

Distributed Systems

Spring Semester 2020

Lecture 13: TreadMarks (Lazy Release Consistency)

John Liagouris
liagos@bu.edu

Lazy Release Consistency (LRC)

M0:

```
a.lock()
  for i:=0;i<100;i++ {
    x=foo(i)
  }
a.unlock()
```

M1:

```
b.lock()
  for (i:=0;i<100;i++) {
    y=bar(i)
  }
b.unlock()
a.lock()
  print x,y
a.unlock()
```

Observation: Nobody should expect to see updates until they acquire a lock. So wait and only send updates of Release only to next Locker (on next lock Acquisition)


Lazy Release Consistency (LRC)

M0:

```
a.lock()
  for i:=0;i<100;i++ {
    x=foo(i)
  }
a.unlock()
```

M1:

```
b.lock()
  for (i:=0;i<100;i++) {
    y=bar(i)
  }
b.unlock()
```



```
a.lock()
  print x,y
a.unlock()
```

Do NOTHING on Release — on acquisition go get diffs from last machine to do Release

LRC Example

M0: a.lck() x=1 a.ulck()

M1: b.lck() y=1 b.ulck()

M2: a.lck() print x a.ulck()

LRC Example

M0: a.lck() x=1 a.ulck()

M1: b.lck() y=1 b.ulck()

M2: a.lck() print x a.ulck()



Nothing happens on M0 and M1 on their unlocks but when M2 acquires lock a then it finds the last owner and requests the diffs for what it has changed

Lock Server or Broadcast to acquire lock and find last owner

LRC Example

M0: a.lck() x=1 a.ulck()

M1: b.lck() y=1 b.ulck()

M2: a.lck() print x a.ulck()



M0 sends its changes M2 applies them and then the lock acquisition is complete and the print will see x=1

LRC Example

M0: a.lck() x=1 a.ulck()

M1: b.lck() y=1 b.ulck()

M2: a.lck() print x a.ulck()



The big advantage of LRC we only send the diff to the one machine that needed to see it vs RC

But consider...

```
M0:  x := 7  // no lock
      a.Lock()
      y = &x
      a.Unlock()
```

```

MI:
    a.Lock()
    b.Lock()
    z = y
    b.Unlock()
    a.Unlock()

```

M2: `b.Lock(); print *z; b.Unlock()`

But consider...

M0: `x := 7 // no lock`
`a.Lock()`
`y = &x`
`a.Unlock()`

M1: `a.Lock()`
`b.Lock()`
`z = y`
`b.Unlock()`
`a.Unlock()`

M2: `b.Lock(); print *z; b.Unlock()`

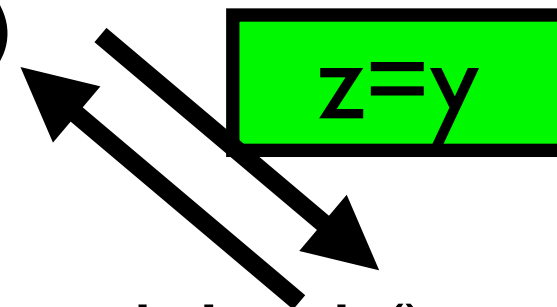


M2's lock of b asks M1 for modifications

But consider...

M0: `x := 7 // no lock`
`a.Lock()`
`y = &x`
`a.Unlock()`

M1: `a.Lock()`
`b.Lock()`
`z = y`
`b.Unlock()`
`a.Unlock()`



M2: `b.Lock(); print *z; b.Unlock()`

M2's lock of `b` asks M1 for modifications. M1 sends back write of `z` — but not necessarily `x`

But consider...

M0: `x := 7 // no lock`
`a.Lock()`
`y = &x`
`a.Unlock()`

M1:
`a.Lock()`
`b.Lock()`
`z = y`
`b.Unlock()`
`a.Unlock()`

M2: `b.Lock(); print *z; b.Unlock()`



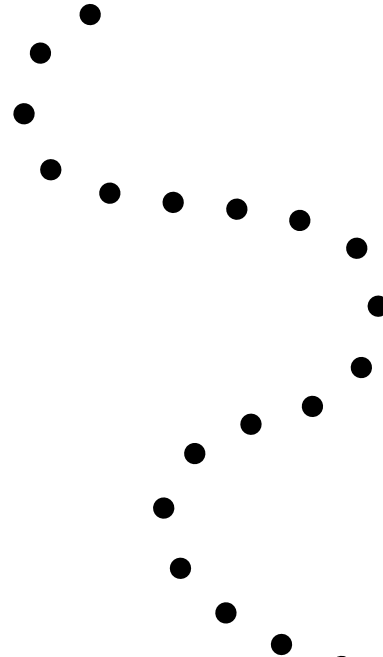
`z=y`

So on M2 `z` is correct but the value at `&x` is garbage!

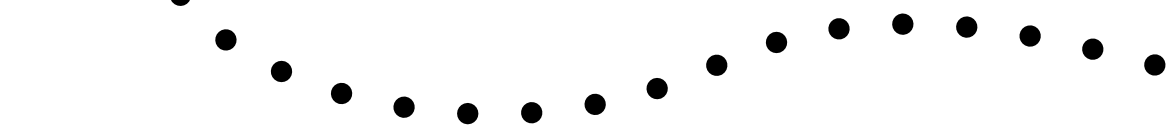
So LRC (as presented
so far) is not really
good enough

TM: Causal Consistency

M0: `x := 7 // no lock`
`a.Lock()`
`y = &x`
`a.Unlock()`

M1: 

`a.Lock()`
`b.Lock()`
`z = y`
`b.Unlock()`
`a.Unlock()`

M2:  `b.Lock(); print *z; b.Unlock()`

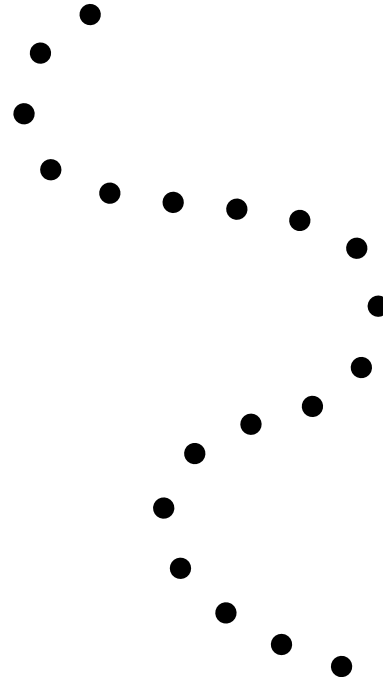
M2 Must see all values that were along the way to computing the value for `z` — that it might depend on

`z=y`

This would avoid the anomaly of seeing `z` and not the things it depends on to be sensible

TM: Causal Consistency

M0: $x := 7$ // no lock
a.Lock()
y = &x
a.Unlock()

M1:  a.Lock()
b.Lock()
z = y
b.Unlock()
a.Unlock()

M2:  b.Lock(); print *z; b.Unlock()

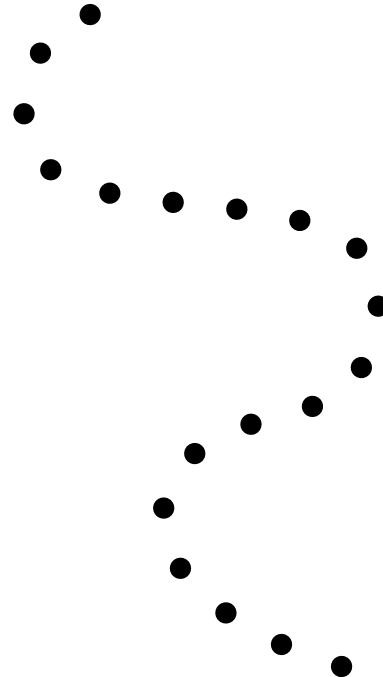
When M1 locks a it picks up all diffs from M0 and keeps them so that it can forward them along

z=y

To Fix this TreadMarks sets up a chain of write diffs so that all diffs from the beginning can make it to M2

TM: Causal Consistency

M0: `x := 7 // no lock`
`a.Lock()`
`y = &x`
`a.Unlock()`

M1: 

`a.Lock()`
`b.Lock()`
`z = y`
`b.Unlock()`
`a.Unlock()`

M2:  `b.Lock(); print *z; b.Unlock()`

If I see version V of the computation then I need to see all value that might have influenced V's computation

All the writes that might have contributed to the writes that you see — see a latter write but don't see a write you know preceded it

To track what info has
to be shipped around

TM uses

Intervals & vector time
stamps

Intervals & vector clocks

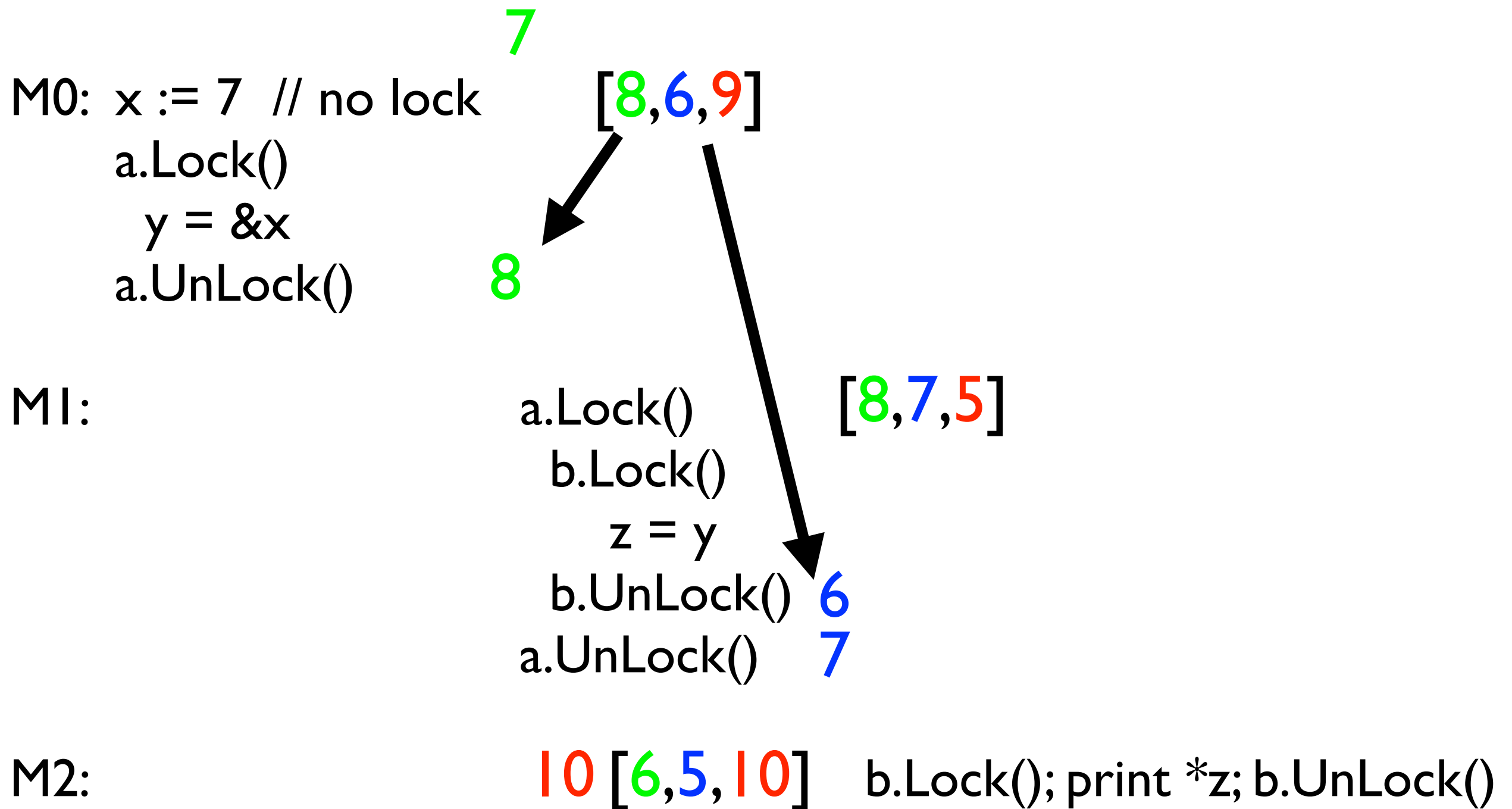
- In ThreadMarks each machine splits up time in to Intervals of Lock Releases — 0,1,2,3...
- Each machine does this independently
- Each machine also maintains a vector of the interval numbers for all other machines when it last did an acquire from them

M0: $x := 7$ // no lock 7
 a.Lock() [8,6,9]
 $y = \&x$
 a.Unlock() 8

M1: a.Lock() [8,7,5]
 b.Lock()
 $z = y$
 b.Unlock() 6
 a.Unlock() 7

M2: 10 [6,5,10] b.Lock(); print *z; b.Unlock()

Each has its own interval count and vector clock



Each has its own interval count and vector clock

M0: $x := 7$ // no lock
 a.Lock()
 $y = \&x$
 a.Unlock()

7

[8, 6, 9]

8

M1:
 a.Lock()
 b.Lock()
 $z = y$
 b.Unlock()
 a.Unlock()

[8, 7, 5]

6

7

M2: 10 [6, 5, 10] b.Lock(); print *z; b.Unlock()

When M2 acquires lock from M1 their vector clocks are compared M1 knows about 2 new M0 updates and 2 M1 updates — necessary transfers are done

7

8

6

