DD2451
Parallel and Distributed Computing

FDD3008
Distributed Algorithms

Lecture 3

Atomic Registers

Mads Dam

Autumn/Winter 2011

This Lecture

- Registers: Shared memory locations that can be *read* and *written*
- Basic building blocks of shared memory concurrency
- What different flavours of registers exist?
- What can they do?
- What can they not do?
  - Coming up in later lectures …
- Premise: NO LOCKS, NO MUTUAL EXCLUSION
  - Already know how to achieve mutual exclusion using locks
- Goal: Wait-free implementations
  - All method calls guaranteed to complete

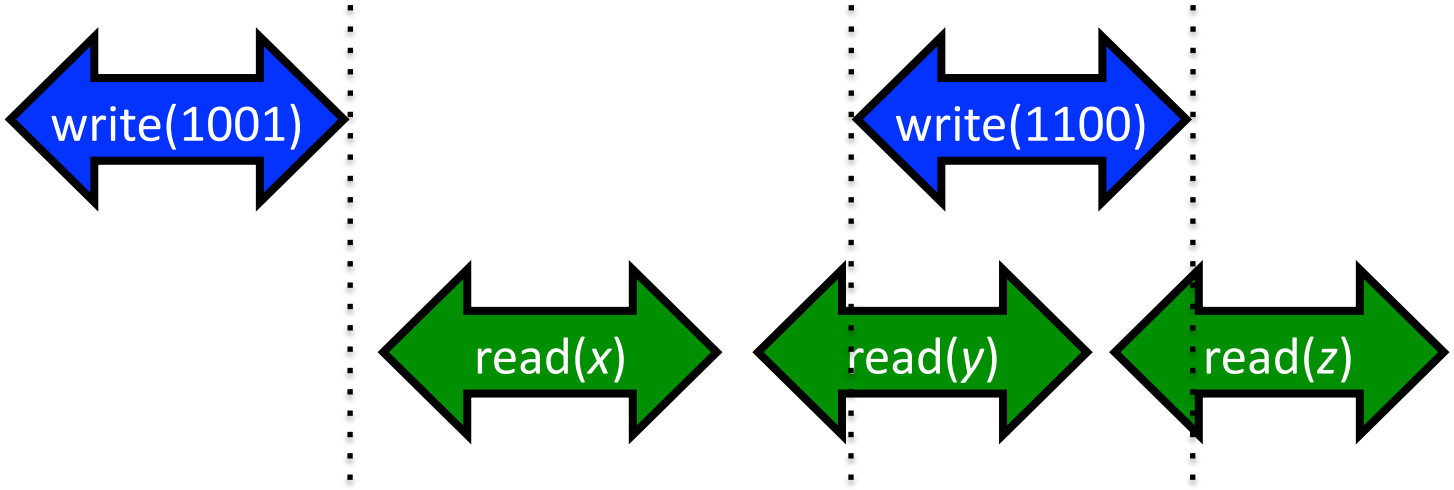
Registers

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

Basic dimensions:

- Number of readers and writers
  - SRSW: Single reader single writer
  - MRSW: Multiple readers single writer
  - MRMW: Multiple readers multiple writers
- Consistency model:
  - Safe, regular, atomic
  - Coming up …

Register Consistency Models



All register types: *x* = 1001 – no overlap

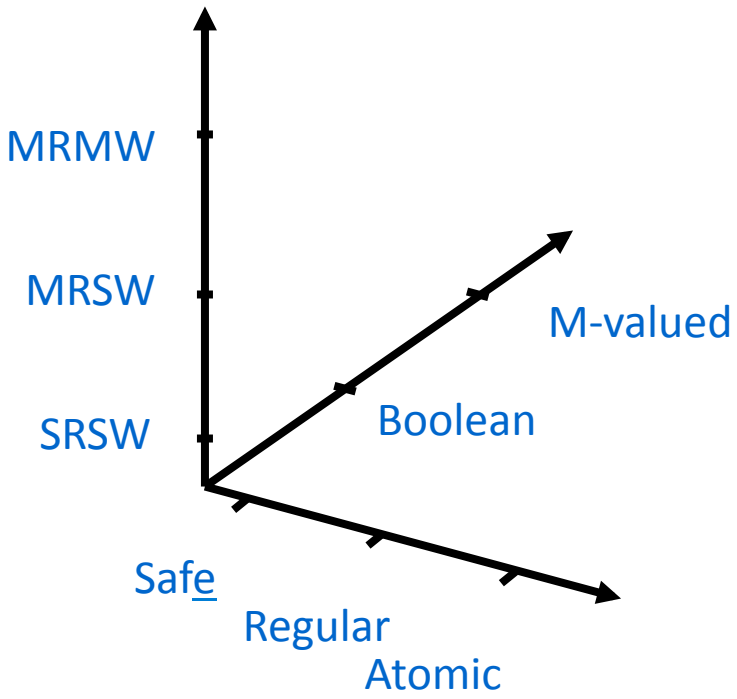
Safe register: *y, z* can be anything (of type T)

Regular register: *y, z* either 1001 or 1100

Atomic register: Regular + *y* = 1100 => *z* = 1100

- Linearizable
- Returns last value written

Register Space

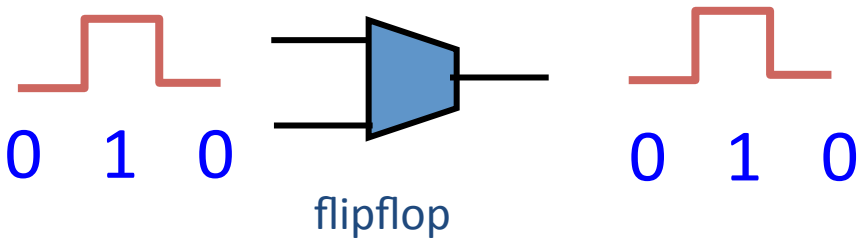


Starting Point

- Safe Boolean SRSW register

Single writer

Single reader



Get correct reading if not during state transition

Results

- Safe SRSW registers sufficient for all other registers
- But there are things that cannot be done with safe SRSW registers – consensus, next lecture
- Road map:

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

- If time: Adaptive store and collect

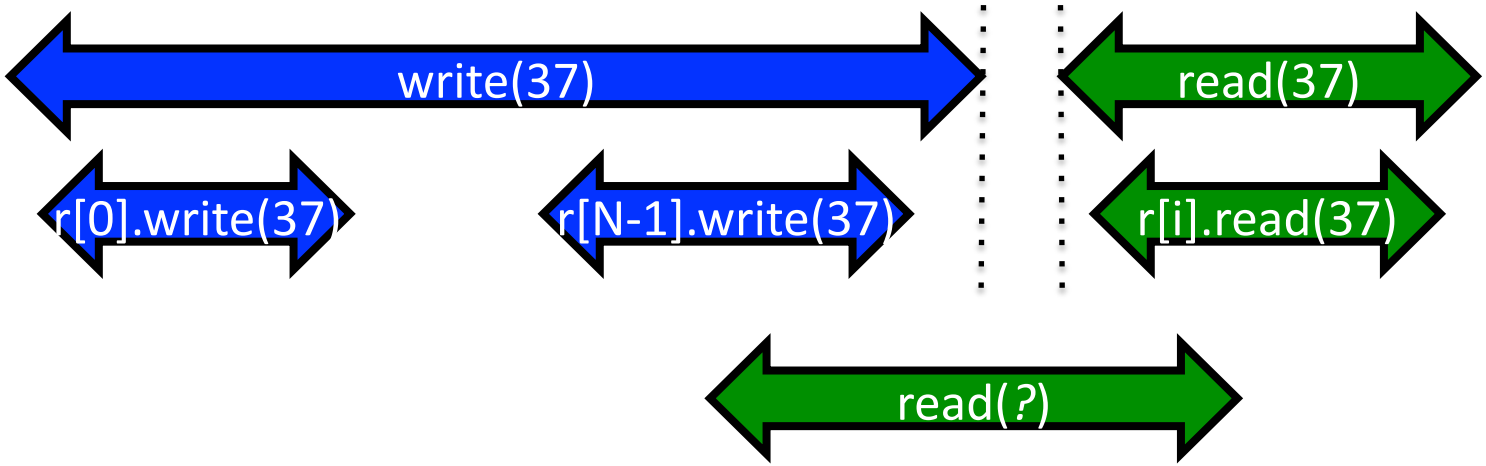
Road Map

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

Safe SRSW -> Safe MRSW

```
public class SafeMRSWRegister<T>
 implements Register<T> {
 private SafeSRSWRegister [] r =
   new SafeSRSWRegister[N];
  public void write(T x) {
   for (int j = 0; j < N; j++)
    r[j].write(x);
  }
  public T read() {
   int i = ThreadID.get();
   return r[i].read();
  }}
```

Safe SRSW -> Safe MRSW




```
public class SafeMRSWRegister<T>
 implements Register<T> {
 private SafeSRSWRegister [] r =
   new SafeSRSWRegister[N];
  public void write(T x) {
   for (int j = 0; j < N; j++)
    r[j].write(x);
  }
  public T read() {
   int i = ThreadID.get();
   return r[i].read(); }}
```

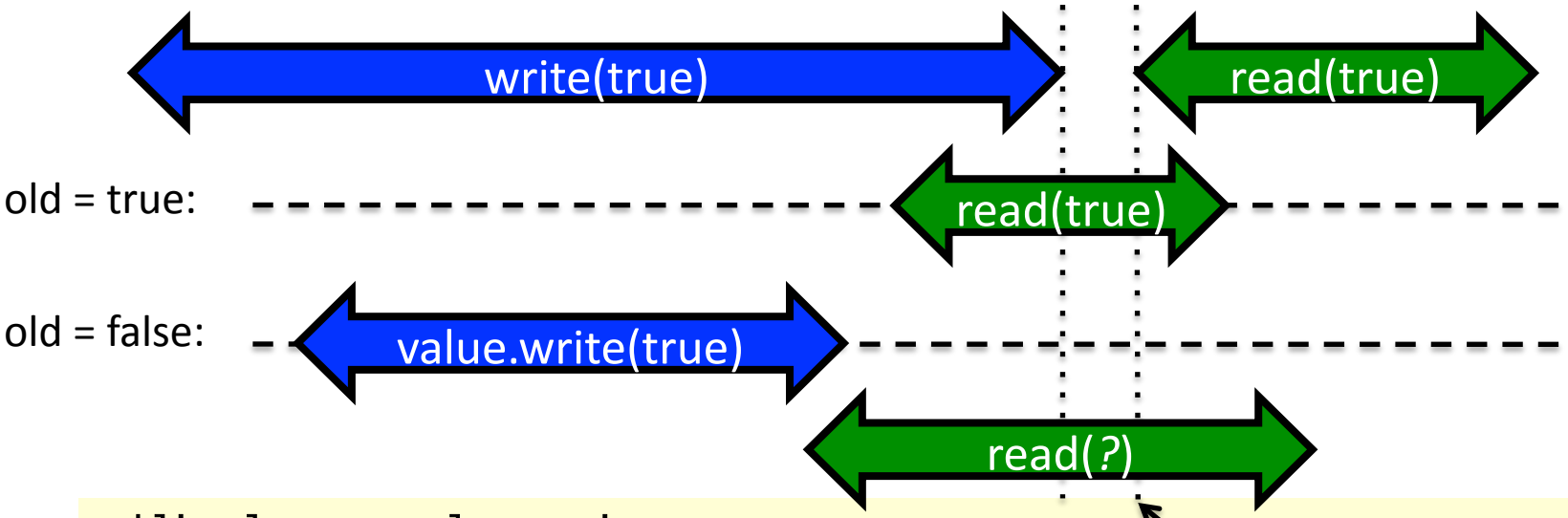
Road Map

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

Safe Boolean MRSW -> Regular 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();
  }}
```

Safe Boolean MRSW -> Regular 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();   }}
```

`value` is safe and can return either true or false. Either will do

Road Map

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

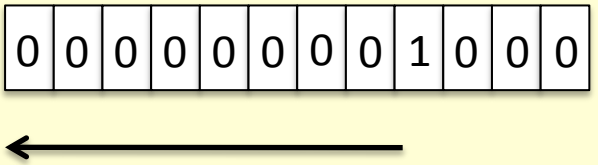
MRSW Regular Boolean -> MRSW Regular

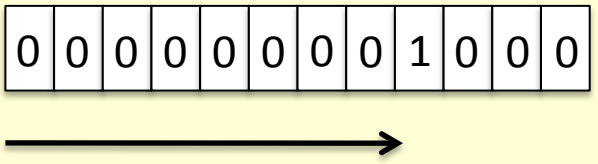
```
public class RegMRSWRegister implements Register{
  RegBoolMRSWRegister[M] bit;

  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;
   }}
```

Unary bit-array

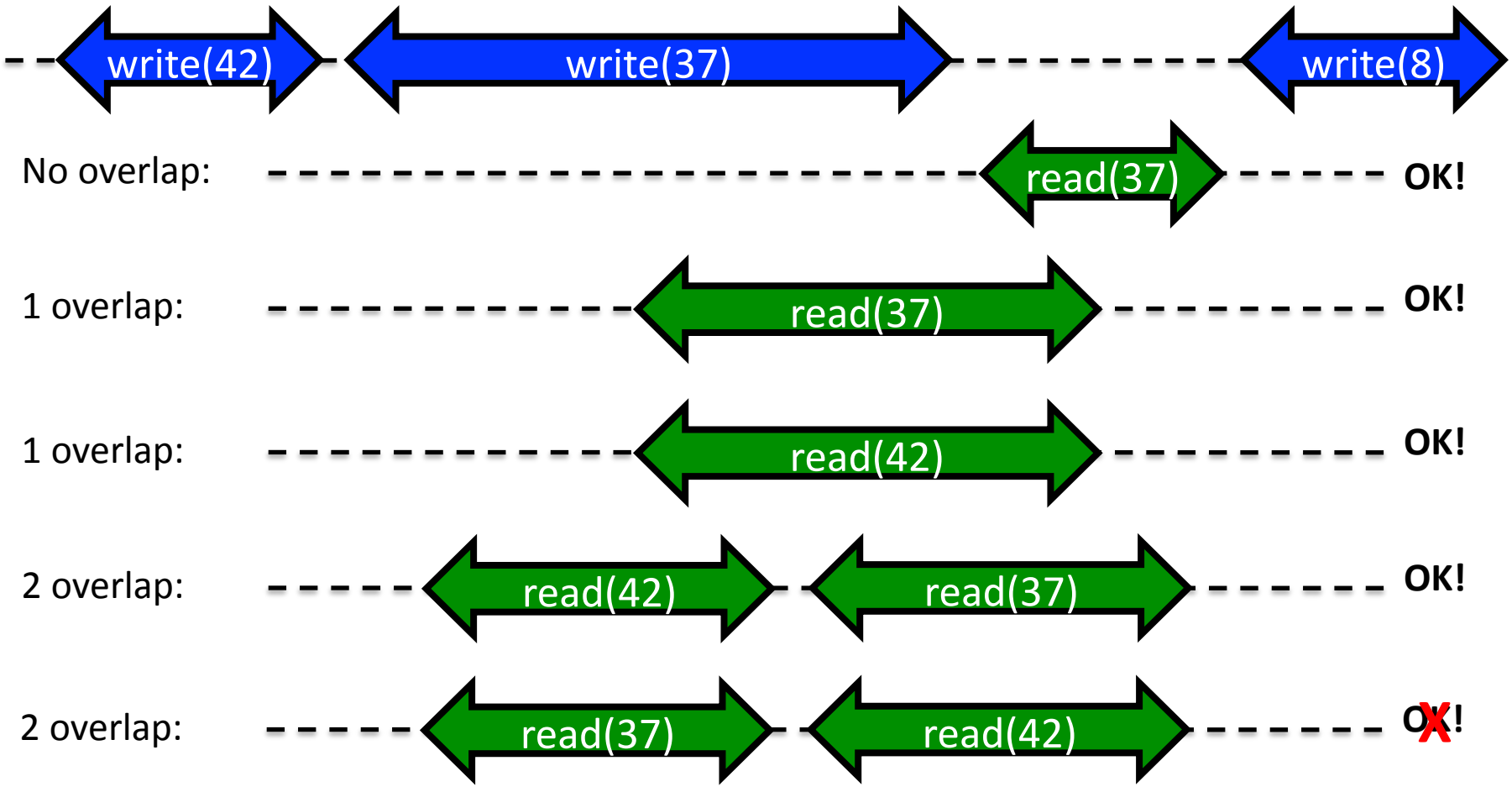




Road Map

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

MRSW Regular -> SRSW Atomic

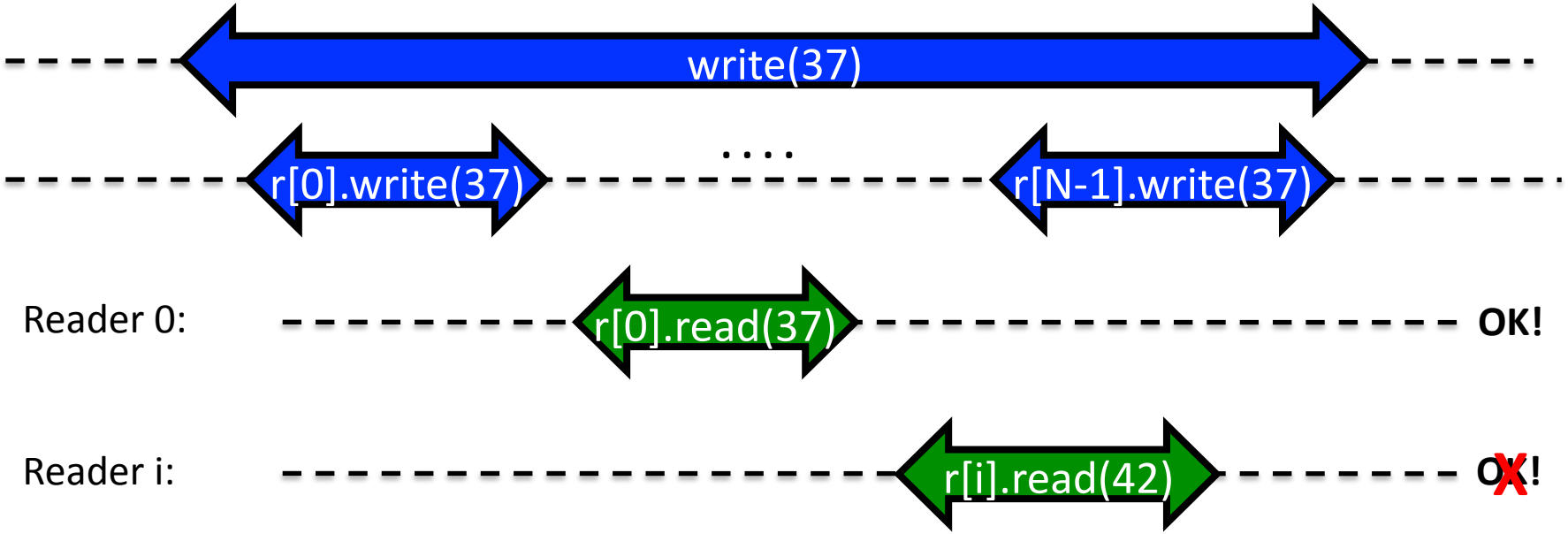


Solution: Time stamping, details in book

Road Map

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

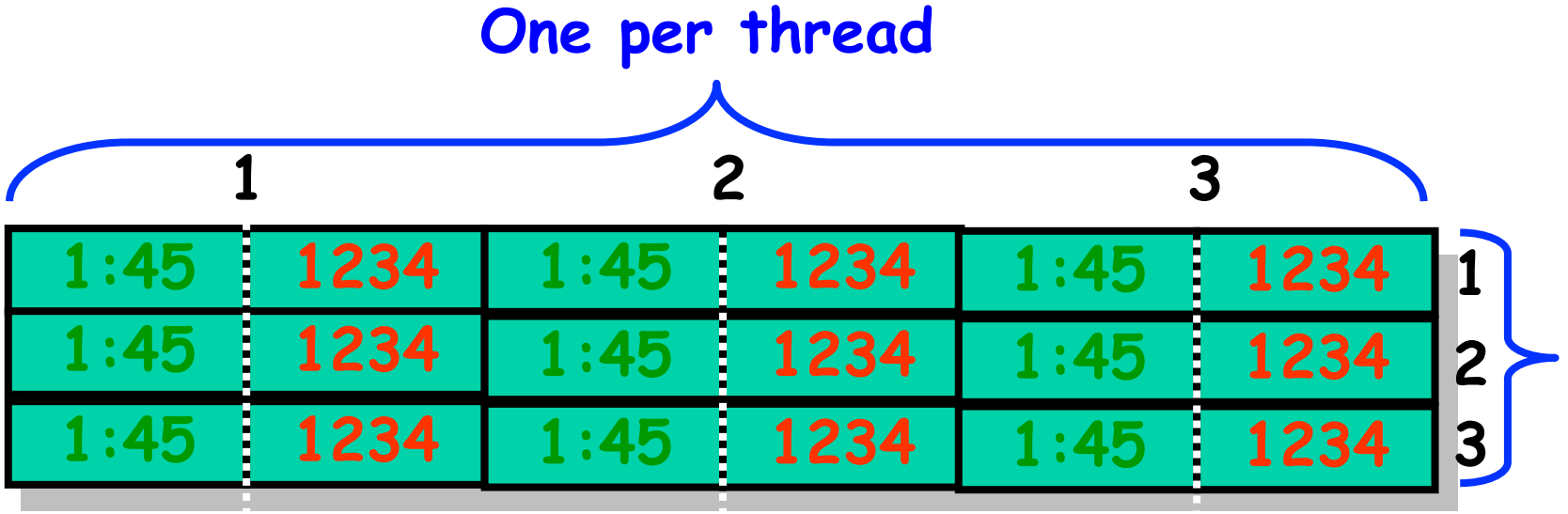
SRSW Atomic -> MRSW Atomic



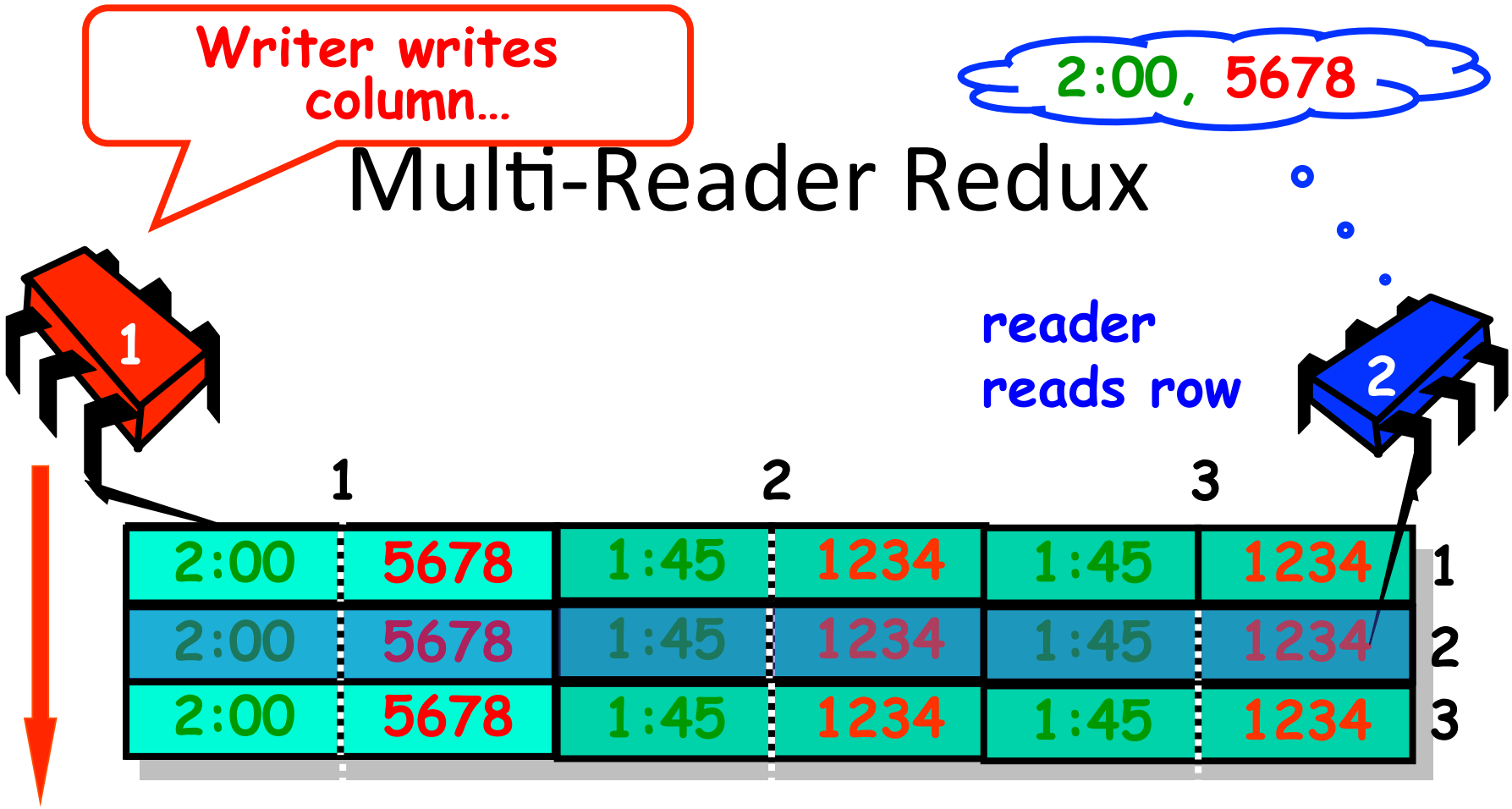
Solution:

- Use time stamps
- Must ensure: If one readers reads new value then all readers do
- Earlier readers tell later readers what they read

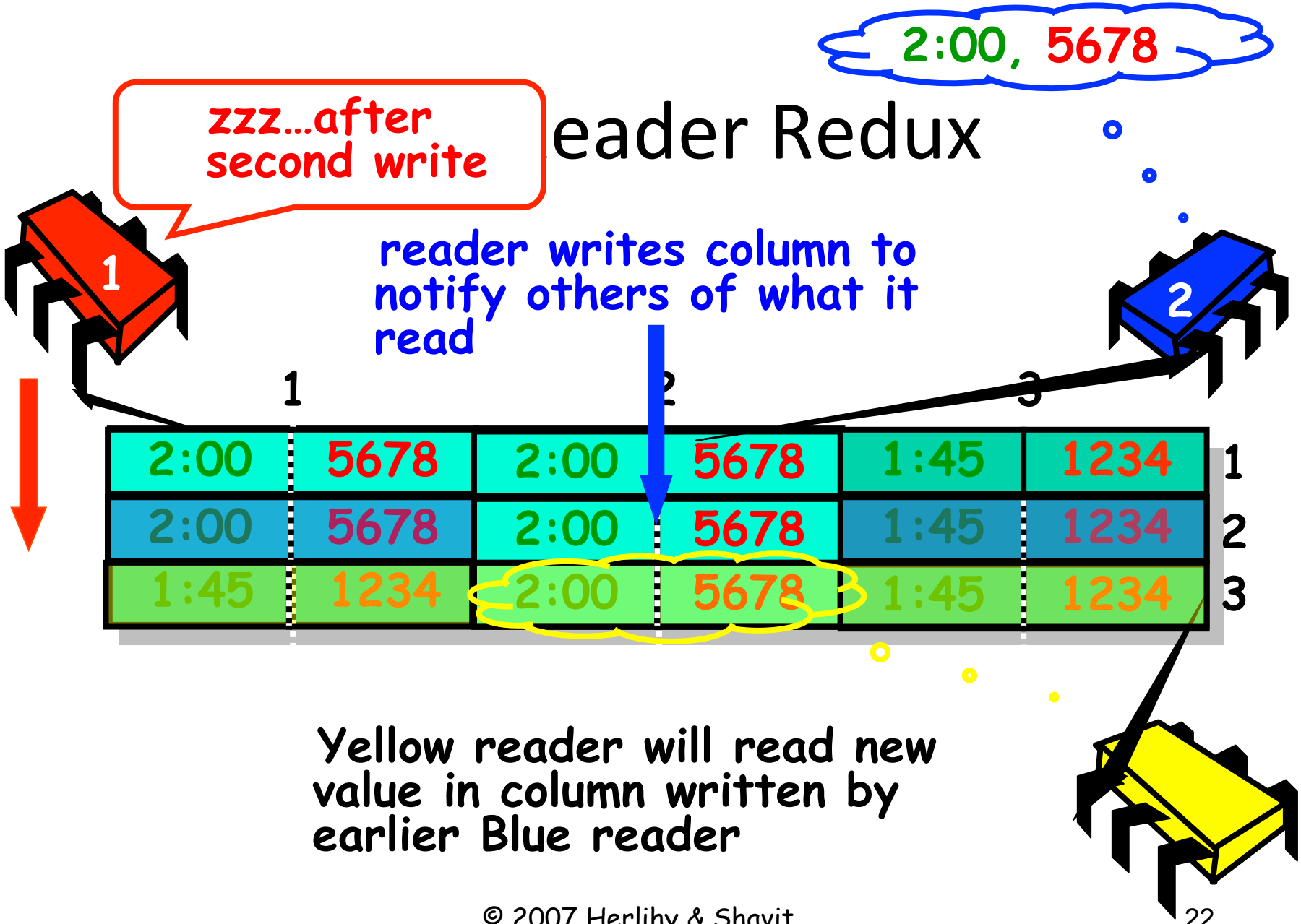
Multi-Reader Redux



Multi-Reader Redux

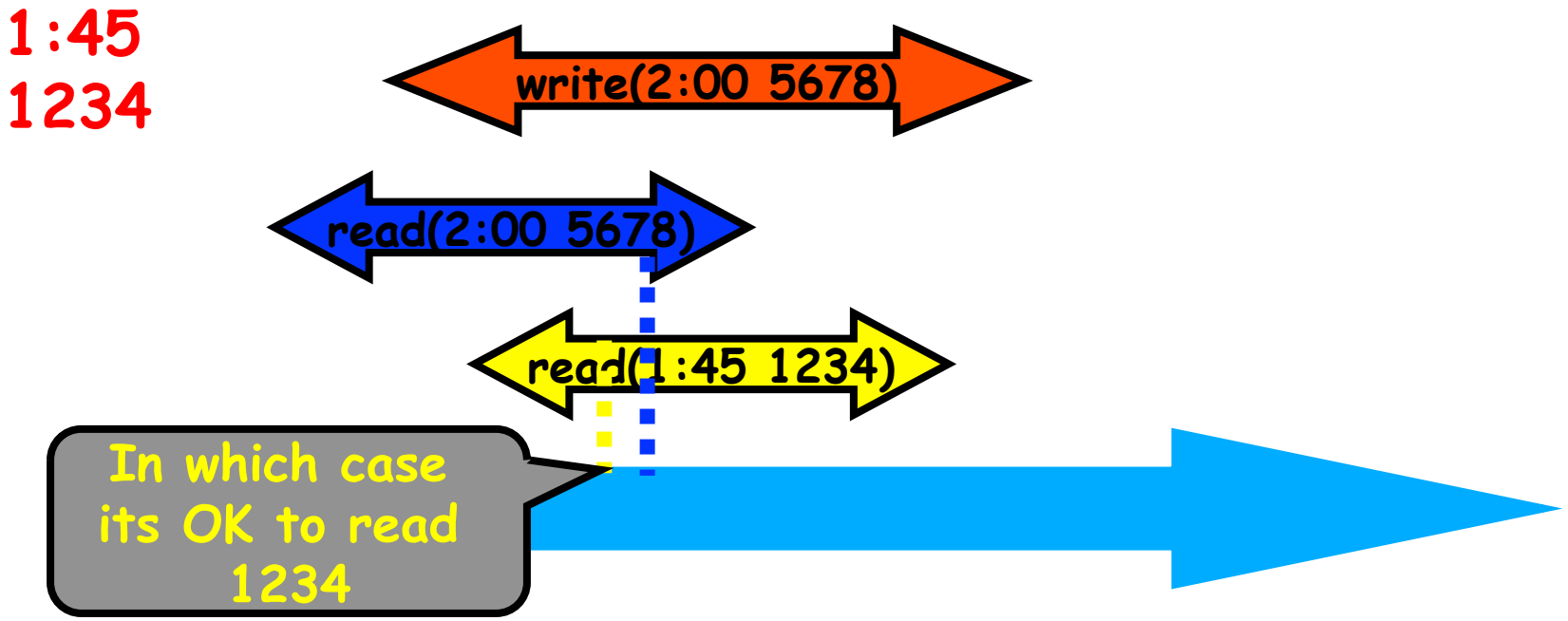


eader Redux

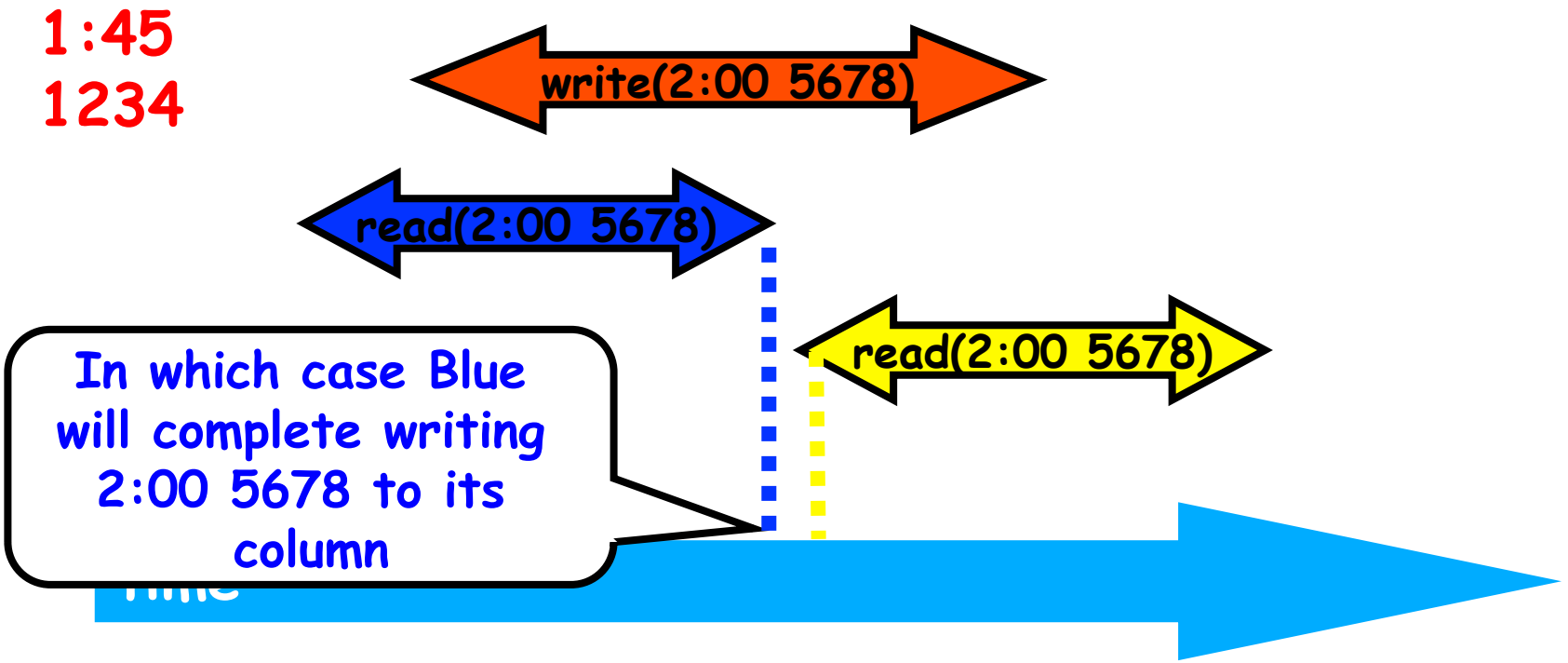


Yellow reader will read new value in column written by earlier Blue reader

Can't Yellow Miss Blue's Update? … Only if Readers Overlap…



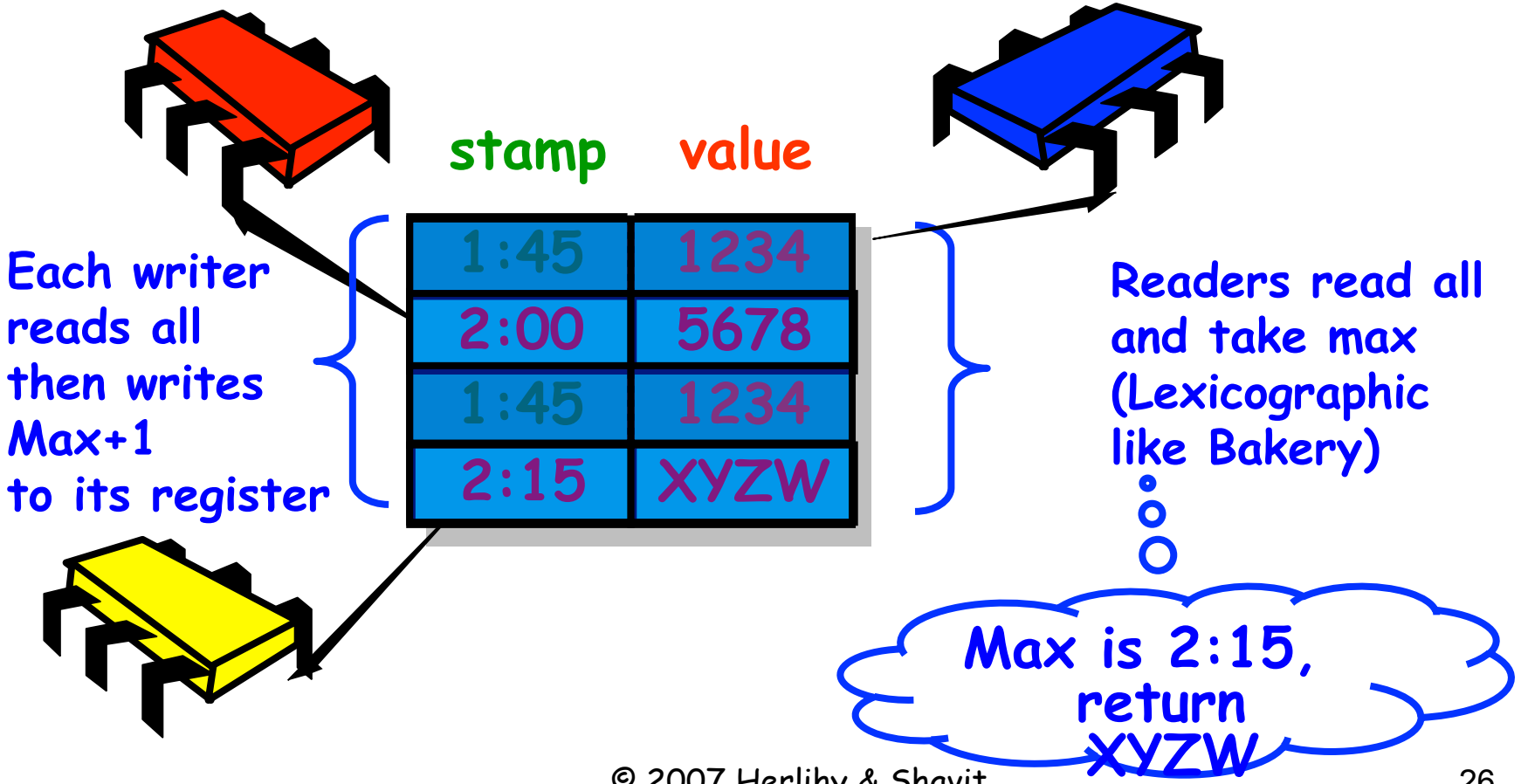
Bad Case Only When Readers Don't Overlap



Road Map

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

Multi-Writer Atomic From Multi-Reader Atomic



Road Map

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

Atomic Snapshot

Given: Array of atomic MRSW registers

Operations:

- Update(*v*): Thread *i* updates the value of register *i* to *v*
- Scan(): Returns an instantaneous snapshot of the register array

Collect:

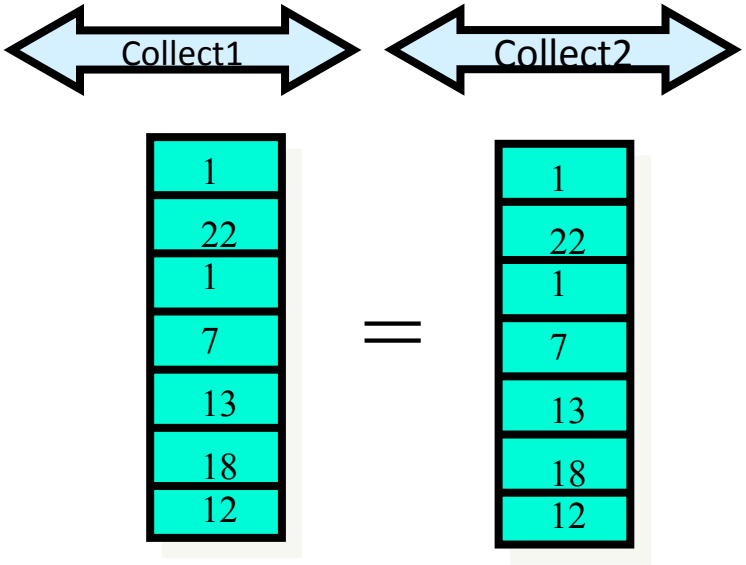
- Read register values one at a time
- May not produce a snapshot

Clean collect:

- Collect during which nothing changed

Simple Snapshot

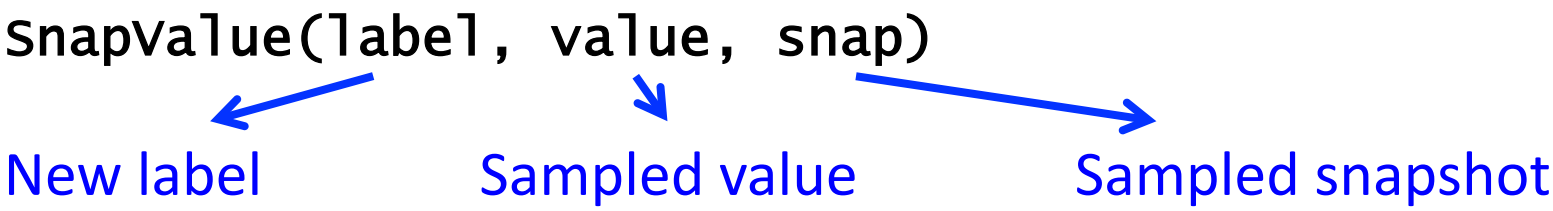
- Put increasing labels on each entry
- Collect twice
- If both agree,
  - We're done
- Otherwise,
  - Try again
- Linearizable
- Update is wait-free
- Scan is not wait-free
- Scan is "obstruction-free"
  - If from some time onwards no-one moves but the scanning thread, then call returns



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 Update




```
public void update(int value) {
 int i = Thread.myIndex();
 int[] snap = this.scan();
        // Complete a scan
 SnapValue oldValue = r[i].read();
        // r is atomic MRSW register array
 SnapValue newValue =
  new SnapValue(oldValue.label+1, value, snap);
        // Increment counter
        // Store value and snap
 r[i].write(newValue);
 }
```

Wait-free Scan

```
public int[] scan() {
 SnapValue[] oldCopy, newCopy;
 boolean[] moved = new boolean[n];
     // Record who moved between collects
 oldCopy = collect();
     // First collect
 collect: while (true) {
 newCopy = collect();
     // Second collect
 for (int j = 0; j < n; j++) {
     // Go through all registers
  if (oldCopy[j].label != newCopy[j].label) {
     …  // (In case of mismatch - next slide …)
 }}
 return getValues(newCopy);   }}}
```

Mismatch Detected

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

(2)

Lemma 1

If collect is clean

- i.e. oldCopy[j].label = newCopy[j].label for all j: 0 <= j < n

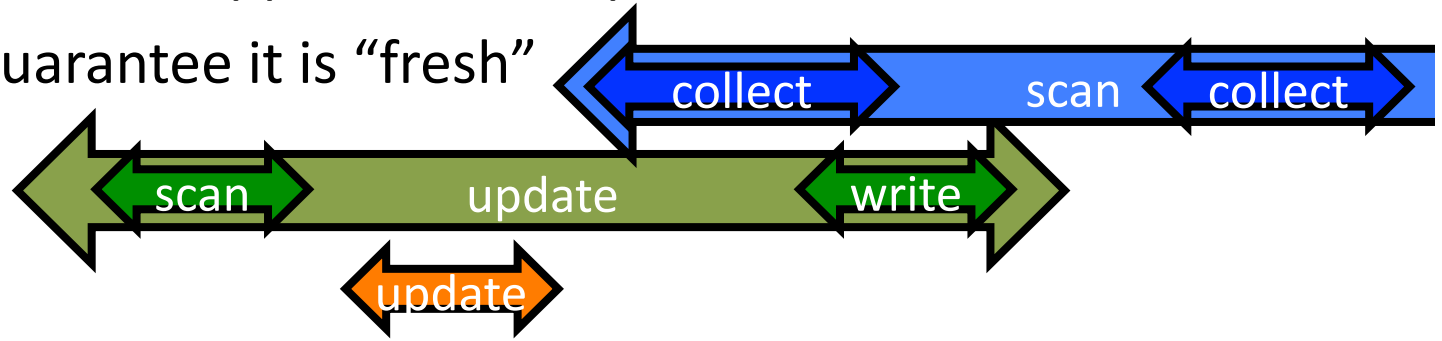
then newCopy is a valid snapshot

```
public int[] scan() {
  SnapValue[] oldCopy, newCopy;
  boolean[] moved = new boolean[n];
        // Record who moved between collects
  oldCopy = collect();
        // First collect
  collect: while (true) {
  newCopy = collect();
        // Second collect
  for (int j = 0; j < n; j++) {
        // Go through all registers
    if (oldCopy[j].label != newCopy[j].label) {
      … // (In case of mismatch - next slide …)
  }}
  return getValues(newCopy);   }}}
```

Mismatch Detected, II

`oldCopy[j].label != newCopy[j].label && ! moved[j]:`

- Thread j update wrote to `r[j]` between the two collects
- Scan may have happened in the past
- Cannot guarantee it is “fresh”

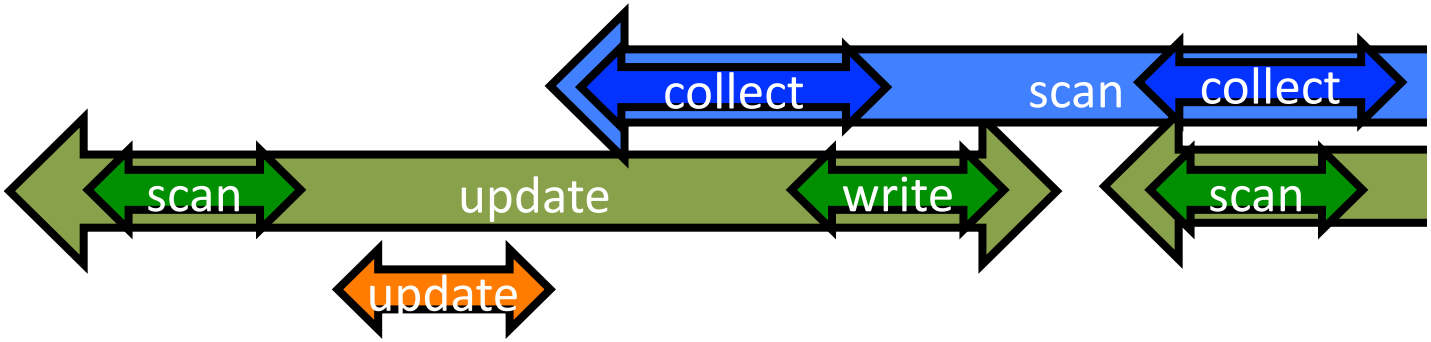



```
➔ if (oldCopy[j].label != newCopy[j].label) {
   if (moved[j]) {
             // second move
    return newCopy[j].snap;
➔  } else {
    moved[j] = true;
             // Record the mismatch and iterate
    oldCopy = newCopy;
    continue collect; }}}
  return getValues(newCopy);    }}}
```

Lemma 2

`oldCopy[j].label != newCopy[j].label && moved[j]:`

- Second time thread j updates r[j] scan must have started after first collect

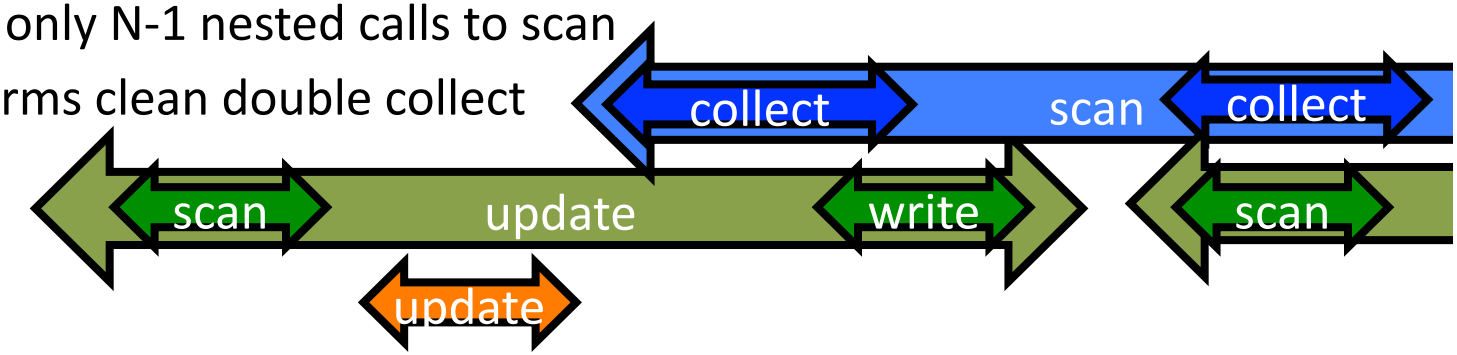



```
➔ if (oldCopy[j].label != newCopy[j].label) {
➔  if (moved[j]) {
            // second move
    return newCopy[j].snap;
   } else {
    moved[j] = true;
            // Record the mismatch and iterate
    oldCopy = newCopy;
    continue collect; }}}
  return getValues(newCopy);    }}}
```

Lemma 3

Scan returns a valid snapshot taken between call and response

- Either blue performs clean double collect
- Else green returns a valid snapshot
- There are only N-1 nested calls to scan
- Last performs clean double collect

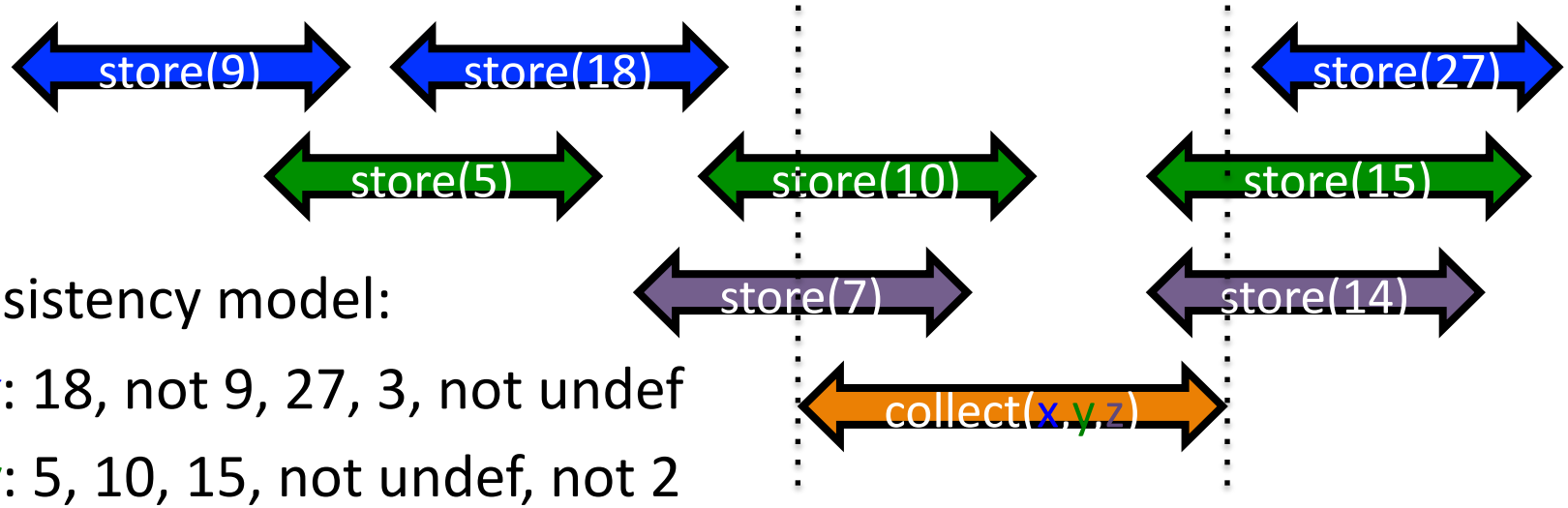



```
if (oldCopy[j].label != newCopy[j].label) {
 if (moved[j]) {
           // second move
  return newCopy[j].snap;
 } else {
  moved[j] = true;
           // Record the mismatch and iterate
  oldCopy = newCopy;
  continue collect; }}}
return getValues(newCopy);    }}}
```

Detour: Store and Collect

```
public interface Register<T> {
  public T collect();
  public void store(T v);
}
```

Want: Regular SRMW register



Consistency model:

- *x*: 18, not 9, 27, 3, not undef
- *y*: 5, 10, 15, not undef, not 2
- *z*: 7, 14, undef, not 16

Trivial Solution

```
algorithm NonadaptiveStoreAndCollect<T> {
  private RegularSRSWRegister [] R =
    new RegularSRSWRegister[n]
  RegularSRSWRegister initially undef
  public void store(T v) {
    R[i] = v }
  public T[] collect() {
    for (int i = 0; i < n; i++) {
      result[i] = R[i]}
    return result ; } }
```

Java pseudocode is going to become more pseudo than Java from now on!

Trivially correct

Step complexity = number of accesses to shared memory

registers = number of threads = *n*

Is this optimal?

Adaptive Store and Collect

Need only sample registers that have been written to!

Example:

- p_i and p_j have been written, all other $R[m]$ are undef
- Then collect need only visit two registers, not n

Assume at time t, $k <= n$ threads have finished or started an operation

Operation (method) m is *adaptive* if step complexity of m is

- Dependent on k
- Independent of n

Splitters

```
algorithm {LEFT,RIGHT,STOP} Splitter() {
  private int X = undef
  private bool Y = false
  enter(int i) {
    X = i ;
    if Y {return RIGHT} {
      Y = true ;
      if X = i {return STOP}{return LEFT}
    }}
```

Process p_i enters splitter *S* by calling enter(*i*)

Recognize this?

Splitters

Lemma: If *k* processes enter a splitter

- At most one process returns STOP
- At most *k* - 1 processes returns LEFT (do. RIGHT)

Proof:

- First process to test Y does not return RIGHT
- Last process to assign X does not return LEFT
- Two processes to return STOP – one must have rewritten others assignment to X

```
algorithm {LEFT,RIGHT,STOP} splitter() {
   private int X = undef
   private bool Y = false
    enter(int i) {
     X = i ;
     if Y {return RIGHT} {
       Y = true ;
       if X = i {return STOP}{return LEFT}
    }}
```

Splitter Tree

Arrange $2^n - 1$ splitters in binary tree

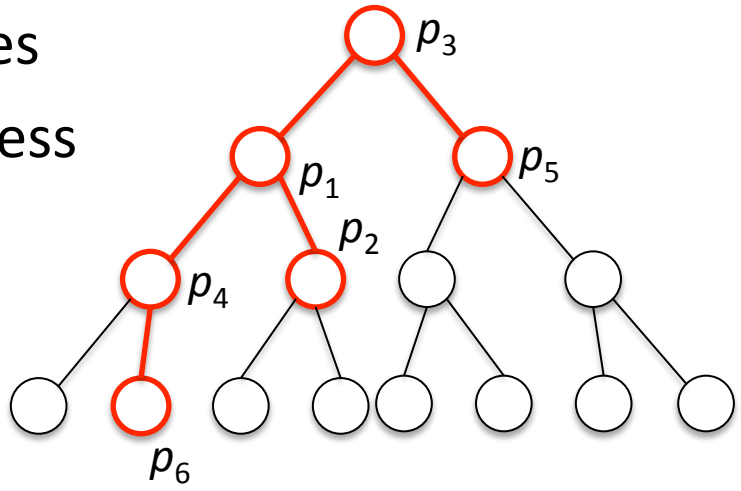
Use splitters to associate process indices to vertices

Marked vertex: Vertex stores a process index

Marking is done when *storing*

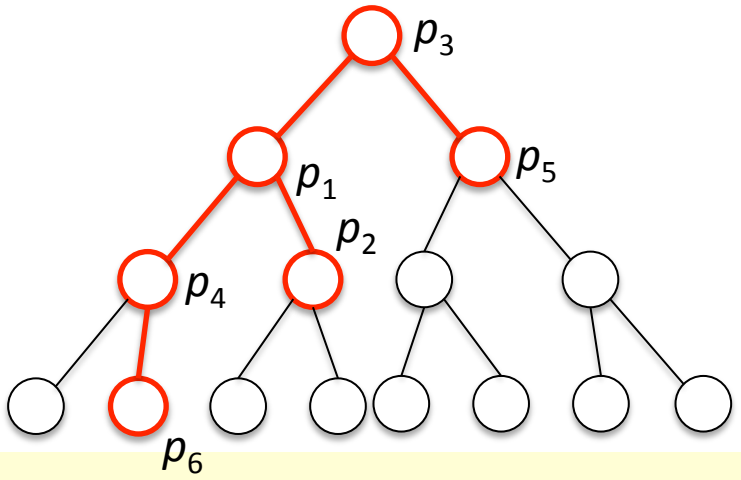
When collecting:

- Traverse only marked vertices
- This is how to get adaptiveness



Store

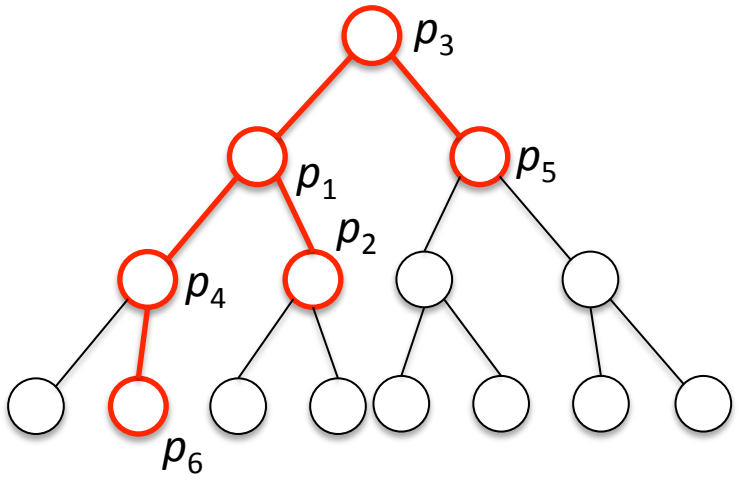
- s splitter
- v binary tree node
- home(s) : {undef,1,…,n}
- marked(s) : bool




```
algorithm store(val) {
   R[thisThread] = val ;
   if first store operation by thisThread {
     v = root node of binary tree ;
     a = splitter(v).enter(thisThread) ;
     marked(splitter(v)) = true
     while a != STOP {
       if a = LEFT {v = leftchild(v)}{v=rightchild(v)} ;
        a = splitter(v).enter(thisThread) ;
        marked(splitter(v)) = true
     } ;
    home(s) = thisThread  }  }
```

Collect

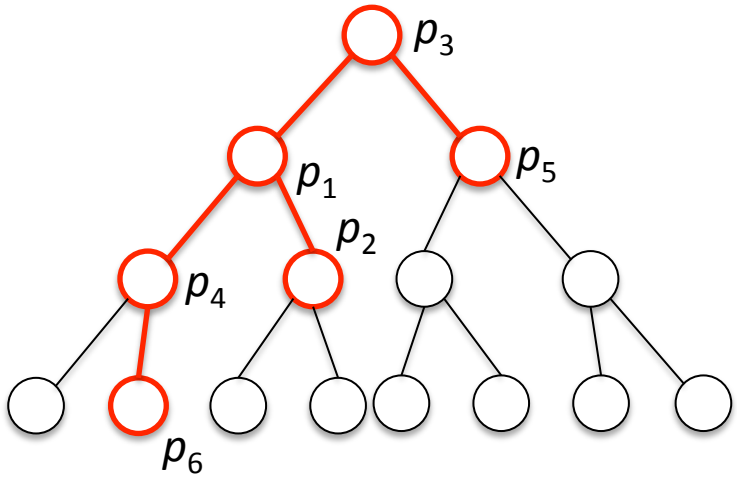
- s splitter
- v binary tree node
- home(s) : {undef,1,…,n}
- marked(s) : bool




```
algorithm collect() {
   for all marked splitters s do {
     if  home(s) != undef {
       i = home(s) ;
       result[i] = R[i]
     }
   } ;
   return result }
```

Store, Analysis

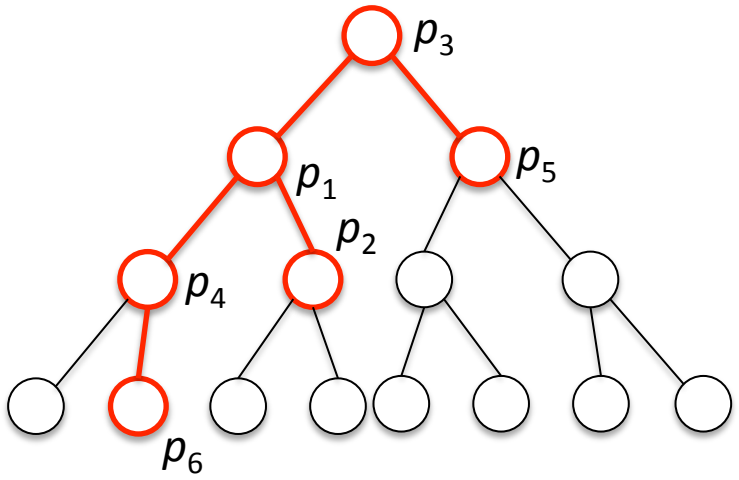
- At most one thread can stop at a splitter
- Every thread stops at max depth $k - 1$
- At most $k - i$ threads reach depth i
- Only 1 thread left at max depth
- That process gets STOP from its splitter
- Step complexity is k




```
algorithm store(val) {
   R[thisThread] = val ;
   if first store operation by thisThread {
     v = root node of binary tree ;
     a = splitter(v).enter(thisThread) ;
     marked(splitter(v)) = true
     while a != STOP {
      if a = LEFT {v = leftchild(v)}{v=rightchild(v)} ;
       a = splitter(v).enter(thisThread) ;
       marked(splitter(v)) = true
     } ;
    home(s) = thisThread  }  }
```

Collect,Analysis

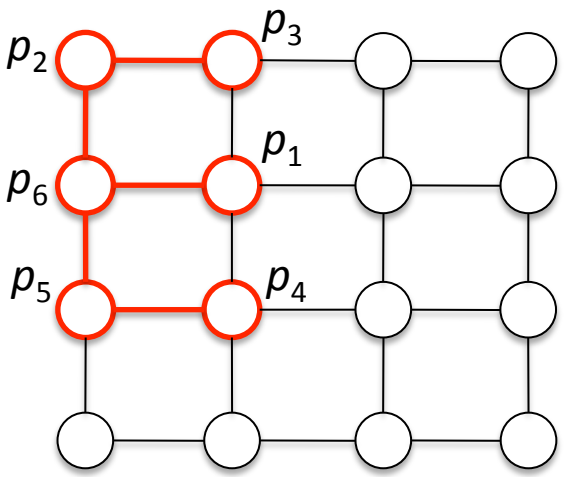
- Marked nodes form subtree
- Marked subtree has size < 2*k*
- Proof by induction (see Wattenhofer's notes)
- Step complexity is *k*




```
algorithm collect() {
   for all marked splitters s do {
     if  home(s) != undef {
       i = home(s) ;
       result[i] = R[i]
     }
   } ;
   return result }
```

Splitter Matrix

- Binary tree step complexity is optimal
- But space complexity is exponential
- Idea: Organize splitter tree in n x n matrix
- Space bound is n^2
- Step complexity of store is $O(k)$ and collect is $O(k^2)$



- FYI: Randomizing the splitter reduces space to O(log n) with high probability

  (whp: with probability approaching 1 exponentially fast)

Thus Far . . .

Have shown SRSW -> atomic snapshot

Constructions are not very practical:

- Unbounded counters (see lecture 1)
- Unary numbers

More practical constructions exist

Also shown: Cool additional construction ;-)

Not the end of the road

- Can do a lot with registers
- But not all!
- Next lecture: Consensus