CS162:
Operating Systems and Systems Programming

Lecture 7:
Synchronization: Lock Implementation, Higher-Level Synchronization

1 July 2015

Charles Reiss
http://cs162.eecs.berkeley.edu/

Recall: Which scheduler should you use?

I care more than anything else about...

- *CPU Throughput*: First-Come First-Served
- *Average Response Time*: SRTF approximation
- *I/O Throughput*: SRTF approximation
- *Fairness – long-term CPU*: something like Linux CFS
- *Fairness – wait time for CPU*: something like RR
- *Meeting deadlines*: Earliest Deadline First
- *Favoring important users*: Strict Priority

Recall: Locks

Recall: Locks

Alice

```
doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {



  buy milk
}
MilkLock.Release()
```

Time

Bob

Critical section: only one thread can enter at a time

```
doHomework()
watchTV()
start MilkLock.Acquire()



finish MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
```

Recall: Locks

Alice

```
doHomework()
watchTV()
MilkLock.Acquire()
if (noMilk) {
```

Bob

"Holding" lock section doesn't prevent context switches. Just stops other threads from Acquiring it.

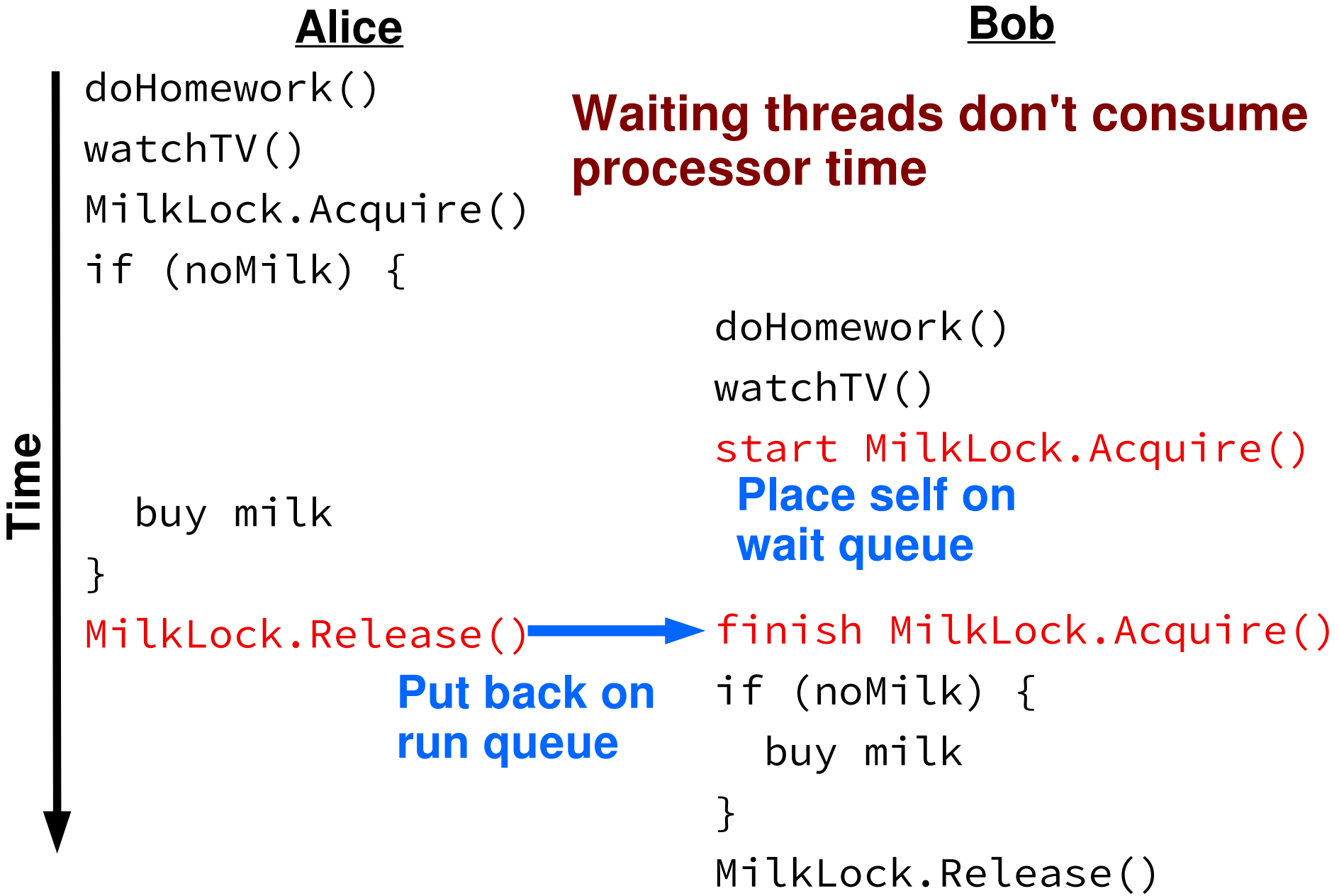
```
doHomework()
watchTV()
start MilkLock.Acquire()
```

```
  buy milk
}
MilkLock.Release()
```

```
finish MilkLock.Acquire()
if (noMilk) {
  buy milk
}
MilkLock.Release()
```

Time

Recall: Locks



Recall: Disabling Interrupts

Critical sections in the kernel

On a single processor, nothing else can run

- Primitive critical section!

Build other properties of lock on top of this

- not excluding *everything else*
- putting thread to sleep and letting other threads run

Recall: Implementing Locks: Single Core

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone waiting) {
    take a thread off queue
  } else {
    Value = FREE;
  }
  enable interrupts;
}
```

Idea: disable interrupts for **mutual exclusion** on accesses to **`value`**

Multiprocessors and Locks

Disabling interrupts doesn't prevent other processor from doing anything

Solution: Hardware support for **atomic operations**

Recall: Atomic Operations

Definition: *an operation that runs to completion or not at all*

- Need some to allow threads to work together

Example: loading or storing words

Some **instructions not atomic**

- e.g. double-precision floating point store (many platforms)

Outline

Hardware support for locks and spin locks

Semaphores

Monitors

Outline

Hardware support for locks and spin locks

Semaphores

Monitors

Atomic Read/Modify/Write

Recall: atomic load/store not good enough

Hardware instructions (or instruction sequences) that **atomically** read a value from (shared) memory and write a new value

Hardware responsible for making work in spite of caches

Read-Modify-Write Instructions

- ```
  test&set (&address) {   /* most architectures */
        result = M[address];
        M[address] = 1;
        return result;
  }
  ```
- ```
  swap (&address, register) { /* x86 */
        temp = M[address];
        M[address] = register;
        register = temp;
  }
  ```
- ```
  compare&swap (&address, reg1, reg2) { /* 68000, x86-64 */
        if (reg1 == M[address]) {
           M[address] = reg2;
           return success;
        } else {
           return failure;
        }
  }
  ```
- ```
  load-linked&store conditional(&address) {
        /* MIPS R4000, alpha */
      loop:
           ll r1, M[address];
           movi r2, 1;      /* Can do arbitrary comp */
           sc r2, M[address];
           beqz r2, loop;
  }
  ```

Locks with test&set

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {
      // do nothing
   }
}

Release() {
   value = 0;
}
```

```
test&set (&address) {
      result = M[address];
      M[address] = 1;
      return result;
}
```

Lock free: test&set reads 0, set value to 1, returns 0 (old value) → acquire finishes

Lock not free: test&set reads 1, sets value to 1, returns 1 (old value) → acquire **waits**

Locks with test&set: Busy waiting

```
int value = 0; // Free

Acquire() {
   while (test&set(value)) {
      // do nothing
   }
}

Release() {
   value = 0;
}
```

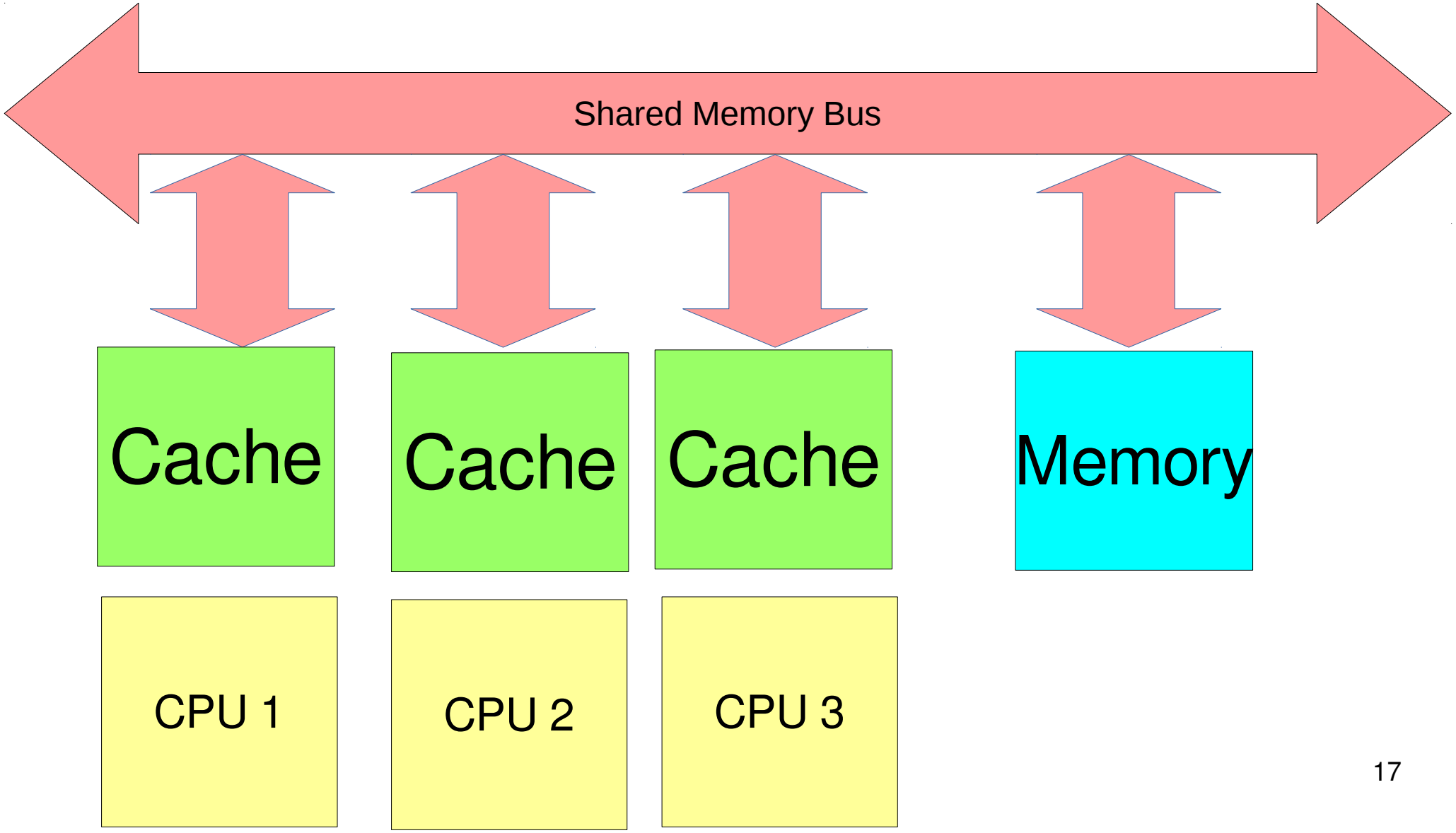
```
test&set (&address) {
      result = M[address];
      M[address] = 1;
      return result;
}
```

Busy-waiting: consumes CPU time while waiting

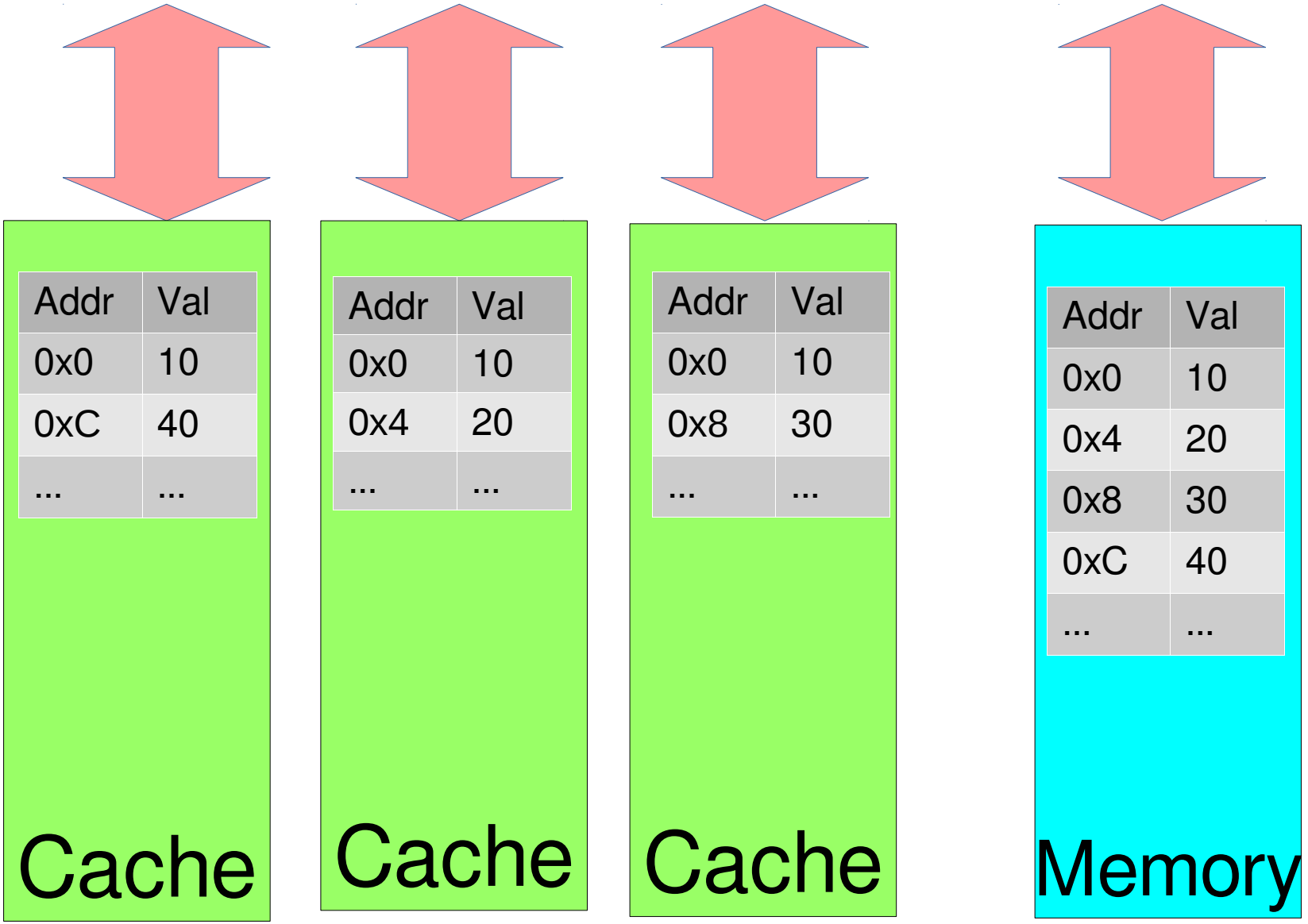
- Keeps other threads from using CPU
- Maybe even the thread holding the lock!

These are called "**spin locks**" (because they spin while busy)

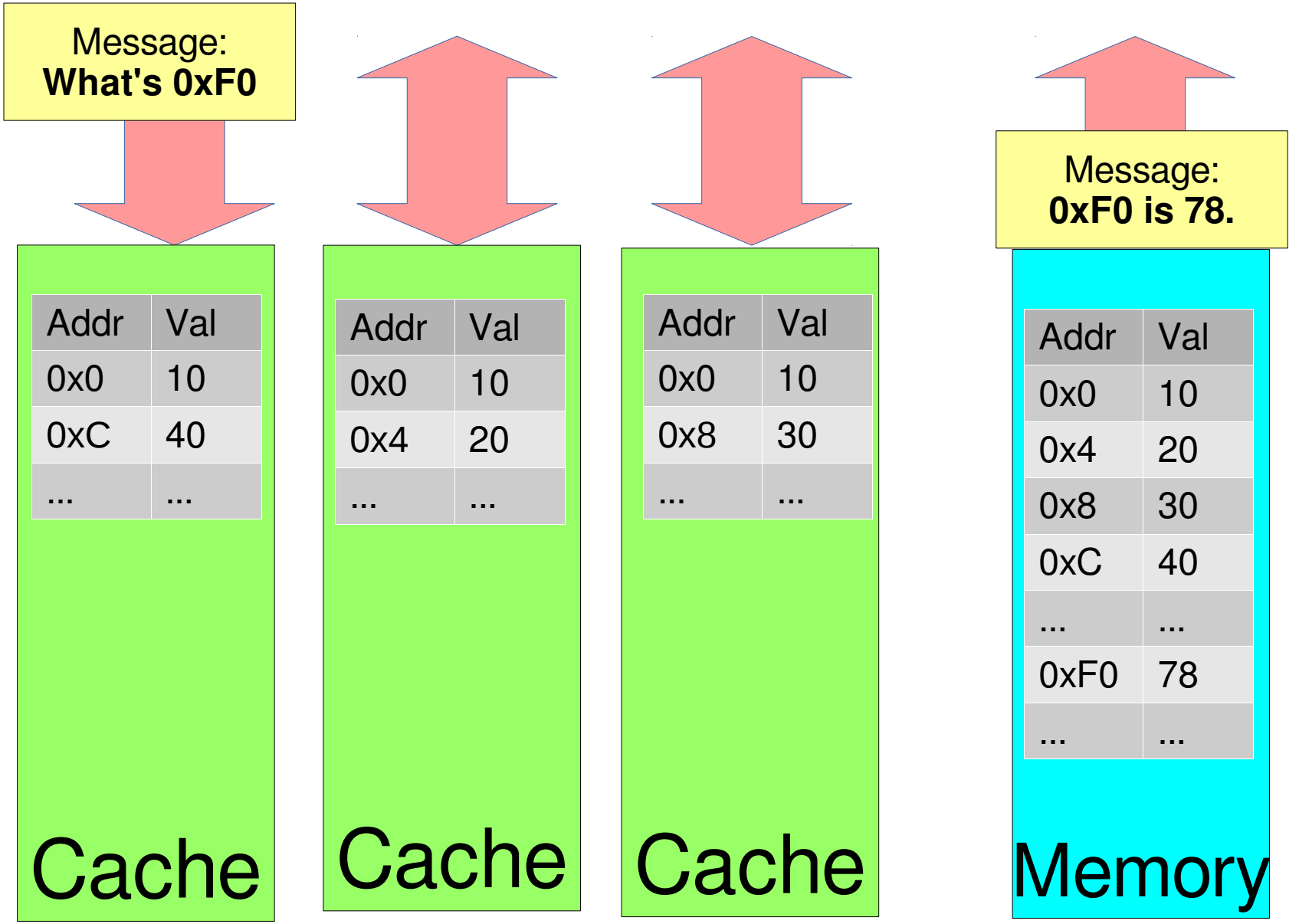
Communicating Between Cores



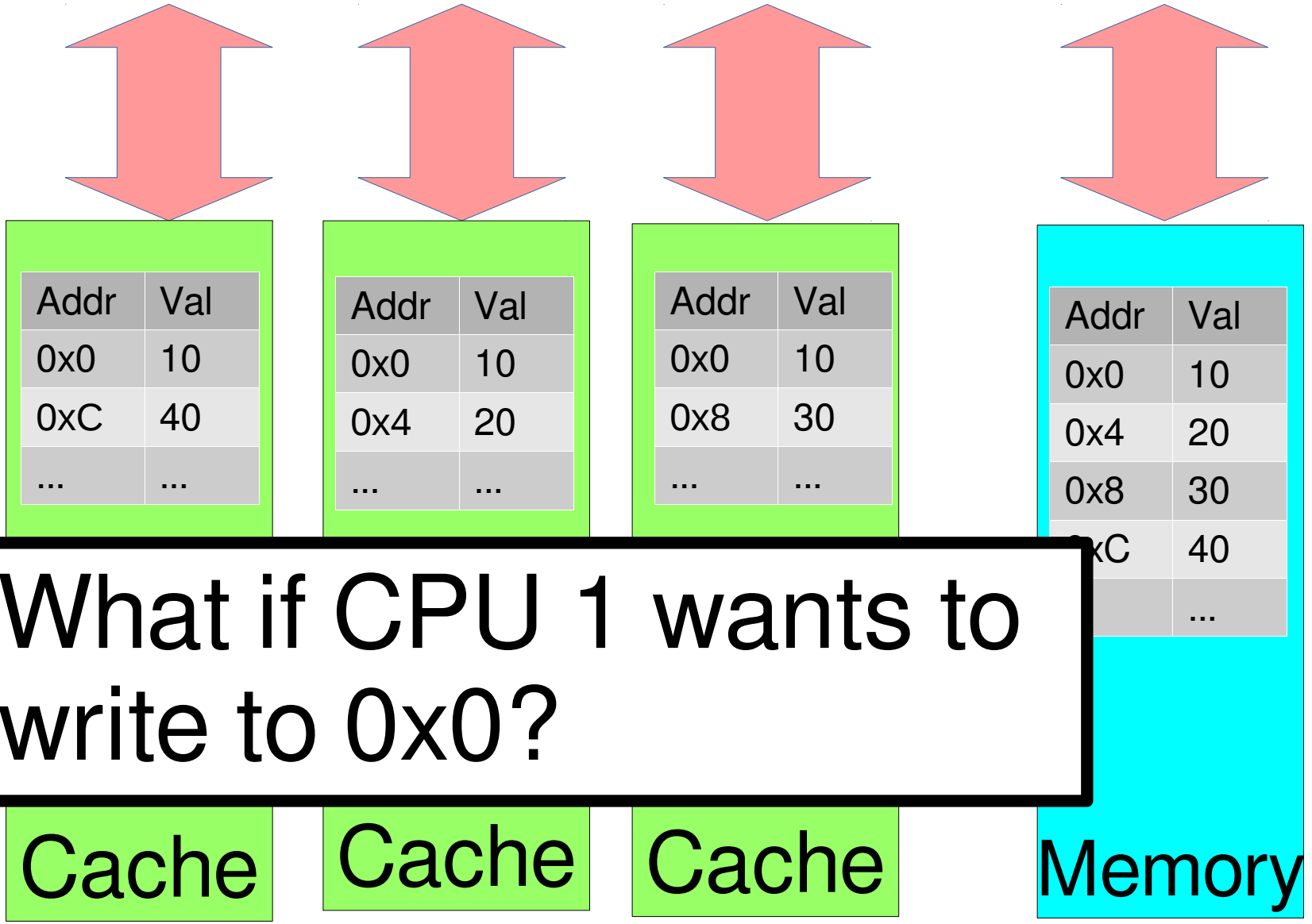
Coherency


Addr
Val
0x0
10
0xC
40
...
...
Cache
Addr
Val
0x0
10
0x4
20
...
...
Cache
Addr
Val
0x0
10
0x8
30
...
...
Cache
Addr
Val
0x0
10
0x4
20
0x8
30
0xC
40
...
...
Memory

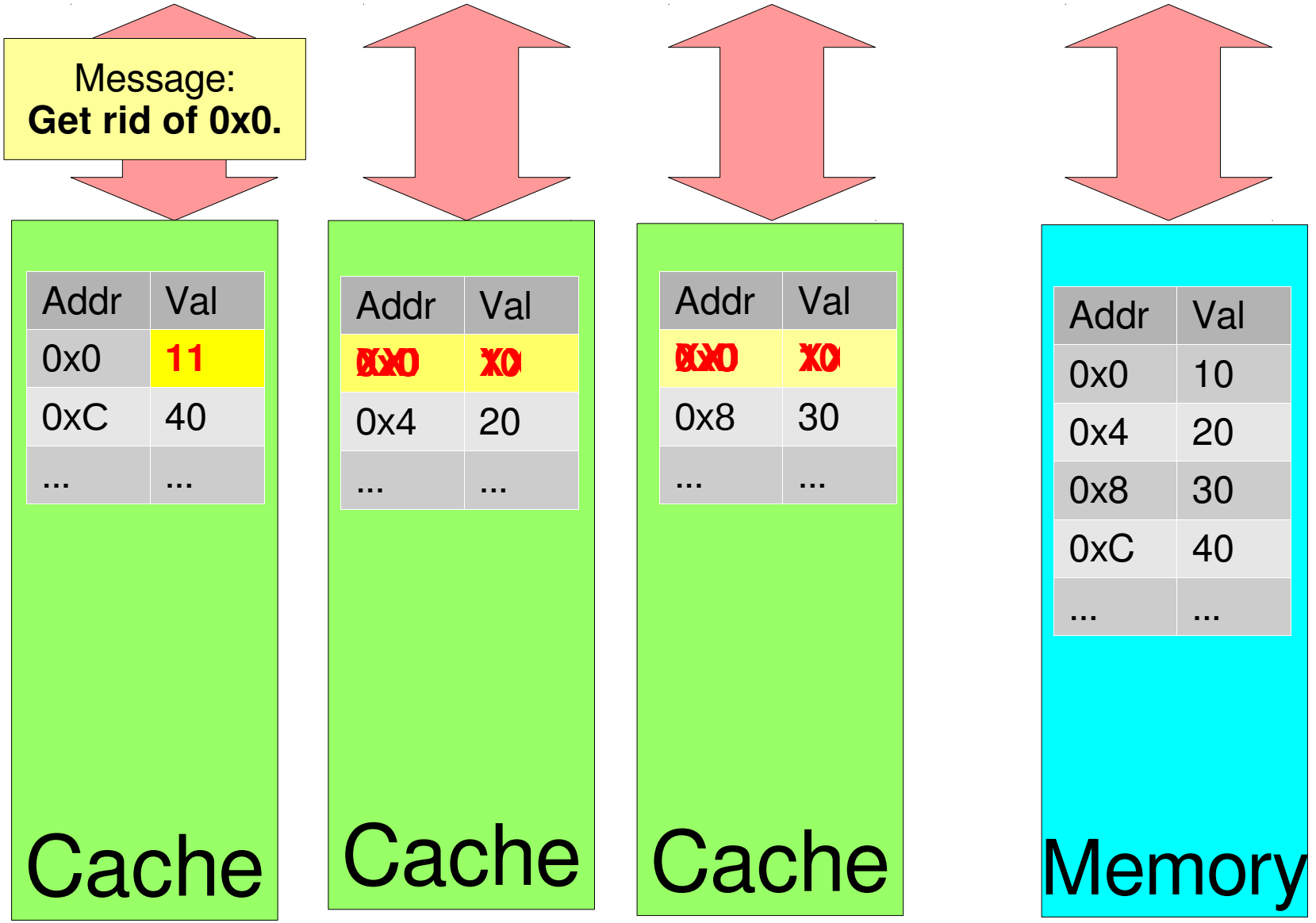
Coherency: Reading Normally



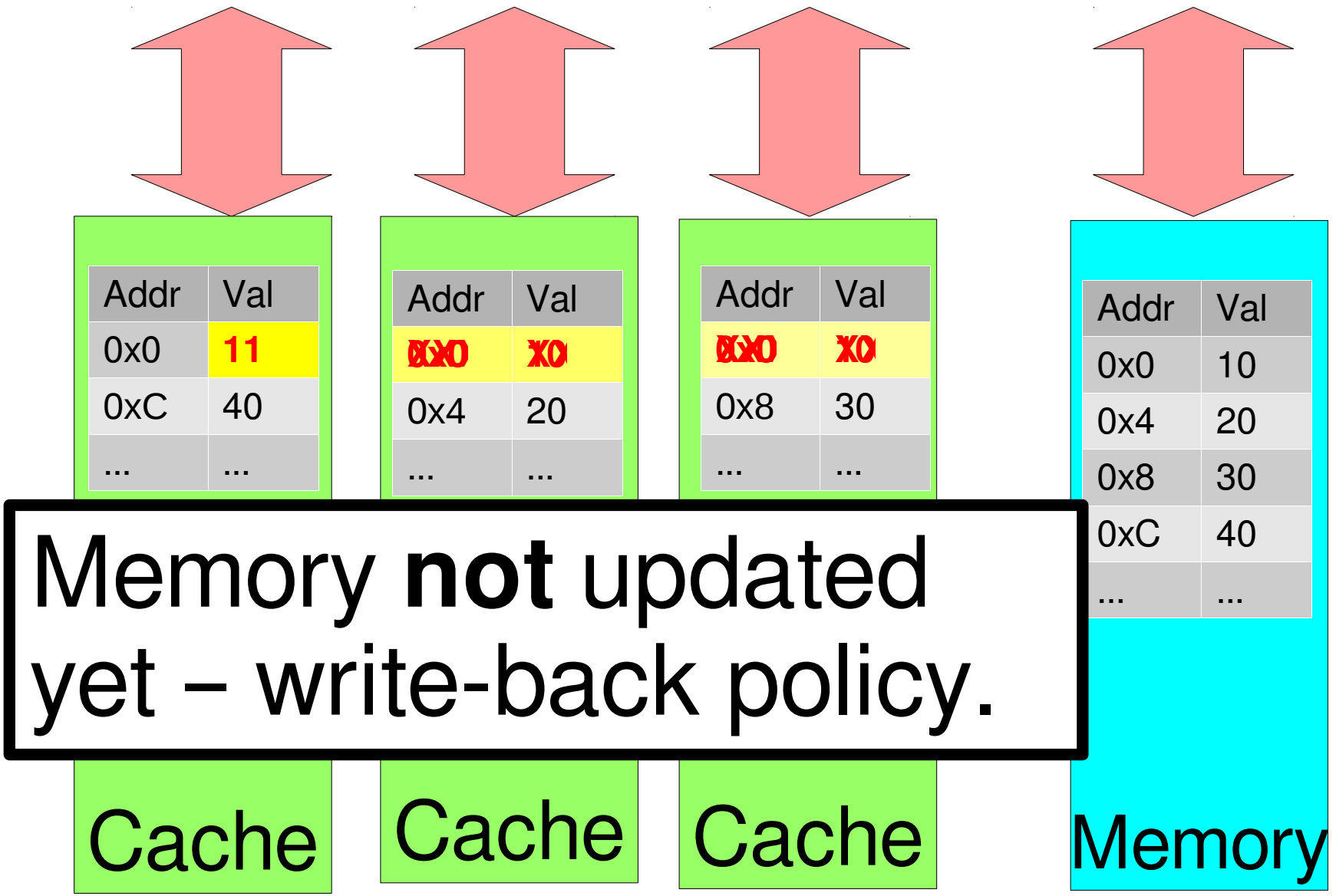
Coherency


Addr Val
0x0 10
0xC 40
...
Addr Val
0x0 10
0x4 20
...
Addr Val
0x0 10
0x8 30
...
Addr Val
0x0 10
0x4 20
0x8 30
...
What if CPU 1 wants to write to 0x0?
Cache
Cache
Cache
Memory

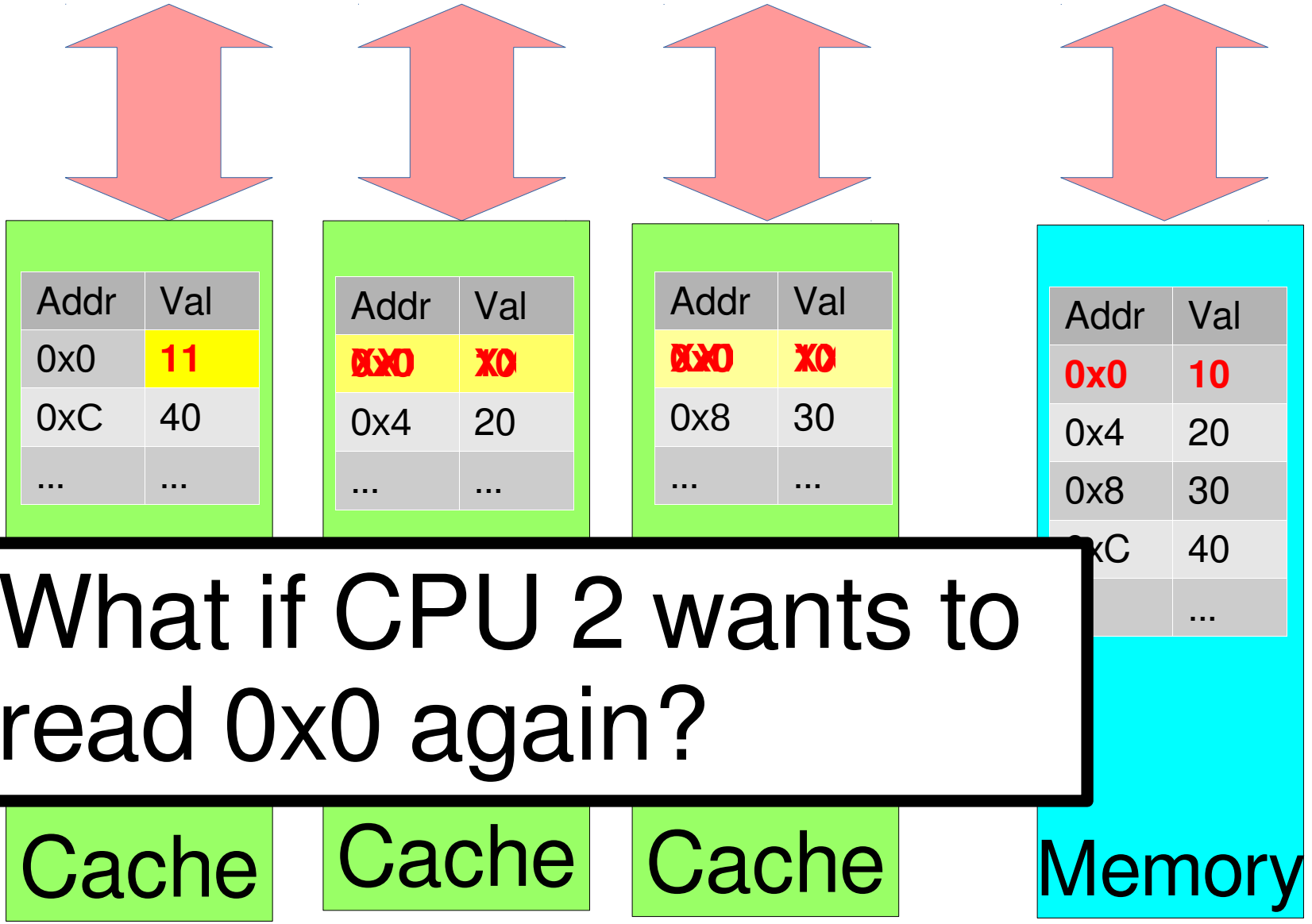
Coherency: Invalidate



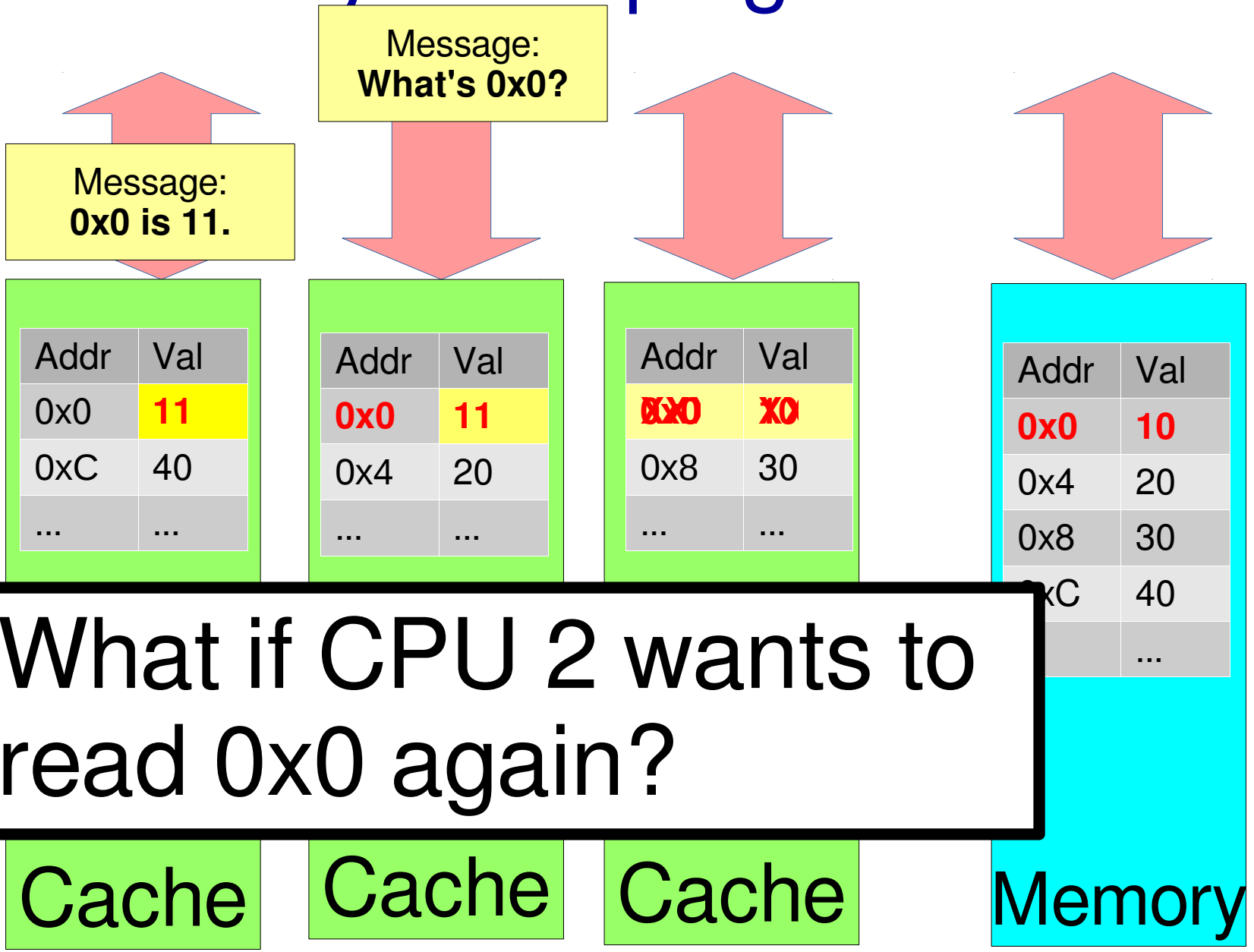
Coherency: Invalidate



Coherency: Snooping



Coherency: Snooping



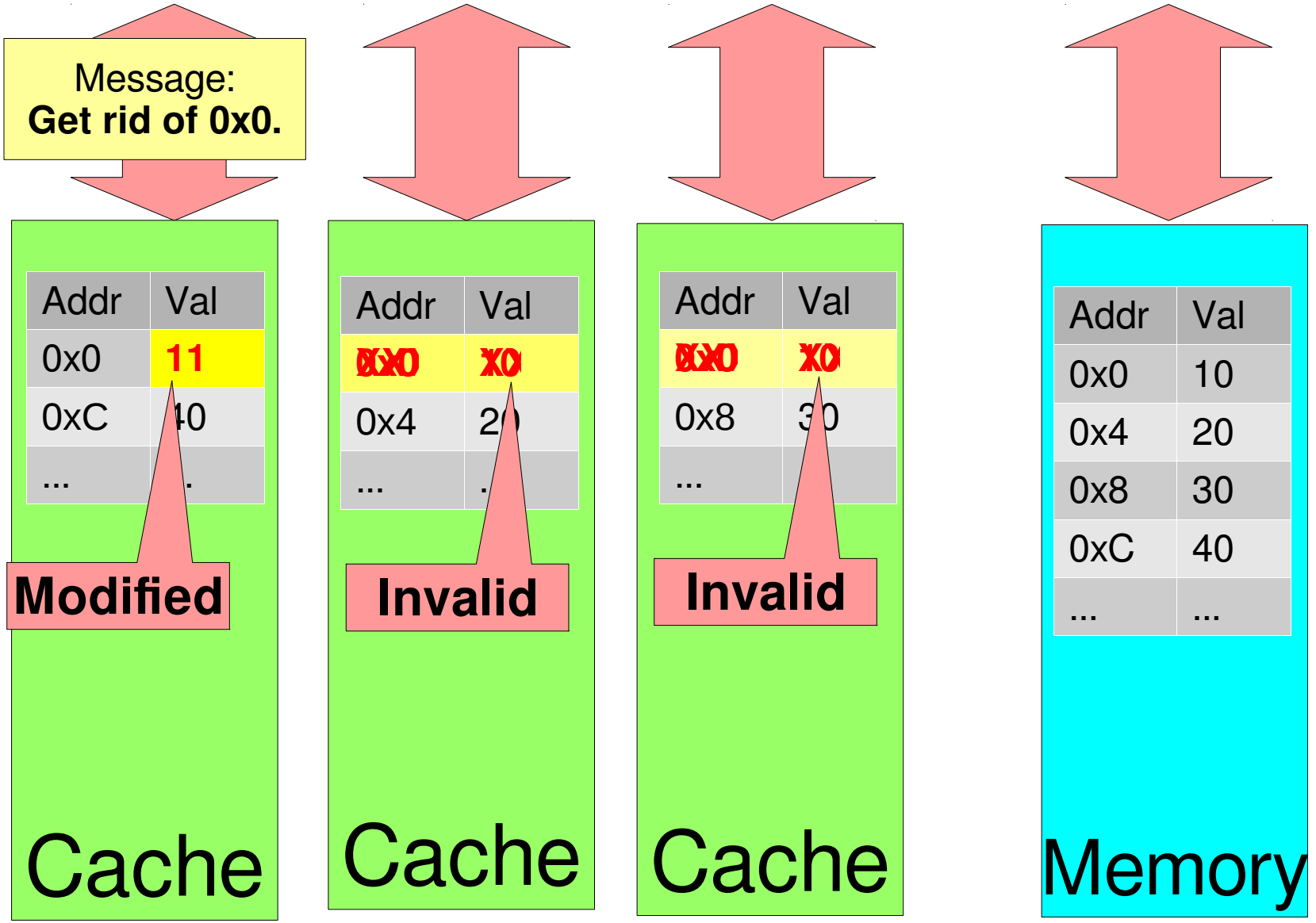
Coherency: States

Caches need to remember if anyone else needs a copy of their items

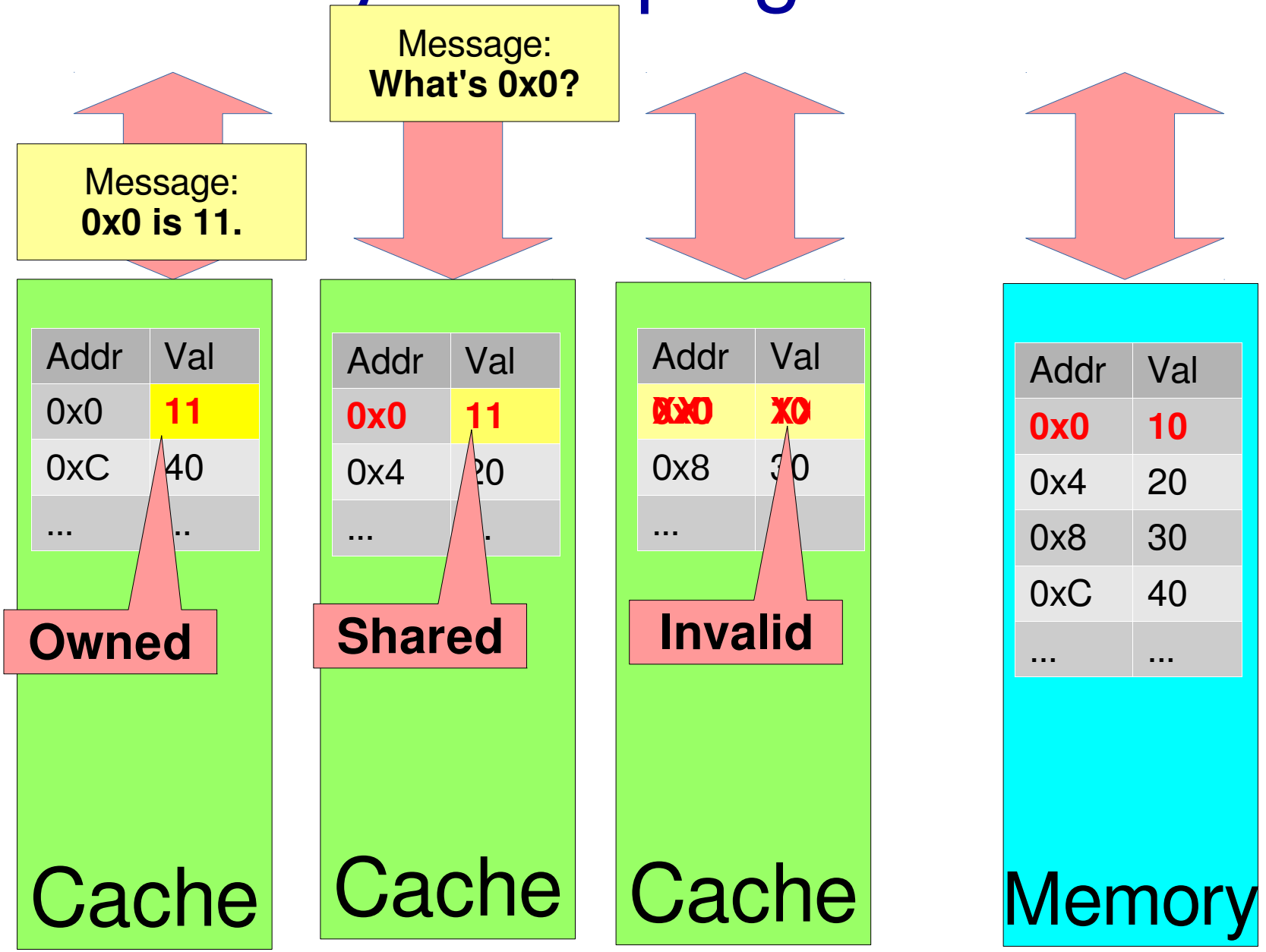
Simple protocol – four states:

- **Modified –** My value is more recent than memory, and I'm the only one who has it
- **Owned** – My value is more recent than memory, but other processors have it
- **Shared** – My value is unmodified (from owner or memory) and most recent
- **Invalid –** My value is junk; don't use it

Coherency: Invalidate



Coherency: Snooping



A Note on Memory Traffic

`test&set` requires communication!

- Local cache needs to "own" the memory address to write to it

Means `while(test&set(value));` spams the memory bus!

- Two processors waiting: ownership flips back and forth
- Hurts the performance of **all processors**

Test and Test and Set

Solution to memory traffic problem:

```
int mylock = 0; // Free

Acquire() {
    do {
        while(mylock);   // Wait until might be free
    } while(test&set(&mylock)); // exit if get lock
}


Release() {
    mylock = 0;
}
```

`while (mylock)` stays in local cache (**shared** state)

- waits for cache invalidation **from lock holder**

Recall: Disable Interrupts

```
int value = FREE;

Acquire() {
  disable interrupts;
  if (value == BUSY) {
    put thread on wait queue;
    run_new_thread()
    // Enable interrupts?
  } else {
    value = BUSY;
  }
  enable interrupts;
}
```

```
Release() {
  disable interrupts;
  if (anyone waiting) {
    take a thread off queue
  } else {
    Value = FREE;
  }
  enable interrupts;
}
```

Idea: disable interrupts for **mutual exclusion** on accesses to **`value`**

Locks with test&set

Use "spinlock" to build better locks

- Like we use disabling interrupts to build "proper" locks

```
int guard = 0;
int value = FREE;
```

```
Acquire() {
   // Short busy-wait time
   while (test&set(guard));
   if (value == BUSY) {
      put thread on wait queue;
      run_new_thread() & guard = 0;
   } else {
      value = BUSY;
      guard = 0;
   }
}
```

```
Release() {
   // Short busy-wait time
   while (test&set(guard));
   if anyone on wait queue {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   guard = 0;
```

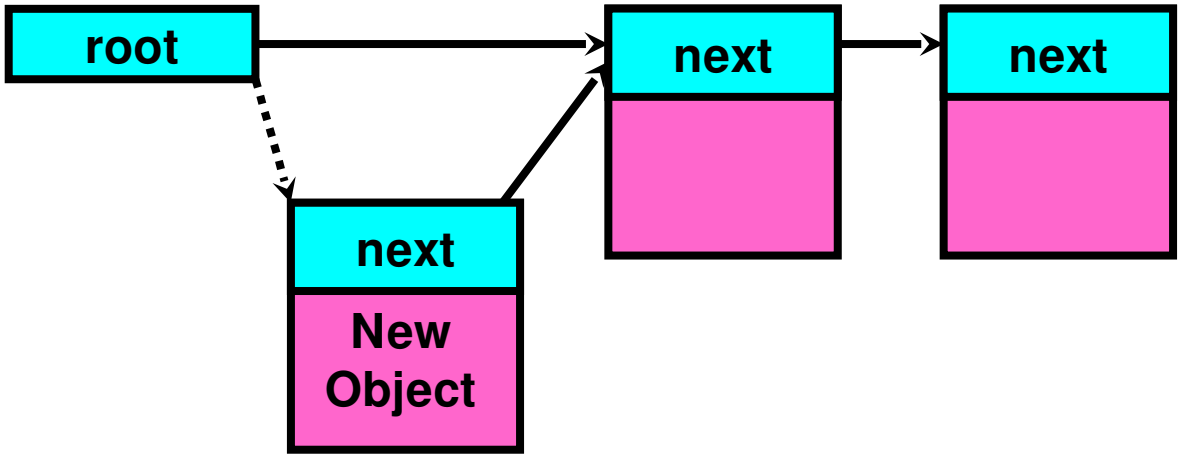
Same problem with releasing lock as enabling interrupts before → need help from scheduler or spurious yield

Queues with Compare&Swap

```
compare&swap (&address, reg1, reg2) {
    if (reg1 == M[address]) {
        M[address] = reg2;
        return success;
    } else {
        return failure;
    }
}
```

Atomic add to linked list – without a lock:

```
addToQueue(&object) {
    do {       // repeat until no conflict
        ld r1, M[root]   // Get ptr to current head
        st r1, M[object]  // Save link in new object
    } until (compare&swap(&root,r1,object));
}
```



Outline

Hardware support for locks and spin locks

Semaphores

Monitors

Semaphores

Generalized lock

Definition: has a non-negative integer value and two operations:

- **P()** or **down** or **wait:** *atomically* wait for semaphore to become positive, then decrement it by 1
- **V()** or **up** or **signal:** *atomically* increment semaphore by 1 (waking up a waiting P() thread)

P, V from Dutch: *proberen* (test), *verhogen* (increment)

Semaphore Like Integers But...

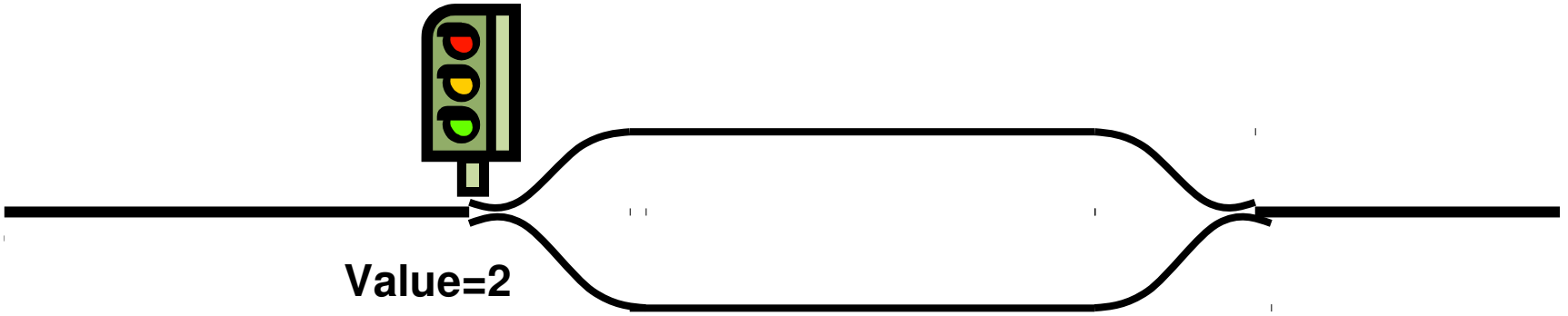
Cannot read/write value directly

- Down (P)/up (V) only
- Exception: initialization

Never negative – something waits instead

- Two down operations can't go below 0 → some thread "wins"

Railway Analogy



Simple Semaphore Patterns

Mutual exclusion: Same as lock (earlier)

- "Binary semaphore"

```
Initial value of semaphore = 1

semaphore.P();
// Critical section goes here
semaphore.V();
```

Signaling other threads, e.g. **ThreadJoin**:

```
Initial value of semaphore = 0

ThreadJoin() {
    semaphore.P();
}

ThreadFinish() {
    semaphore.V();
}
```

Intuition for Semaphores

What do you need to wait for?

- Example: critical section to be finished
- Example: queue to be non-empty
- Example: array to have space for new items

What can you count that will be 0 when you need to wait?

- Ex: # of threads that can start critical section now
- Ex: # of items in queue
- Ex: # of empty spaces in array

Then: make sure semaphore operations maintain count

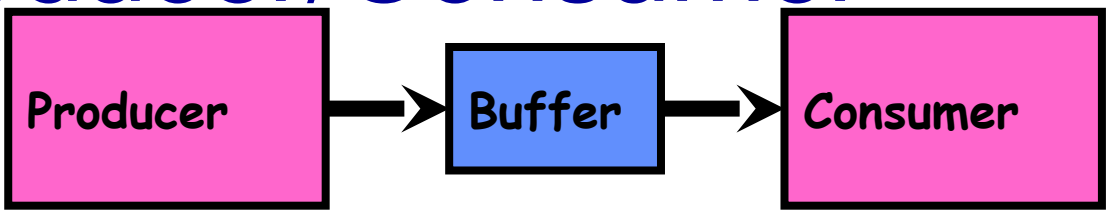
Higher-Level Synchronization

Want to make synchronization convenient, correct

Semaphores first solution (1963)

First-class support for **waiting for a condition to become true**

Example: Producer/Consumer



Shared buffer (queue) – fixed size

- Producer inserts items
- Consumer removes items

Producer/consumer don't need to work in lockstep

Example: C compiler

- preprocessor → compiler → assembler → linker

Producer/Consumer Correctness

Scheduling constraints:

- Consumer waits for producer if buffer empty
- Producer waits for consumer if buffer full

Mutual exclusion constraint:

- Only one thread manipulates buffer at a time

One semaphore per constraint:

- `Semaphore fullBuffers; // consumer's constraint`
- `Semaphore emptyBuffers;// producer's constraint`
- `Semaphore mutex;       // mutual exclusion`

Producer/Consumer Code

```
Semaphore fullBuffer = 0;  // Initially, buffer empty
Semaphore emptyBuffers = numBuffers;
    // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
   emptyBuffers.P(); // Wait until space
   mutex.P();  // Wait until buffer free
   Enqueue(item);
   mutex.V();
   fullBuffers.V();  // Tell consumers there is more data.
}
Consumer() {
   fullBuffers.P();  // Check if there's an item
   mutex.P();  // Wait until buffer free
   item = Dequeue();
   mutex.V();
   emptyBuffers.V(); // tell producer need more
   return item;
}
```

Producer/Consumer Code

```
Semaphore fullBuffer = 0;  // Initially, buffer empty
Semaphore emptyBuffers = numBuffers;
    // Initially, num empty slots
Semaphore mutex = 1; //

Producer(item) {
    emptyBuffers.P(); //
    mutex.P();  // Wait
    Enqueue(item);
    mutex.V();
    fullBuffers.V();  // Tell consumers there is more data.
}
Consumer() {
    fullBuffers.P();  // Check if there's an item
    mutex.P();  // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

fullBuffers always <= number of items on queue

emptyBuffers always <= number of free slots on queue

Producer/Consumer Code

```
Semaphore fullBuffer = 0;  // Initially, buffer empty
Semaphore emptyBuffers = numBuffers;
    // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P();  // Wait until space
    mutex.P();  // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();  // Tell consumers there is more data.
}
Consumer() {
    fullBuffers.P();  // Check if there's an item
    mutex.P();  // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

Can we do:

```
mutex.P()
emptyBuffers.P()
```

instead?

Producer/Consumer Code

```
Semaphore fullBuffer = 0;  // Initially, buffer empty
Semaphore emptyBuffers = numBuffers;
    // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P(); // W
    mutex.P();  // Wait un
    Enqueue(item);
    mutex.V();
    fullBuffers.V();  // T
}
Consumer() {
    fullBuffers.P();  // C
    mutex.P();  // Wait un
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // t
    return item;
}
```

Can we do:

```
mutex.P()
emptyBuffers.P()
```

instead?

No. Consumer could block on mutex.P() and never V() emptyBuffers

Then producer **waits forever**!

Called deadlock (later!)

Producer/Consumer Code

```
Semaphore fullBuffer = 0;  // Initially, buffer empty
Semaphore emptyBuffers = numBuffers;
    // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptyBuffers.P(); // Wait until space
    mutex.P();  // Wait until buffer free
    Enqueue(item);
    mutex.V();
    fullBuffers.V();  // Tell consumers there is more data.
}
Consumer() {
    fullBuffers.P();  // Check if there's data
    mutex.P();  // Wait until buffer free
    item = Dequeue();
    mutex.V();
    emptyBuffers.V(); // tell producer need more
    return item;
}
```

Can we do:

```
emptyBuffers.V()
mutex.V()
```

instead?

Still correct, possibly less efficient.

Producer/Consumer: Discussion

Producer:

`emptyBuffer.P(), fullBuffer.V()`

Consumer:

`fullBuffer.P(), emptyBuffer.V()`

Two consumers or producers?

- Same code!

Problems with Semaphores

Our textbook doesn't like semaphore:

- "Our view is that programming with locks and condition variables is superior to programming with semaphores."

Arguments:

Clearer to have separate constructs for

- **waiting for a condition to become true** and
- **only allowing one thread to manipulate something at a time**

Need to make sure some thread calls P() for every V()

- Other interfaces let you be sloppier (later)

Outline

Hardware support for locks and spin locks

Semaphores

Monitors

Monitors and Condition Variable

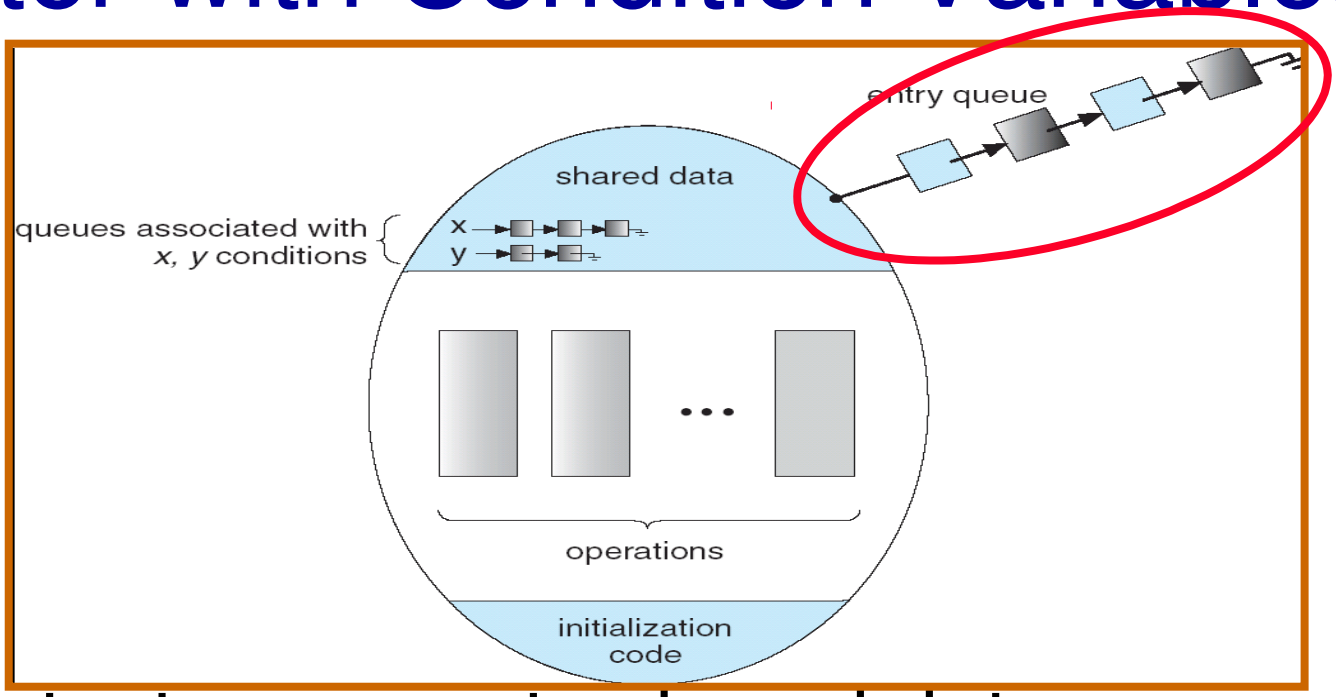
Locks for mutual exclusion

Condition variables for waiting

A **monitor** is a lock and zero or more condition variables with some associated data and operations

- Java provides this natively
- POSIX threads: provides **locks** and **condvars**, build your own

Monitor with Condition Variables

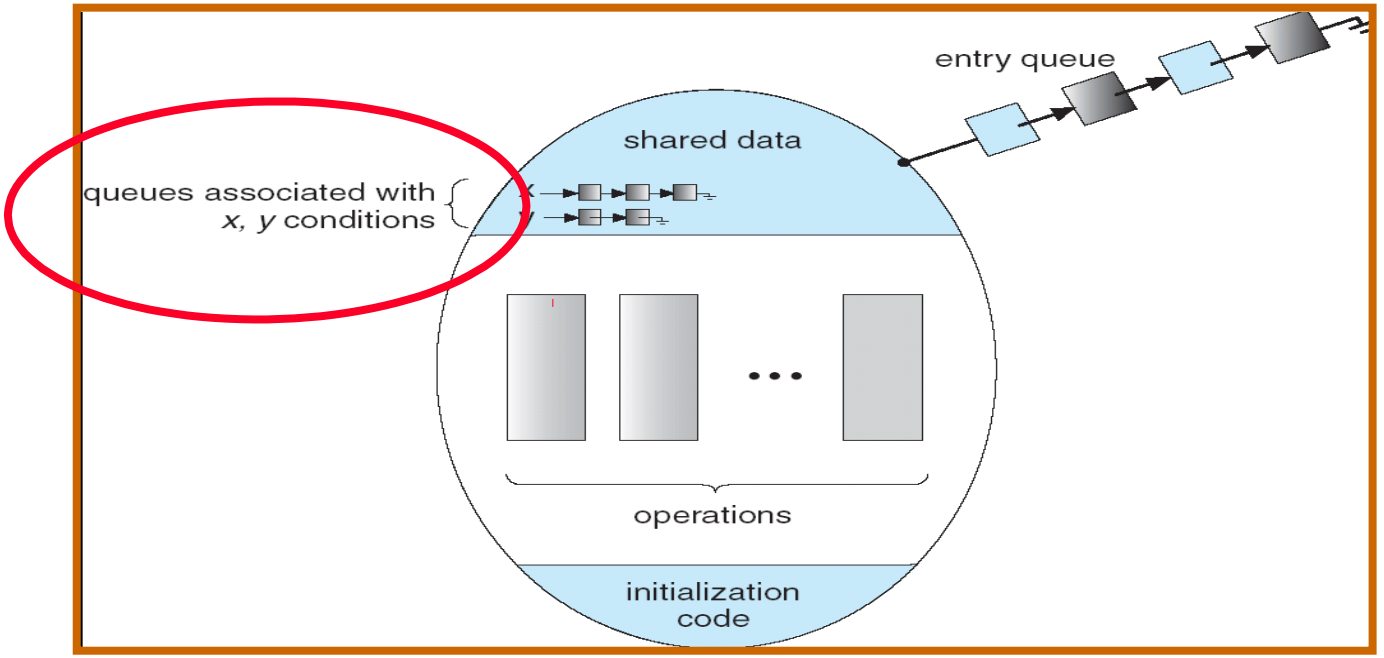


Lock: protects access to shared data

- **Rule: always acquire lock while accessing**

Queue of threads waiting to enter monitor

Monitor with Condition Variables



Condition variables: queue of threads waiting for something to become true **inside** a critical section

- **atomically** release the lock *and* start waiting (why?)
- another thread in the monitor will signal them
- the "something" is a function of the shared data

Condition Variables

Condition variable: **queue of threads waiting inside a critical section**

Operations:

- **`Wait(&lock)`**: Atomically release lock and go to sleep. Re-acquire lock before returning.
- **`Signal()`**: Wake up one waiter (if there is one)
- **`Broadcast()`**: Wake up all waiters

Rule: **Hold lock when using condition variable**

Monitor Example: Queue

```
Lock lock;
Condition dataready;
Queue queue;


AddToQueue(item) {
   lock.Acquire();   // Get Lock
   queue.enqueue(item); // Add item
   dataready.signal();  // Signal a waiter, if any
   lock.Release();   // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();   // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue(); // Get next item
   lock.Release();   // Release Lock
   return(item);
}
```

Why the loop? (1)

```
while (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
```

and **not**:

```
if (queue.isEmpty()) {
   dataready.wait(&lock); // If nothing, sleep
}
```

When a thread is woken up by **signal()**, it is just put on the ready queue – it **might not reacquire the lock immediately!**

- Another thread could "sneak in" and empty the queue
- Consequence: need a loop

Why the loop? (2)

Couldn't we "hand-off" the lock to the signaled thread so nothing can sneak in?

- Yes. Called ***Hoare-style* monitors.**
- What many textbooks describe

Most OSs implement ***Mesa-style monitors***

- Allow other threads to "sneak in"
- Much easier to implement
- Even easier if you allow "spurious wakeups" (returning when nothing signaled in rare cases)
  - POSIX does this

Comparing High-Level Synchronization

Semaphores can implement locks:

- `Acquire() { semaphore.P() }`
- `Release() { semaphore.V() }`

Monitors are a superset of locks.

Can monitors implement semaphores?

Semaphores with Monitors

```
Lock lock;
int Count = initial value of semaphore;
CondVar atOne;

P() {
  lock.Acquire();
  while (count == 0) {
    atOne.wait(&lock);
  }
  count--;
  lock.Release()
}
```

```
V() {
  lock.Acquire();
  count++;
  if (count == 1) {
    atOne.signal();
  }
  lock.Release()
}
```

Comparing High-Level Synchronization

Semaphores can implement locks:

- `Acquire() { semaphore.P() }`
- `Release() { semaphore.V() }`

Can monitors implement semaphores? Yes.

Can semaphores implement monitors?

CVs with Semaphores: Attempt 1

Attempt 1:

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
     semaphore.V();
}
```

Problem:

- Wait() signals non-waiting threads (in the future)
- What about broadcast()?

CVs with Semaphores: Attempt 2

Attempt 2 to construct CV from semaphore:

```
Wait(Lock lock) {
    lock.Release();
    semaphore.P();
    lock.Acquire();
}
Signal() {
    Atomically {
        if semaphore queue is not empty
            semaphore.V();
     }
}
```

Problem: "is queue empty" isn't a semaphore op

There is actually a solution

- See textbook (for Hoare scheduling)

CVs with Semaphores:

Is actually a solution!

Trick: queue of semaphores

- Protected by semaphore acting as lock
- Each waiting thread has a semaphore
- Call V() on waiter's semaphore and remove it from queue
-

Comparing High-Level Synchronization

Semaphores can implement locks:

- `Acquire() { semaphore.P() }`
- `Release() { semaphore.V() }`

Monitors are a superset of locks.

Can monitors implement semaphores? Yes.

Can semaphores implement monitors? Yes.

Summary: Lock Implementation

Hardware support for locks:

- **interrupt disabling** – suitable for single-processor
- **atomic read/modify/write** – spinlocks for multiproc

Blocking threads nicely

- Move to **queue for each lock**
- Run new thread atomically (no chance to not wakeup)

Summary: High-Level Sync

Semaphores – integers with restricted interface

- P() / down: Wait until non-zero then decrement
- V() / up: Increment and wake sleeping task

Monitors: Lock + Condition Variables

- Shared state determines when to wait
- Condition variables to wait "within critical section" on shared state changing
- Wait(), Signal(), Broadcast()