Distributed Systems

Spring Semester 2020

Lecture 13: TreadMarks (Lazy Release Consistency)

John Liagouris
liagos@bu.edu

Today's Paper

How do we take a bunch of workstations on a LAN and make them into a system that can be easily programmed

- their approach — Distributed Shared Memory (DSM)
  - Take familiar parallel programming model — threads — and move it over to a network of computers
  - Our focus intro DSM and explore Consistency — Two Key Ideas from the paper
    - Lazy-release consistency (introduced by the paper)
    - Version vectors (a common implementation technique in distributed systems)

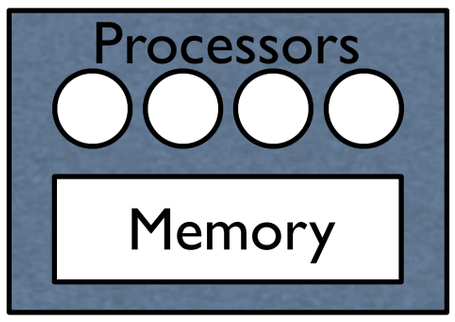
The Big Idea

- Computational models for distributed computing
- How to implement a framework that hides details of all the machines?
- How to keep programming model “simple”?
- How to achieved the performance goals?

Today’s paper trademarks. Map reduce falls in to this category and Spark which we will look at later

Consider

System Interface



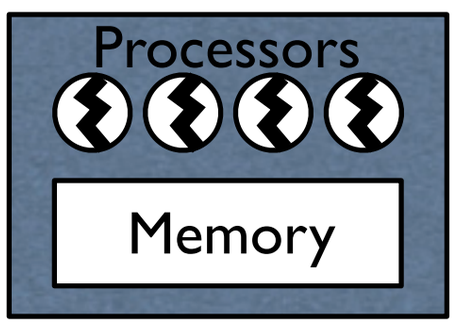
“Unit” of HW

Consider

```
for i:=0; i<4; i++ { go work(i,results) }
```

System Interface

Know how to write parallel code… fork off threads that work together by reading and updating variables — data structures

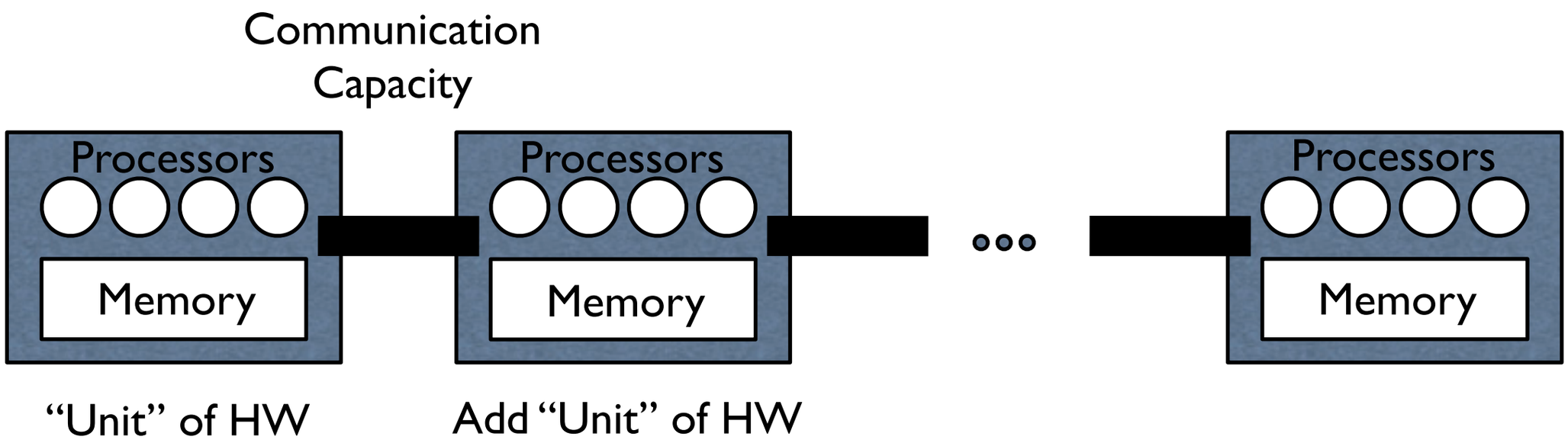


"Unit" of HW

Consider

System Interface

Make a bigger system

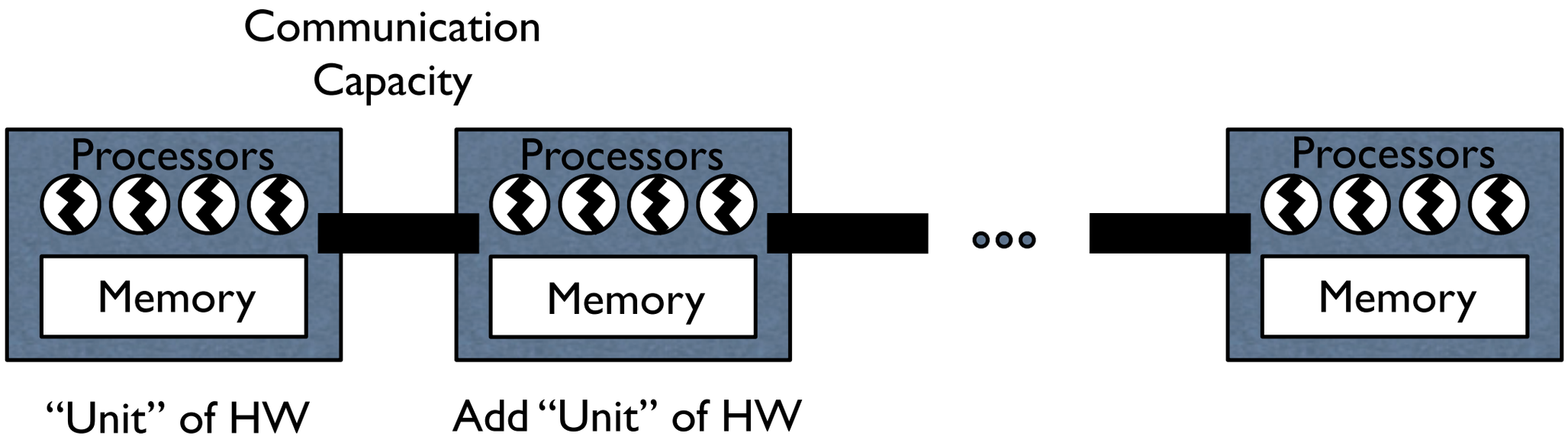


Consider

```
for i:=0; i<n; i++ { go work(i,results) }
```

System Interface

Wouldn’t it be nice if …

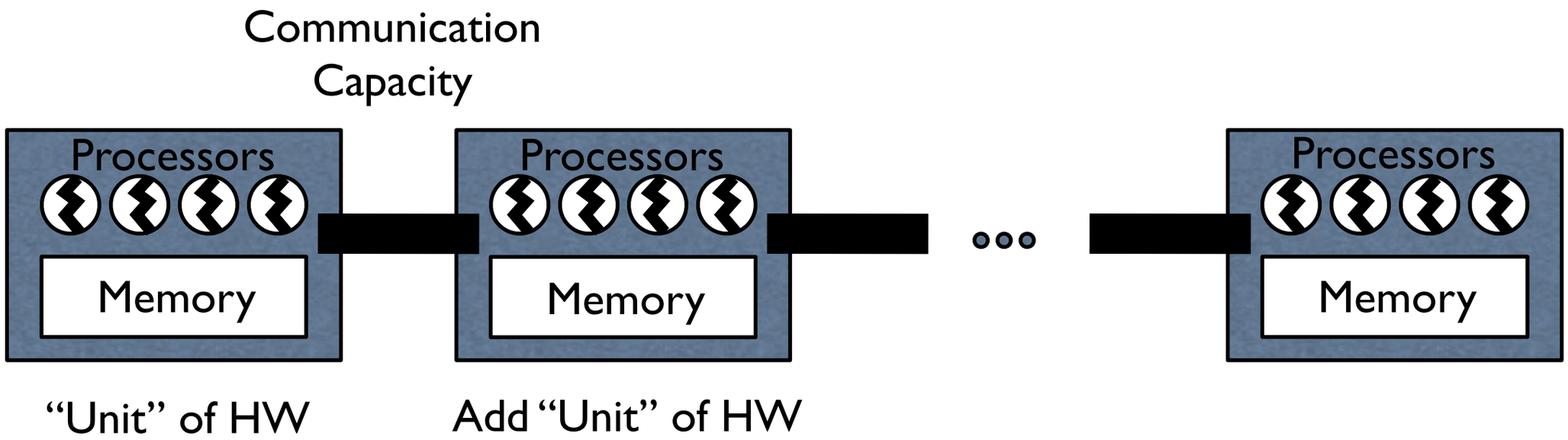


Consider

```
for i:=0; i<n; i++ { go work(i,results) }
```

System Interface

So what are we missing?

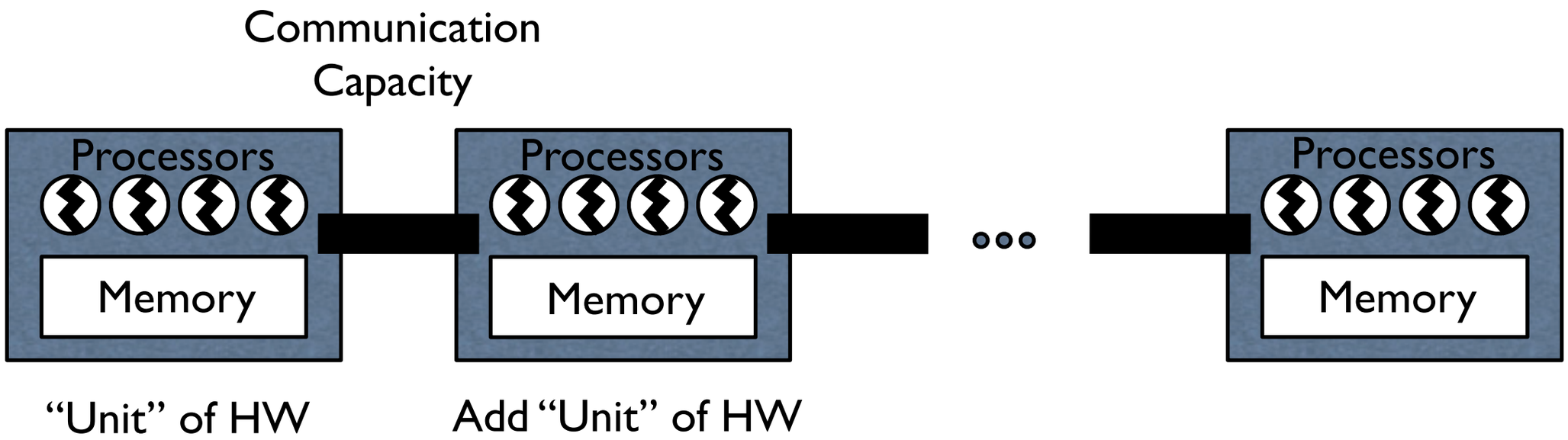


Consider

```
for i:=0; i<n; i++ { go work(i,results) }
```

System Interface

SHARED MEMORY!

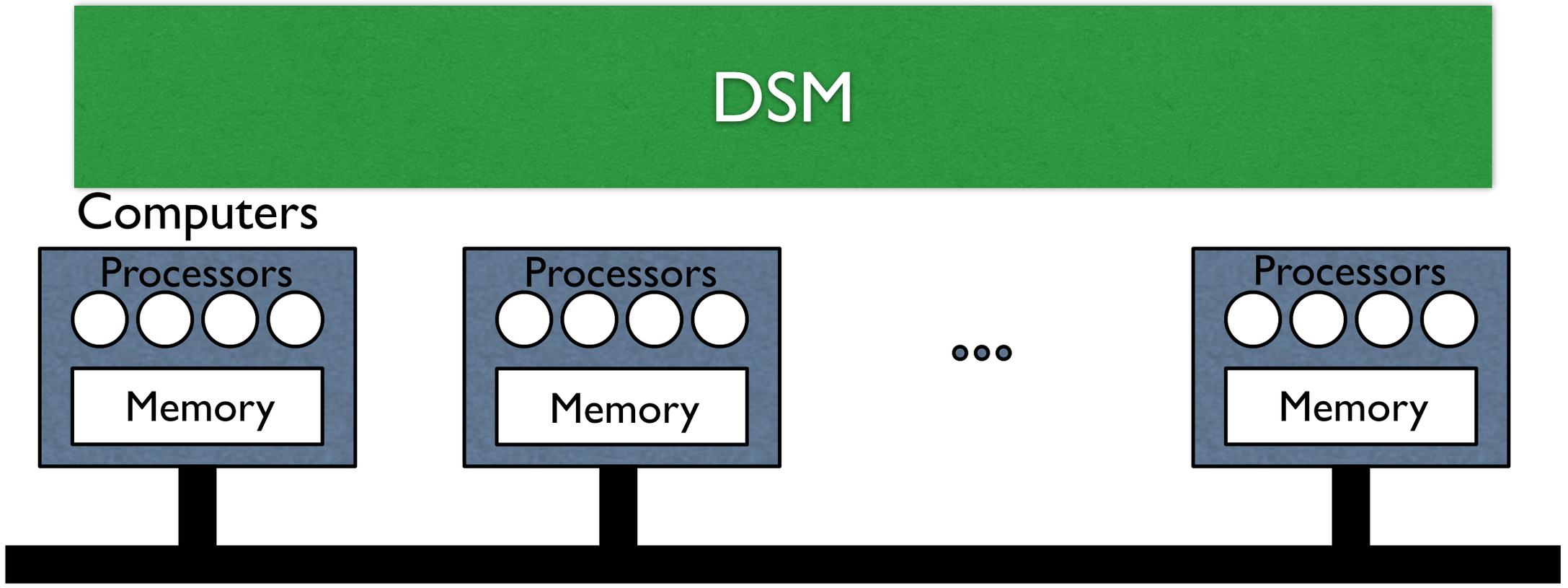


DSM: Simulate Shared Memory on a Distributed Collection of Computers

DSM

Apps don't directly do communication … instead DSM software provide the primitives for a language like go — Shared Address space, threads, locks, channels, etc.

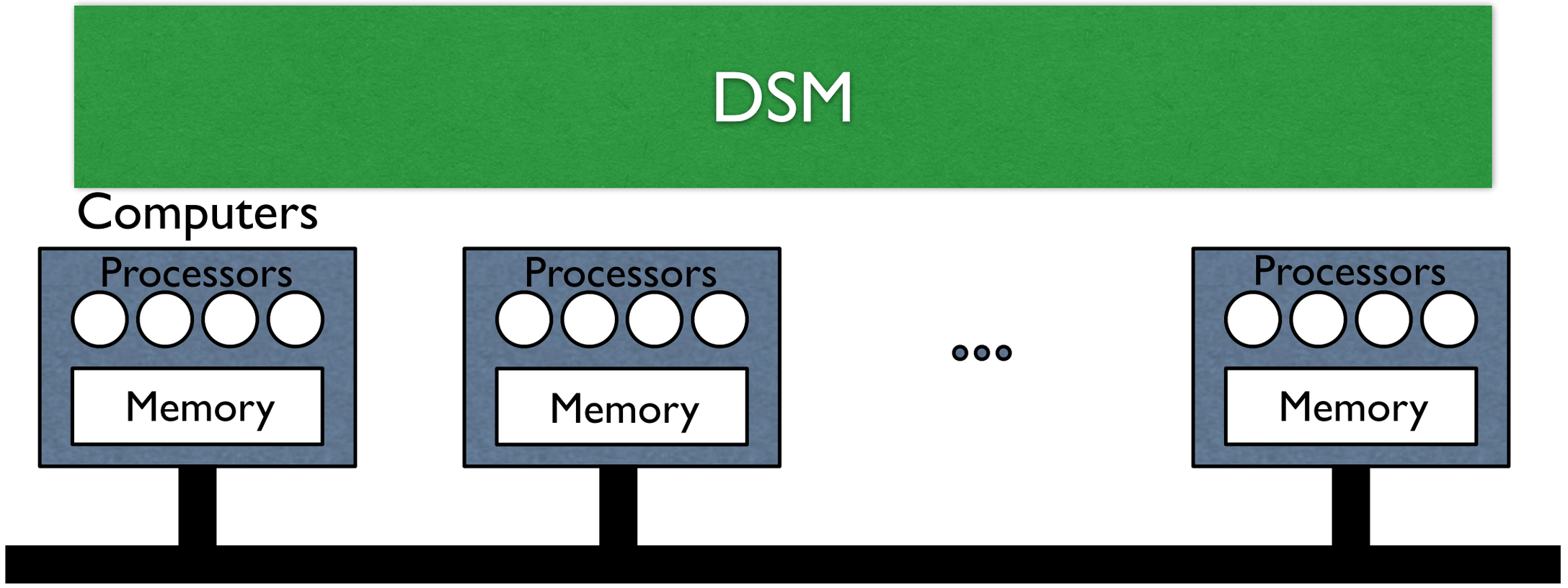
```
for i:=0; i<n; i++ { go work(i,results) }
```



DSM

1. Familiar Model
2. Very General Purpose
3. Huge collection of existing threaded libraries

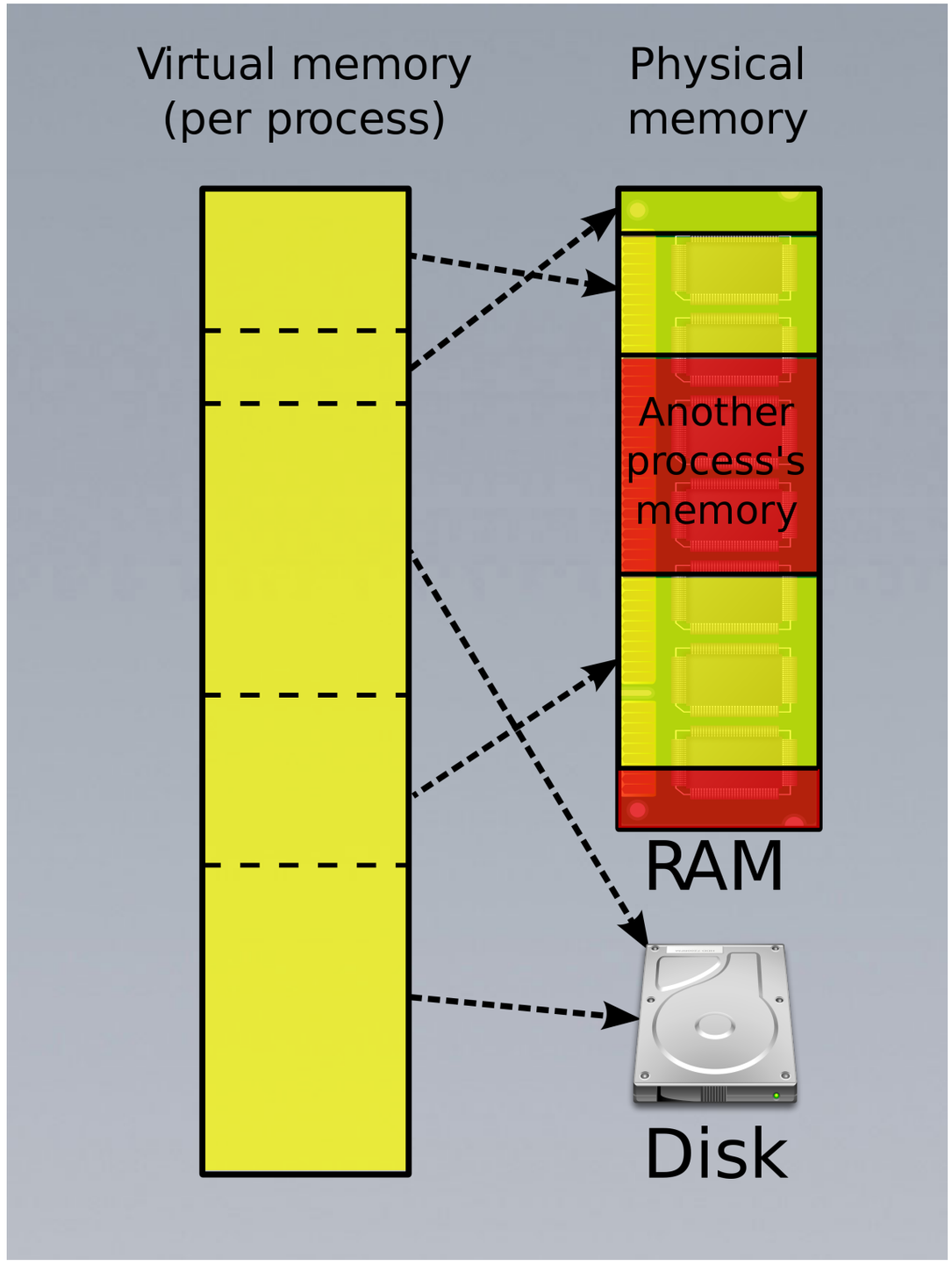
```
for i:=0; i<n; i++ { go work(i,results) }
```



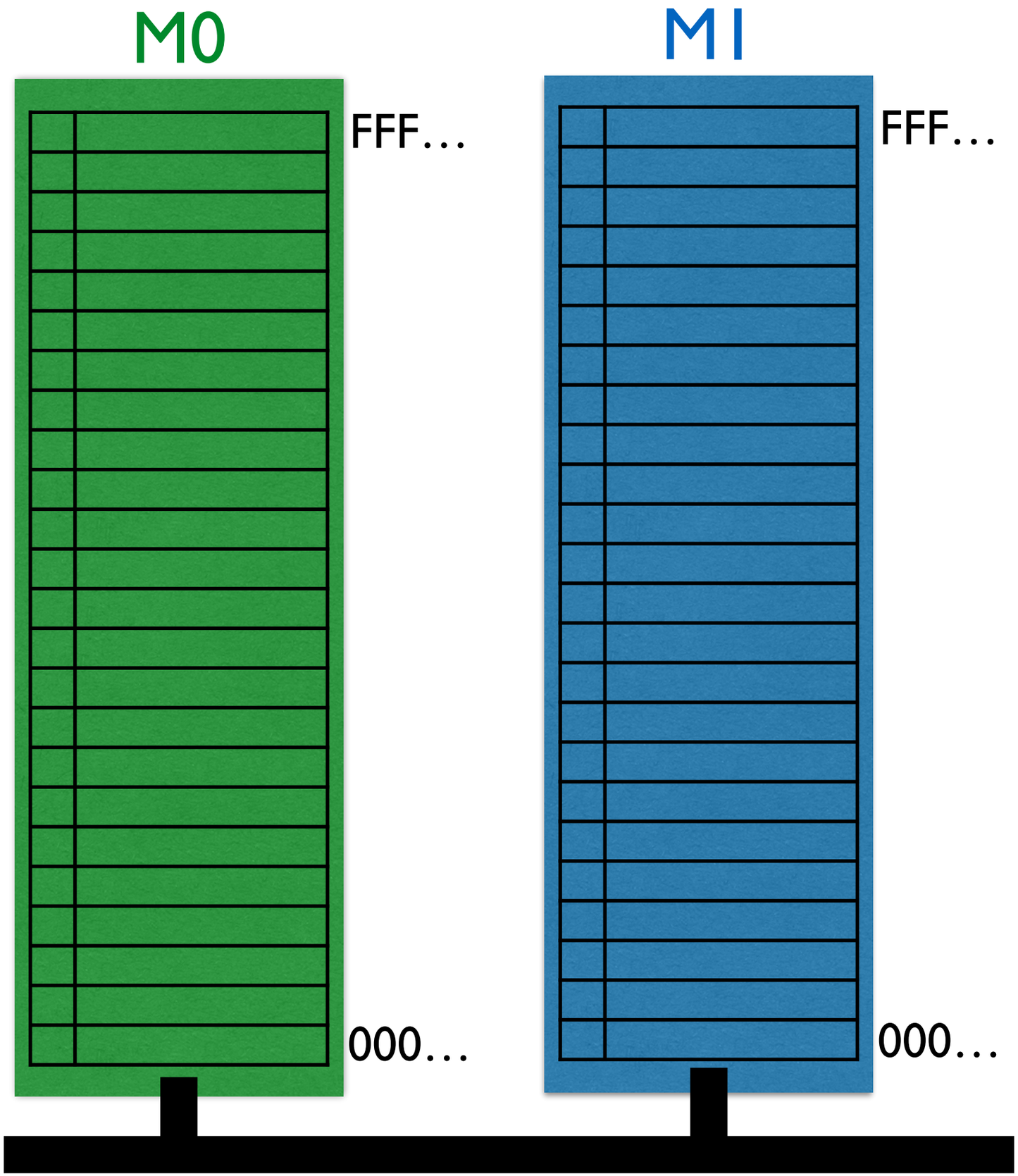
So how are we going to share memory?

DSM Basic Plan (SLOW)

- Exploit the fact that hardware already has support for “Virtual Memory”
- Paging Hardware and OS software trampoline to DSM SW

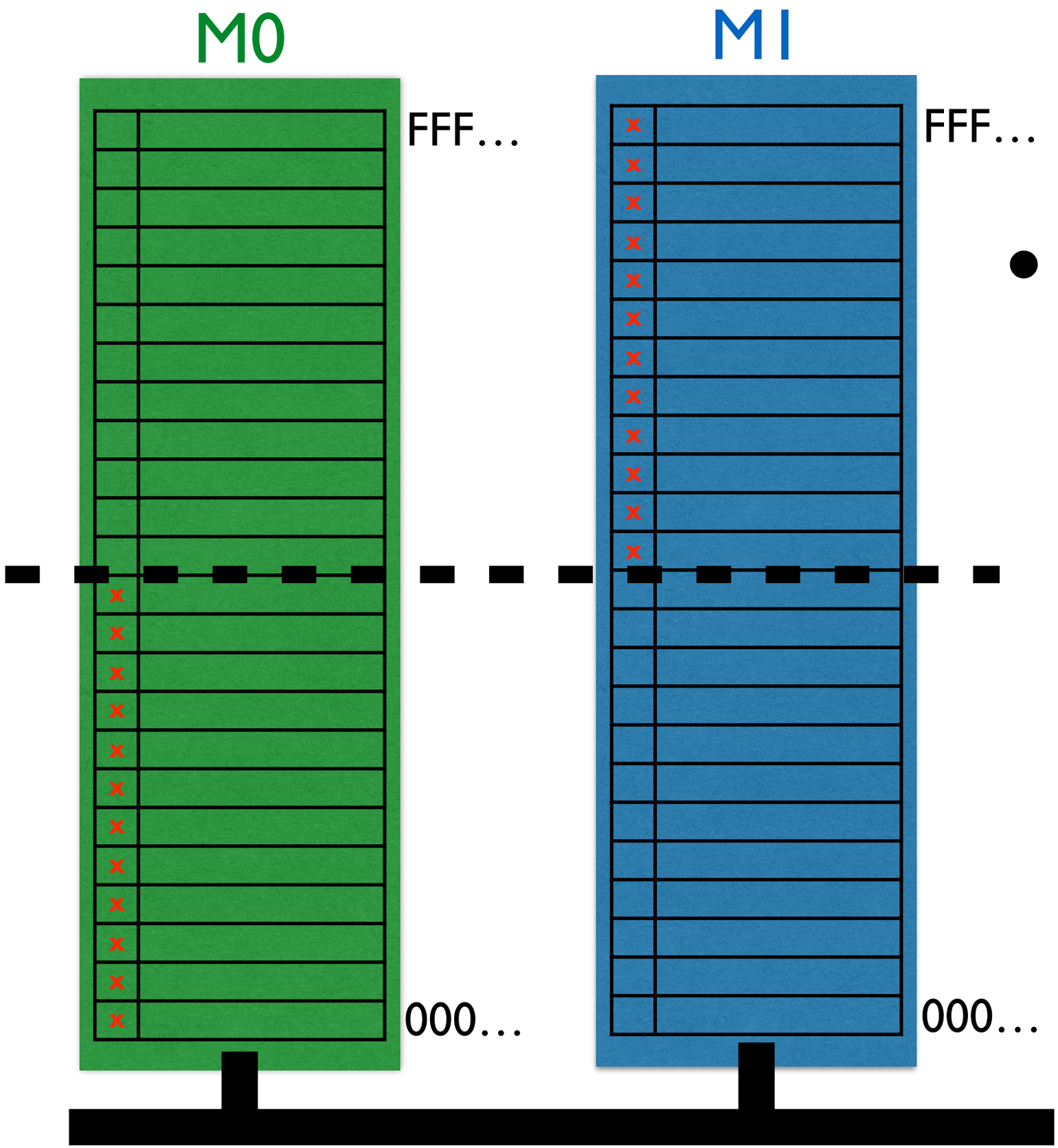


DSM Basic Plan



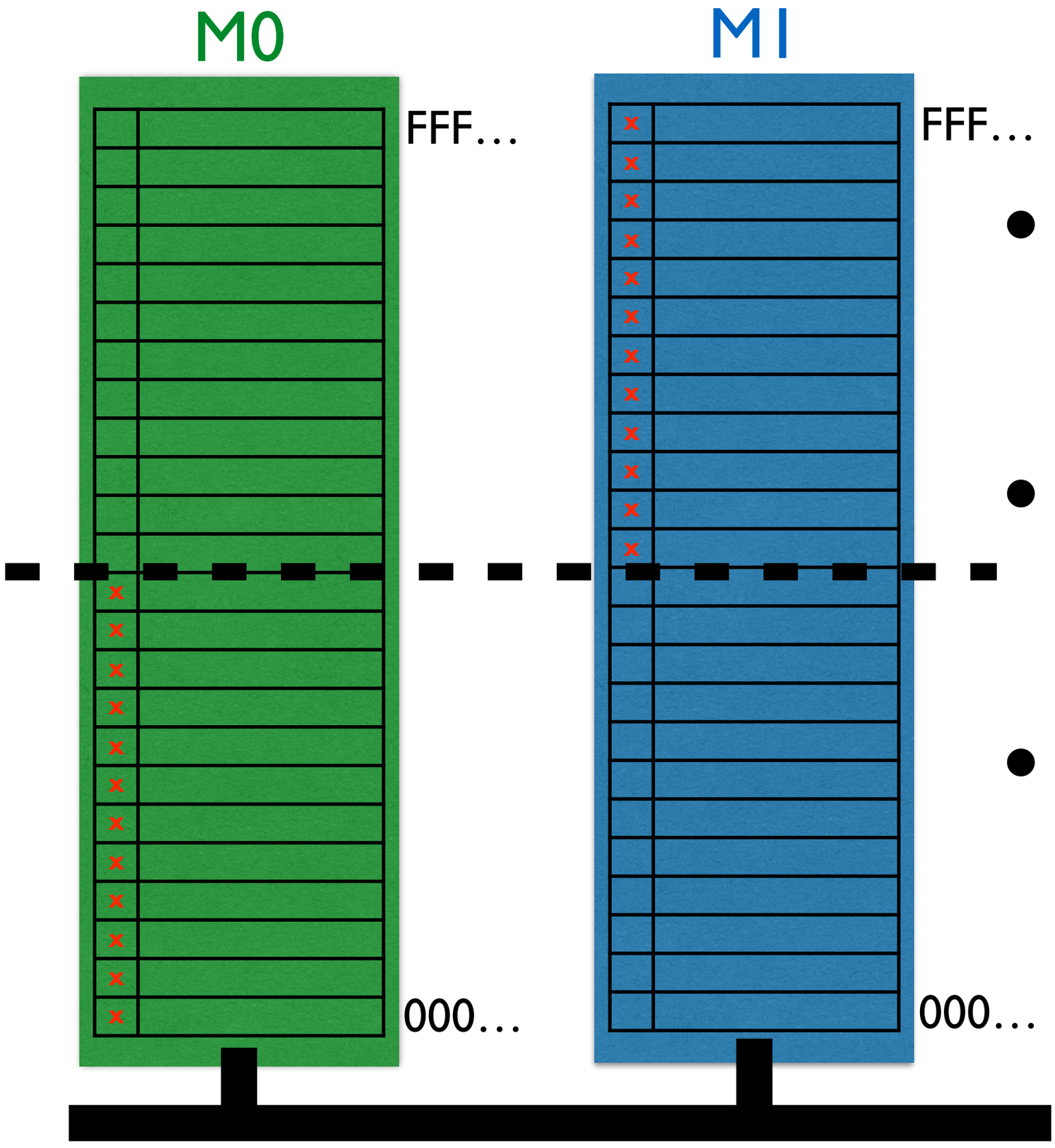
- Two machines on a network
- Address Space controlled by page table
- Pages can be marked R,W,RW,NONE

DSM Basic Plan



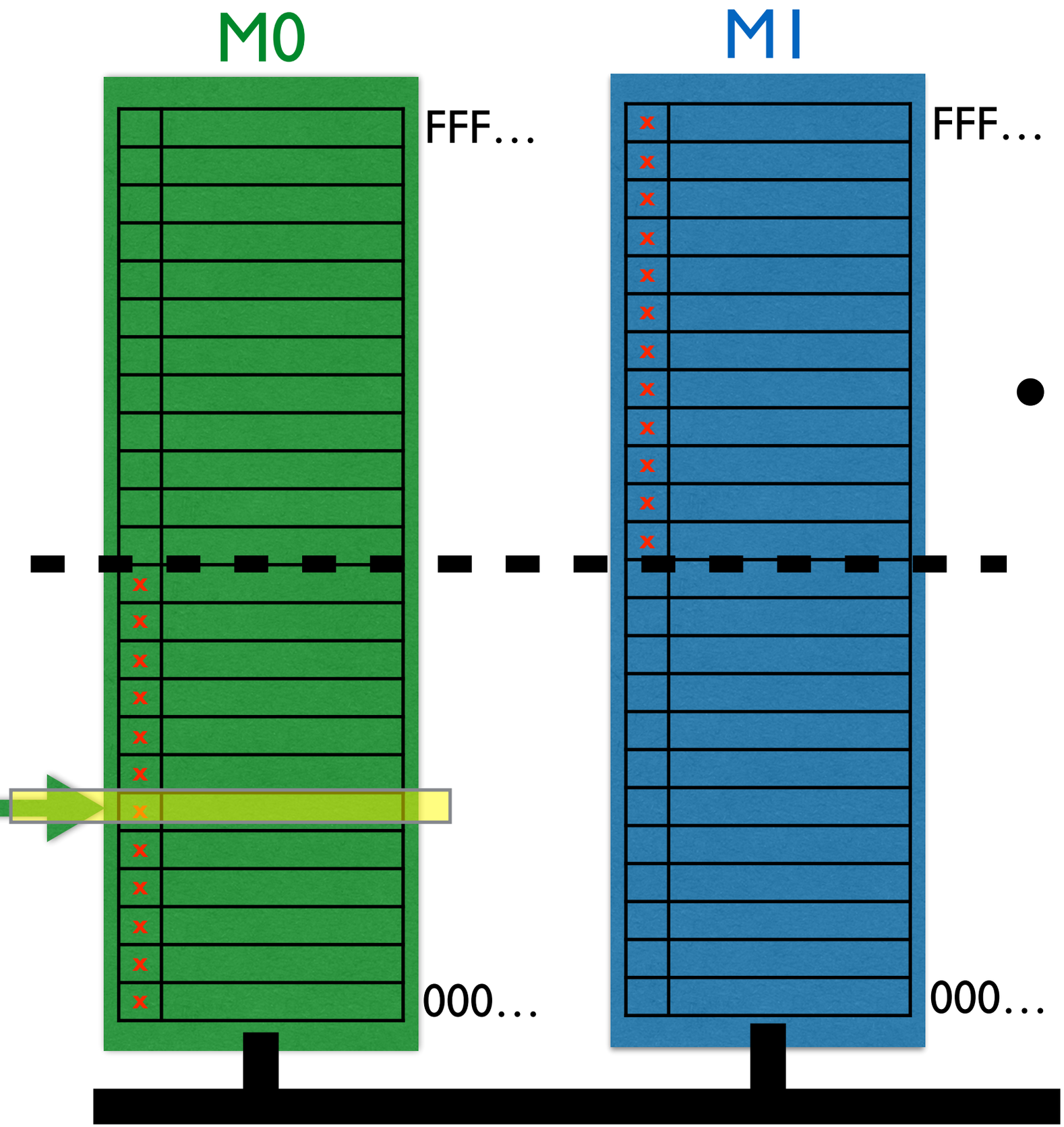
- Simple setup where at first we split the "Shared" Address Space in Half
- Top accessible "owned" by M0
- Bottom accessible "owned" by M1

DSM Basic Plan



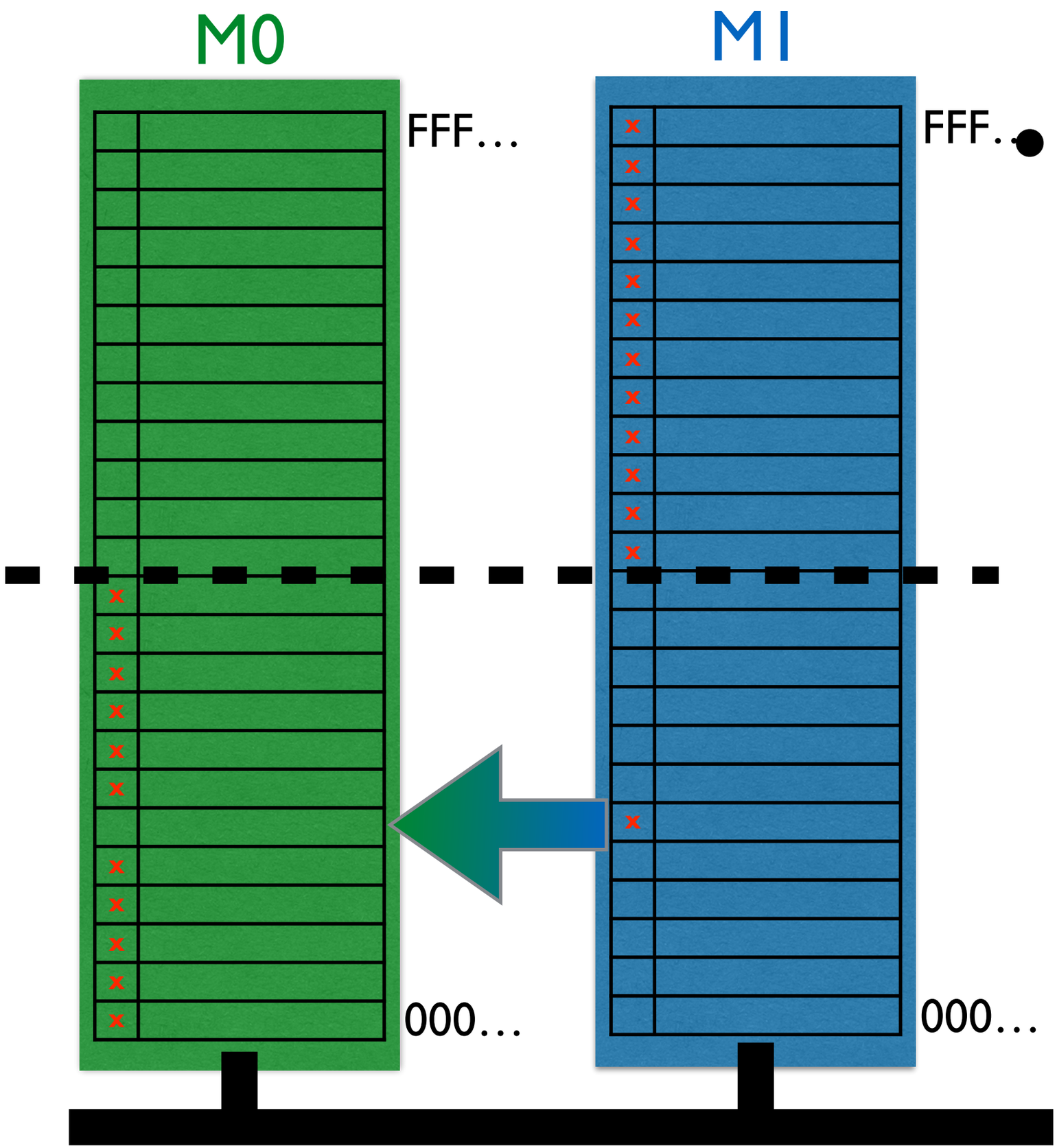
- For any one page only accessible on a single machine
- A thread on M0 that only touch upper half is fine
- A thread on M1 that only touch's lower half is fine

DSM Basic Plan



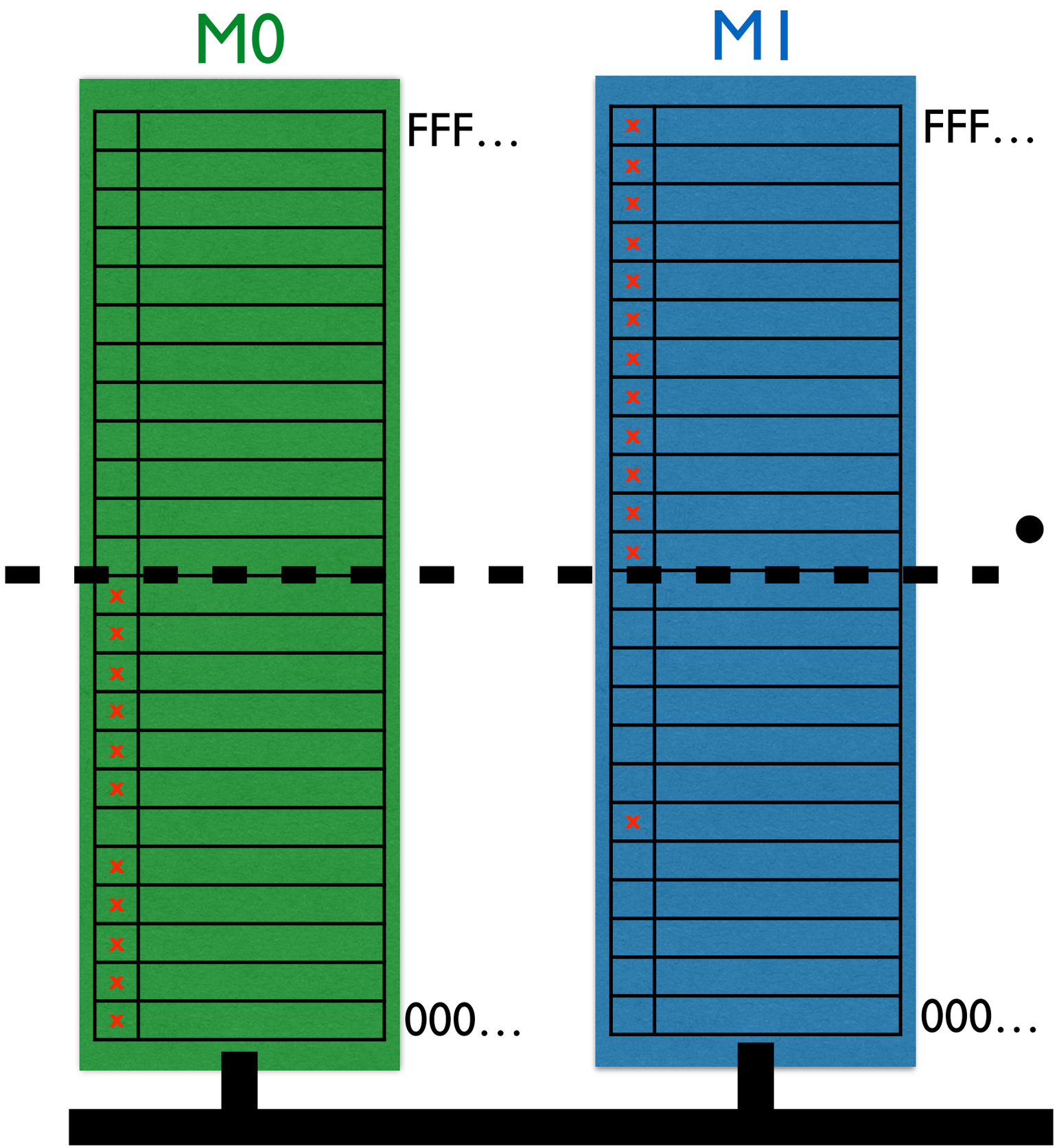
- When a thread on M0 touches a page in the lower half — DSM software kicks in
- hw generates fault

DSM Basic Plan



- When a thread on M0 touches a page in the lower half — DSM software kicks in
  - hw generates fault
  - DSM code handles:
    - copies page and marks page inaccessible on M1
    - and accessible on M0

DSM Basic Plan



- Threaded code just works! — eg. Matrix Multiply, sort, etc.

An Example to think about

```
X = Y = 0


Thread 0                Thread 1
   X=1                     Y=1
   if Y==0 {               if X==0 {
     print "yes"             print “yes”
   }                       }
```

Can both these print yes?

An Example to think about

```
X = Y = 0


Thread 0                Thread 1
   X=1                     Y=1
   if Y==0 {               if X==0 {
     print "yes"             print “yes”
   }                       }
```

Can both these print yes?

The “go” memory model (https://golang.org/ref/mem)
Could allow one to see two yes’s !!!! YIKES

In other words go’s memory model is not the one you might have expected! (like many modern languages/ machines)

Memory Model?

- Explains how read's and writes in different threads interact with each other
- Its a contract — there are many many memory models out there — Why? Tension between

1. Give compiler/hw freedom to optimize — the more relaxed the MM greater optimization
2. Programmer Guaranties — Things people can reason about that when writing code — stricter easier — no funny business

TreadMarks is trying to address these issues

1. False Sharing
2. Write Amplification

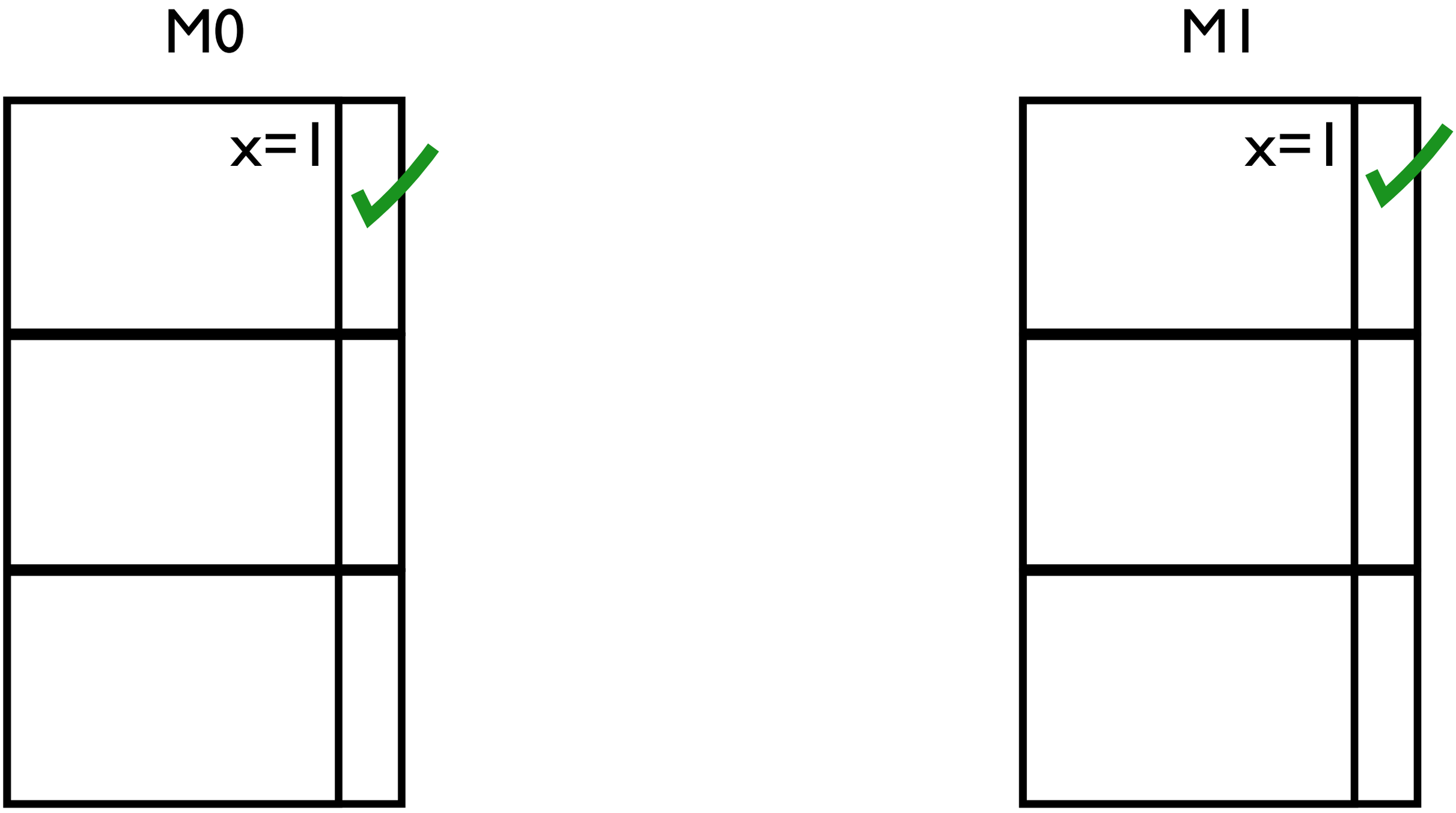
False Sharing

- false sharing: two machines r/w different vars on same page
  - M1 writes x, M2 writes y
  - M1 writes x, M2 just reads y
- Q: what does the general approach do in this situation?

Write Amplification

- write amplification: a one byte write turns into a whole-page transfer

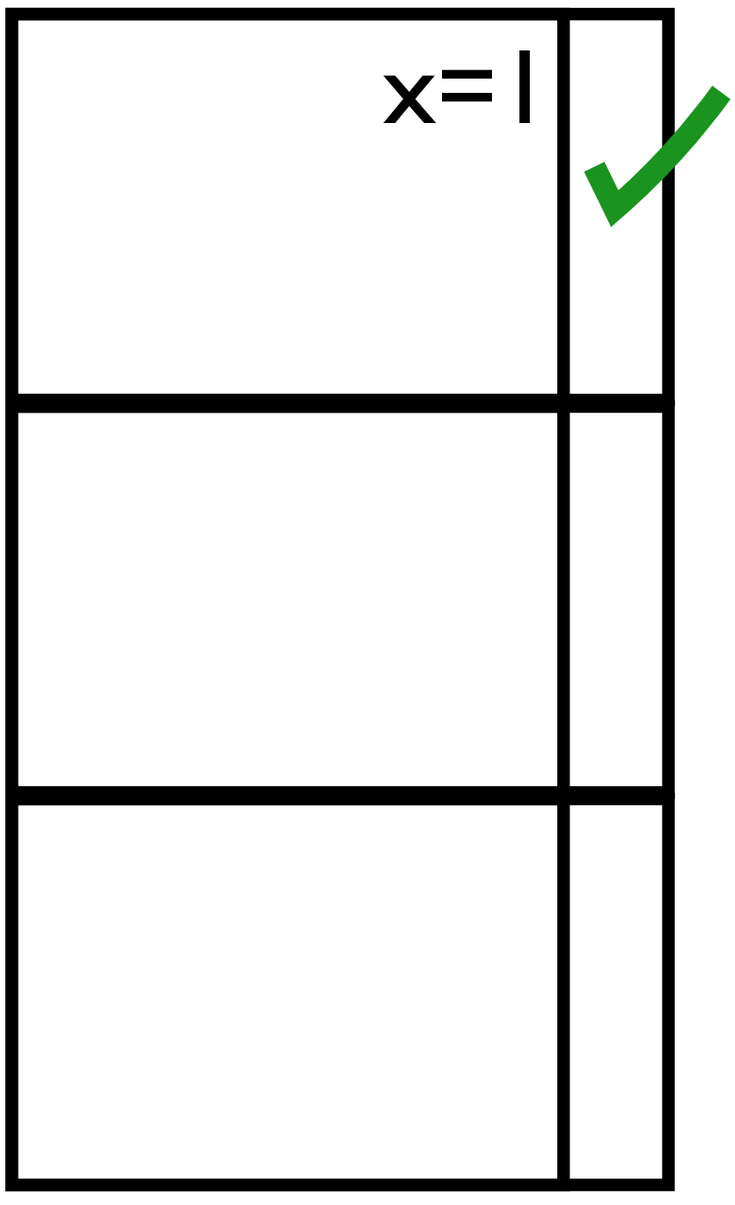
TM:Fix for Write Amplification — Write Diffs



M0: x=2

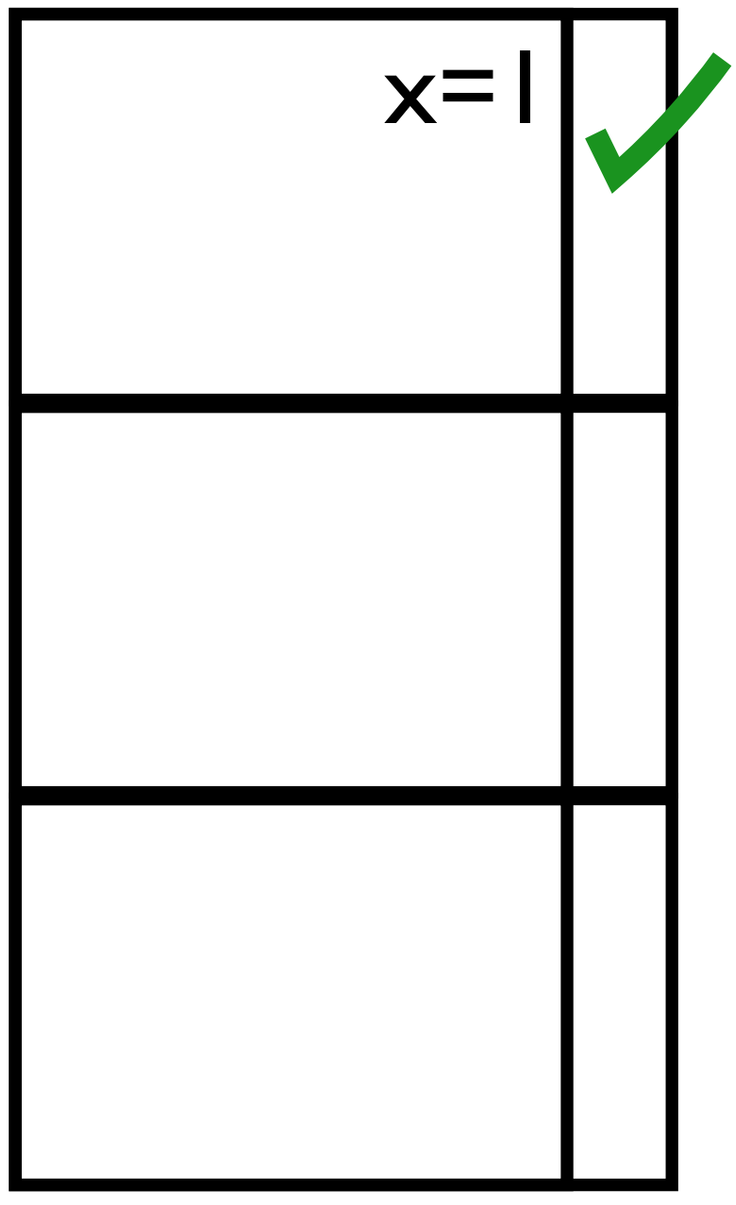
M0

x=1

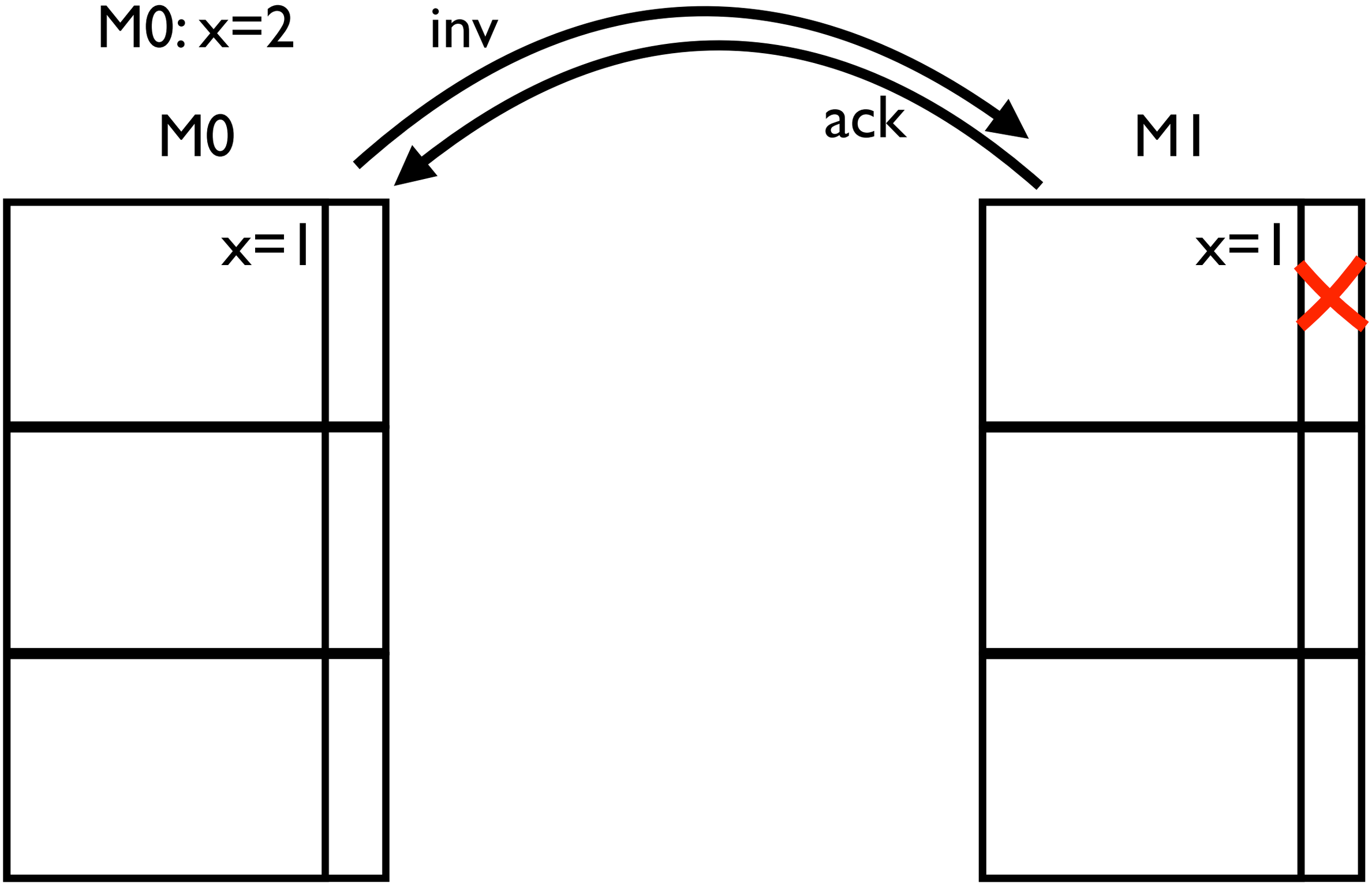


M1

x=1

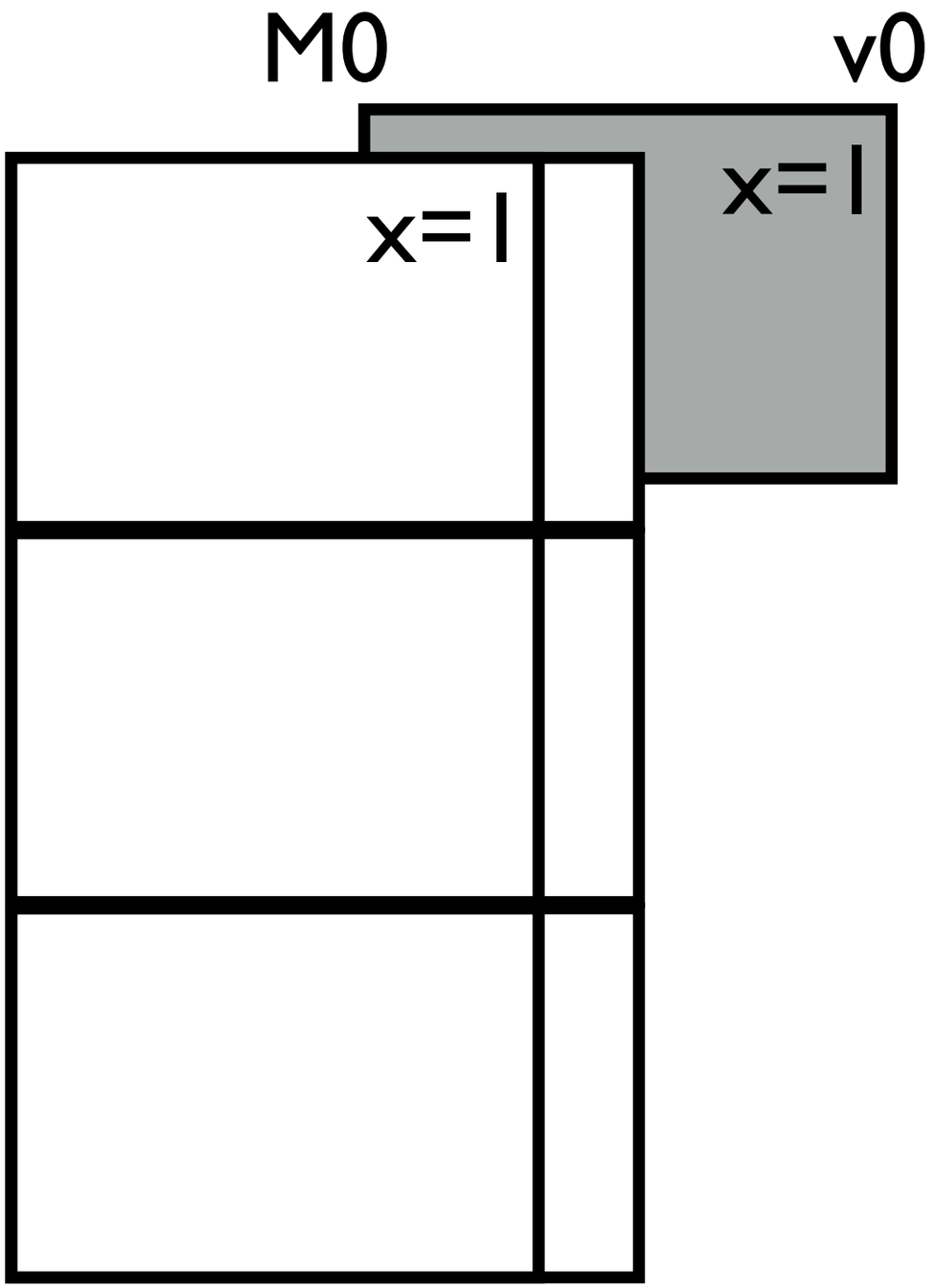


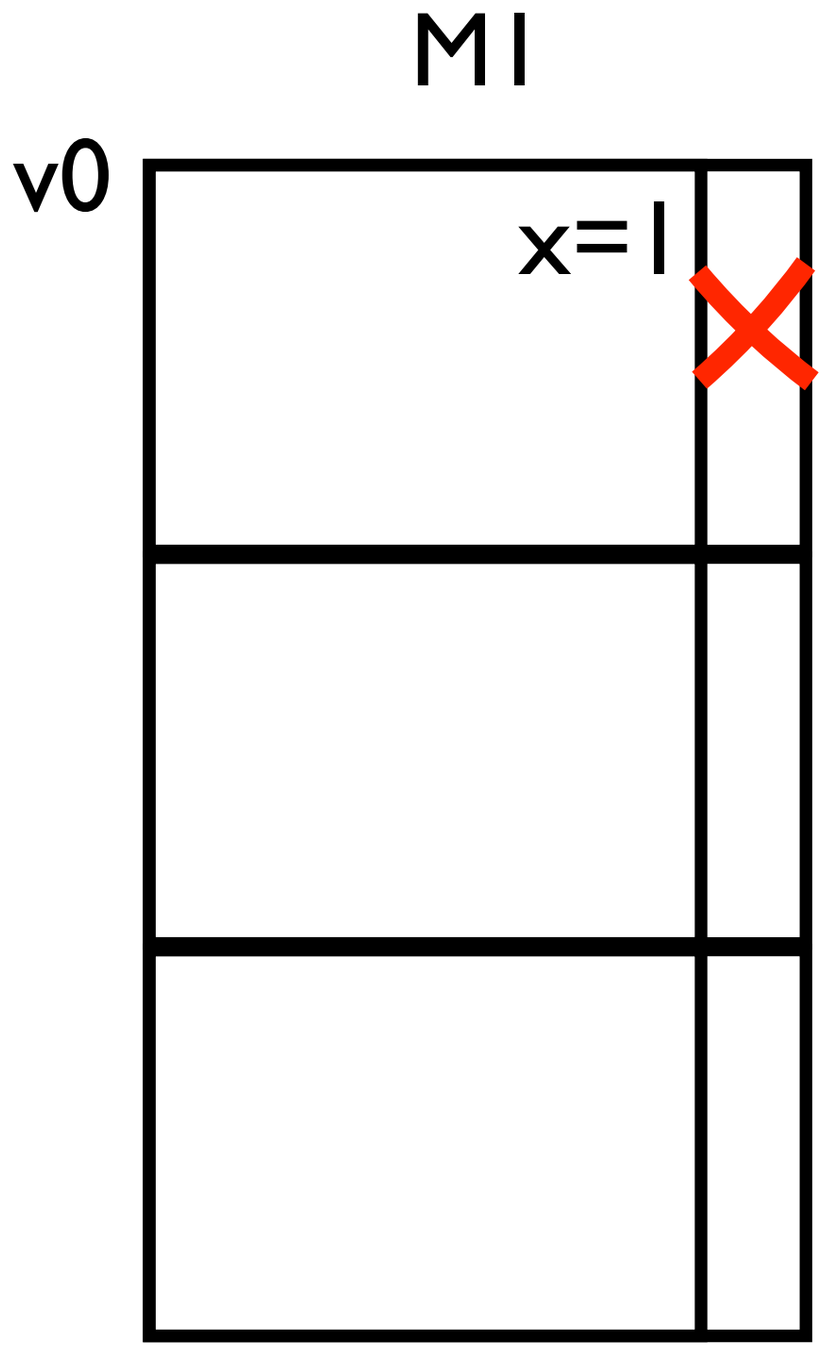
1: Invalidate remote copies but keep



2: make local (shadow) before changing

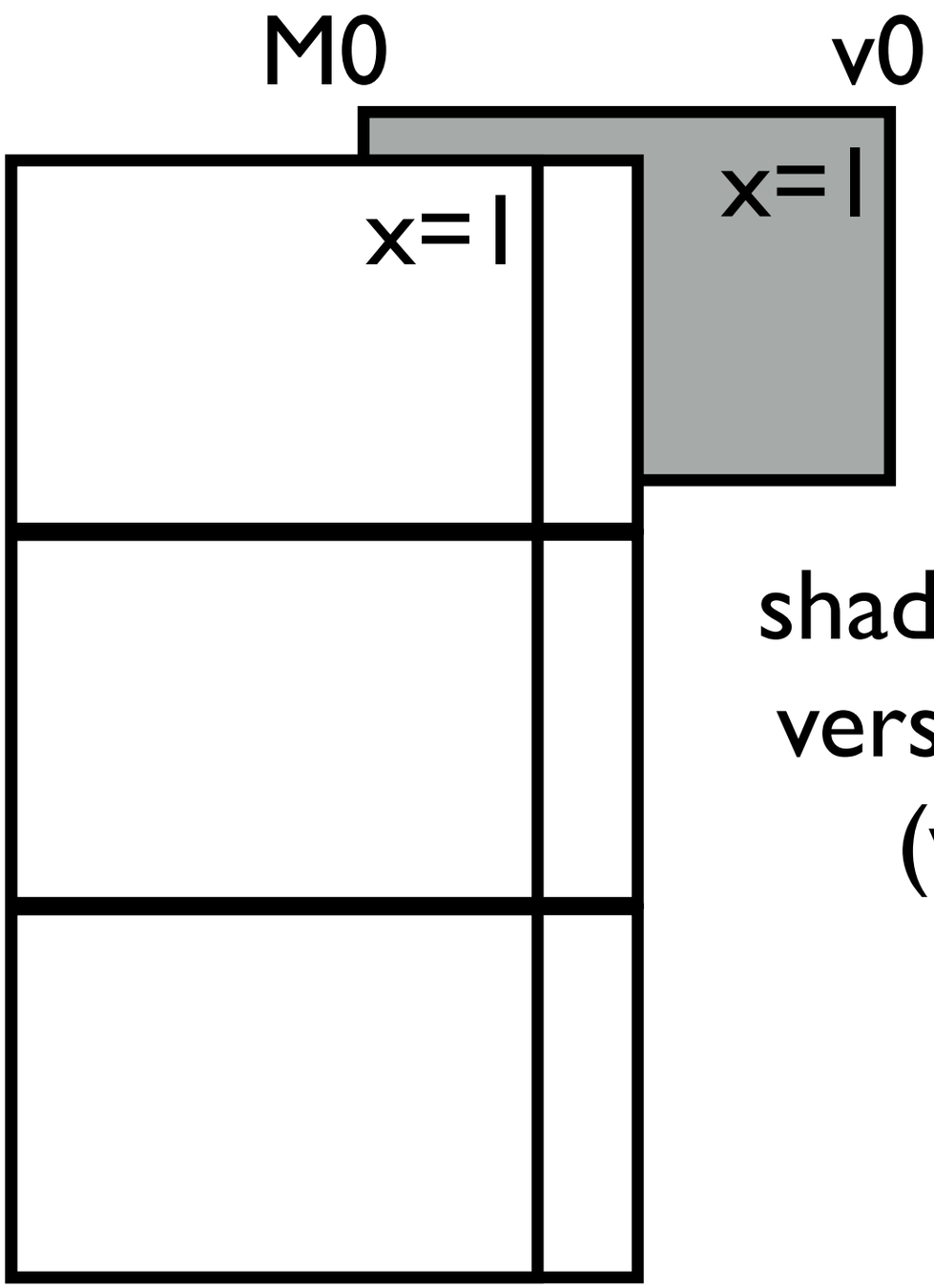
M0: x=2



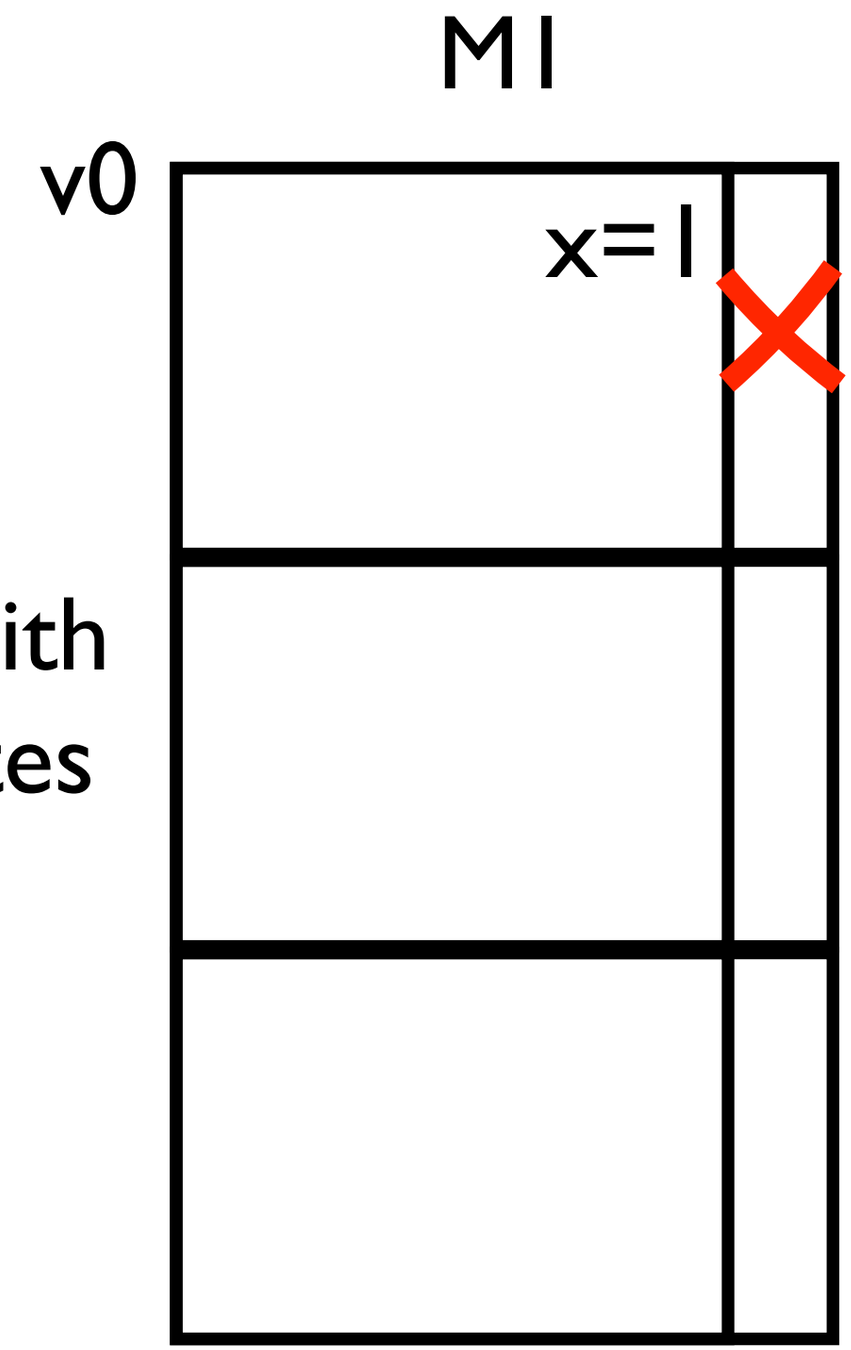


2: make local (shadow) before changing

M0: x=2

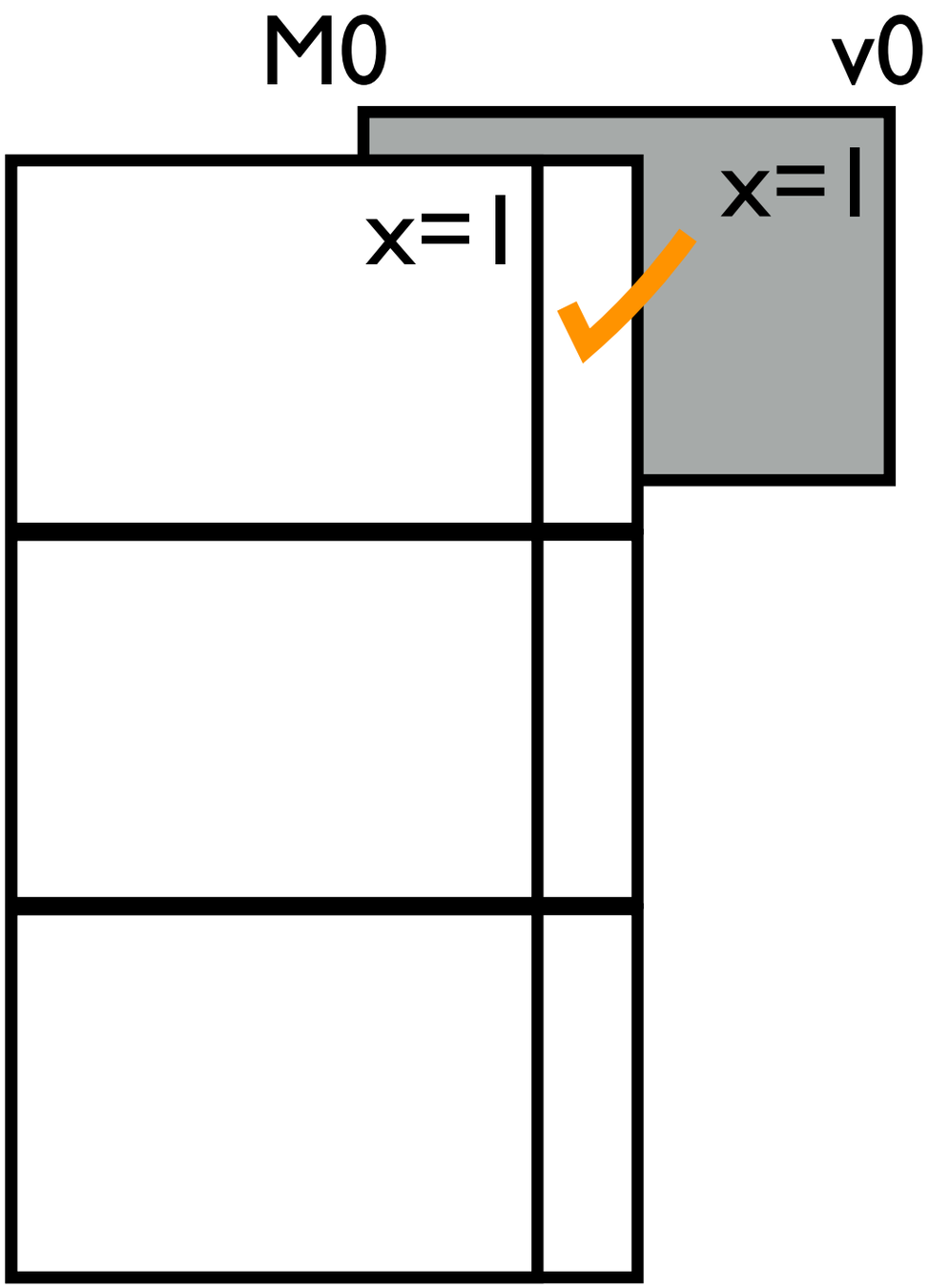


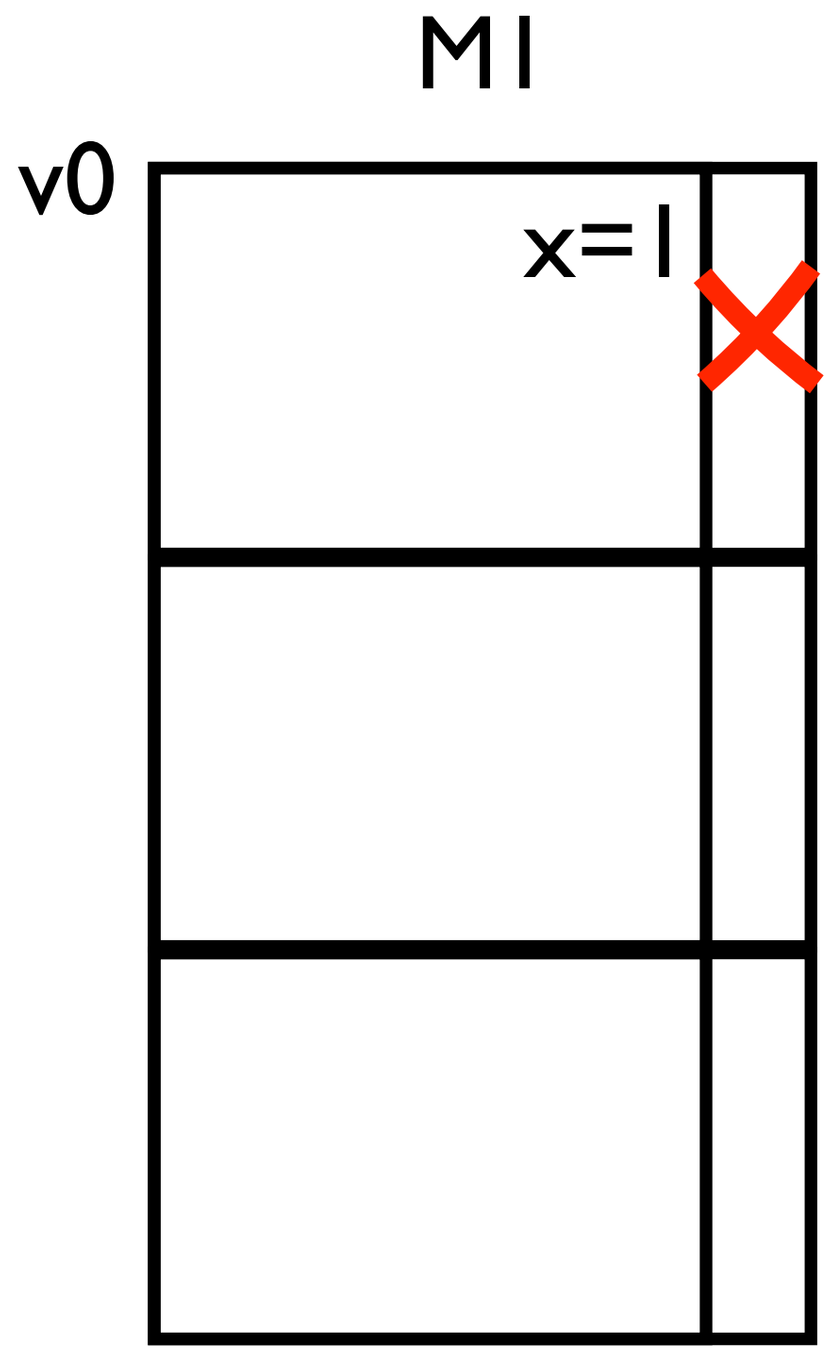
shadow frozen with version at remotes (which is not accessible)



3: mark r/w on M0

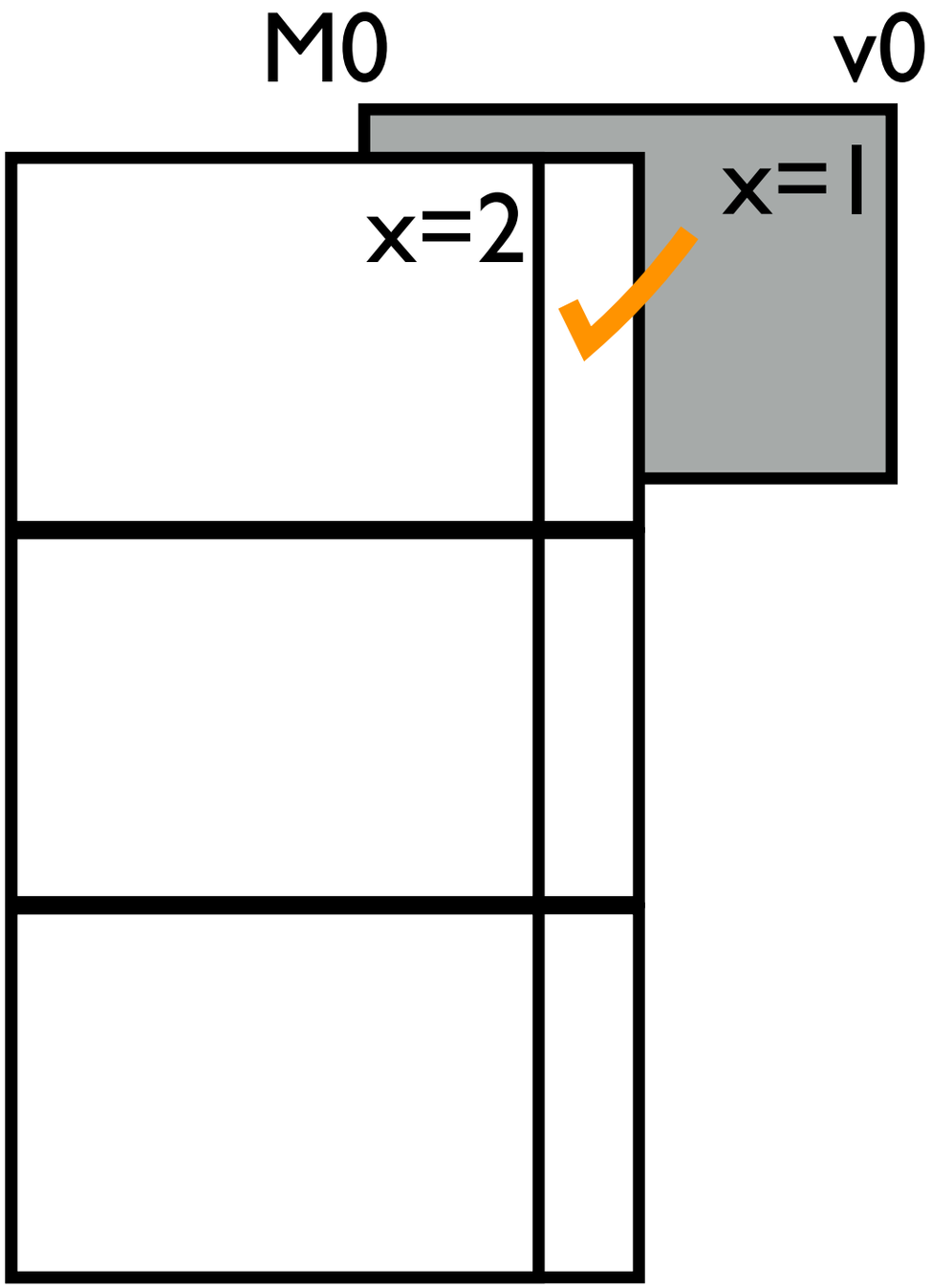
M0: x=2

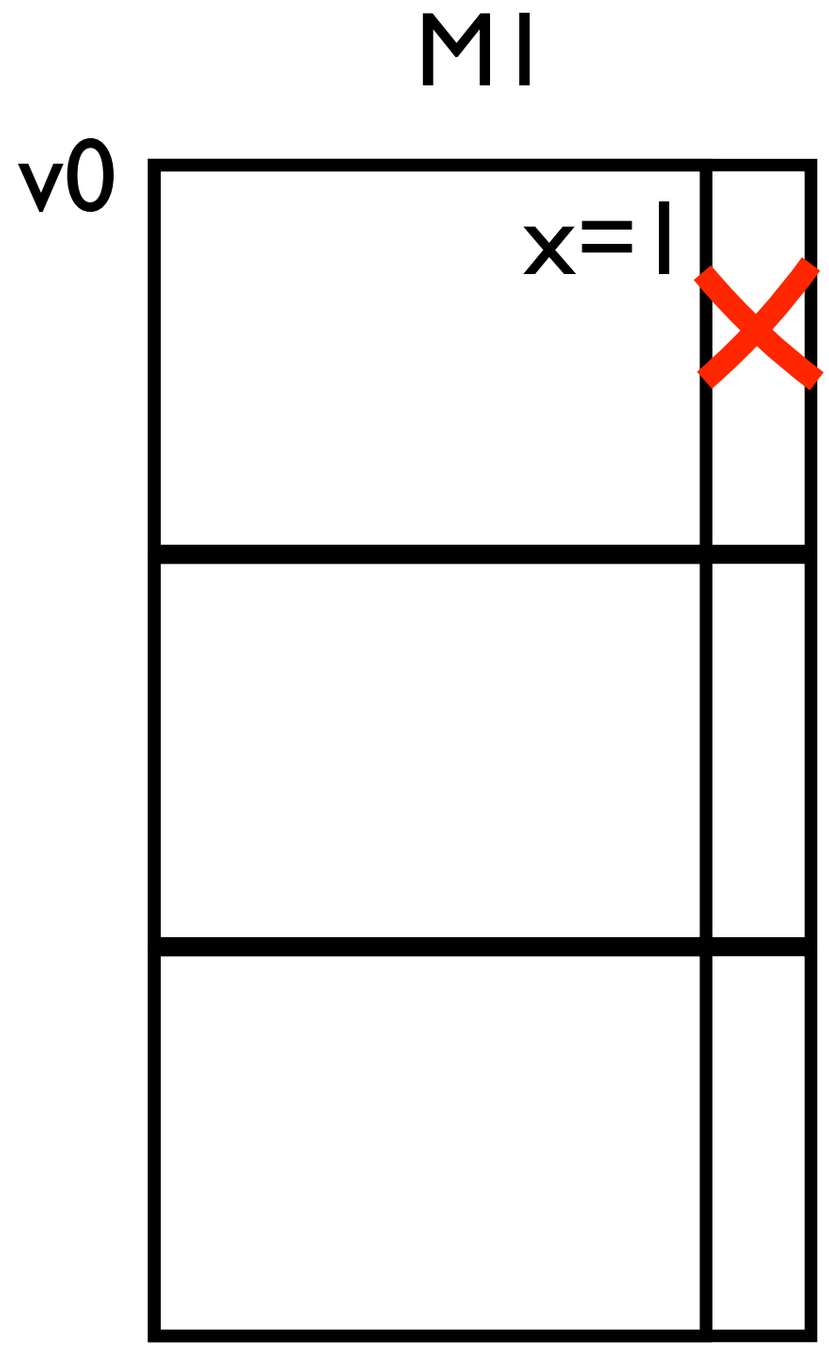




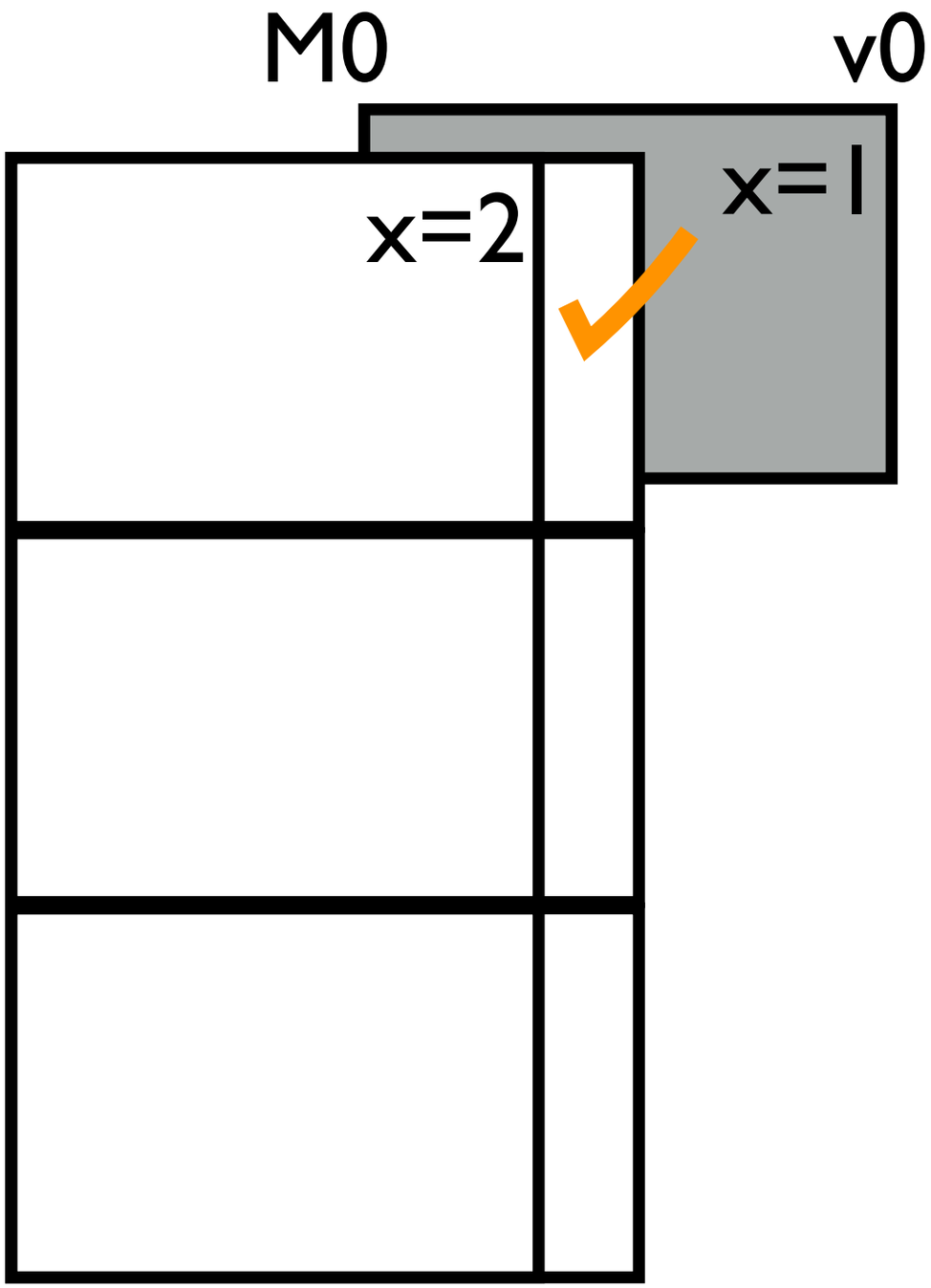
4: Changes to page can proceed (only on M0)

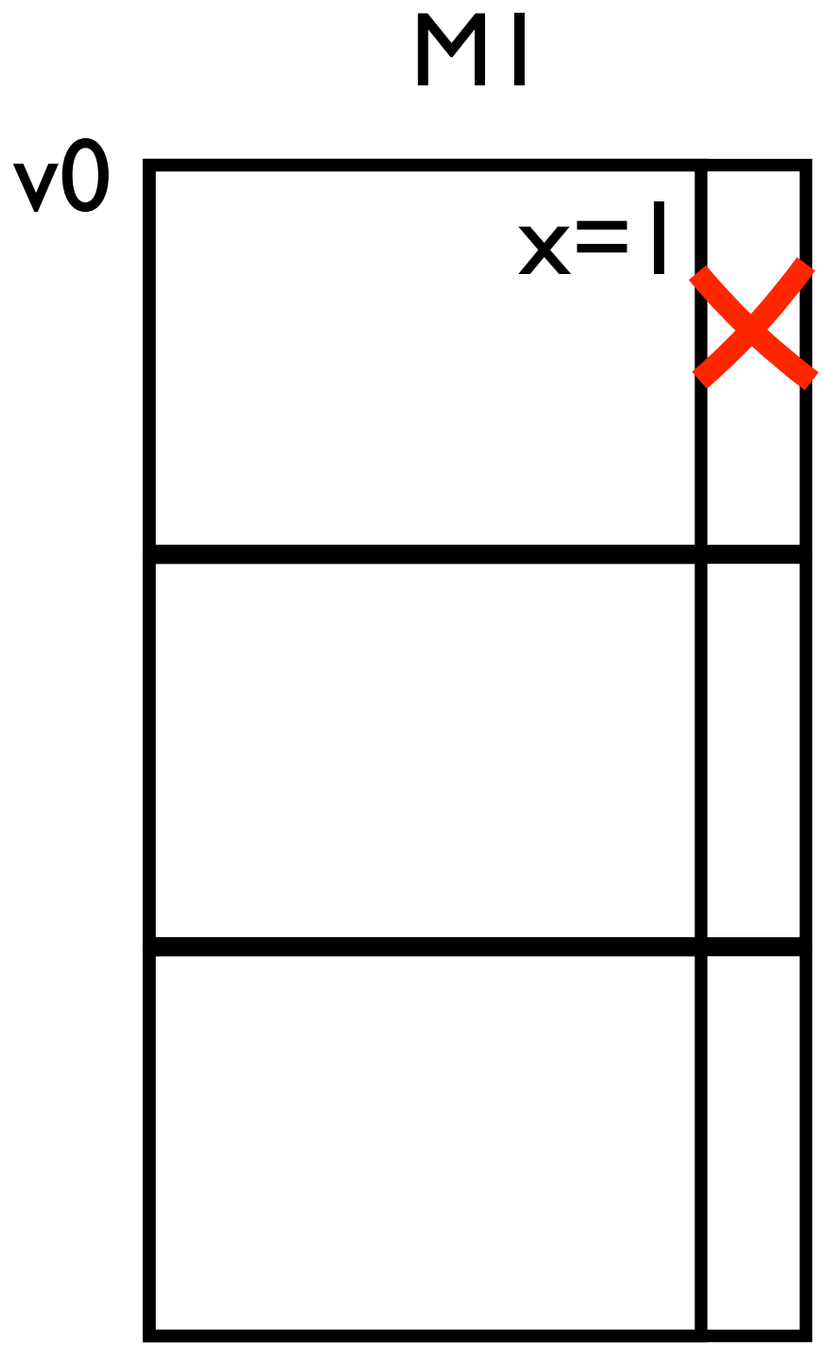
M0: x=2





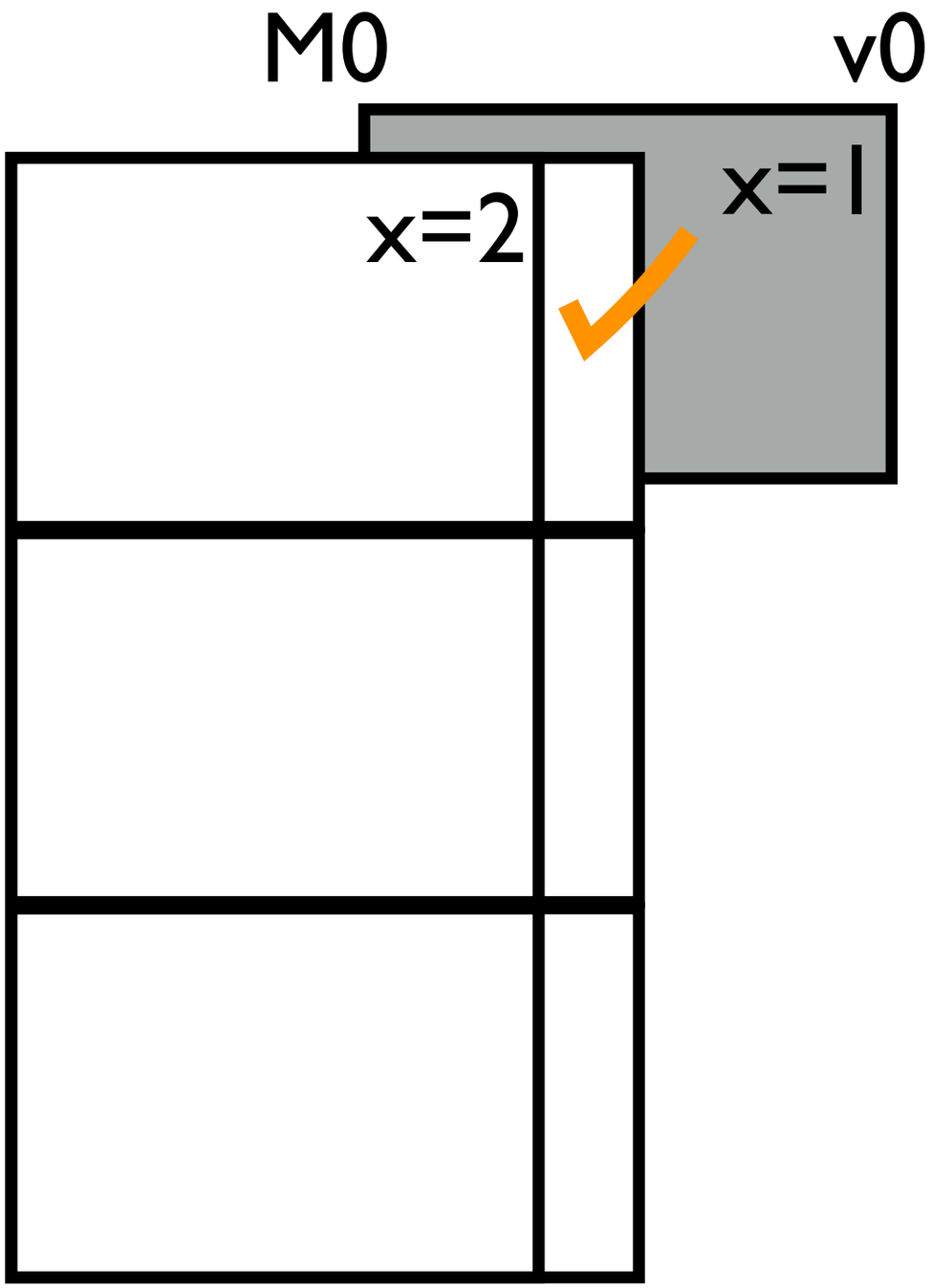
4: Changes to page can proceed (only on M0)

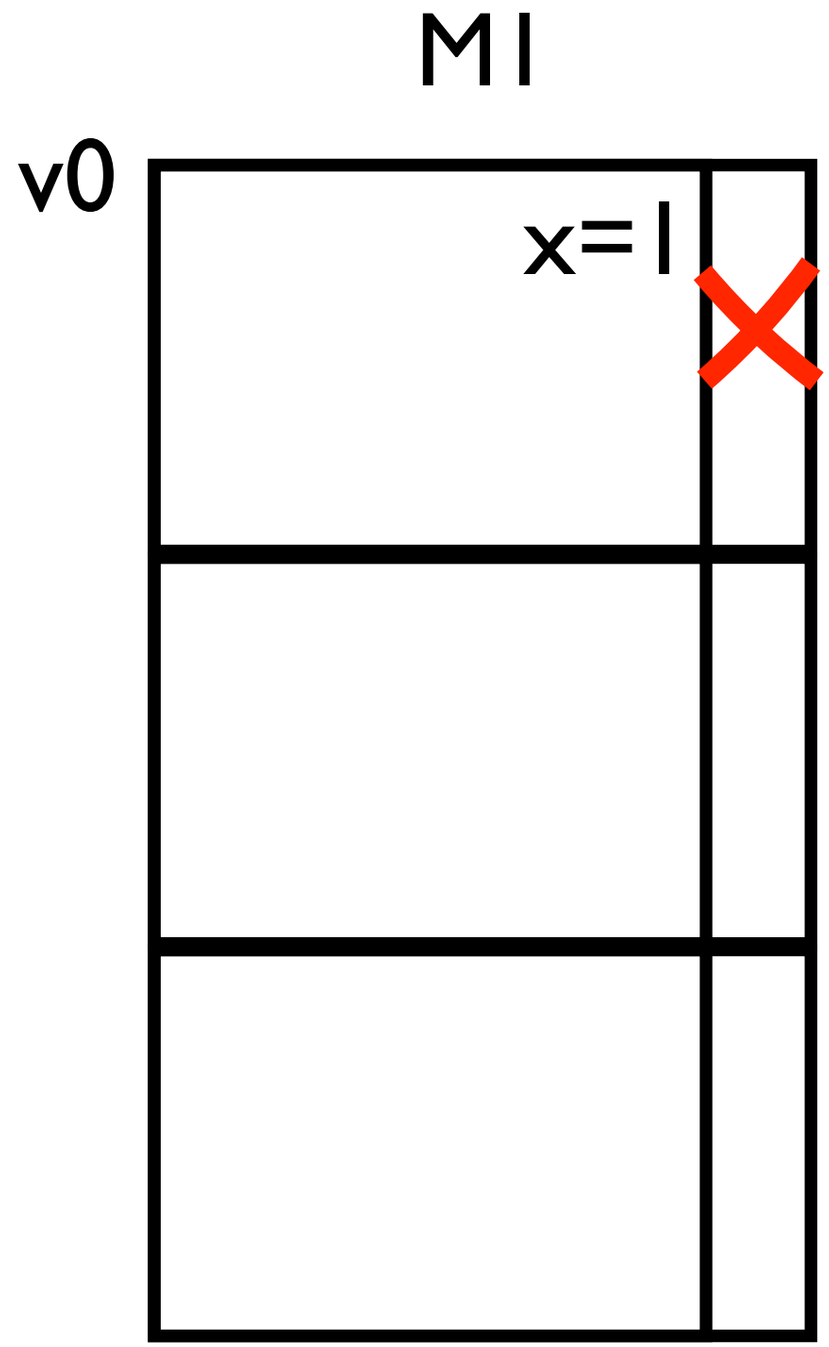




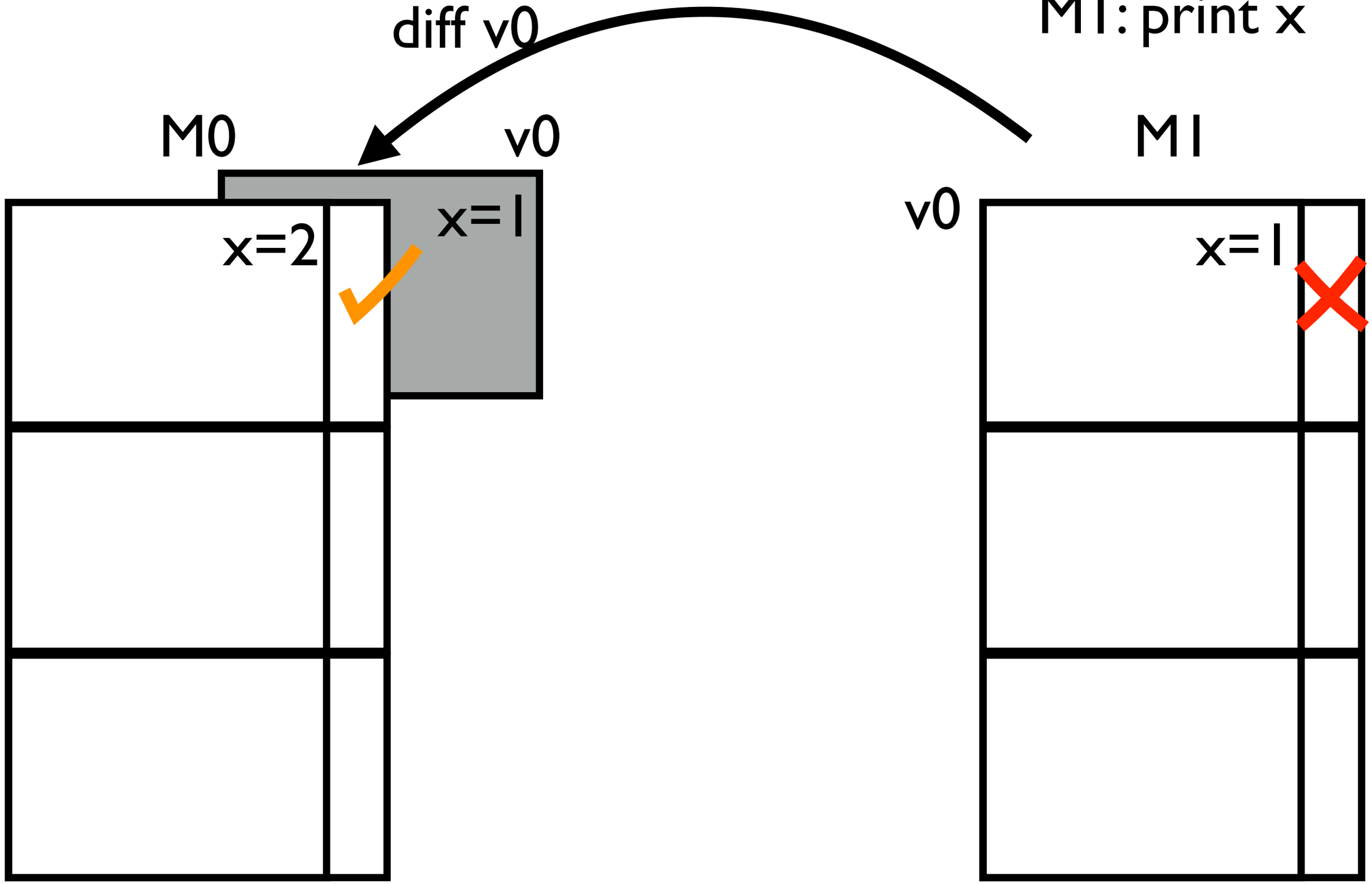
5: At some time M1 accesses Page

M1: print x

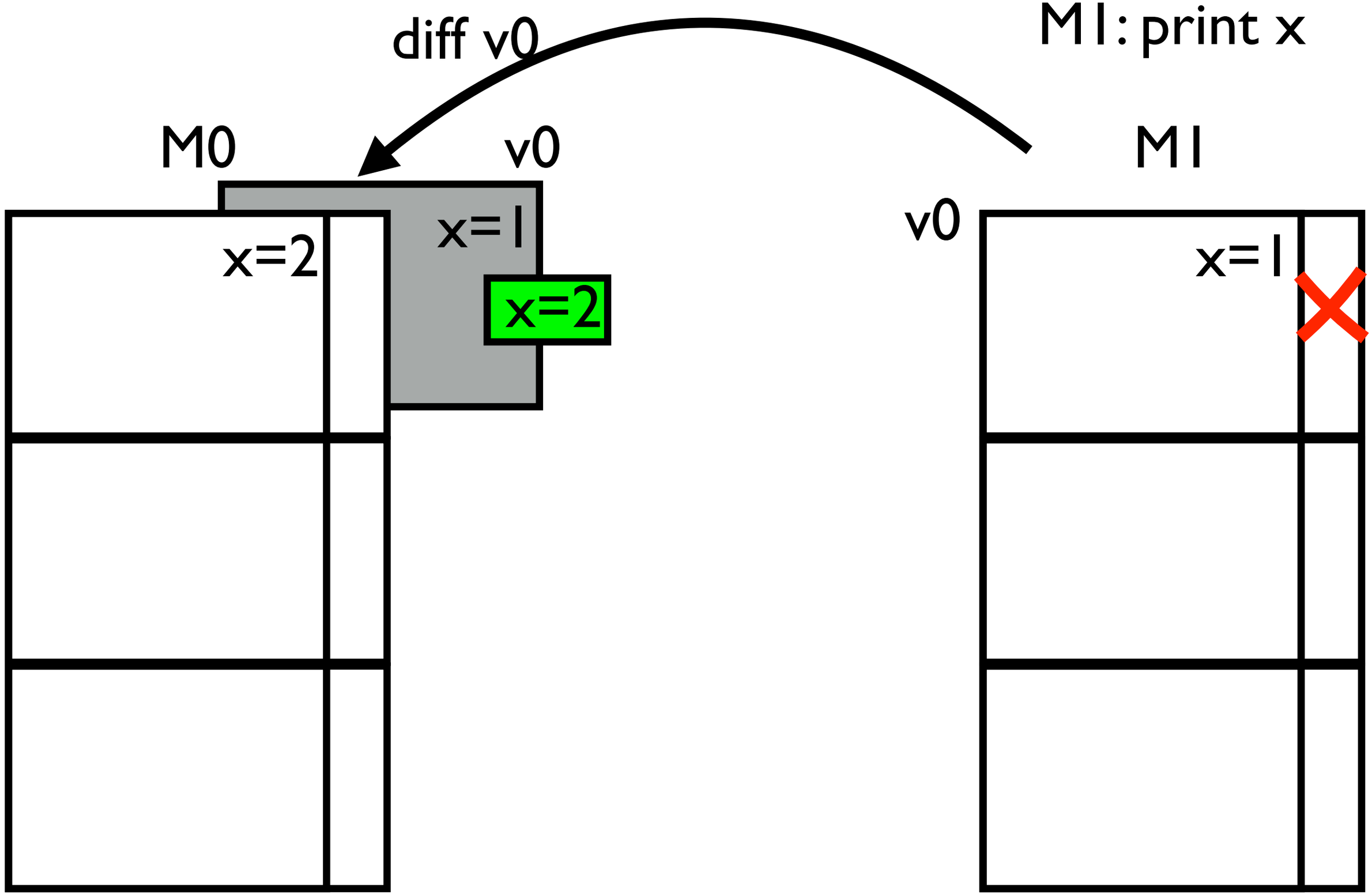




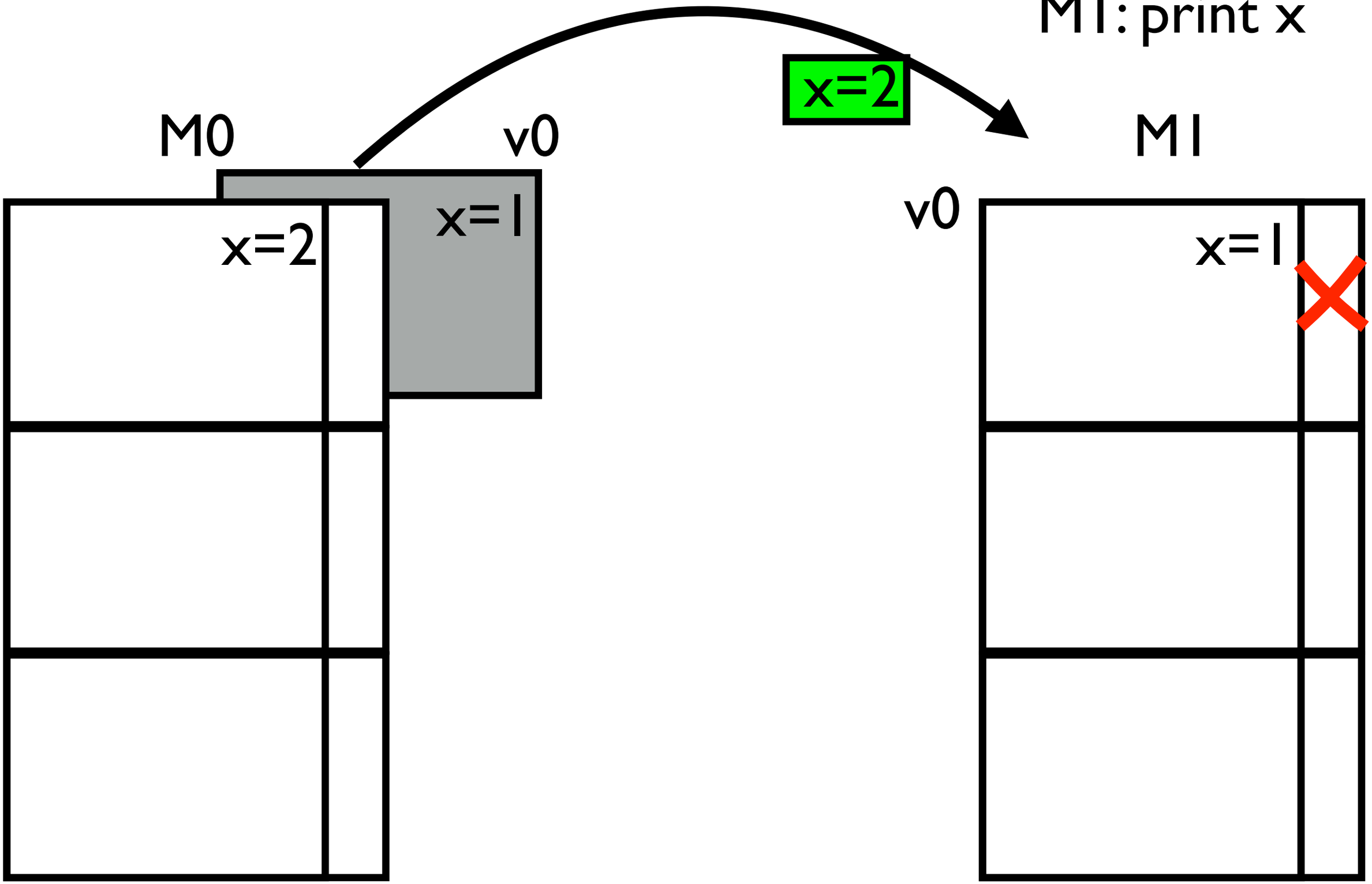
6: Fault — req diff wrt v0 from owner



7: Lock Page and create diff

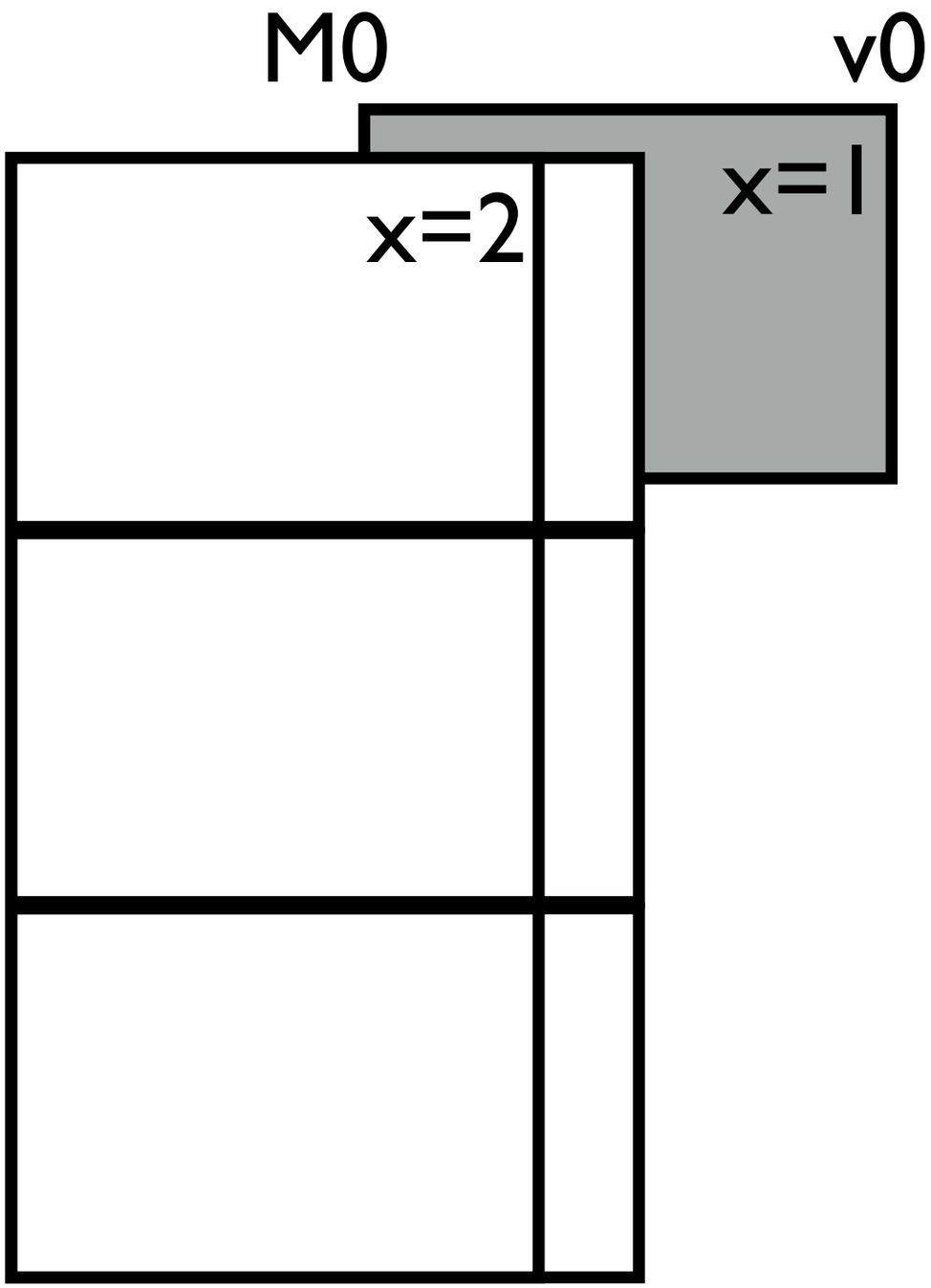


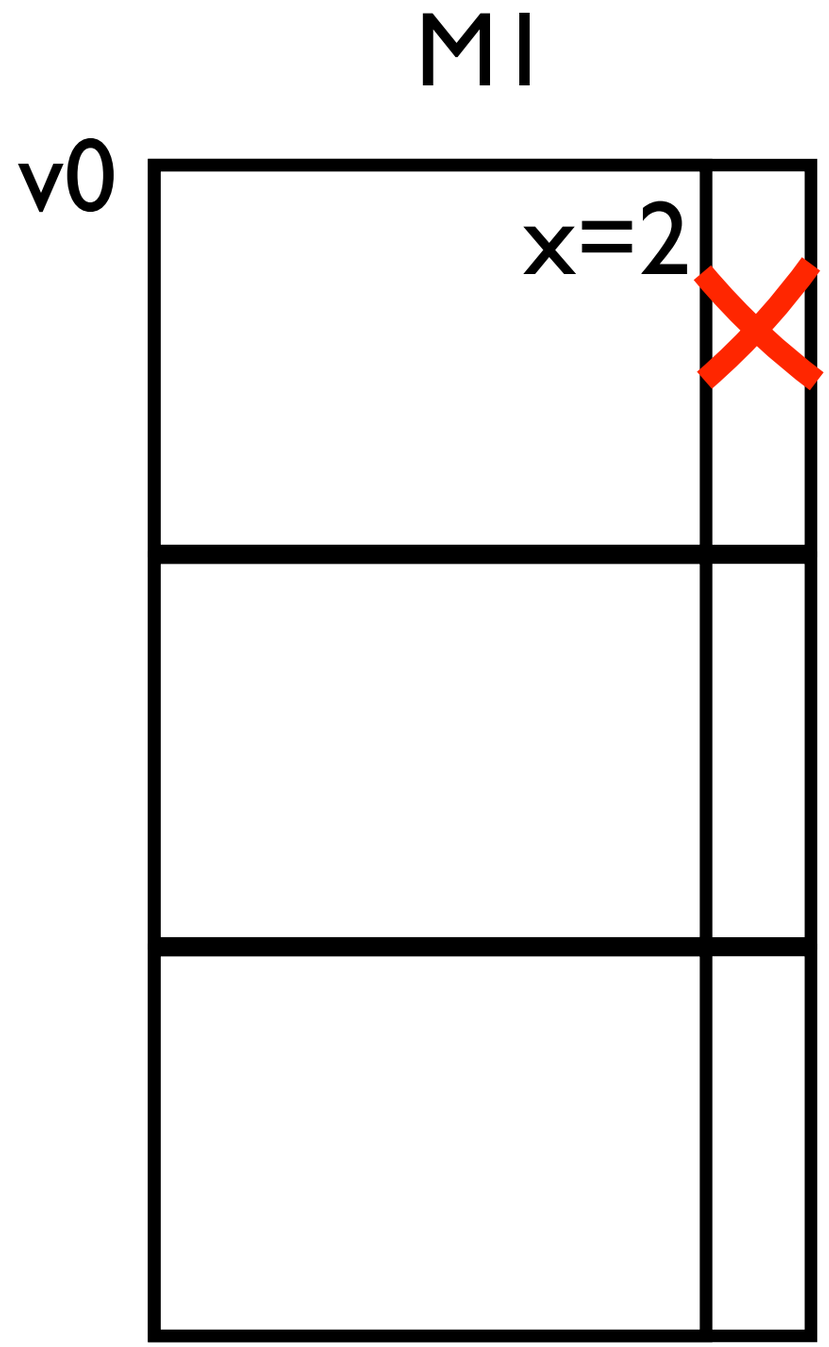
8: send diff to M1



9: Apply Diff

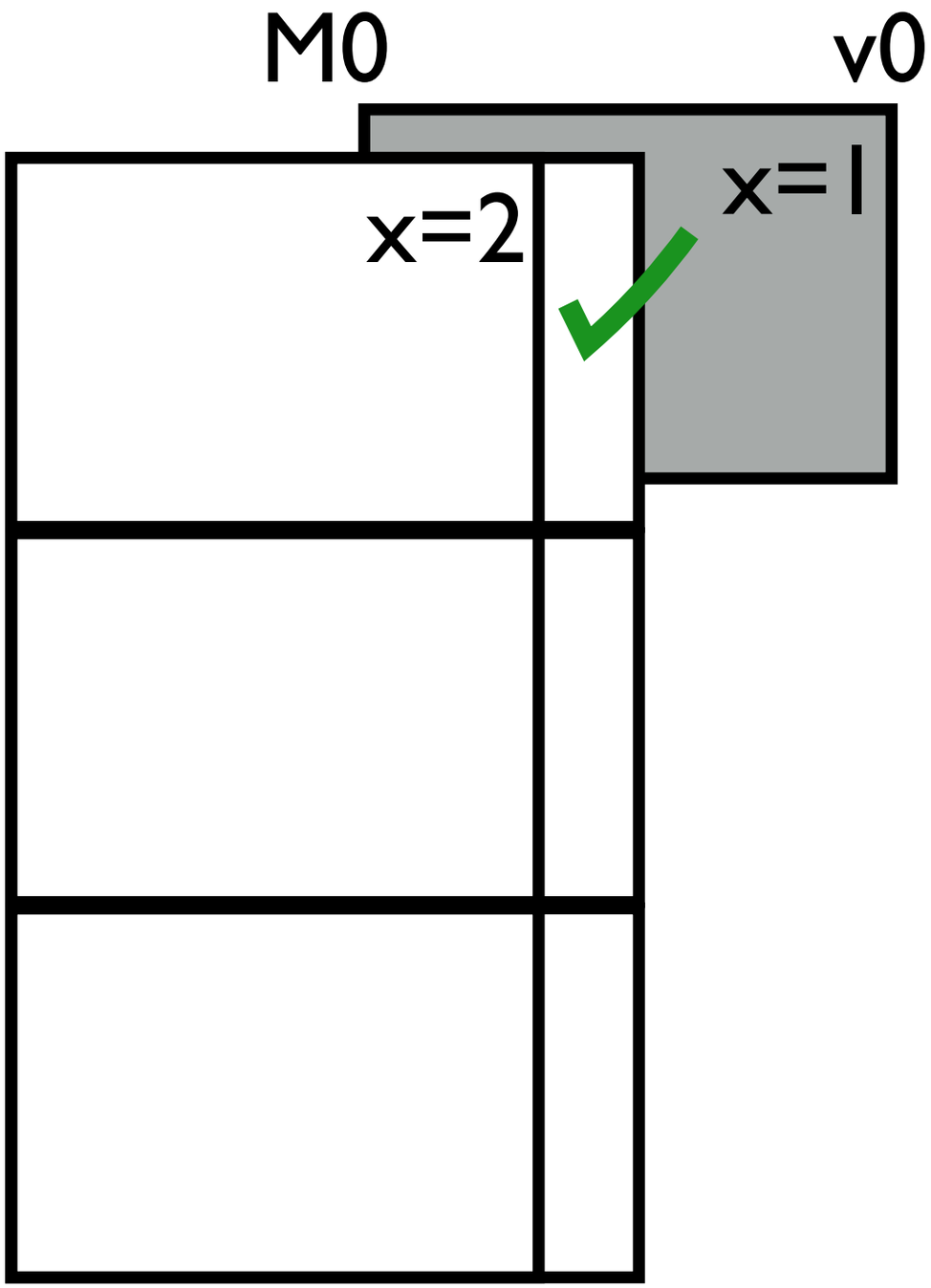
M1: print x

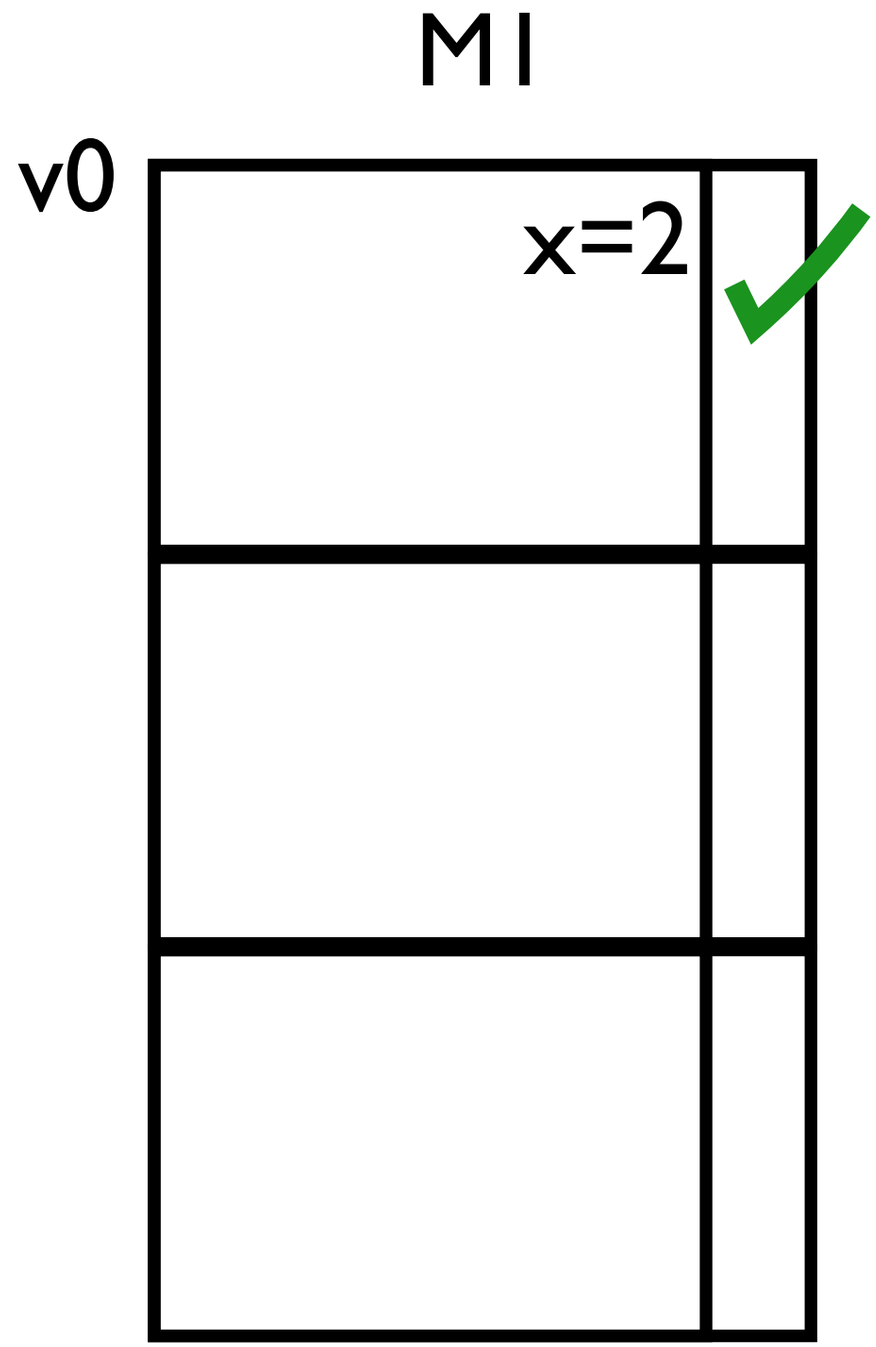




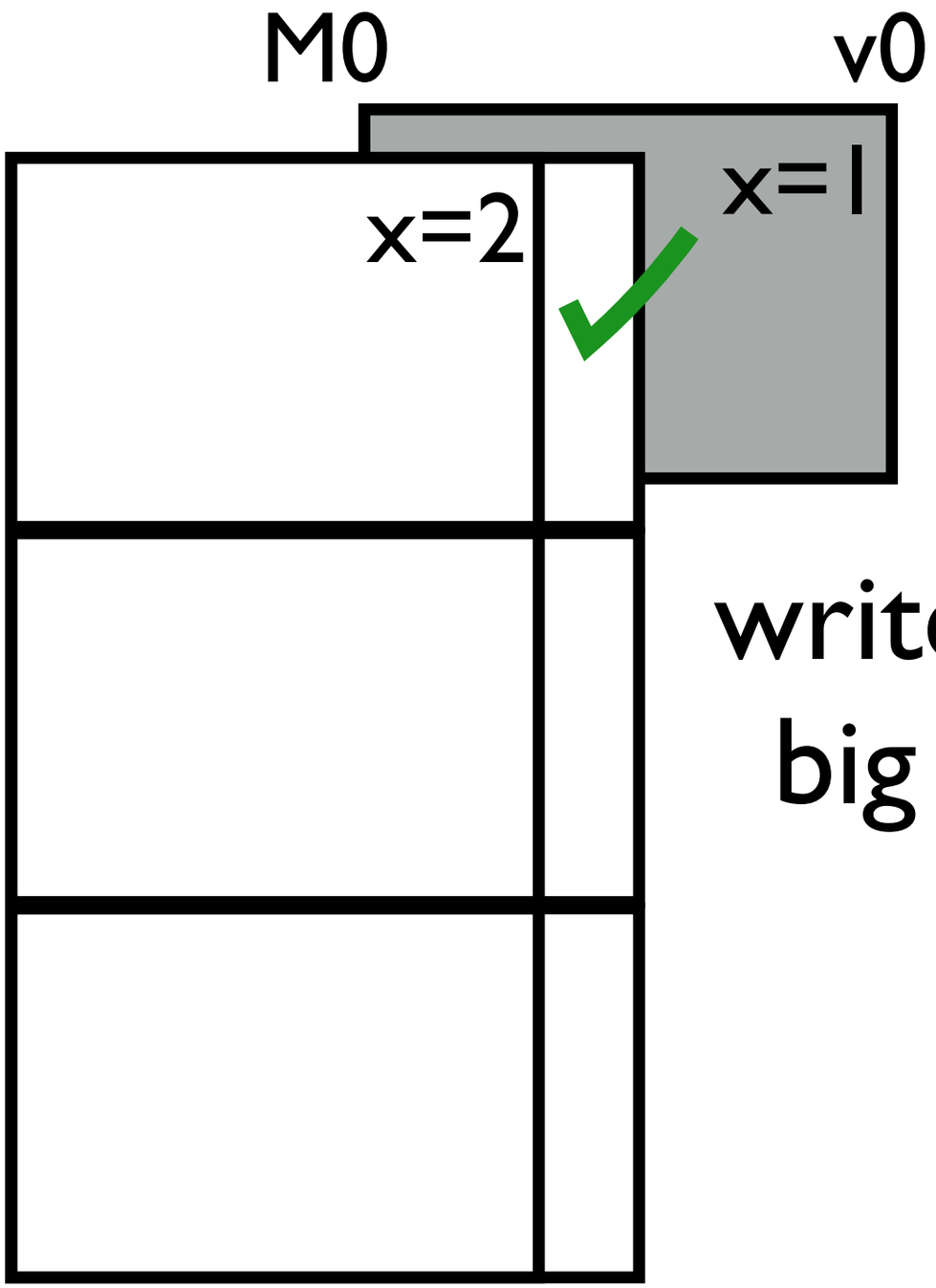
10: Both M0 and M1 proceed with RO mapping

M1: print x



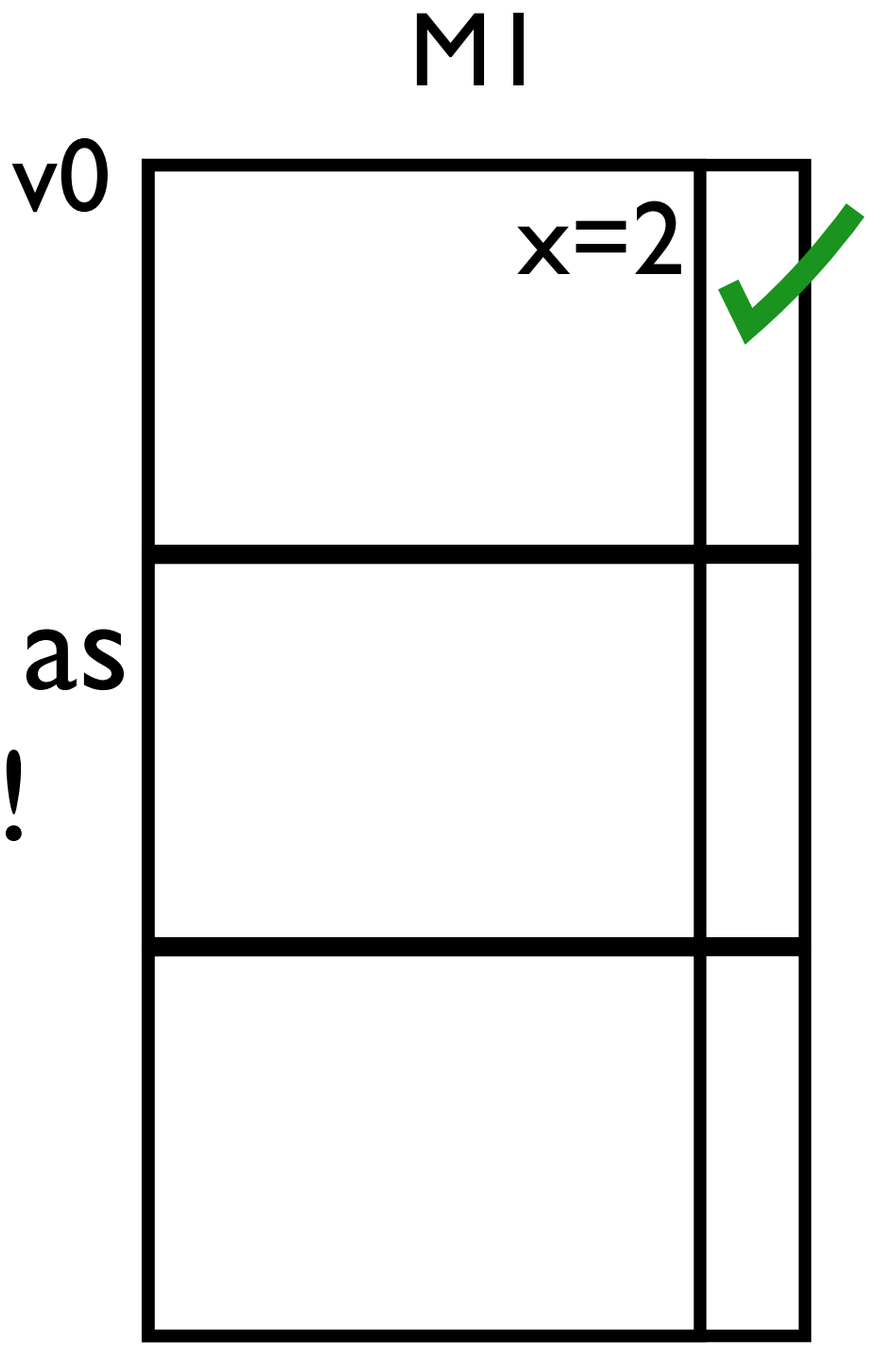


Write Amplification FIXED



writes are only as big as changes!

x=2



do write diffs change
the consistency model?

do write diffs change the consistency model?

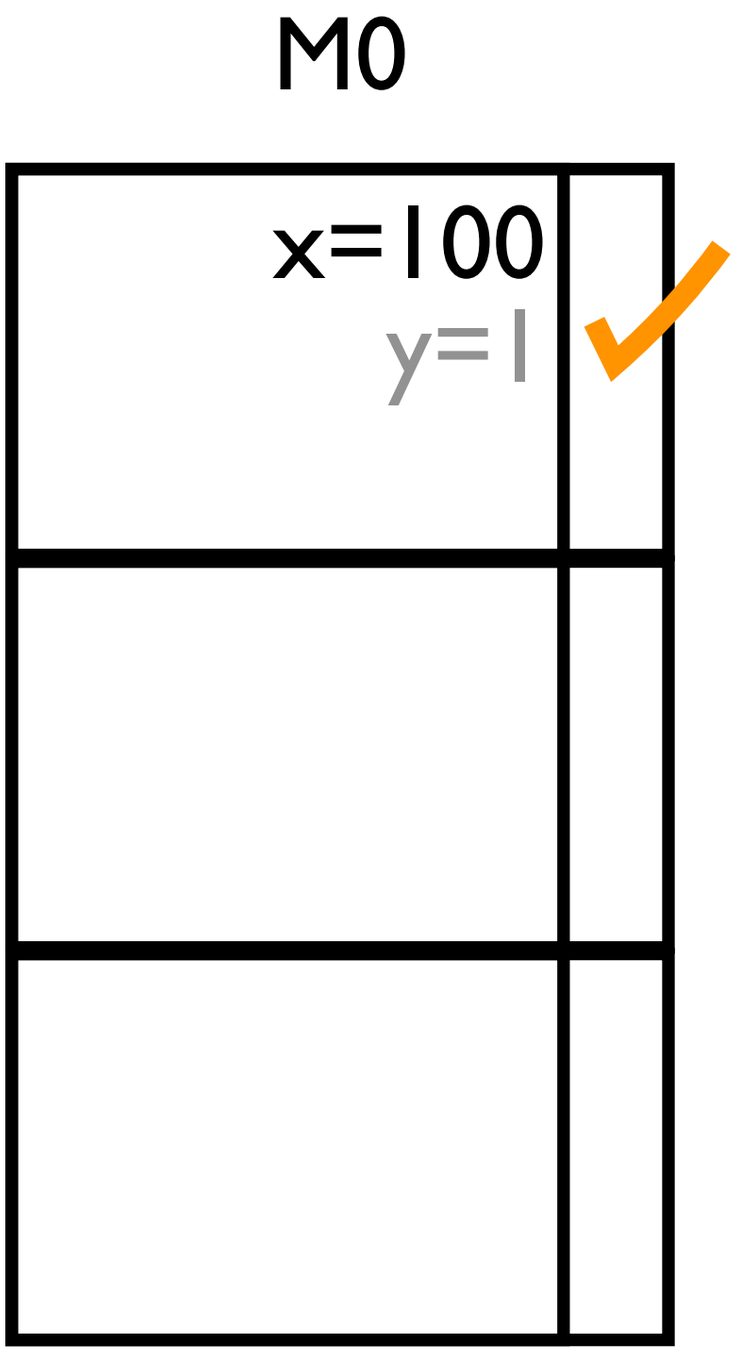
At most one writeable copy, so writes are ordered
No writing while any copy is readable, so no stale reads
Readable copies are up to date, so no stale reads

No

False Sharing

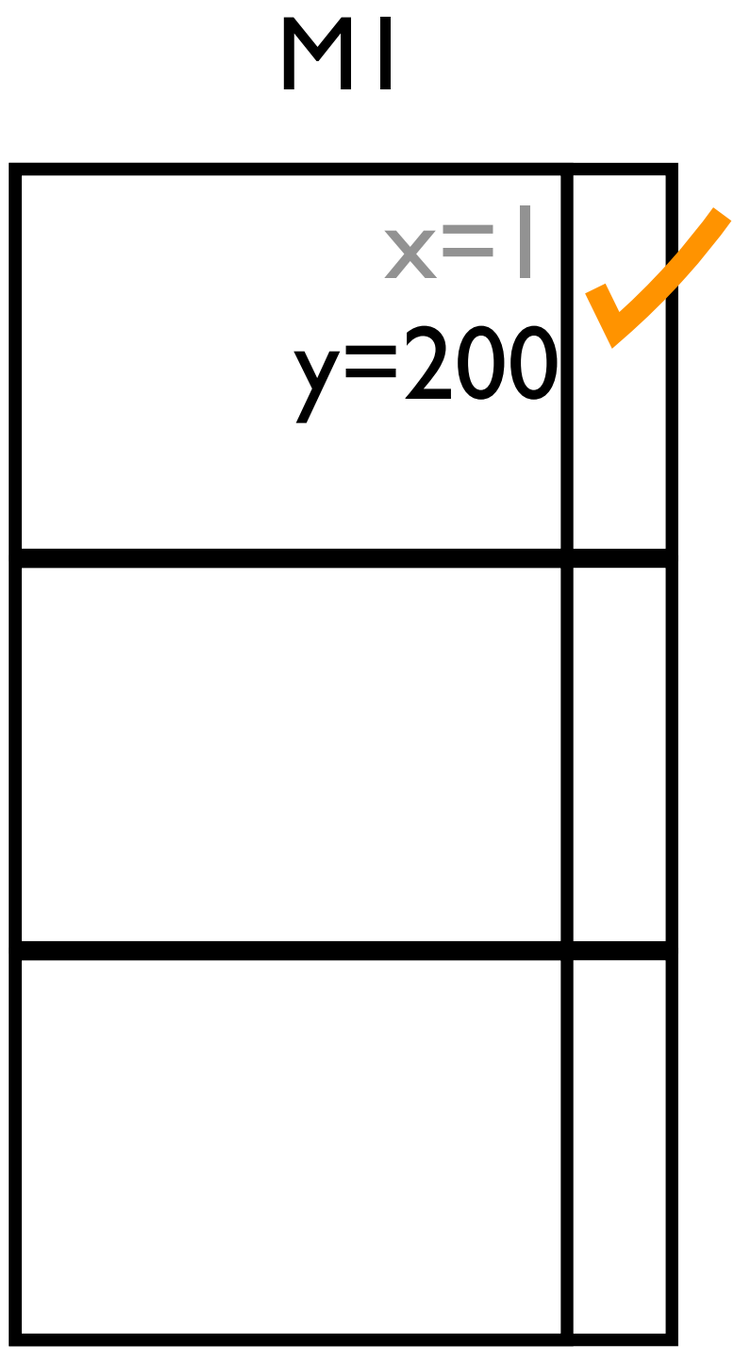
- So far an invariant of the system only single write copy
  - Multiple read copies ok but only one write
  - This ensure a strict ordering — sequential consistency
- Can we do something?
  - Can we let two machines write to the same page — have write copies?
  - write diffs will be useful but not the whole solution — allow us to track local changes

False Sharing



Some how can we have each machine have r/w access to the pages and only send around the diffs of the changes they make as needed?

eg. Fix false sharing



But… real code uses locks to coordinate updates

M0

x=100

y=1

M0:

a.lock()

x = 100

a.unlock()

M1:

b.lock()

y=200

b.unlock()

M1

x=1

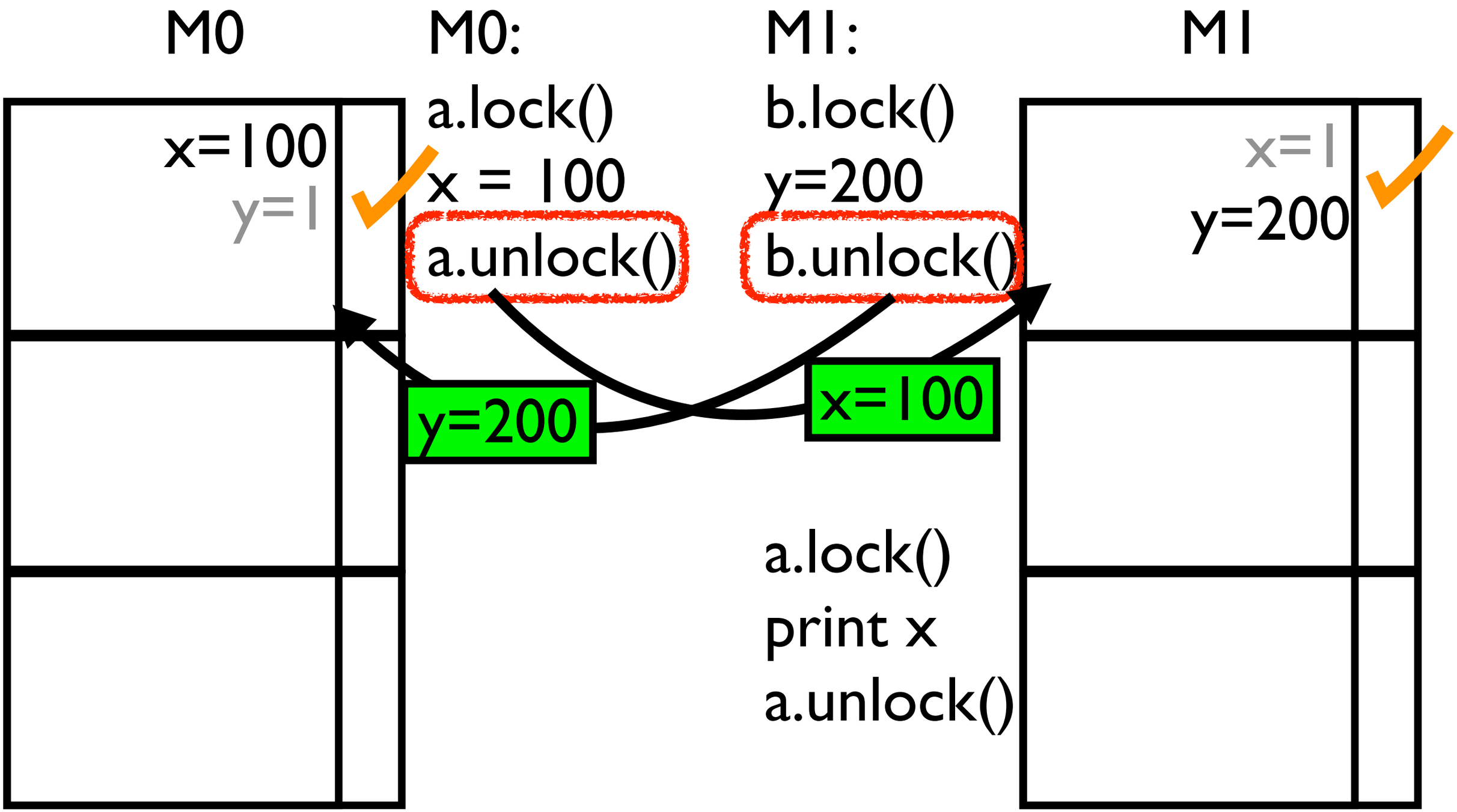
y=200

a.lock()

print x

a.unlock()

Lock Releases
used to track and communicate changes



Work happens at the time of Lock Release

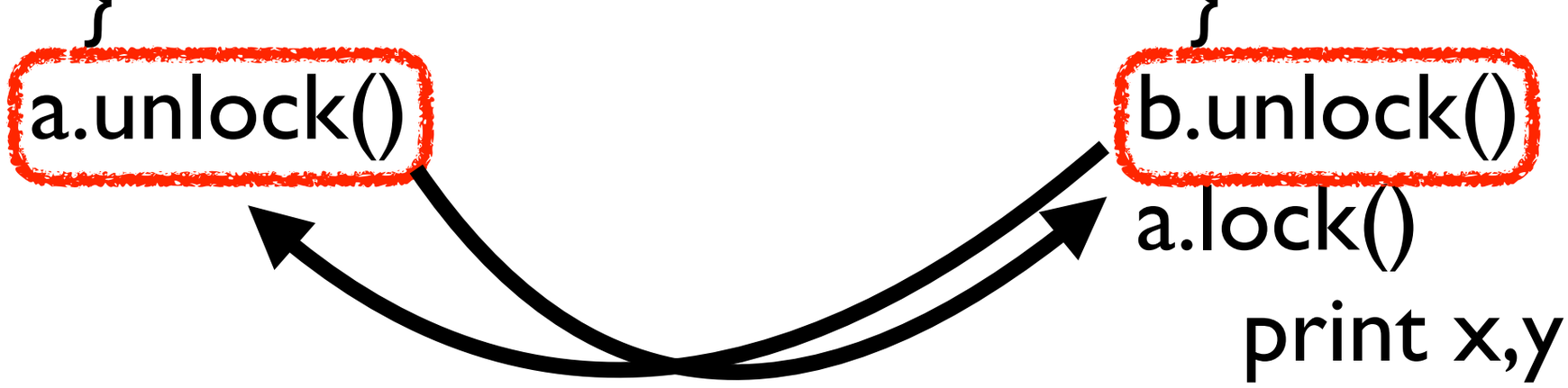
Assume x and y on the SAME page

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()
```



On Releases — unlocks

1) diff of ALL local changes since LAST RELEASE
2) sends diffs to all other machines
3) All machines with copies of those pages will apply
4) Then unlock will finish

Huge Perf Benefit —

no Ping Pong of False Sharing

Is the Programming Model different than basic DSM?

YES! In basic DSM you can see all writes — roughly a read will see last write to any memory

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)

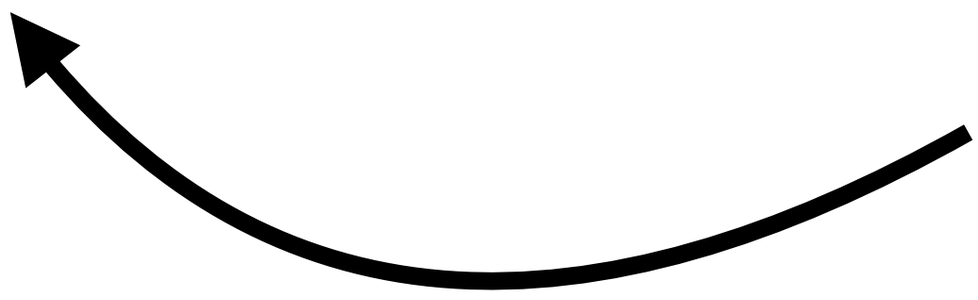
Lazy Release Consistency (LRC)

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