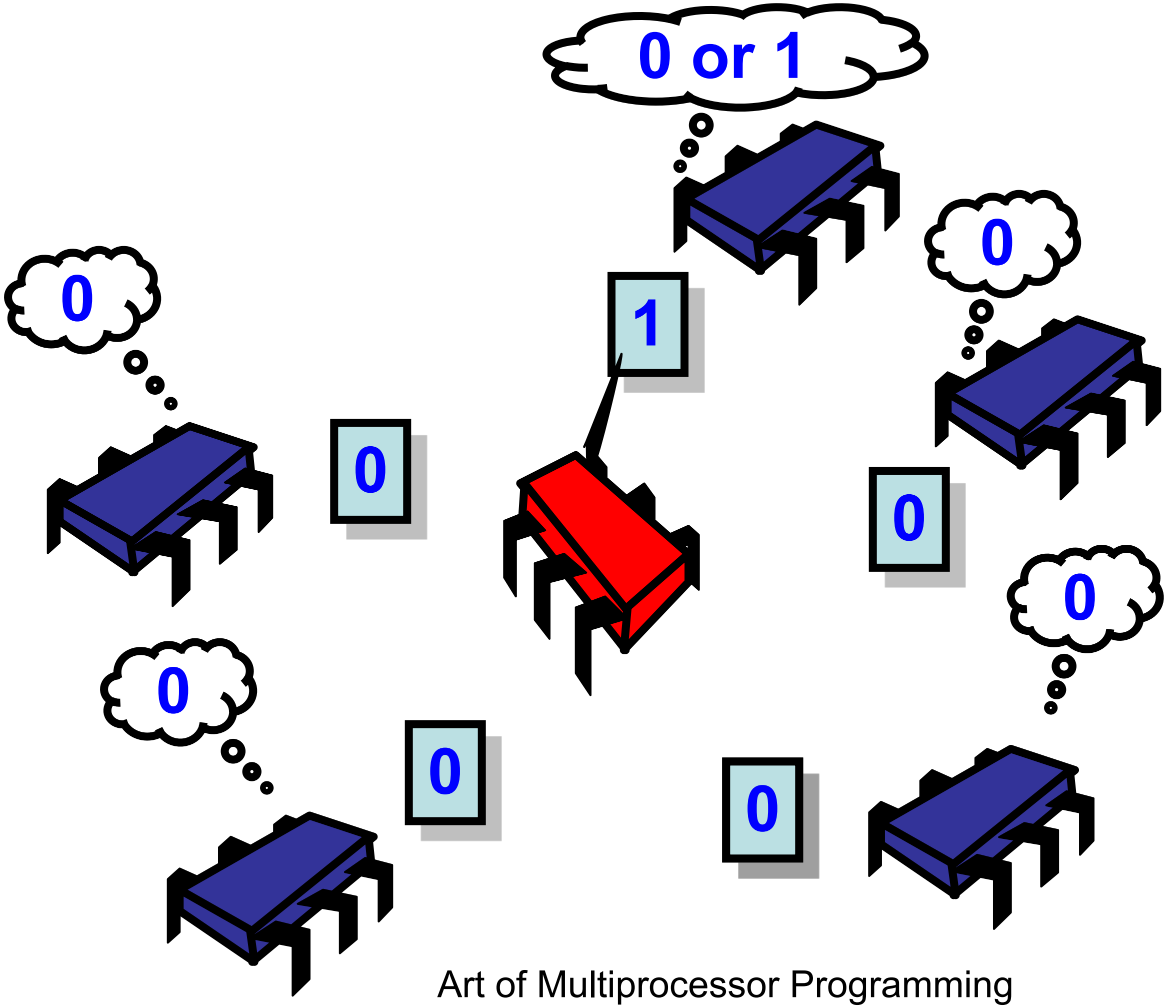
Safe Boolean MRSW from Safe Boolean SRSW



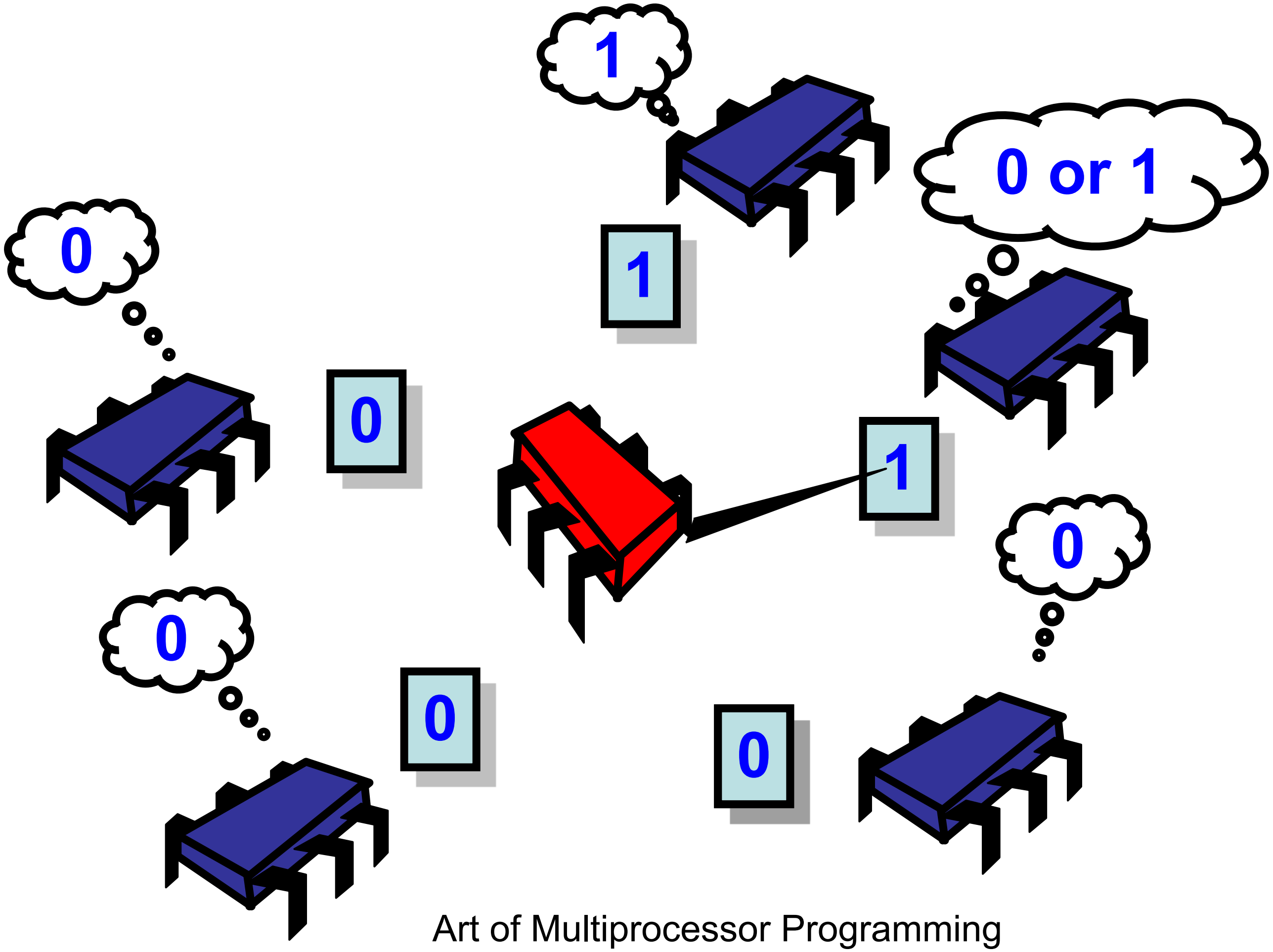
Safe Boolean MRSW from Safe Boolean SRSW



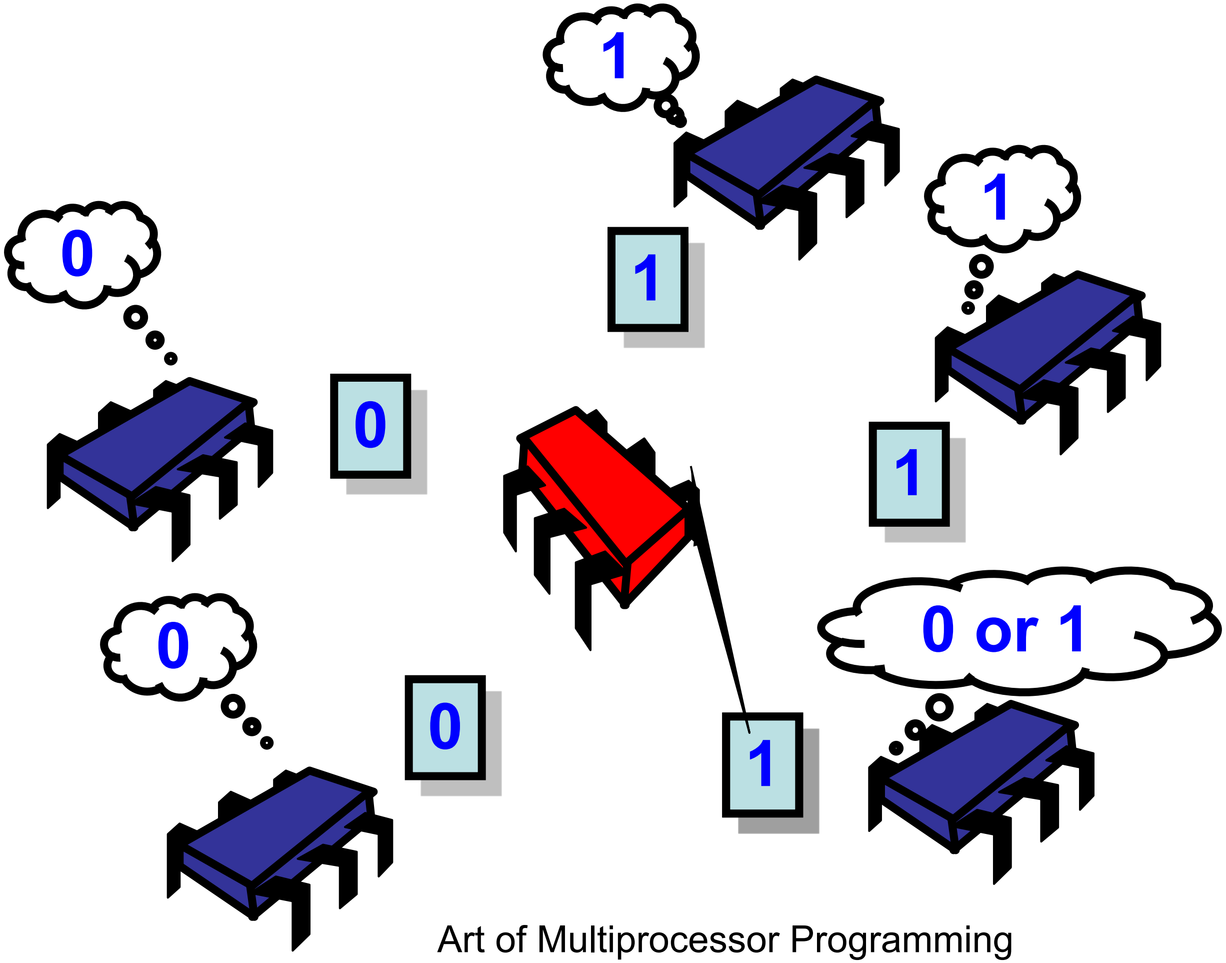
Safe Boolean MRSW from Safe Boolean SRSW



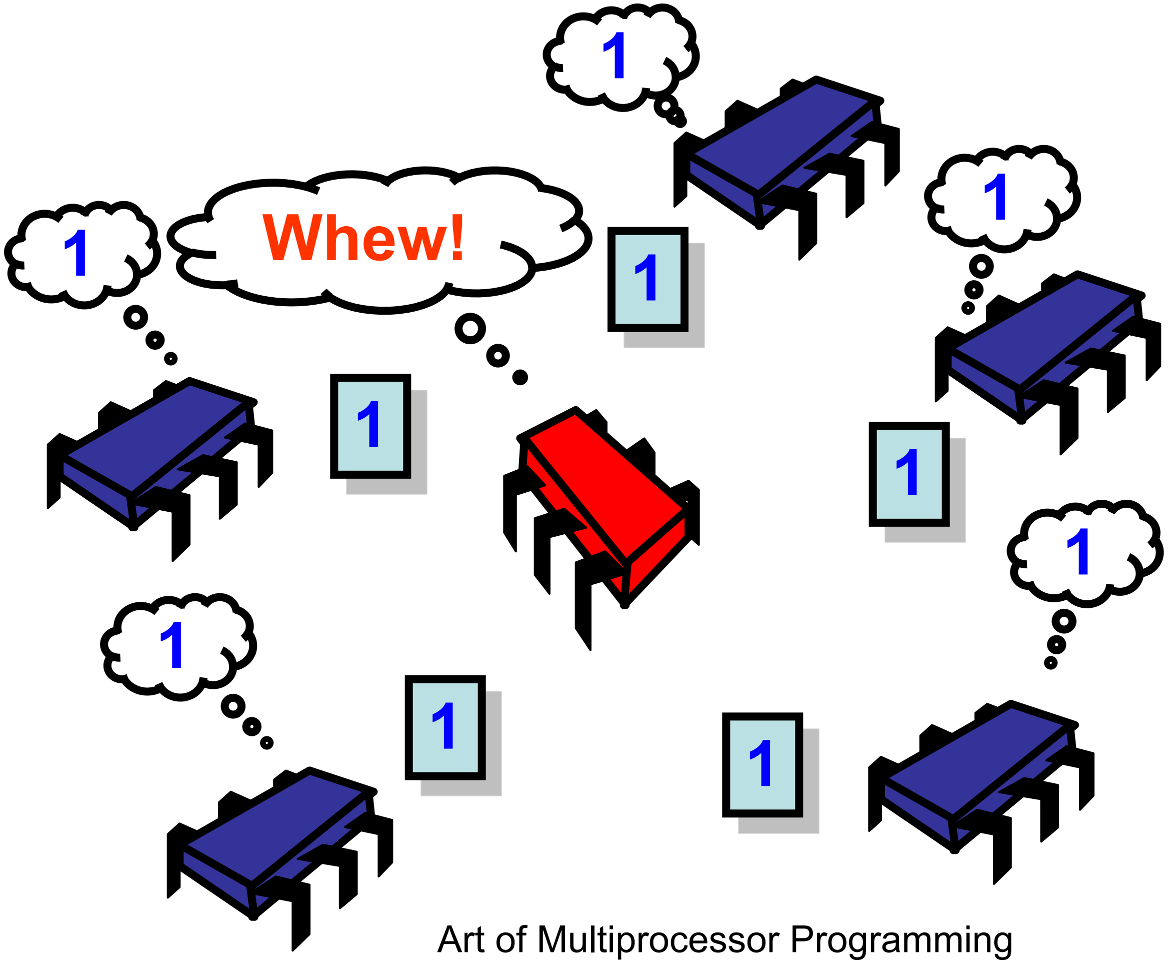
Safe Boolean MRSW from Safe Boolean SRSW



Safe Boolean MRSW from Safe Boolean SRSW



Safe Boolean MRSW from Safe Boolean SRSW



Safe Boolean MRSW from Safe Boolean SRSW

```
public 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();
  }}
```

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBoolMRSWRegister
 implements BooleanRegister {
 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();
  }}
```

Each thread has own safe SRSW register

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBoolMRSWRegister
 implements BooleanRegister {
 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();
  }}
```

write method

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBoolMRSWRegister
 implements BooleanRegister {
 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();
  }}
```

Write each thread's register one at a time

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBoolMRSWRegister
 implements BooleanRegister {
 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();
  }}
```

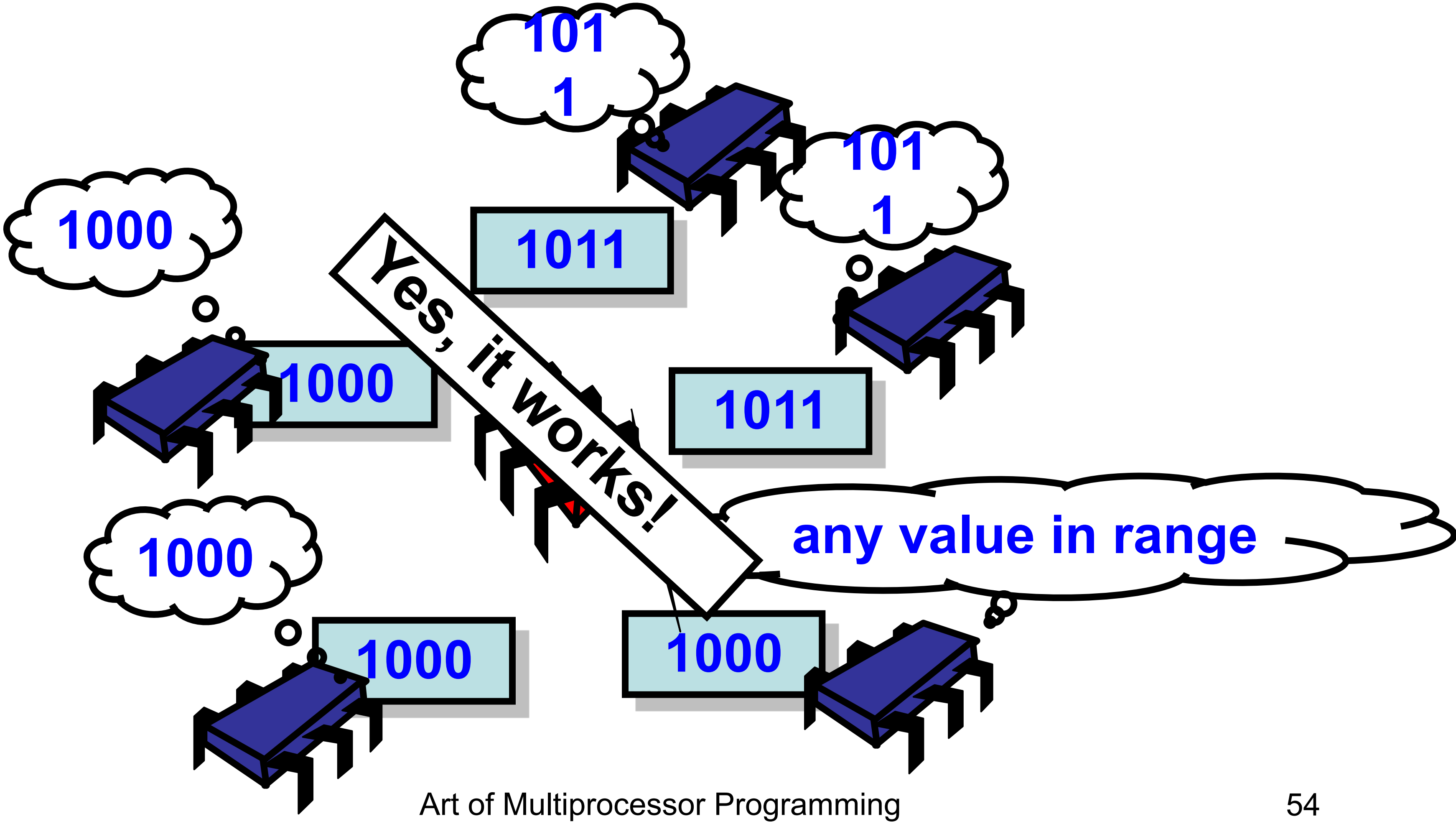
read method

Safe Boolean MRSW from Safe Boolean SRSW

```
public class SafeBoolMRSWRegister
 implements BooleanRegister {
 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();
  }}
```

Read my own register

Safe Multi-Valued MRSW from Safe Multi-Valued SRSW?



Road Map

- SRSW safe Boolean
- MRSW safe Boolean

- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

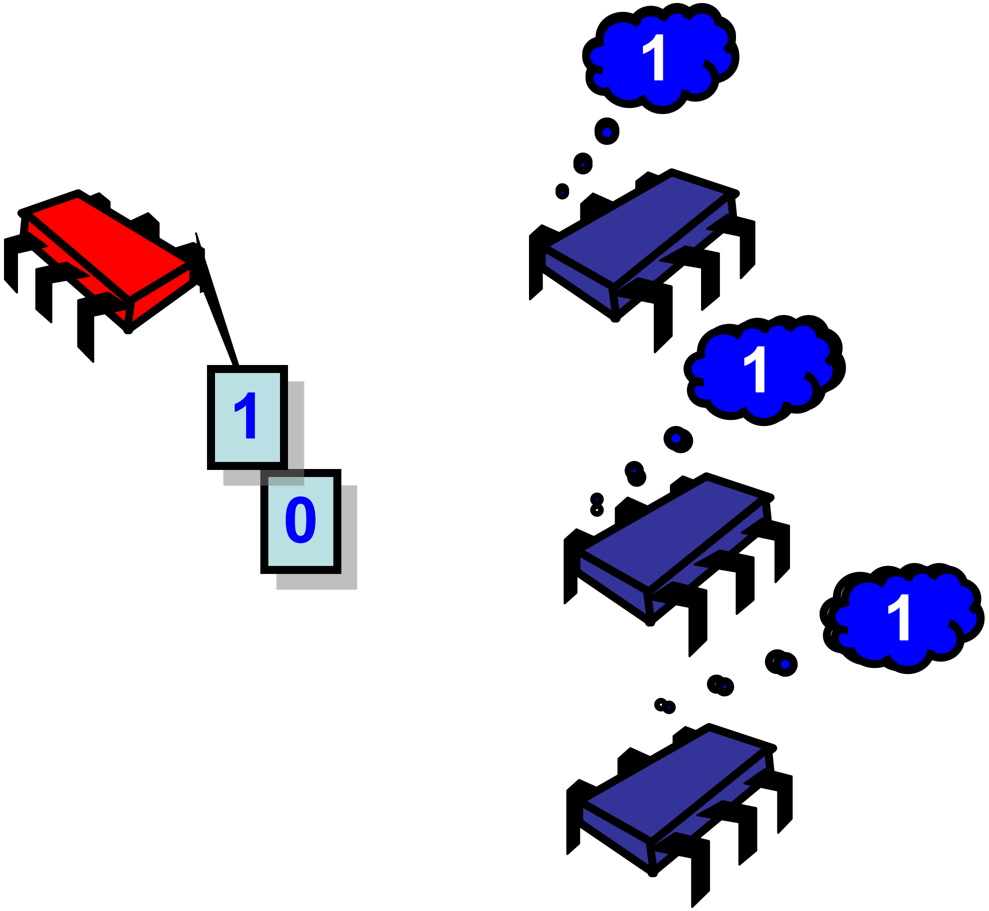
Questions?

Road Map

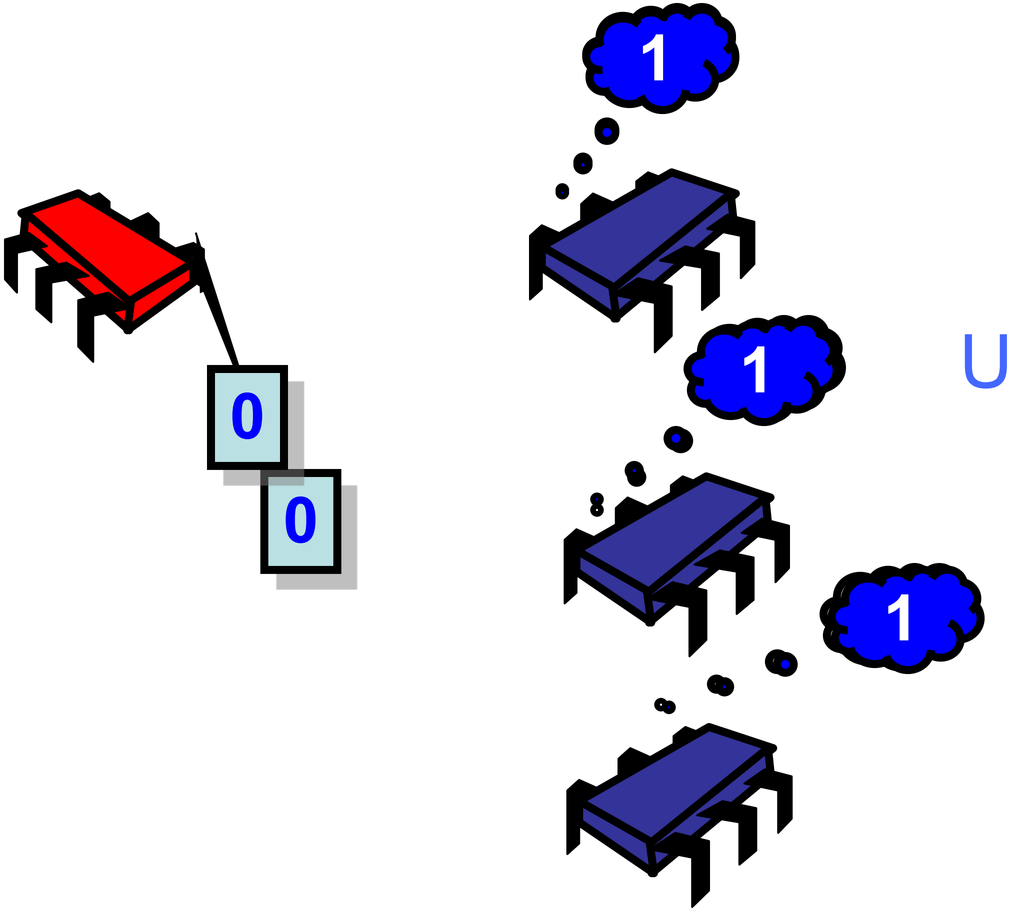
- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Regular Boolean MRSW from Safe Boolean MRSW

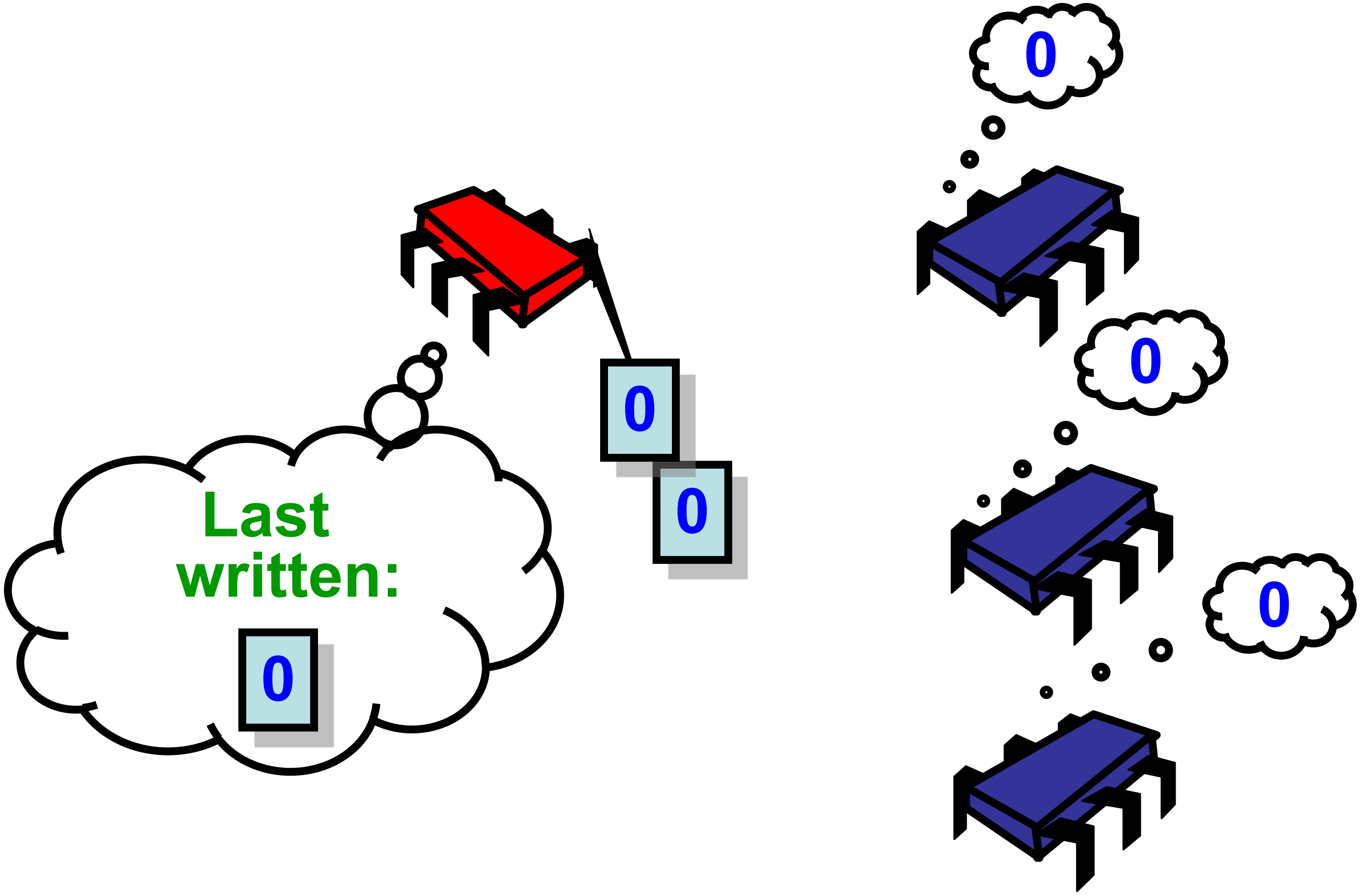


Regular Boolean MRSW from Safe Boolean MRSW



Uh, oh!

Regular Boolean MRSW from Safe Boolean MRSW



Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  private boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Last bit this thread wrote

(made-up syntax)

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Actual value

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

Is new value different from last value I wrote?

Regular Boolean MRSW from Safe Boolean MRSW

```
public class RegBoolMRSWRegister
 implements Register<Boolean> {
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
  }}
```

If so, change it (otherwise don't!)

Regular Boolean MRSW from Safe Boolean MRSW

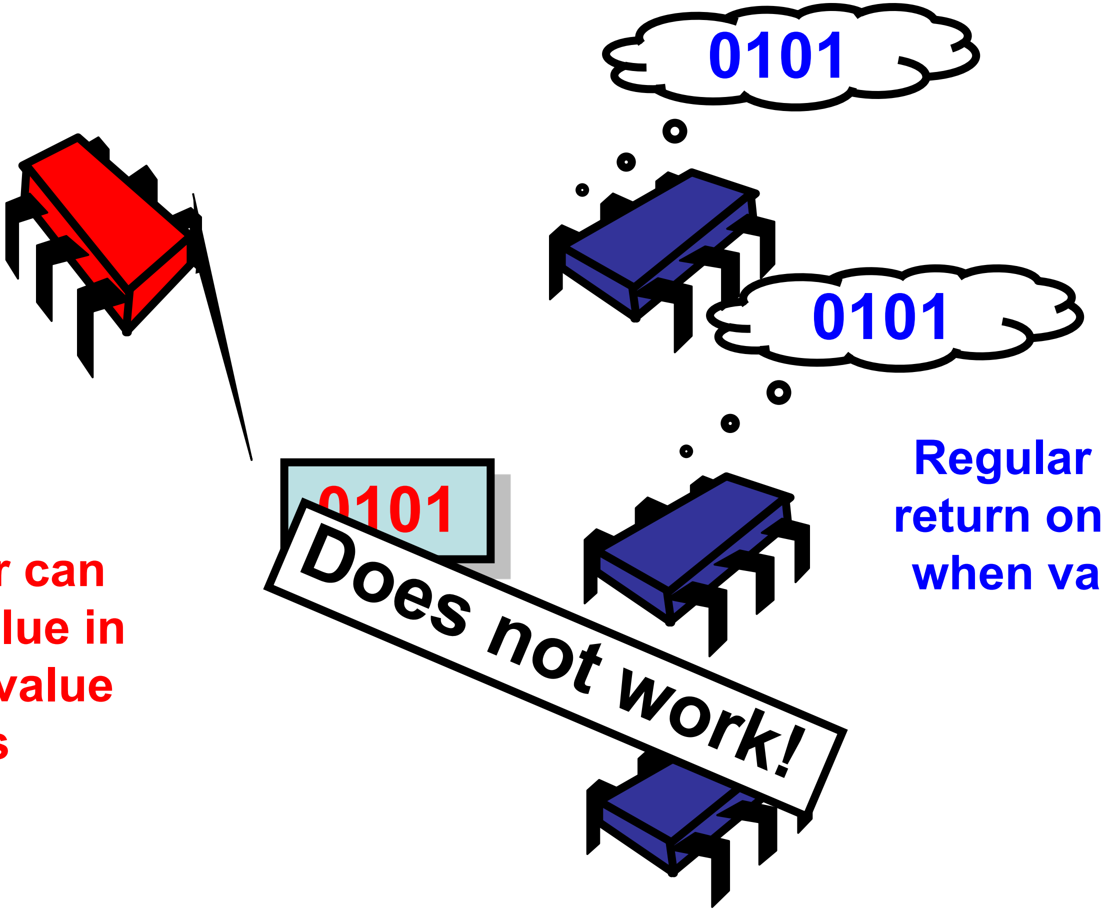
```
public class RegBoolMRSWRegister
 implements Register<Boolean>{
  threadLocal boolean old;
  private SafeBoolMRSWRegister value;
  public void write(boolean x) {
   if (old != x) {
    value.write(x);
    old = x;
   }}
  public boolean read() {
   return value.read();
 }}
```

Overlap? What overlap?

No problem

either Boolean value works

Regular Multi-Valued MRSW from Safe Multi-Valued MRSW?



Safe register can return any value in range when value changes

Regular register can return only old or new when value changes

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Questions?

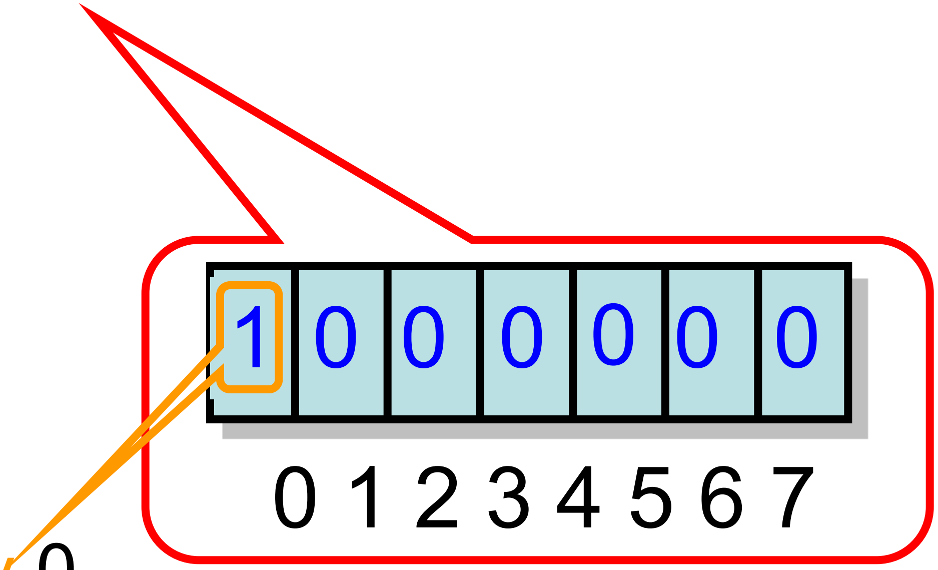
Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Representing *m* Values

Unary representation:
bit[i] means value i



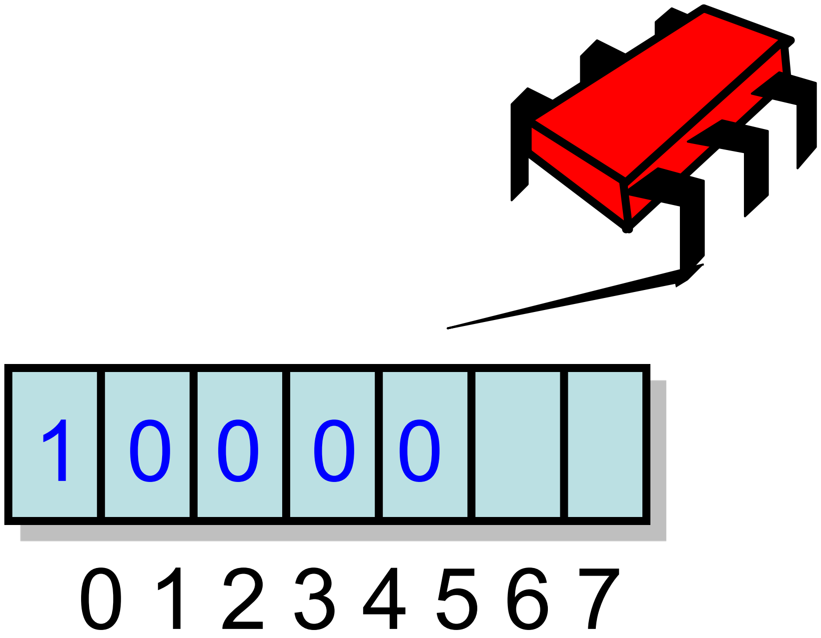
1 0 0 0 0 0 0

0 1 2 3 4 5 6 7

Initially 0

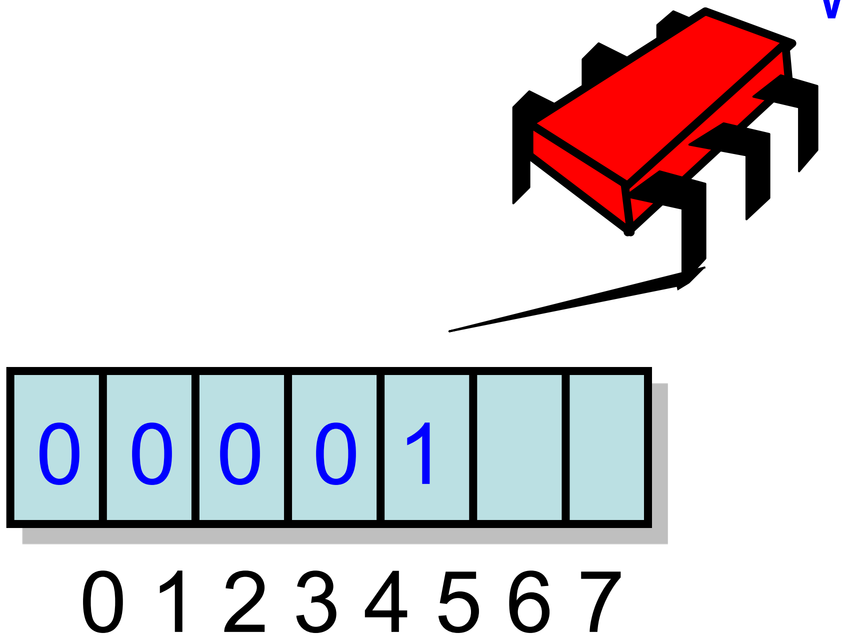
Writing *m*-Valued Register

Write 5



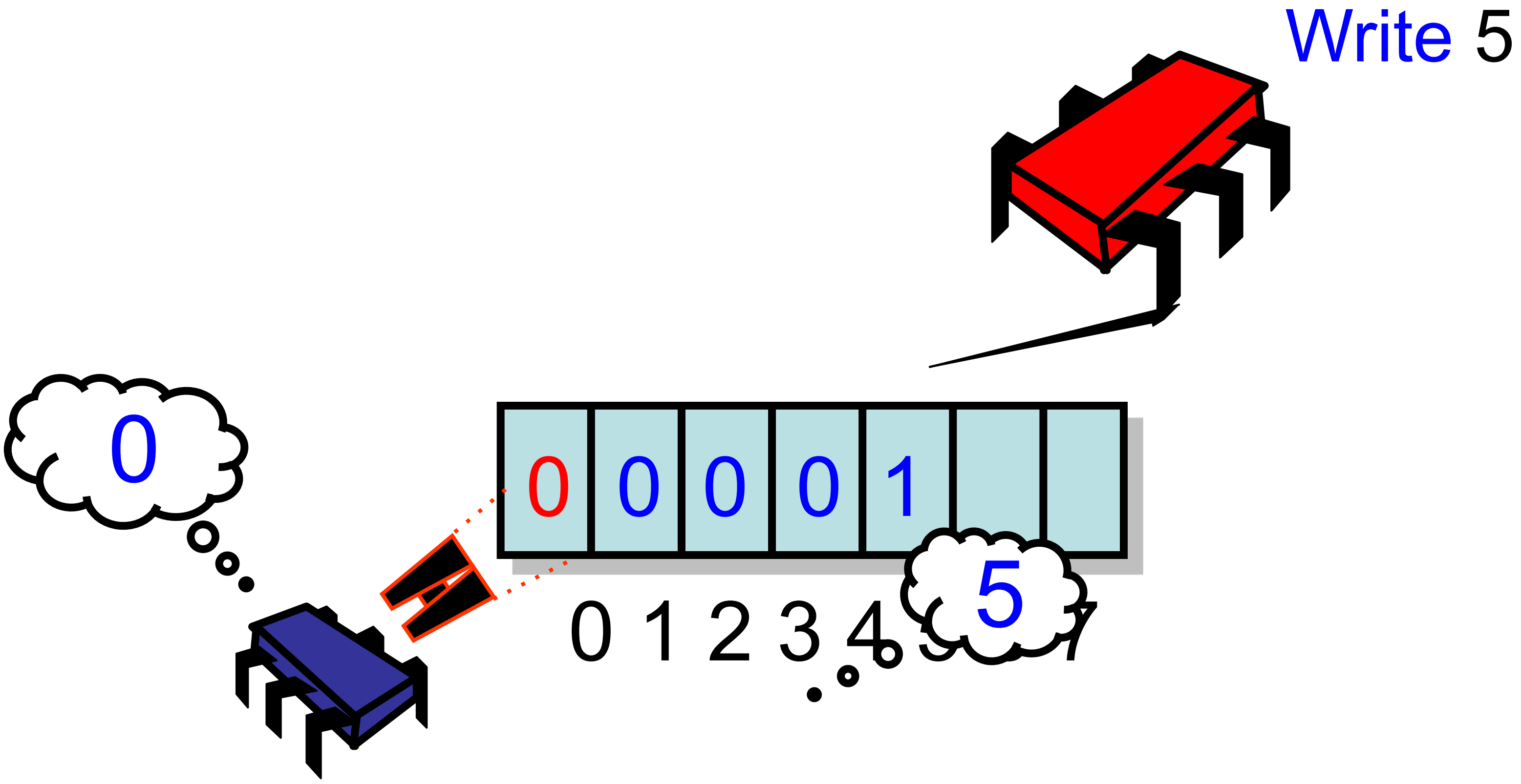
Writing *m*-Valued Register

Write 5



Initially 0

Writing *m*-Valued Register



MRSW Regular *m*-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register{
 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;
  }}
```

MRSW Regular *m*-valued from MRSW Regular Boolean

```
public class RegMRSWRegister implements Register{
  RegBoolMRSWRegister[M] bit;

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

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

Unary representation: bit[i] means value i

MRSW Regular *m*-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements Register {
  RegBoolMRSWRegister[m] bit;

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

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

set bit x

MRSW Regular *m*-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements Register {
  RegBoolMRSWRegister[m] bit;

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

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

Clear bits from higher to lower

MRSW Regular *m*-valued from MRSW Regular Boolean

```
public class RegMRSWRegisterimplements Register {
  RegBoolMRSWRegister[m] bit;

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

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

Scan from lower to higher & return first bit set

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Questions?

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

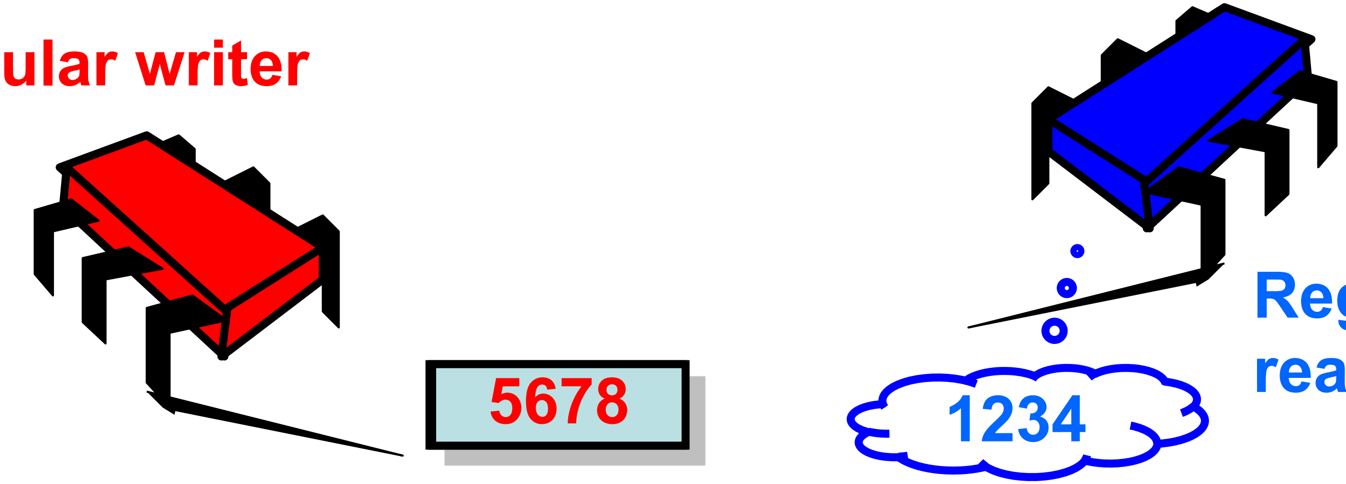
Road Map (Slight Detour)

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

SRSW Atomic

SRSW Atomic From SRSW Regular

Regular writer



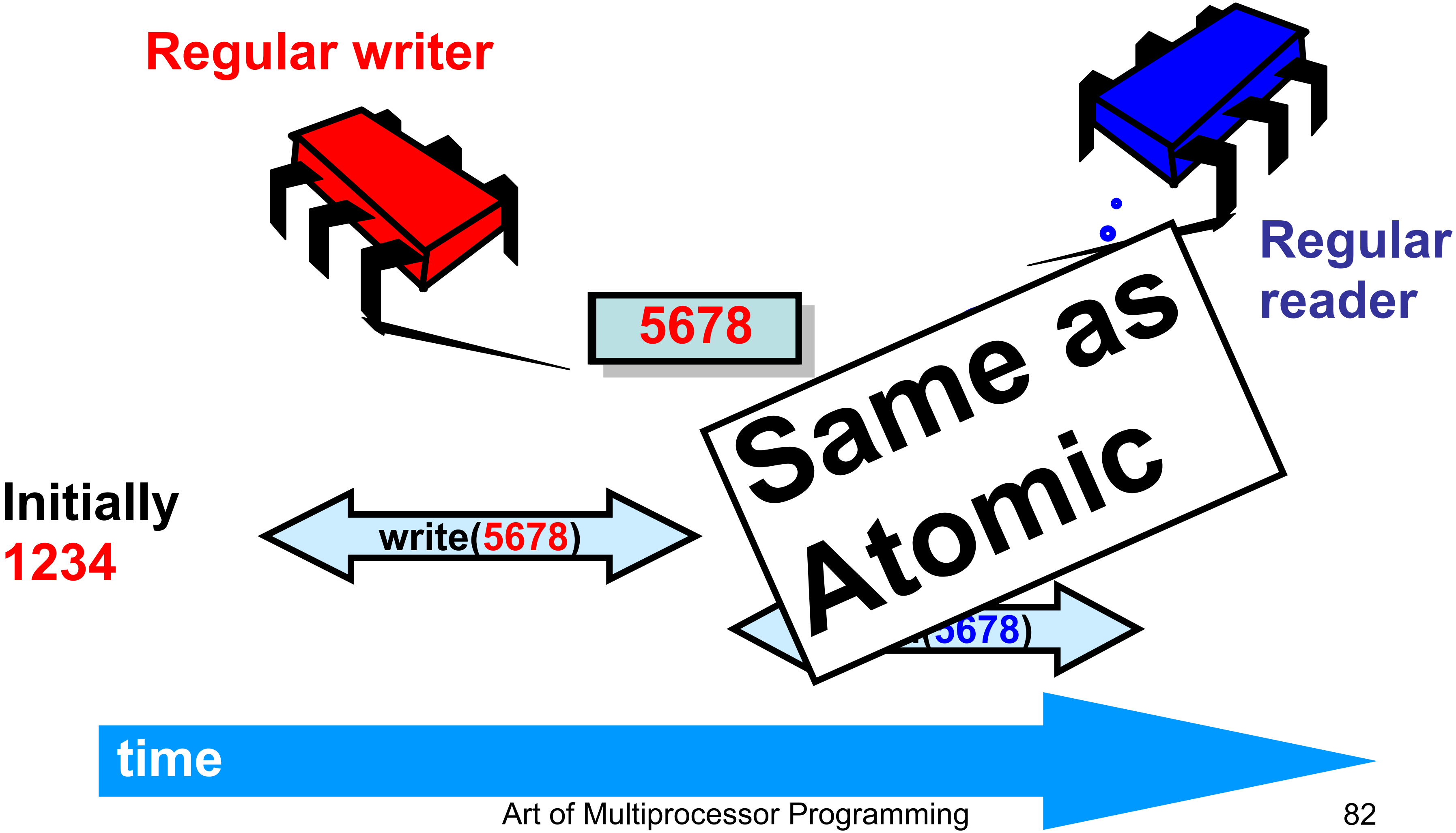
Regular reader

Instead of 5678…

Concurrent Reading

When is this a problem?

SRSW Atomic From SRSW Regular



Regular writer

5678

Regular reader

Same as Atomic

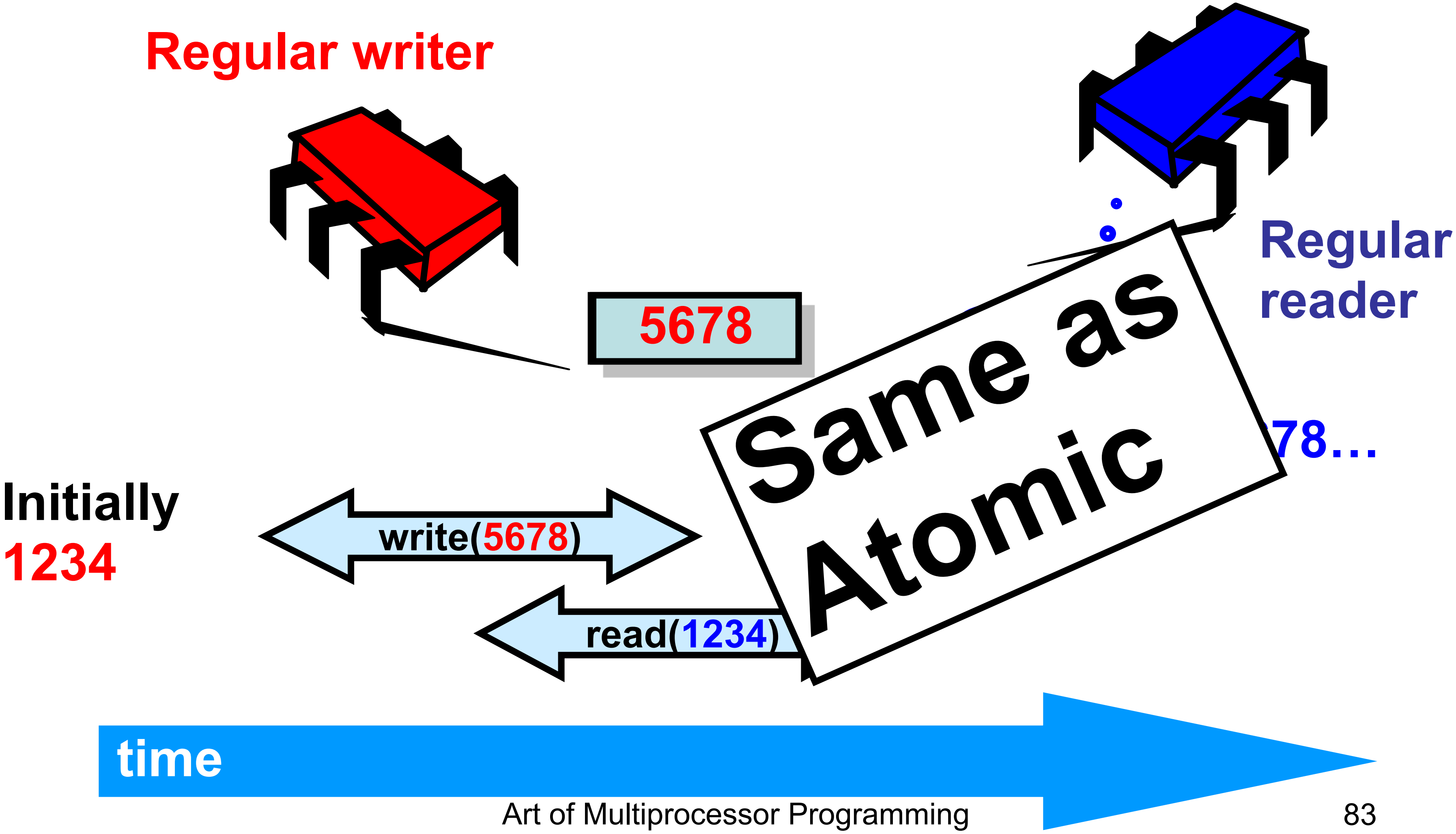
Initially 1234

write(5678)

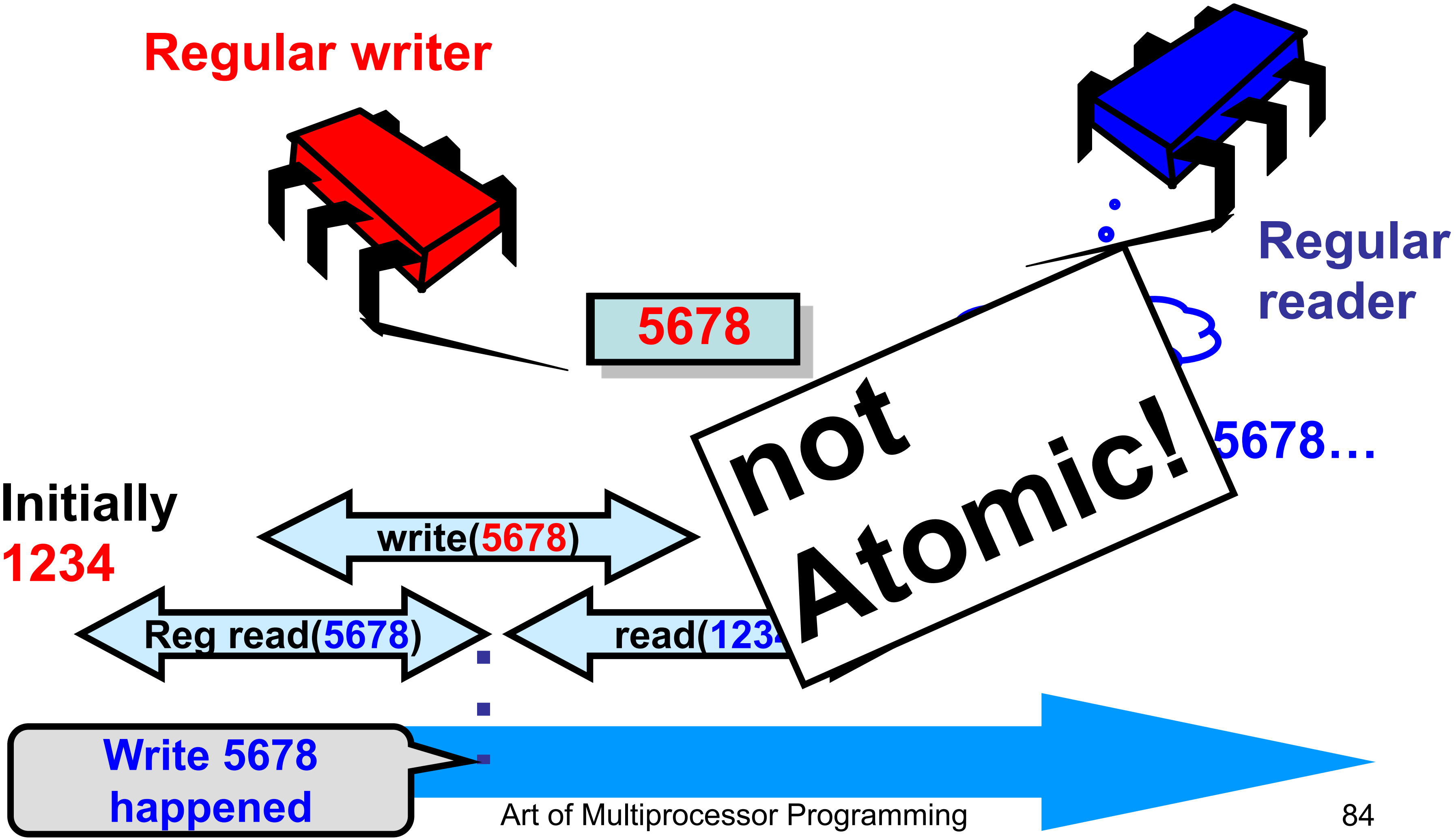
(5678)

time

SRSW Atomic From SRSW Regular



SRSW Atomic From SRSW Regular



Regular writer

5678

Regular reader

5678…

Initially
1234

write(5678)

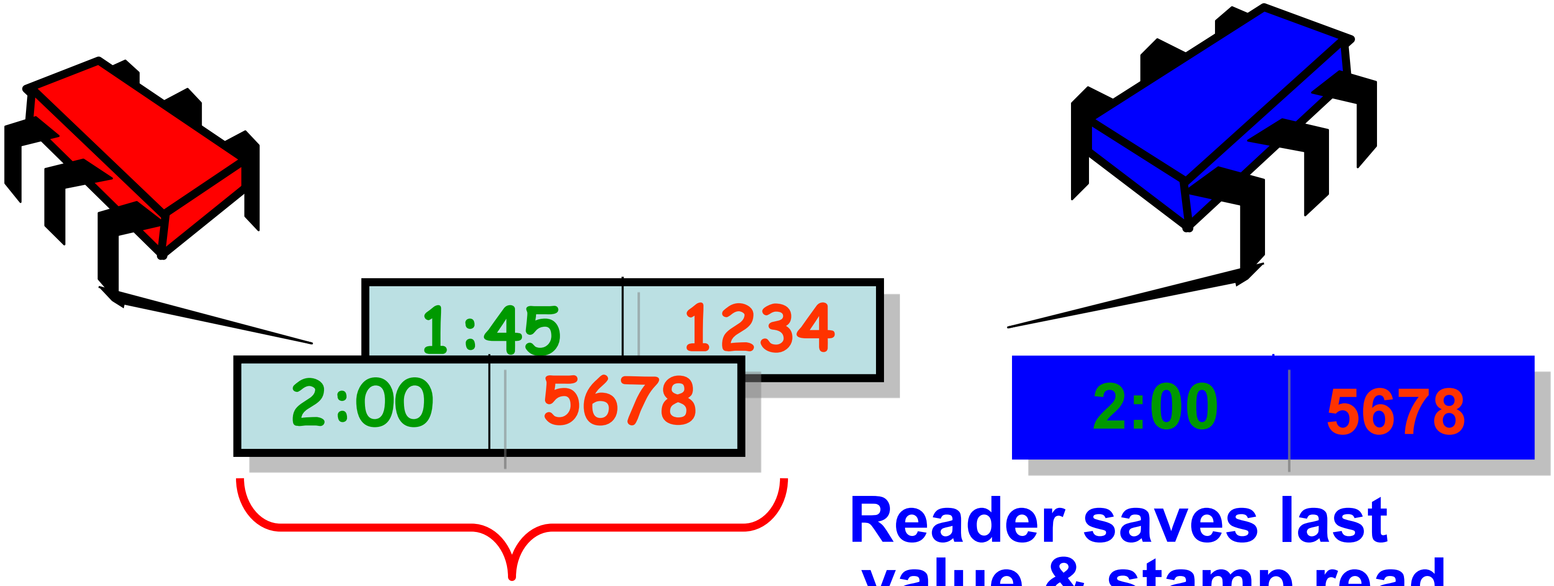
Reg read(5678)

read(123

Write 5678 happened

not Atomic!

Timestamped Values

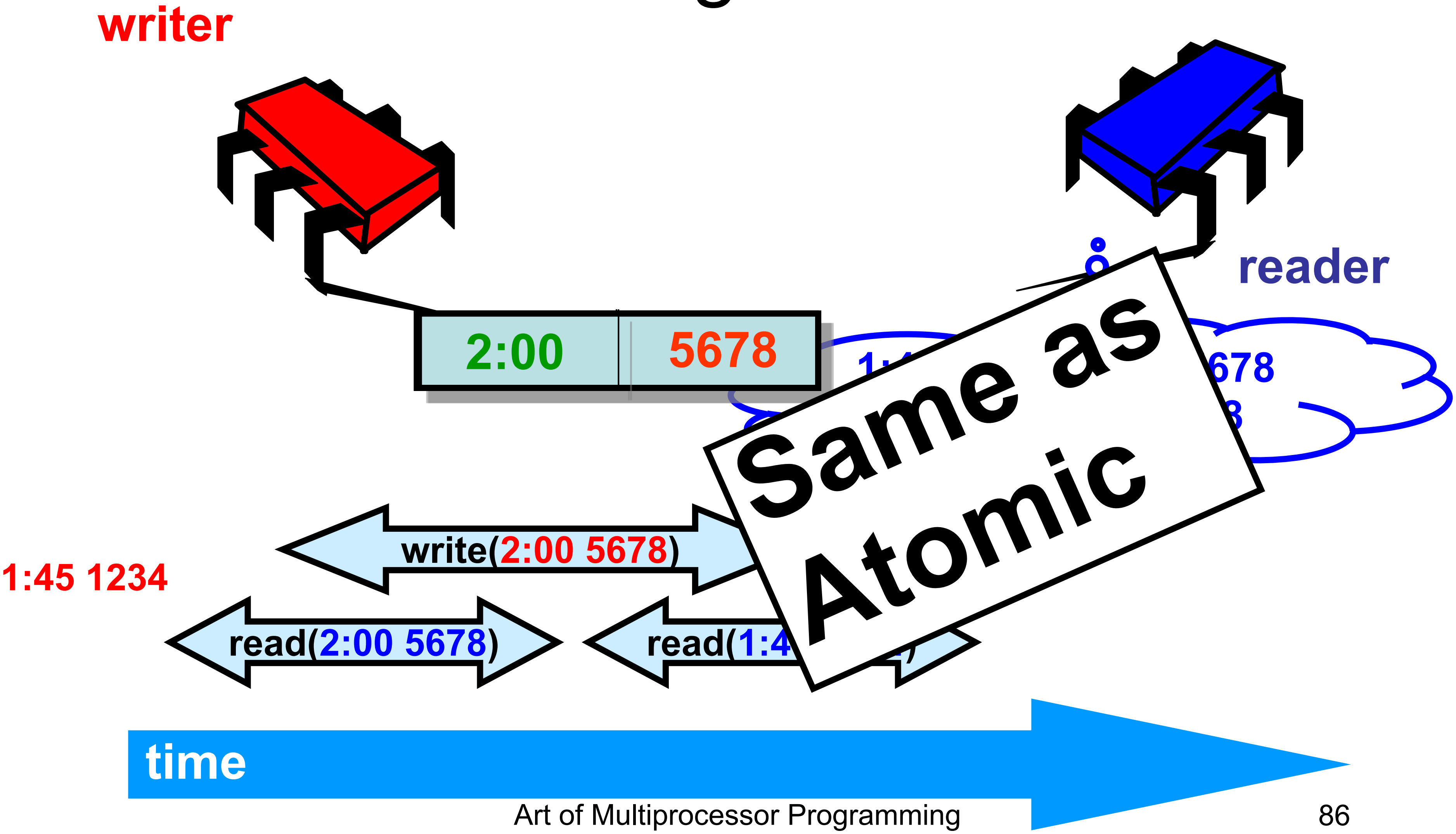


Writer writes value and stamp together

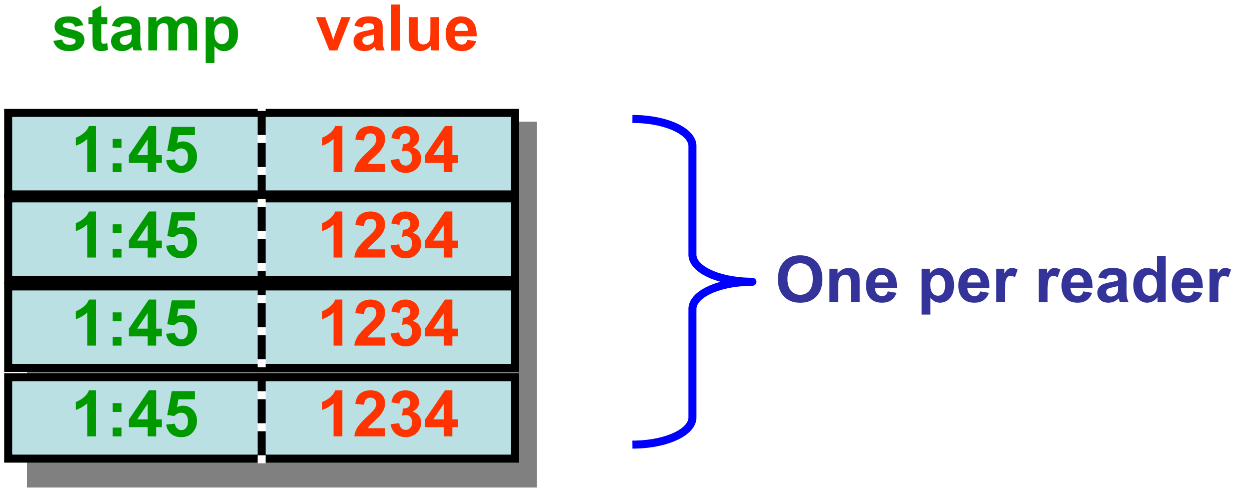
Reader saves last value & stamp read

returns new value only if stamp is higher

SRSW Atomic From SRSW Regular

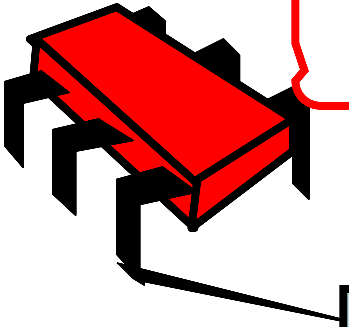


Atomic Single-Reader to Atomic Multi-Reader



stamp	value
1:45	1234
1:45	1234
1:45	1234
1:45	1234

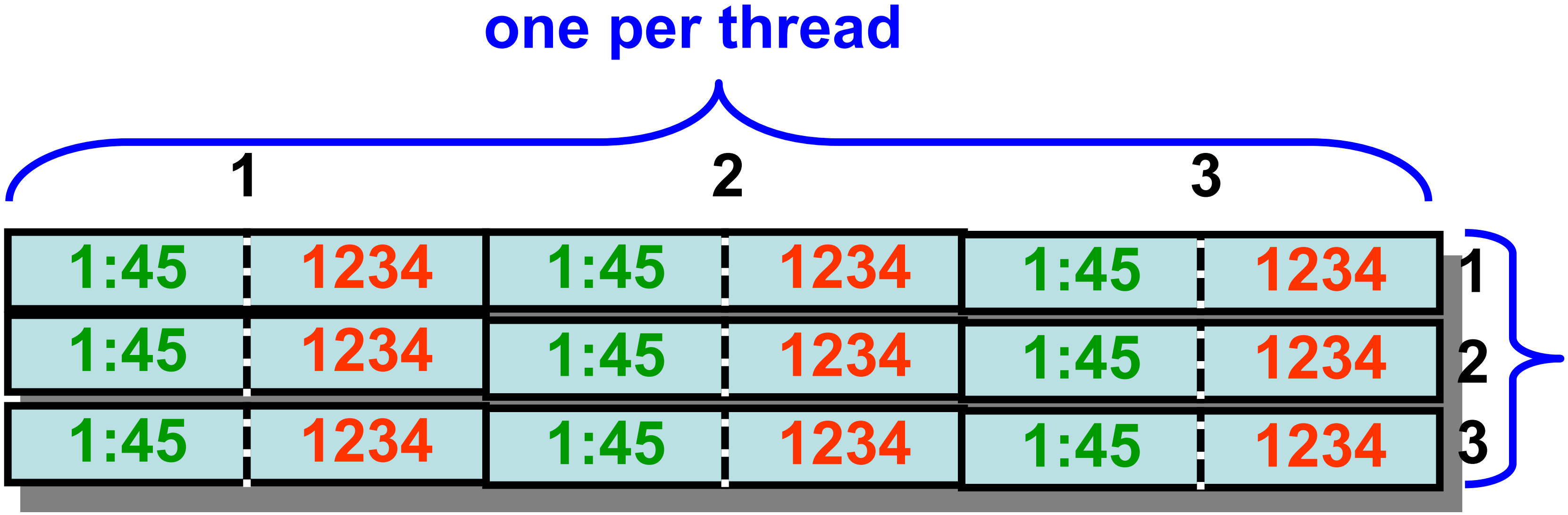
Another Scenario



Writer starts write…

stamp	value
2:00	5678
1:45	1234
1:45	1234
1:45	1234

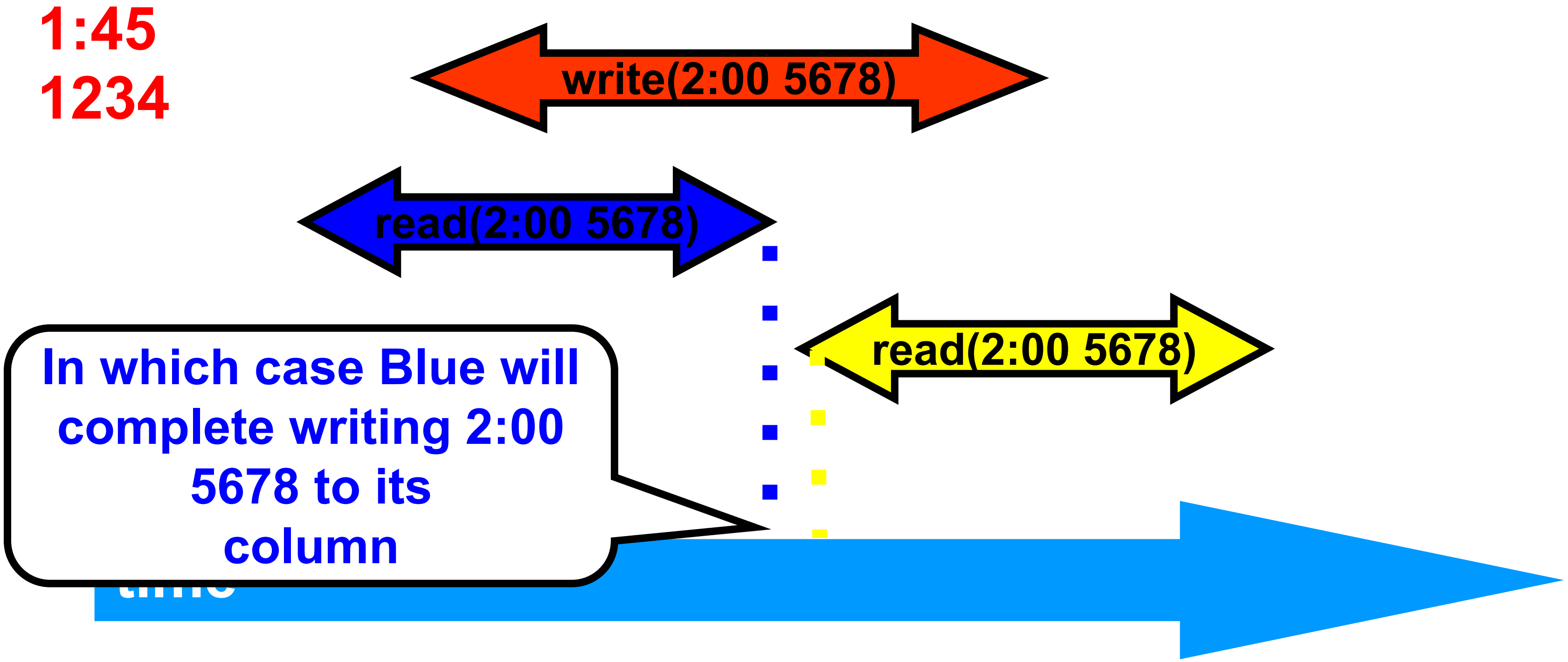
Multi-Reader Redux



one per thread

1		2		3		
1:45	1234	1:45	1234	1:45	1234	1
1:45	1234	1:45	1234	1:45	1234	2
1:45	1234	1:45	1234	1:45	1234	3

Bad Case Only When Readers Don’t Overlap



1:45
1234

write(2:00 5678)

read(2:00 5678)

read(2:00 5678)

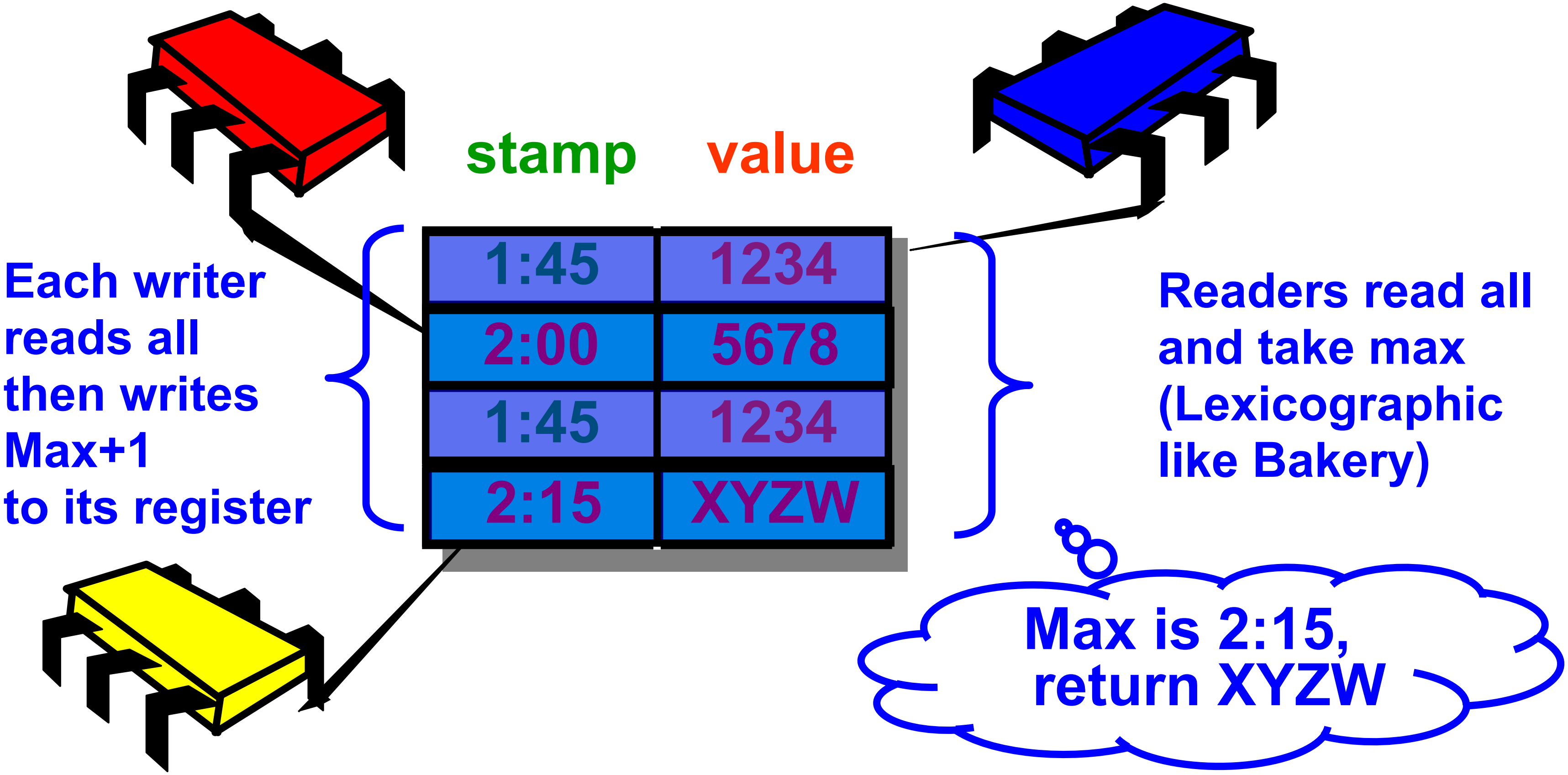
In which case Blue will complete writing 2:00 5678 to its column

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Multi-Writer Atomic From Multi-Reader Atomic



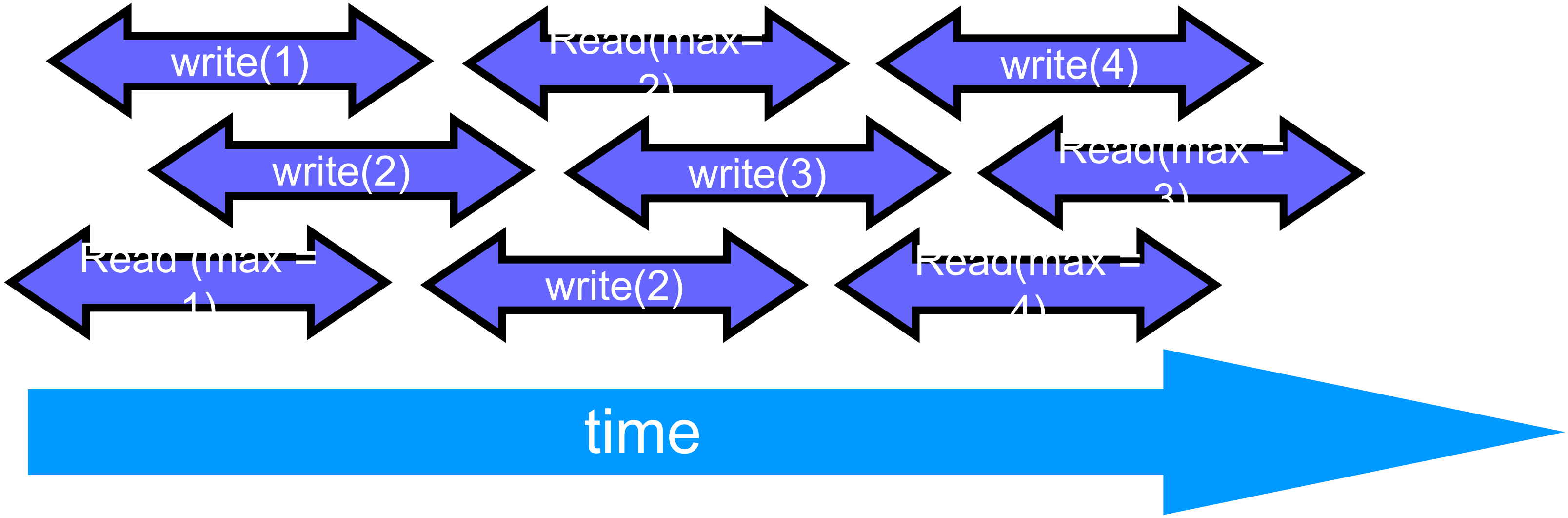
stamp	value
1:45	1234
2:00	5678
1:45	1234
2:15	XYZW

Each writer reads all then writes Max+1 to its register

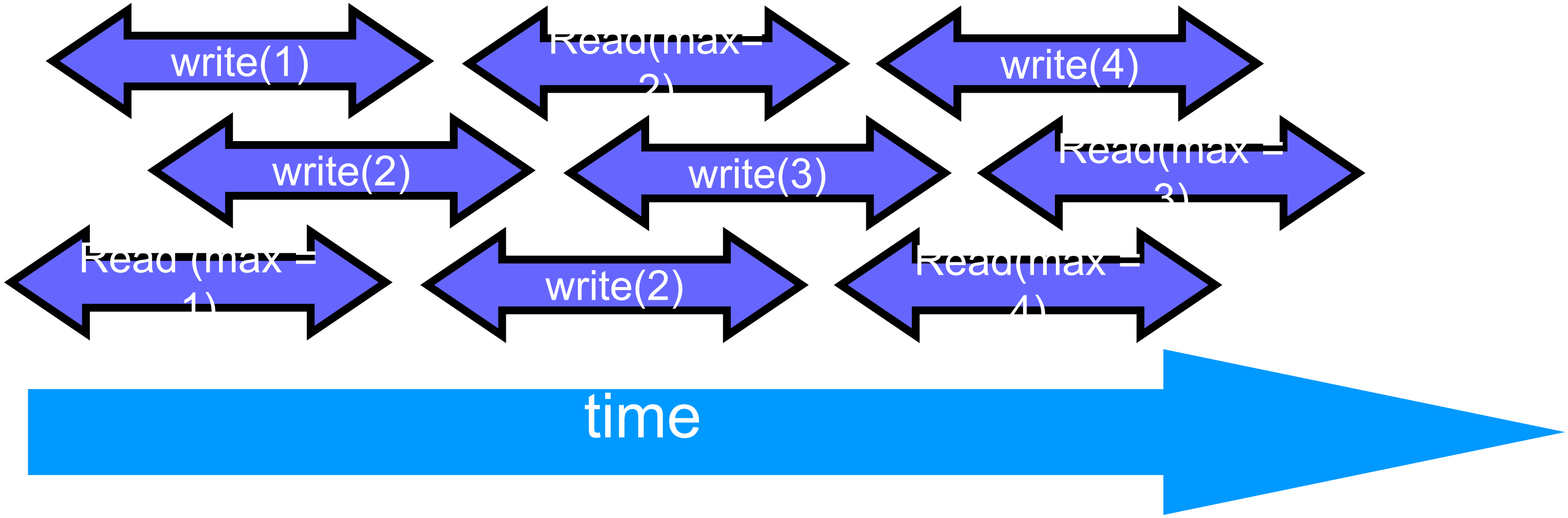
Readers read all and take max (Lexicographic like Bakery)

Max is 2:15, return XYZW

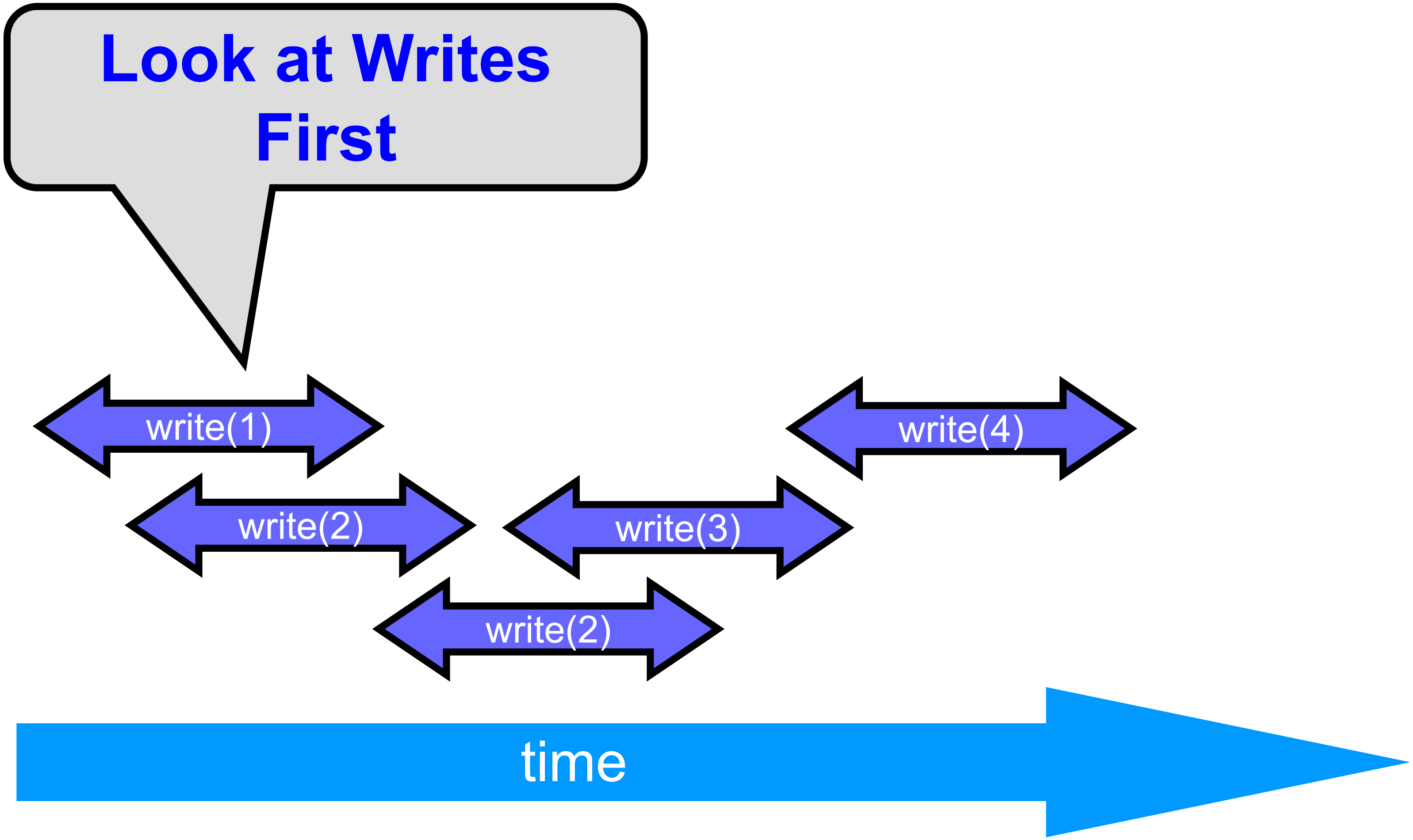
Atomic Execution Means it is Linearizable



Linearization Points

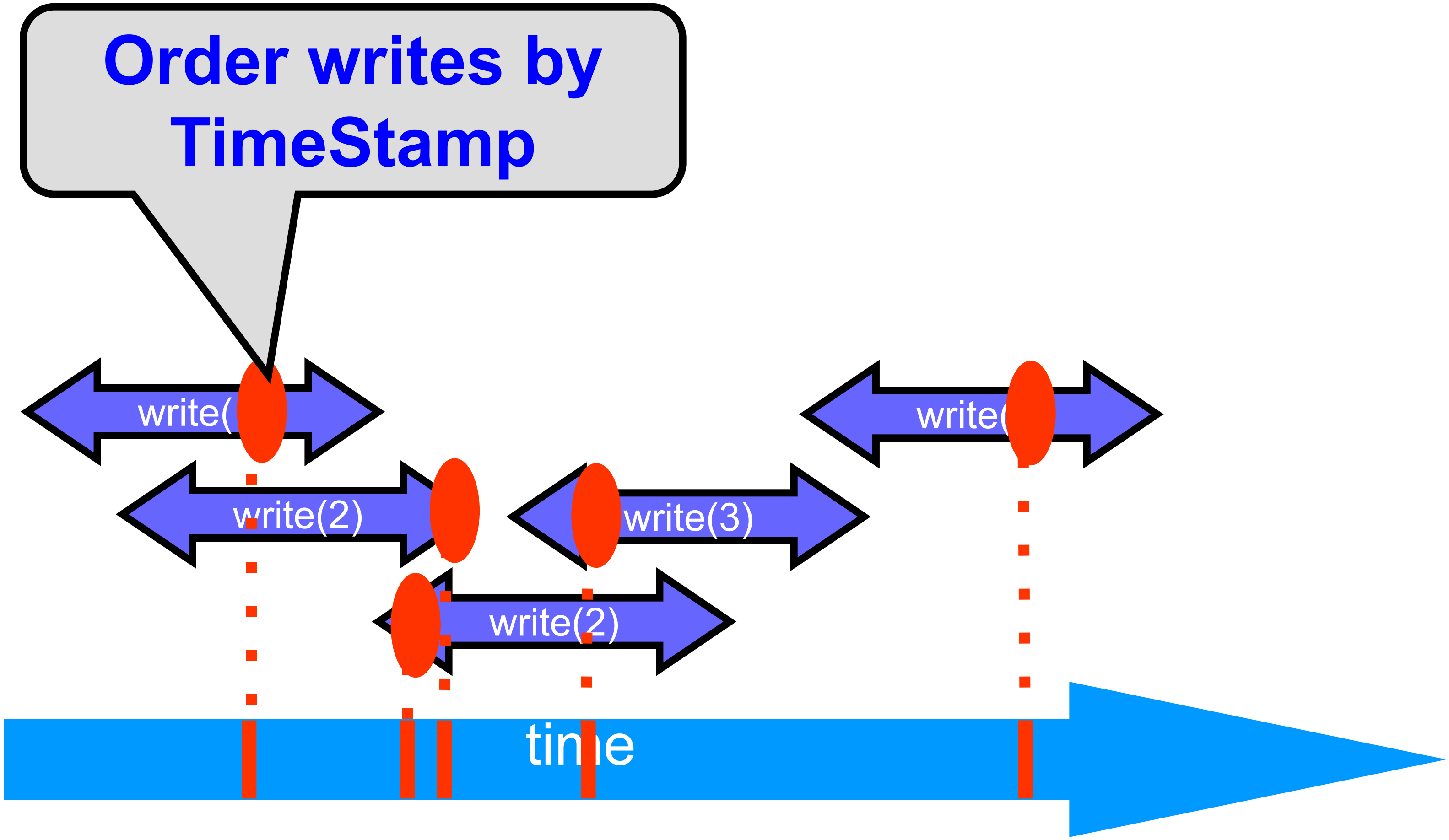


Linearization Points



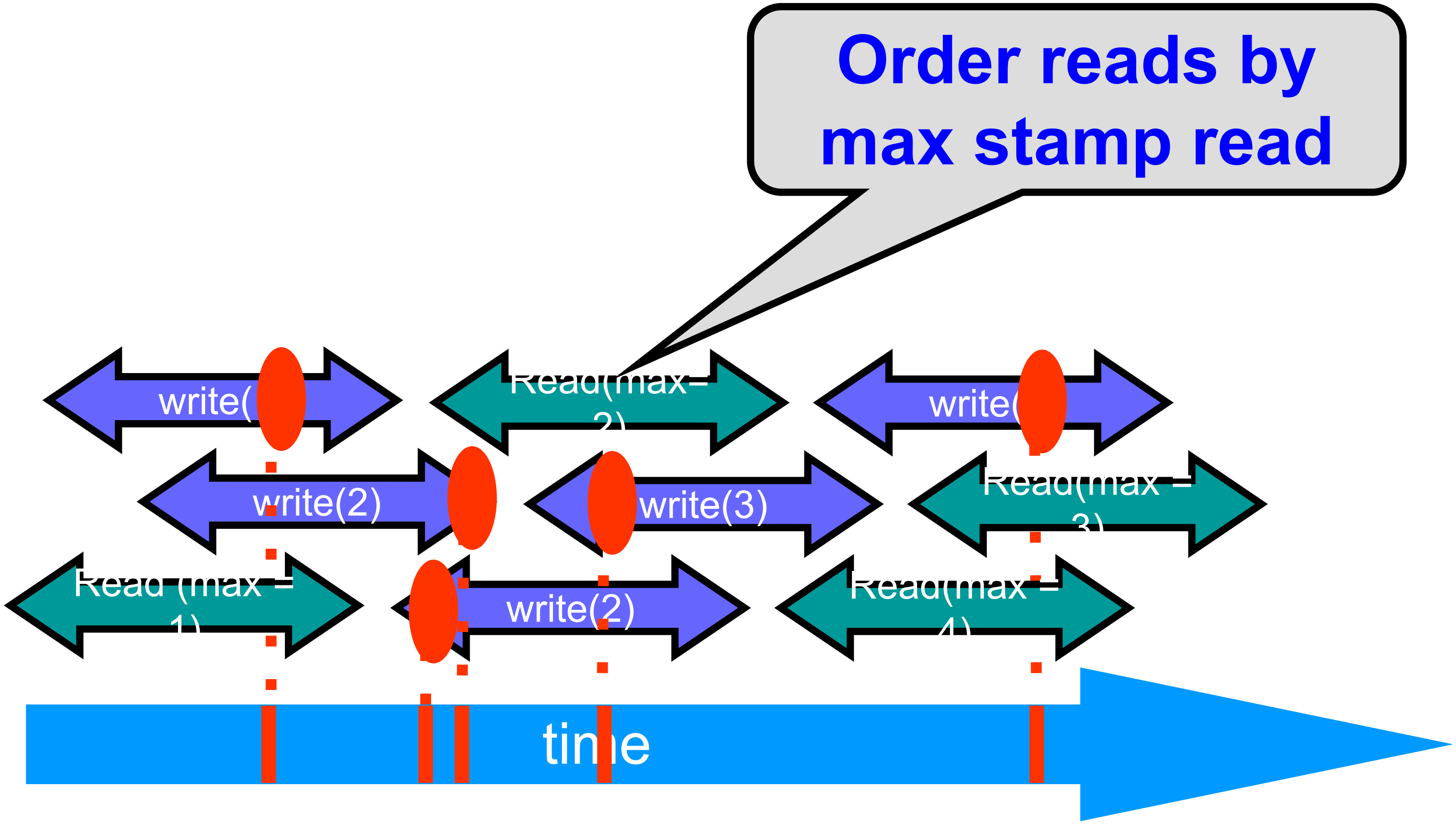
Look at Writes First

Linearization Points



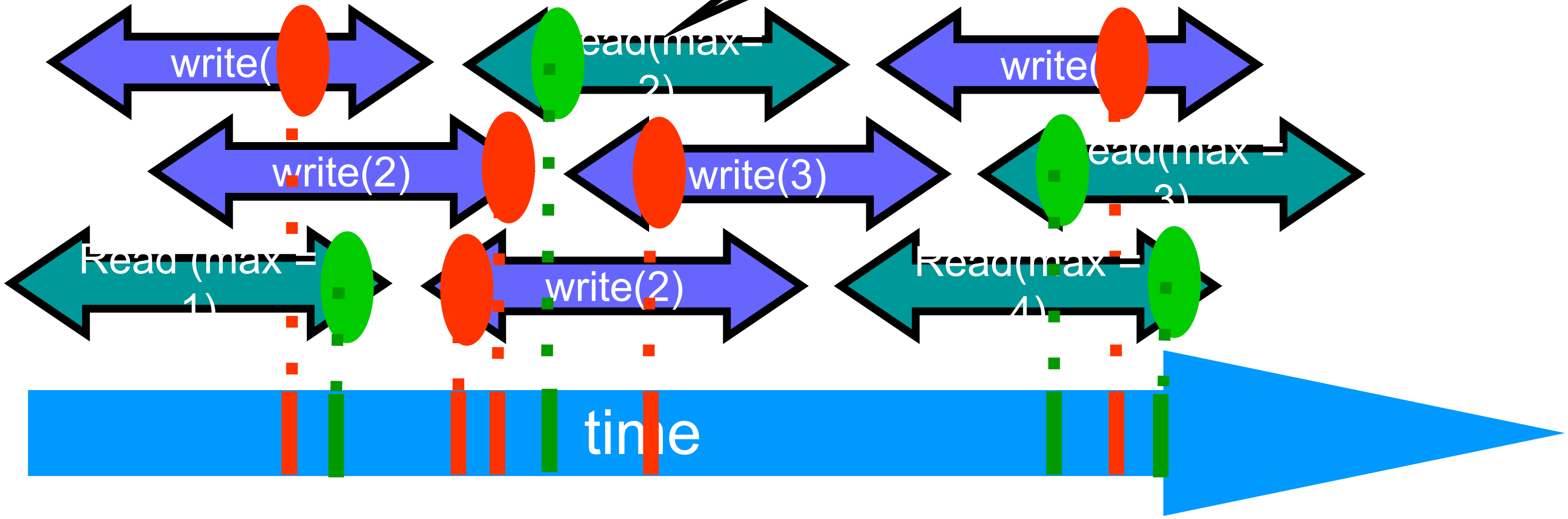
Order writes by TimeStamp

Linearization Points


Order reads by max stamp read

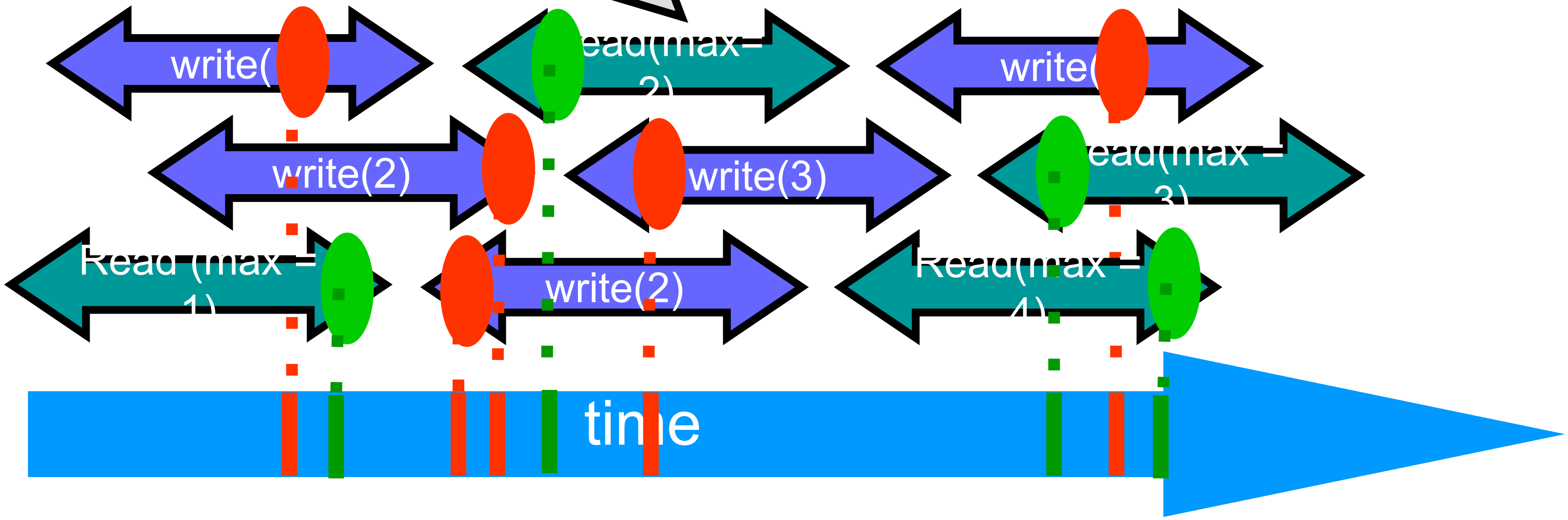
Linearization Points

Order reads by max stamp read



Linearization Points

The linearization point depends on the execution (not a line in the code)!



Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

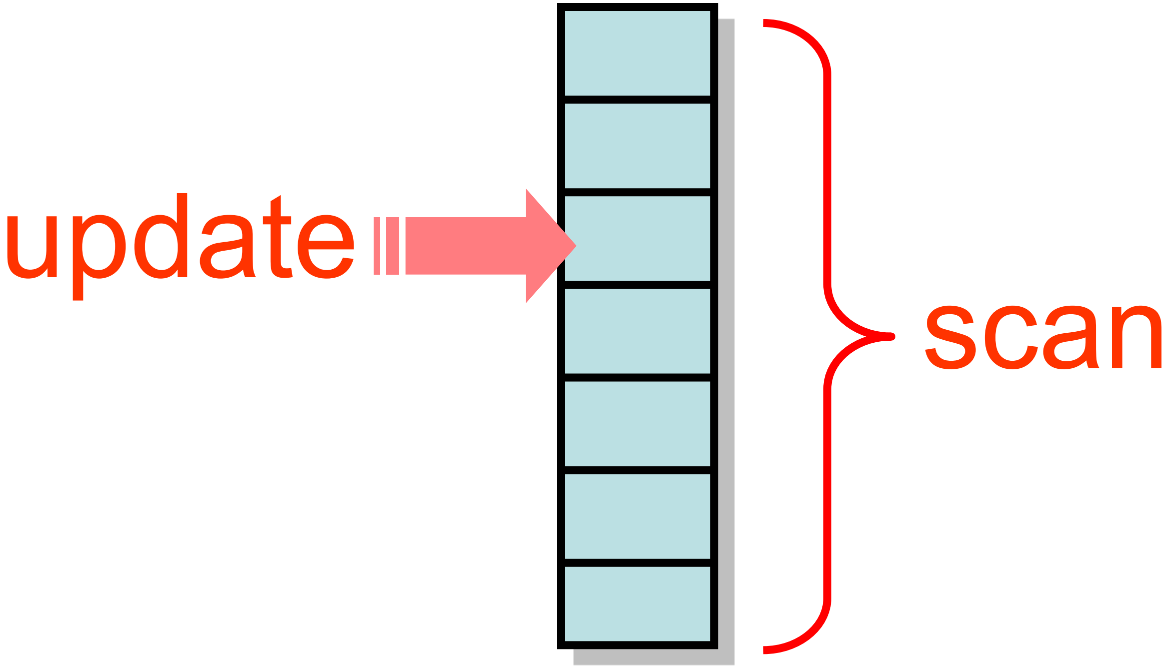
Questions?

Road Map

- SRSW safe Boolean
- MRSW safe Boolean
- MRSW regular Boolean
- MRSW regular
- MRSW atomic
- MRMW atomic
- Atomic snapshot

Next

Atomic Snapshot



Atomic Snapshot

- Array of SWMR atomic registers
- Take instantaneous snapshot of all
- Generalizes to MRMW registers …

Snapshot Interface

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Snapshot Interface

Thread i writes v to its register

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Snapshot Interface

Instantaneous snapshot of all theads’ registers

```
public interface Snapshot {
  public int update(int v);
  public int[] scan();
}
```

Atomic Snapshot

- Collect
  - Read values one at a time
- Problem
  - Incompatible concurrent collects
  - Result not linearizable

Clean Collects

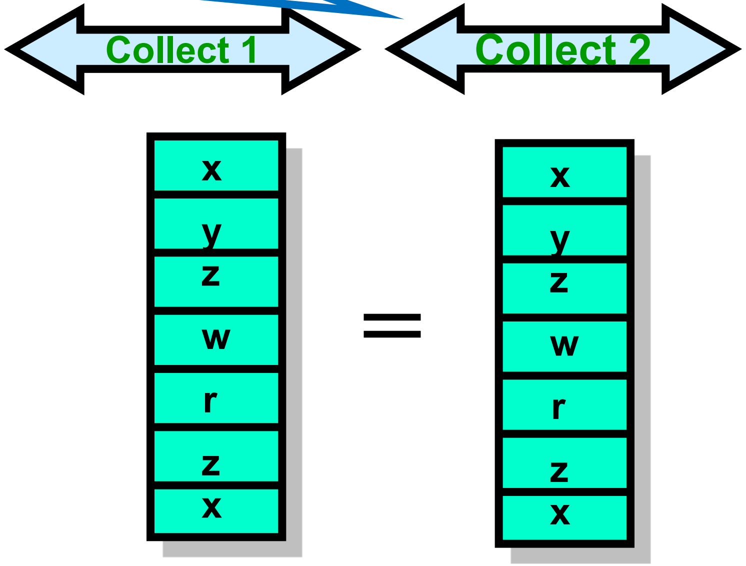
- Clean Collect
  - Collect during which nothing changed
  - Can we make it happen?
  - Can we detect it?

Simple Snapshot

- Put increasing labels on each entry

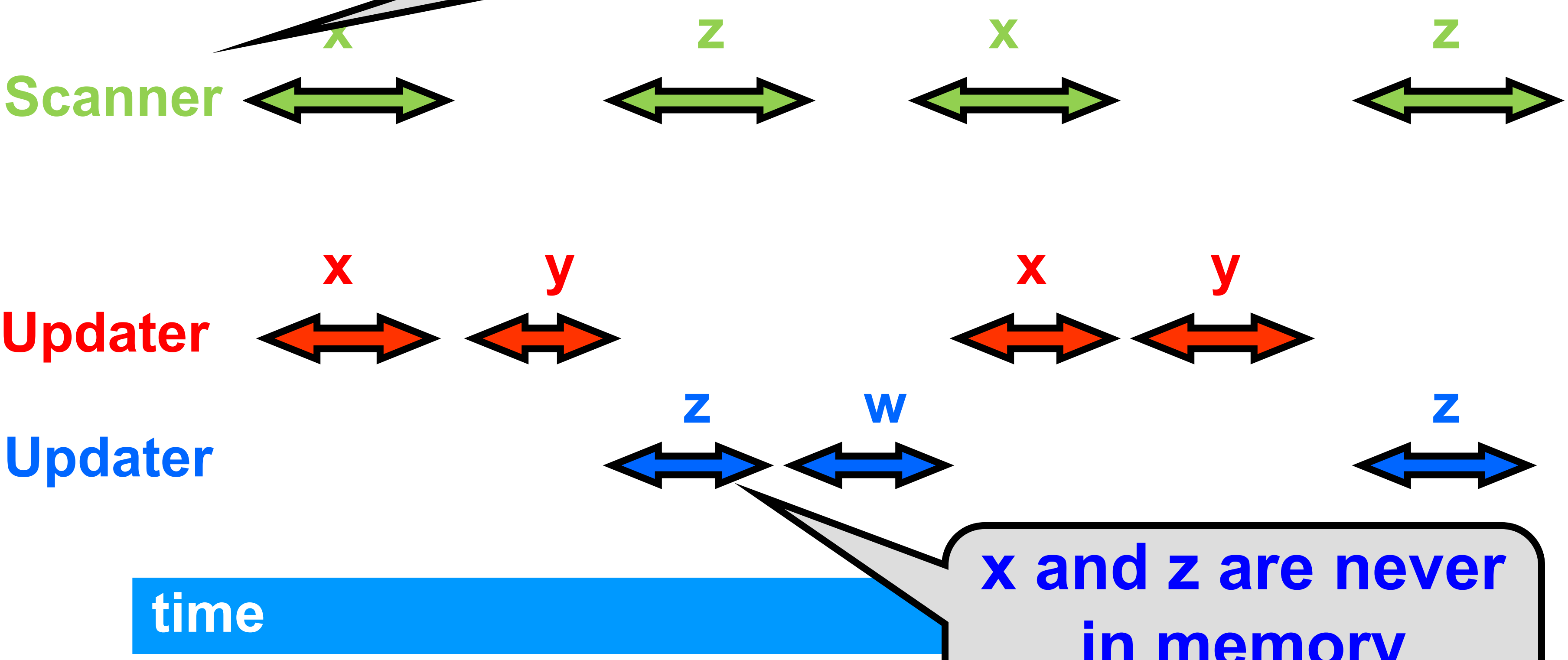
Problem: Scanner might not be collecting a snapshot!

  - We're done
- Otherwise,
  - Try again



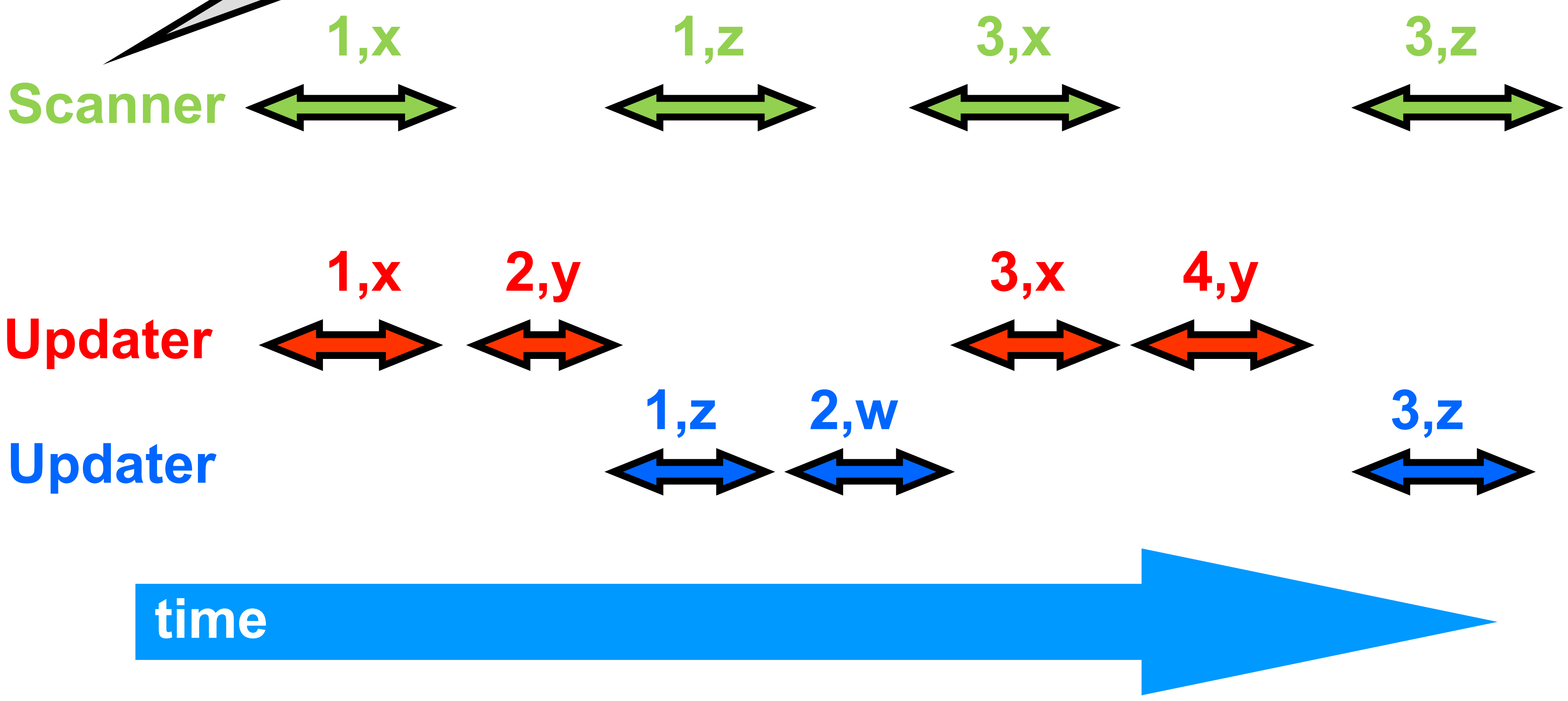
Claim:

But scanner sees x and z together!



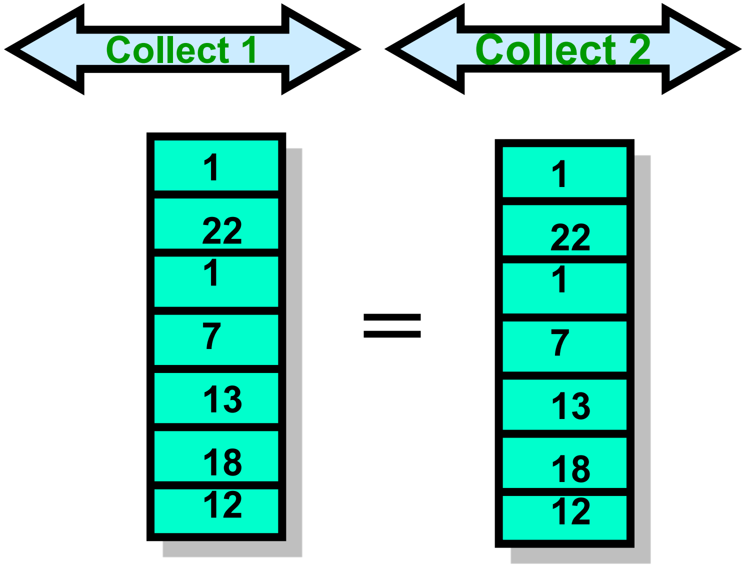
x and z are never in memory together

Scanner reads x and z with different labels and recognizes collect not clean



Simple Snapshot

- Collect twice
- If both agree,
  - We're done
- Otherwise,
  - Try again



Simple Snapshot: Update

```
public 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: Update

```
public 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);
  }
```

One single-writer register per thread

Simple Snapshot: Update

```
public 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);
  }
```

Write each time with higher label

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;
}
```

Simple Snapshot

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

Just read each register into array

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: 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);
}}
```

Collect once

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);
}}
```

Collect once

Collect twice

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);
}}
```

Collect once

Collect twice

On mismatch, try again

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);
}}
```

Collect once

Collect twice

On match, return values

Simple Snapshot

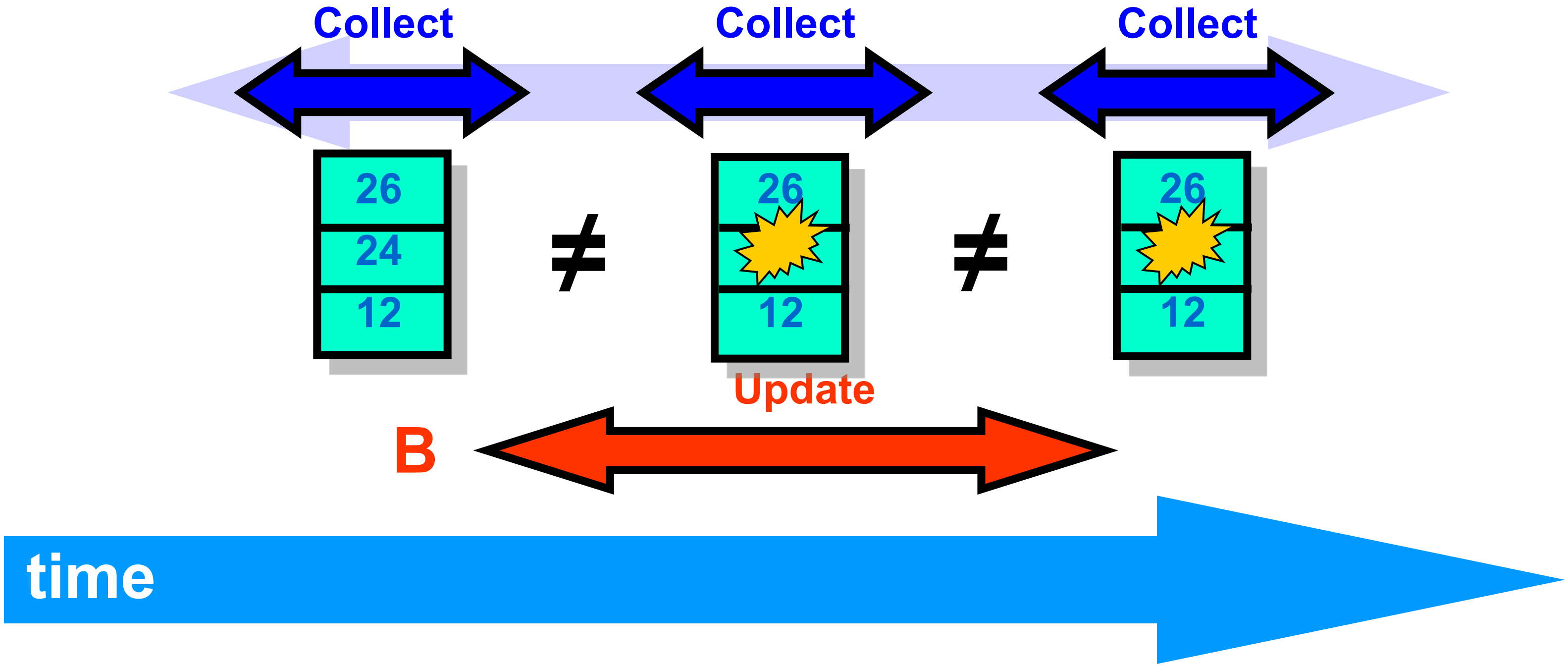
- Linearizable
- Update is wait-free
  - No unbounded loops
- But Scan can starve
  - If interrupted by concurrent update

Wait-Free Snapshot

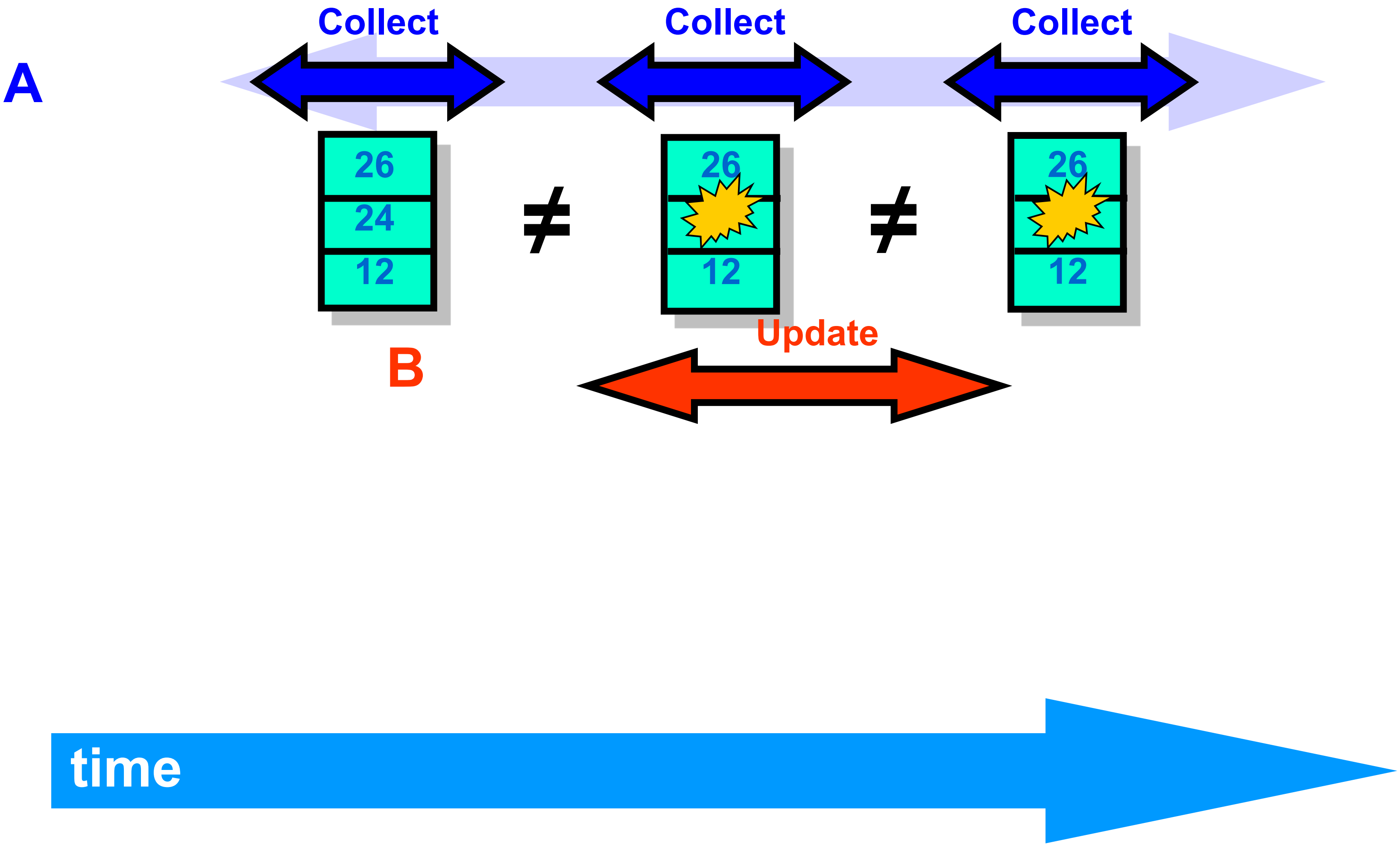
- Add a scan before every update
- Write resulting snapshot together with update value
- If scan is continuously interrupted by updates, scan can take the update's snapshot

Wait-free Snapshot

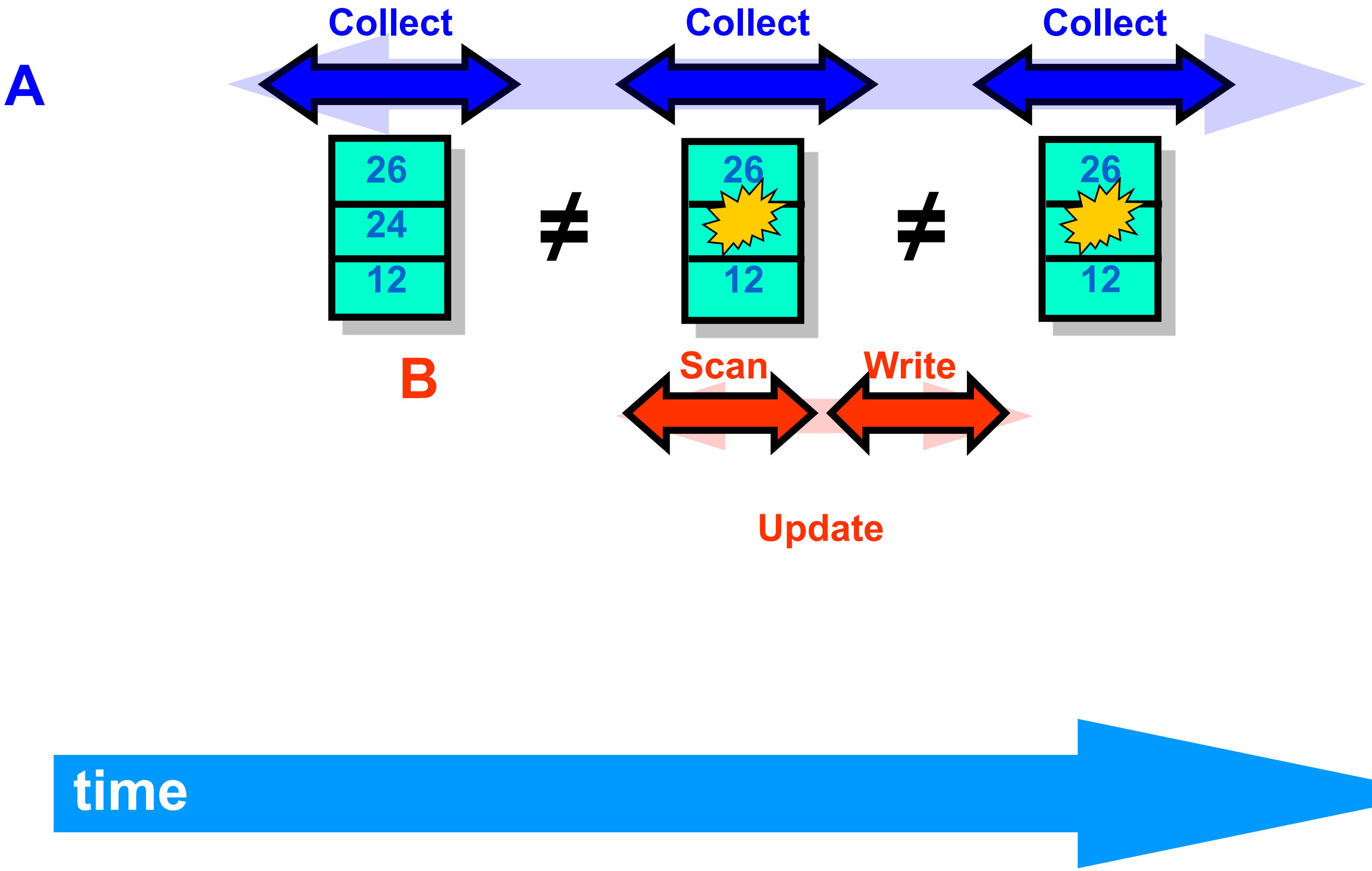
If A's scan observes that B moved ***twice***, then B completed an update while A's scan was in progress



Wait-free Snapshot

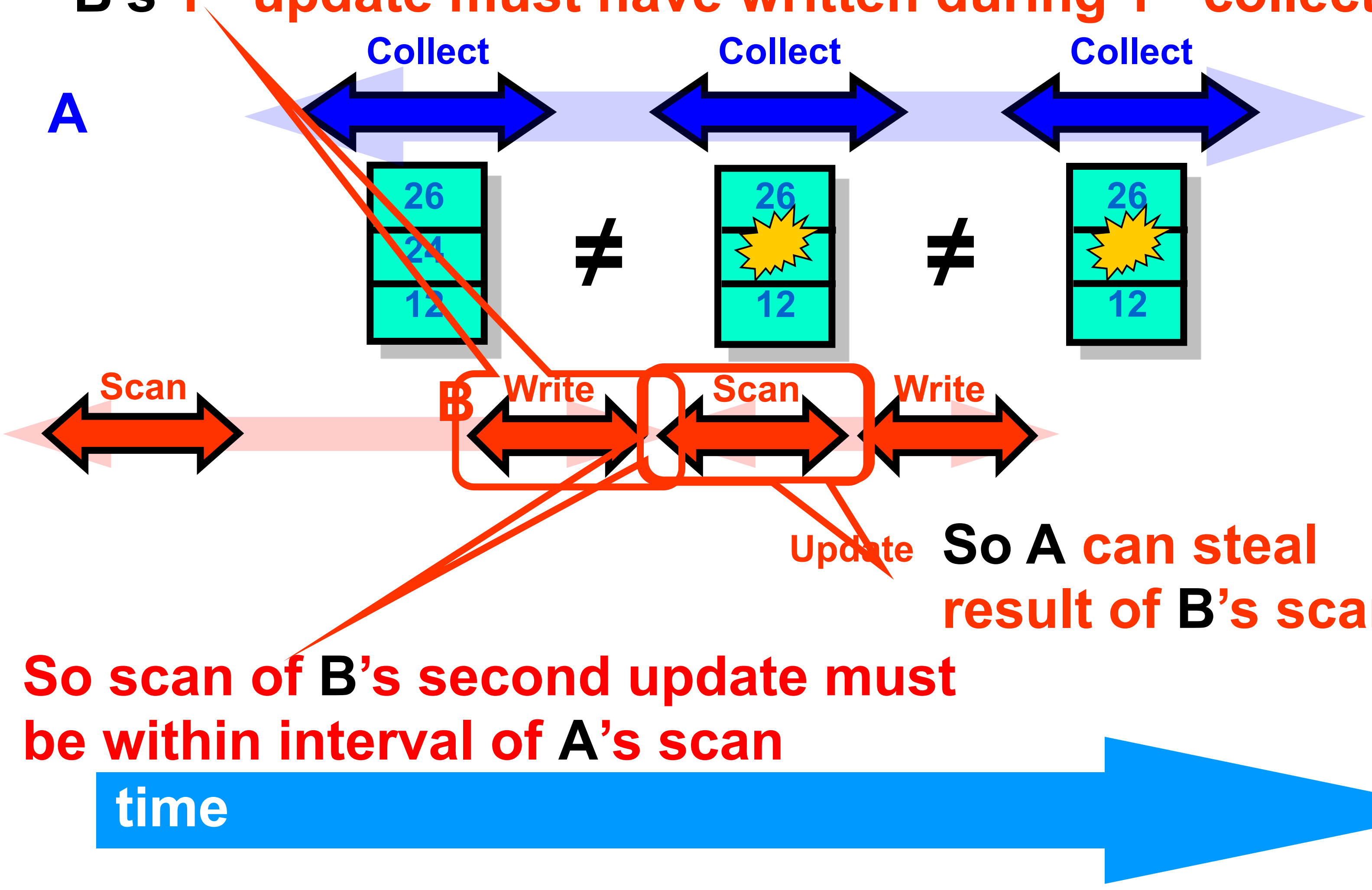


Wait-free Snapshot



Wait-free Snapshot

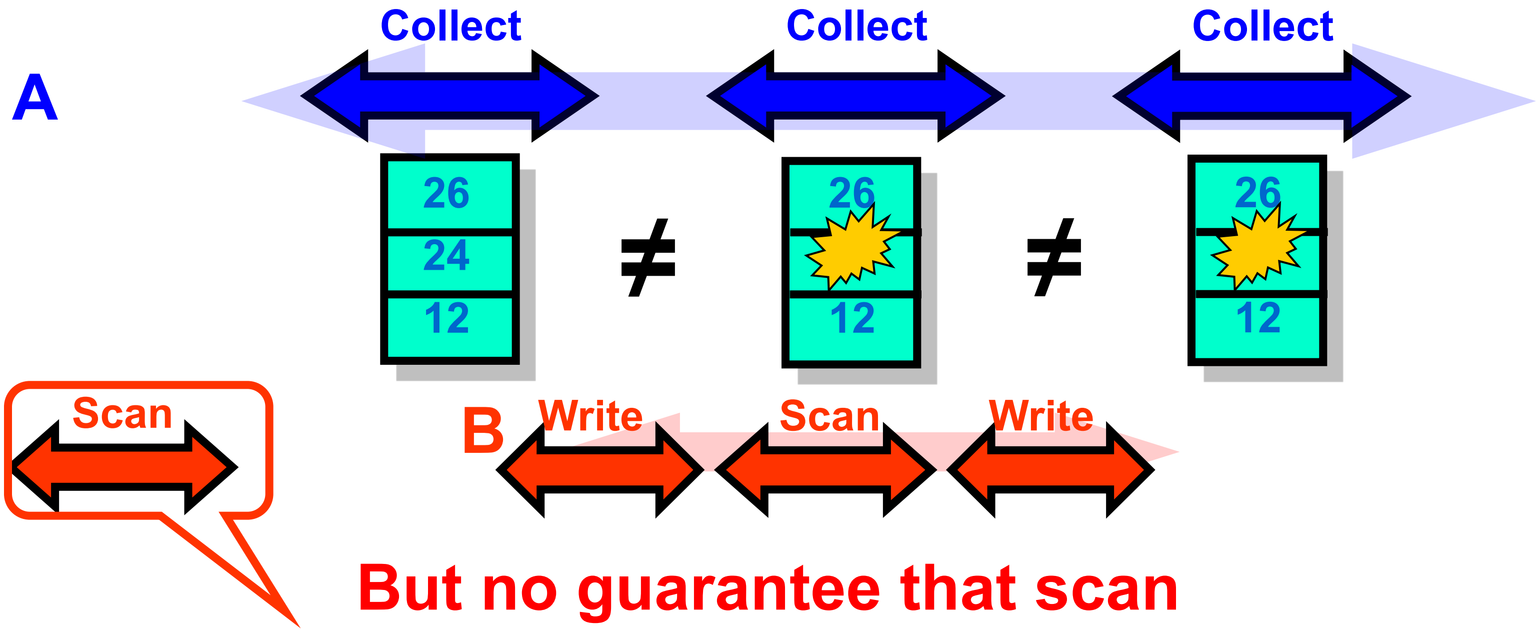
B's 1st update must have written during 1st collect



So A can steal result of B's scan

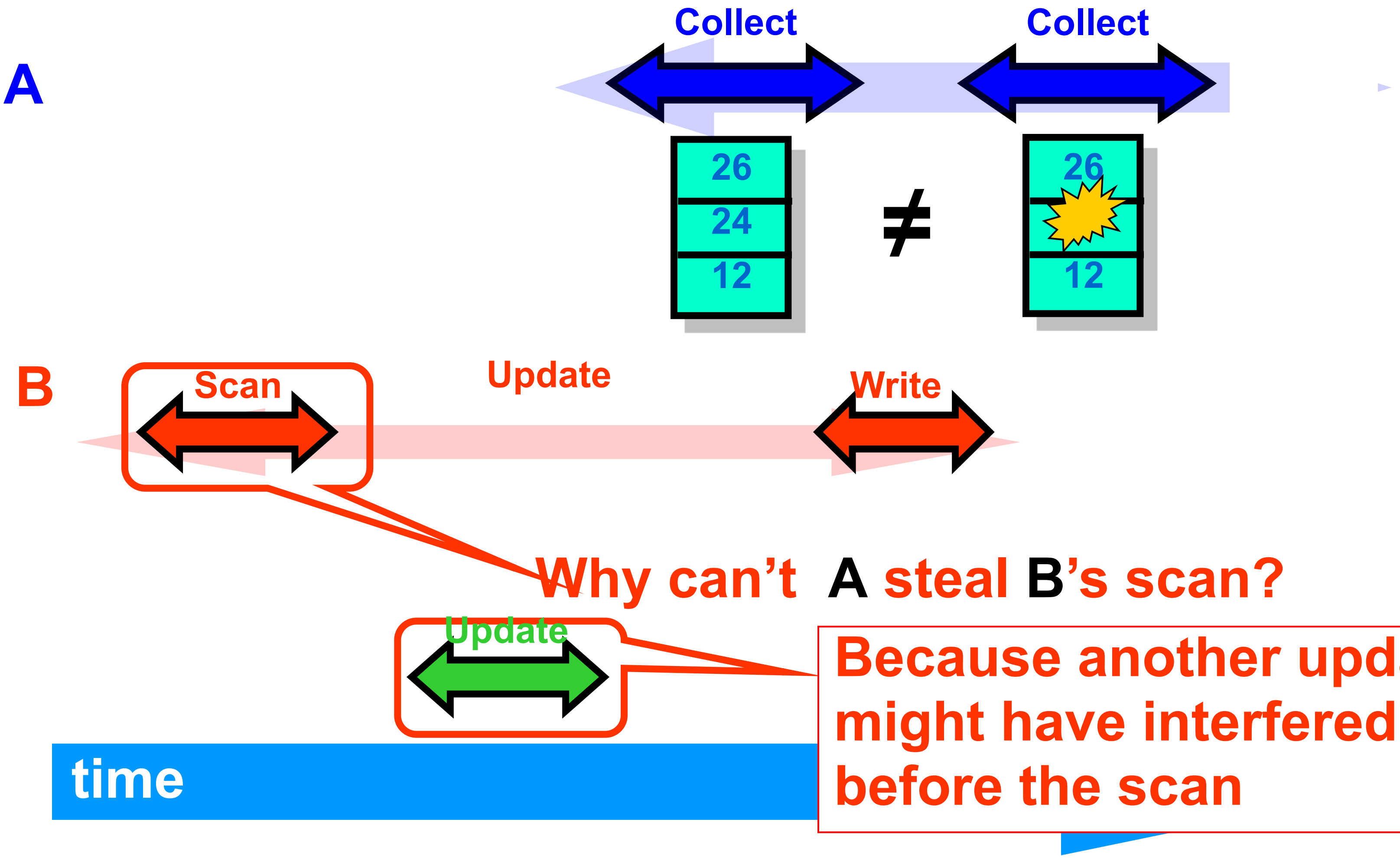
So scan of B's second update must be within interval of A's scan

Wait-free Snapshot



But no guarantee that scan of B's 1st update can be used… Why?

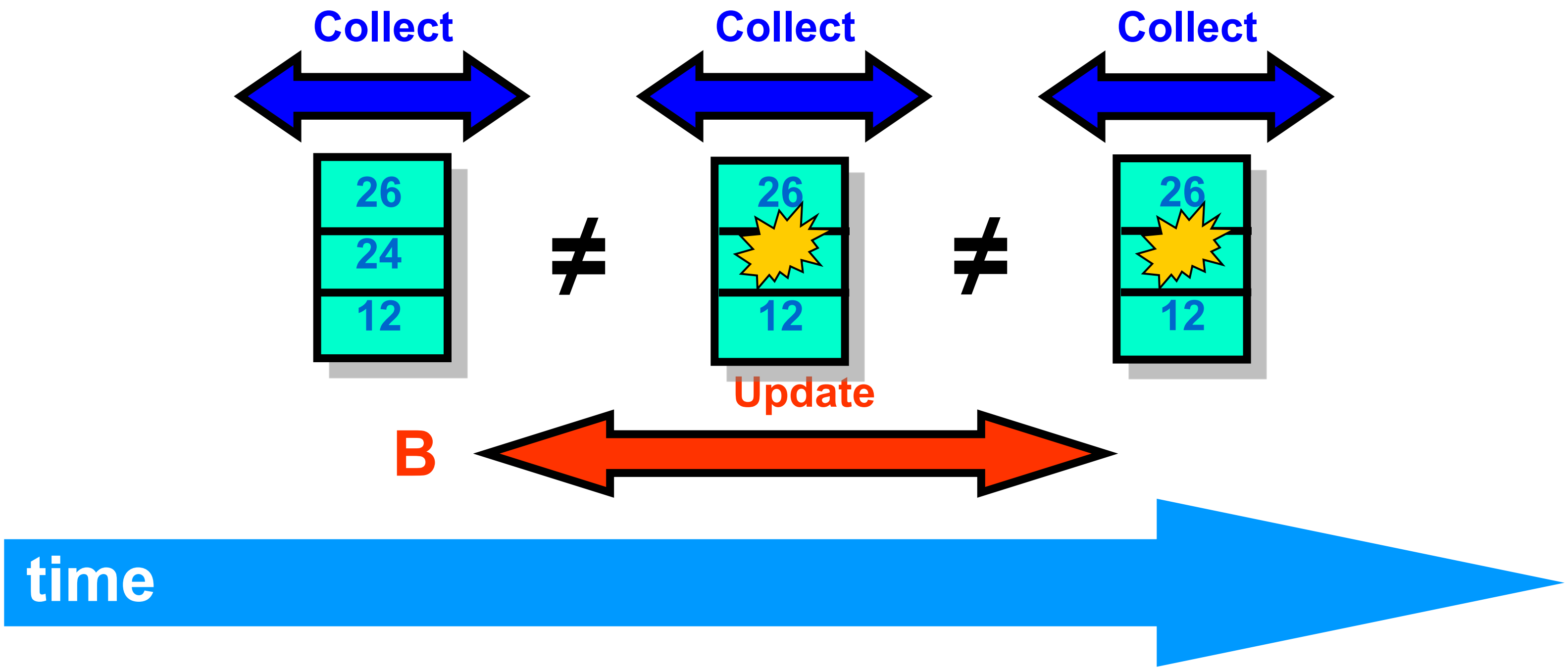
Once is not Enough



Why can't A steal B's scan?

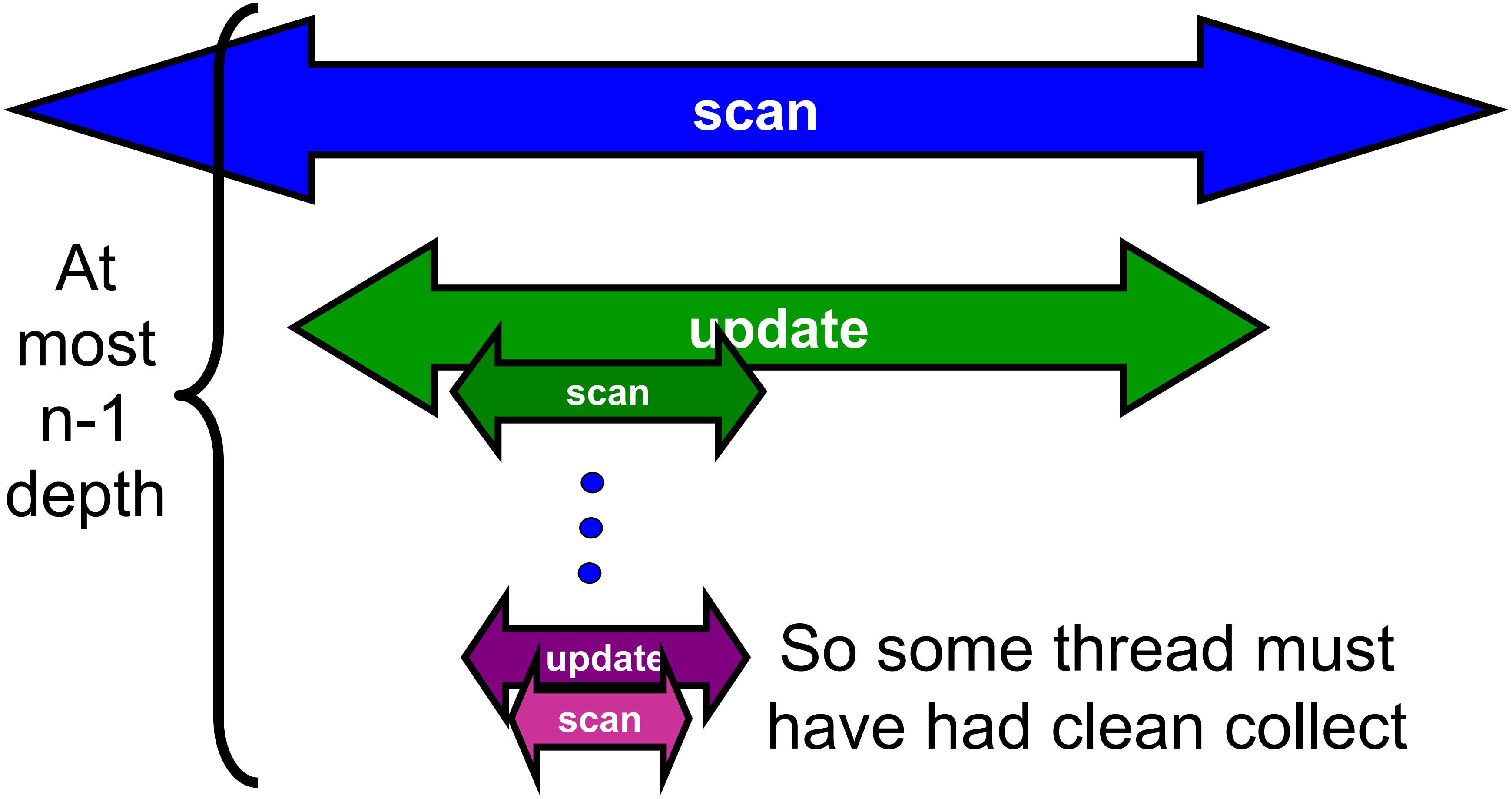
Because another update might have interfered before the scan

Someone Must Move Twice



If we collect *n* times…some thread must move twice (pigeonhole principle)

Scan is Wait-free



So some thread must have had clean collect

Wait-Free Snapshot Label

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}
```

Wait-Free Snapshot Label

```
public class SnapValue {
  public int   label;
  public int   value;
  public int[] snap;
}
```

Counter incremented with each snapshot

Wait-Free Snapshot Label

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}
```

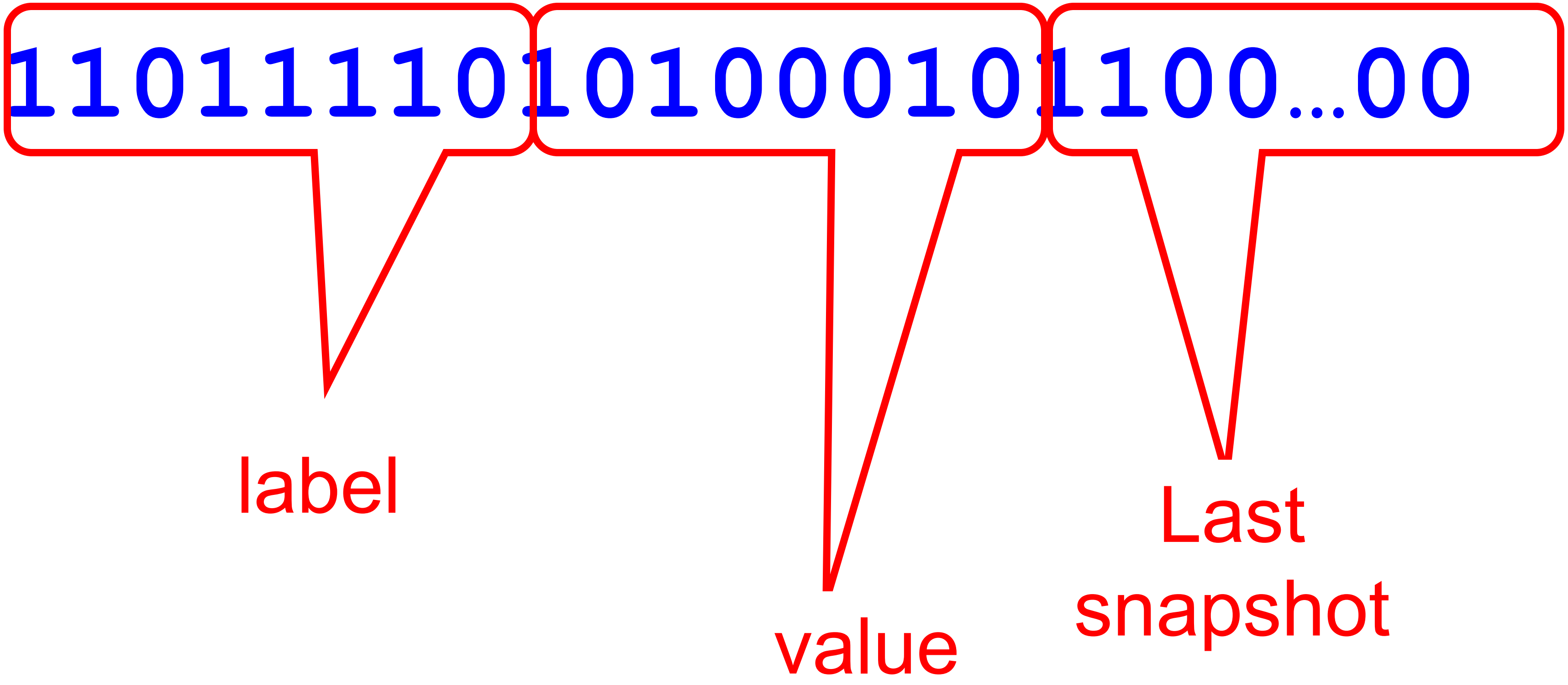
Actual value

Wait-Free Snapshot Label

```
public class SnapValue {
 public int   label;
 public int   value;
 public int[] snap;
}
```

most recent snapshot

Wait-Free Snapshot Label



110111101010001011100...00

label

value

Last snapshot

Wait-free Update

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

Wait-free Scan

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

Take scan

Wait-free Scan

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

Take scan

Label value with scan

Wait-free Scan

```
 public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
      if (oldCopy[j].label != newCopy[j].label) {
       …
  }}
  return getValues(newCopy);
}}}
```

Wait-free Scan

```
public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
     if (oldCopy[j].label != newCopy[j].label) {
      …
  }}
  return getValues(newCopy);
}}}
```

Keep track of who moved

Wait-free Scan

```
 public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
     if (oldCopy[j].label != newCopy[j].label) {
      …
  }}
  return getValues(newCopy);
}}}
```

Repeated double collect

Wait-free Scan

```
 public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
  oldCopy = collect();
  collect: while (true) {
  newCopy = collect();
  for (int j = 0; j < n; j++) {
     if (oldCopy[j].label != newCopy[j].label) {
      …
  }}
  return getValues(newCopy);
}}}
```

If mismatch detected…

Mismatch Detected

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

Mismatch Detected

```
   if (oldCopy[j].label != newCopy[j].label) {
   if (moved[j]) {
    return newCopy[j].snap;
   } else {
    moved[j] = true;
    oldCopy = newCopy;
    continue collect;
  }}}
  return getValues(newCopy);
}}}
```

If thread moved twice, just steal its second snapshot

Mismatch Detected

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

Remember that thread moved

Observations

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

Summary

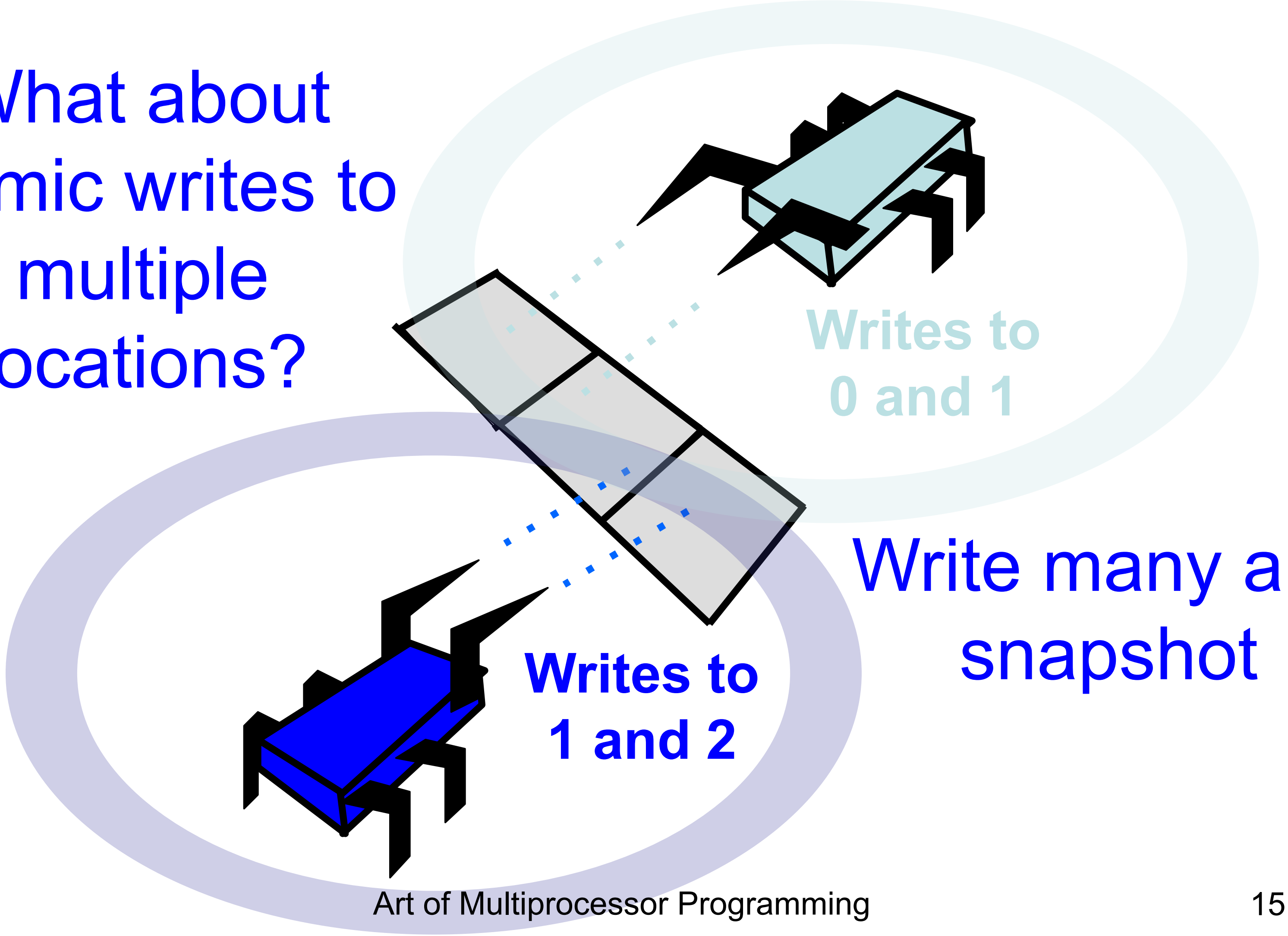
- We saw we could implement MRMW multi valued snapshot objects
- From SRSW binary safe registers (simple flipflops)
- But what is the next step to attempt with read-write registers?

Grand Challenge

- Snapshot means
  - Write any one array element
  - Read multiple array elements

Grand Challenge

What about atomic writes to multiple locations?



Write many and snapshot