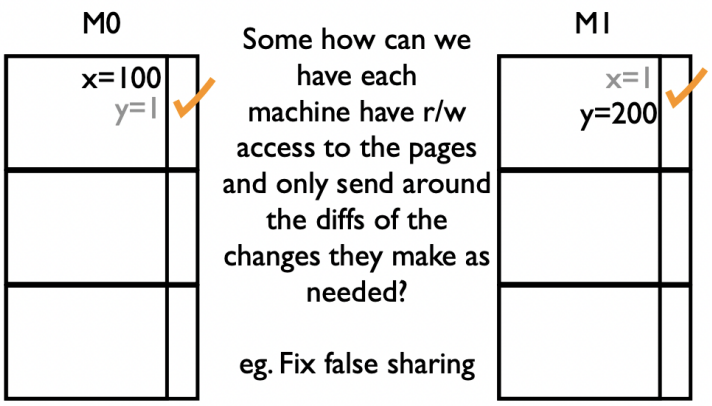
CAS CS 350
Lec 18

Shared Memory System

1. Overview
    a. Treadmarks→ System that provides abstraction regarding address space (although in reality, it is distributed in different space)
    b. Allows programmers to write program as if it is in a single machine → make faster
    c. Similar to MapReduce
    d. Difference
        i. Mapreduce → workers do not share memory and they communicate with each other with only messages
        ii. MapReduce → express the program into map and reduce functions but you do not need to do in treadmarks
    e. Guarantees that process running in different machine does not see stale data
    f. How basic scheme works
        i. Partitions the space
        ii. There is only one copy
        iii. When one tries to write on another machine's space
            1. Makes the space inaccessible to other machines and makes accessible to the machine who tries (guarantees nobody can read stale data)
    g. Change of single byte in a page → transfer the whole page to the machine → can be a problem
2. False Sharing
    a. False sharing: two machines r/w different vars on same page (at least one write)
    b. M1 writes x, M2 writes y
    c. M1 writes x, M2 just reads y
    d. What does general approach do in this situation?
        i. Ownership goes to M1 and writes x
        ii. Ownership goes back to M2 and writes y
        iii. Whole ownership moves
    e. So far an invariant of the system only single write copy
        i. Multiple read copies ok but only one write
        ii. This ensure a strict ordering
    f. Can we do something?
        i. Can we let two machines write to the same page - have write copies?

ii. Write diffs will be useful but not the whole solution - allow us to track local changes

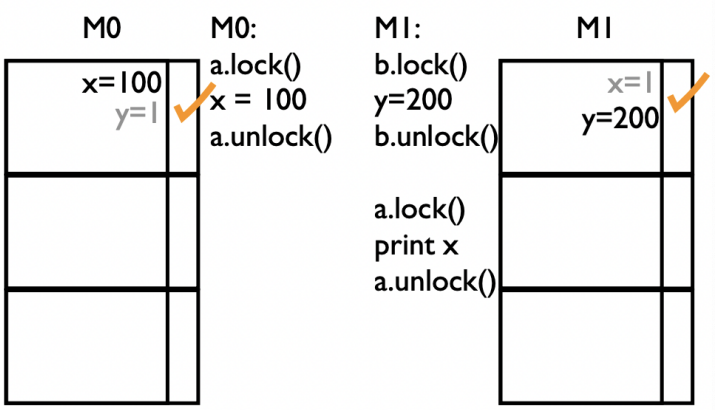
g.



h. But… real code uses lock to coordinate updates - use lock to update variables

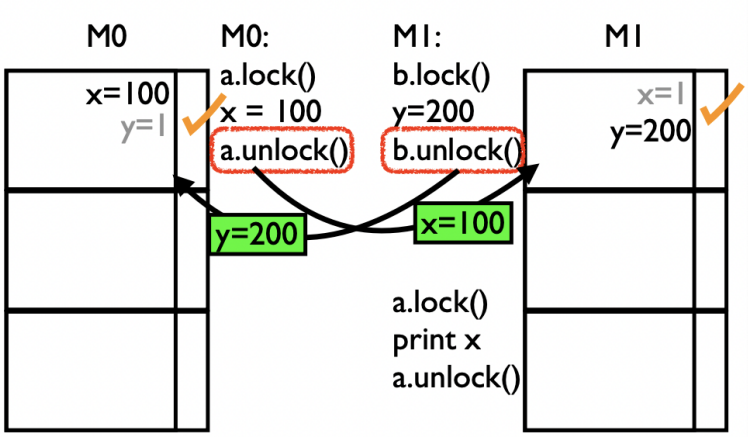
i. X and y belong to the same space

j.

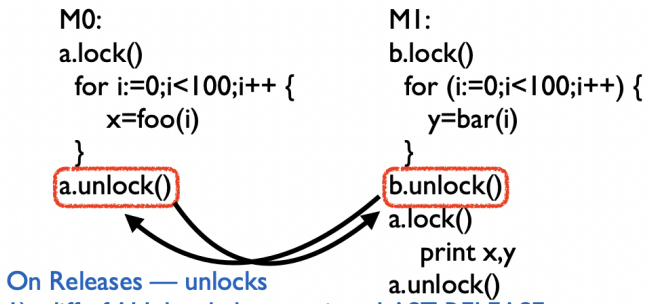


k. Lock Releases: used to track and communicate changes (assuming that locks are used properly), propagate updates only after the lock

l.



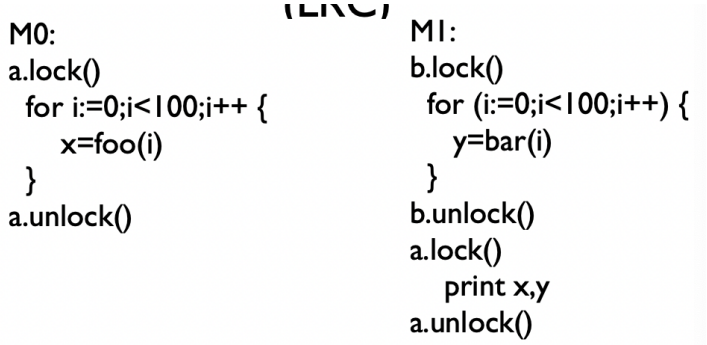
m. Work happens at the time of Lock Release Assume



n.

o. Allows difference process to write on the same pages (more than one writable copies of the same page)
p. Huge performance benefit
q. Is the programming model different than basic DSM?
    i. Yes. In basic DSM you can see all writes - roughly a read will see last write to any memory
    ii. Basic model is stronger and the released consistency is weaker (tend to perform better) because you allow different machines to write on different variables on the same page

3. Lazy release Consistency (LRC)



a.

b. Nobody should expect to see updates until they acquire a lock
c. So wait and only send updates of Release only to next locker (on next lock application)
d. Everytime you update, you lock all other machines
    i. Some of the machines do not use to this page and do not have copies of this page → do not need to broadcast the lock to these machines (only lock the machines that use the variable)
e. Do nothing on release – on acquisition go get diffs from last machine to do release

4. LRC Example

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

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

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

a.

b. First machine updates x, second machine updates y, third machine prints x

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

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

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

c.

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

e. Lock server or broadcast to acquire lock and find last owner

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

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

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

f.

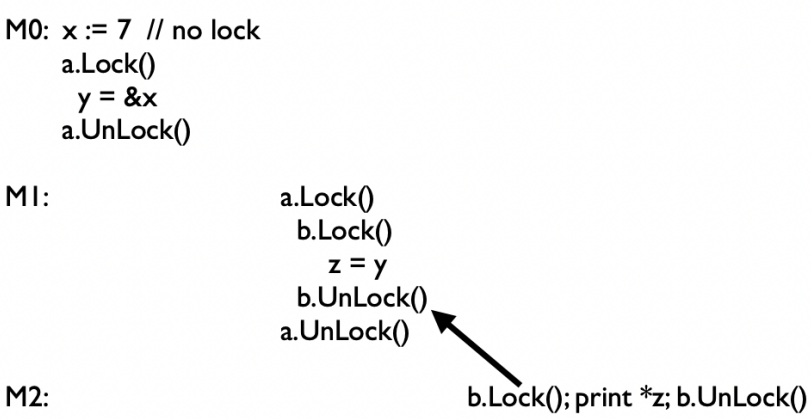
g. M0 sends its changes M2 applies them and then the lock acquisition is complete and the print will see x = 1

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

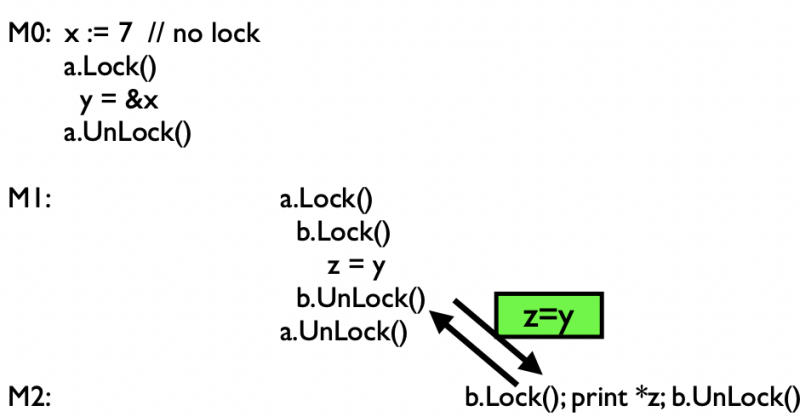
5. But consider…

```
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()
```

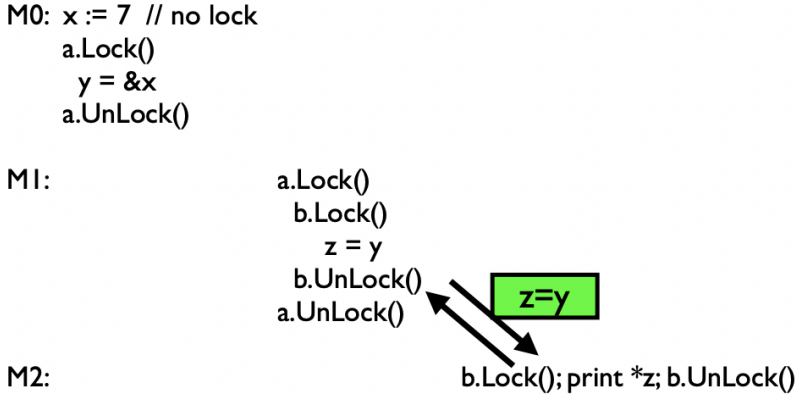
a.



b. M2's lock of b asks M1 for modifications



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

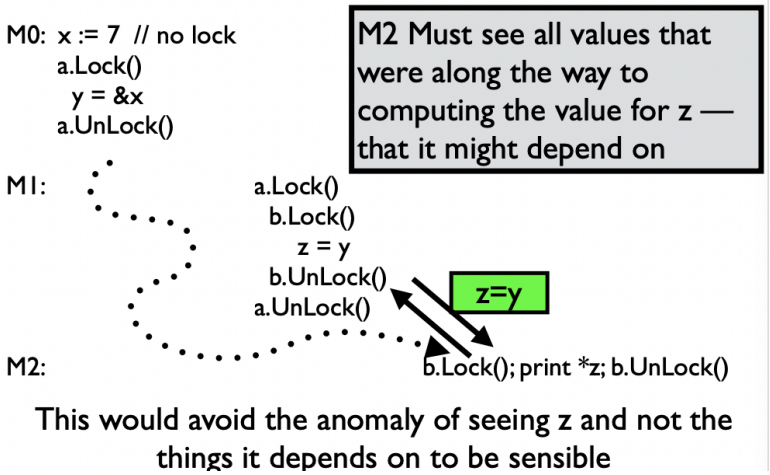


d. So on M2 z is correct but the value at &x is garbage!

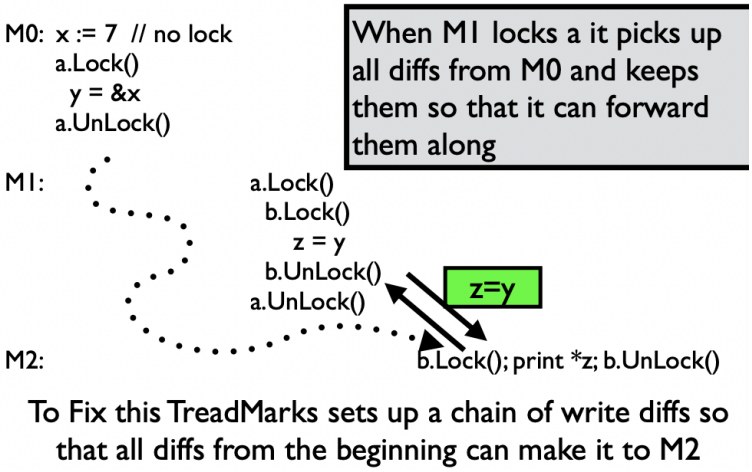
e. So LRC (as presented so far) is not really good enough

6. TM: Causal Consistency

a.



b.



c.