ADVANCED PARALLEL COMPUTING
LECTURE 04 - SYNCHRONIZATION

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Some material by Falsafi, Hardavellas, Nowatzyk of EPFL, Northwestern, CMU

SYNCHRONIZATION BASICS

SYNCHRONIZATION BASICS

Synchronization is as crucial as communication for parallel programming

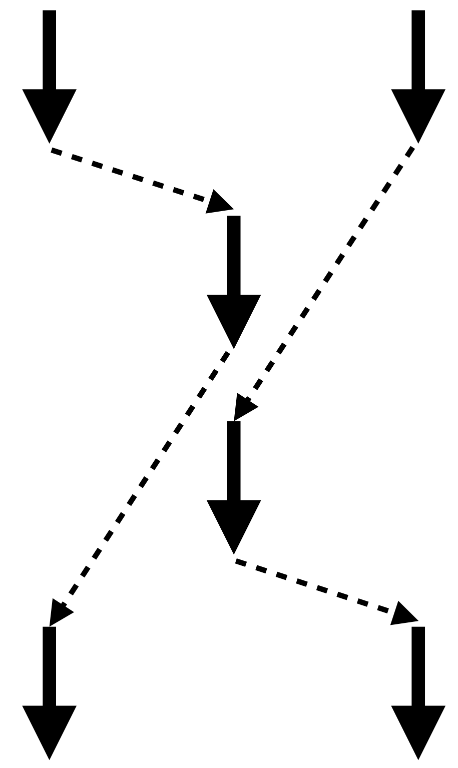
- Message passing vs. shared memory => implicit/explicit synchronization

Objectives

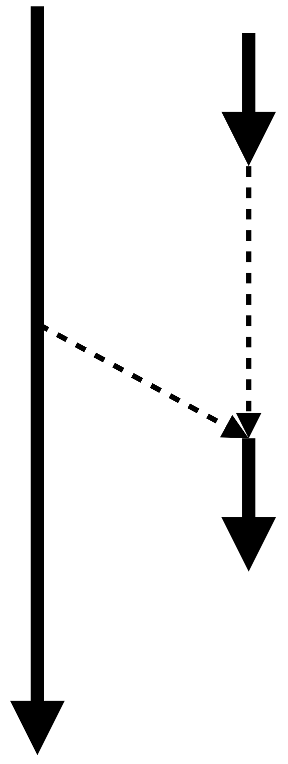
- Low overhead – no limitations with regard to scalability
- Correctness – synchronization failures are extremely difficult to debug (race conditions etc.)
- Coordination of HW and SW – SW semantics must be tightly specified to prove correctness, HW can improve efficiency dramatically

SYNCHRONIZATION FORMS

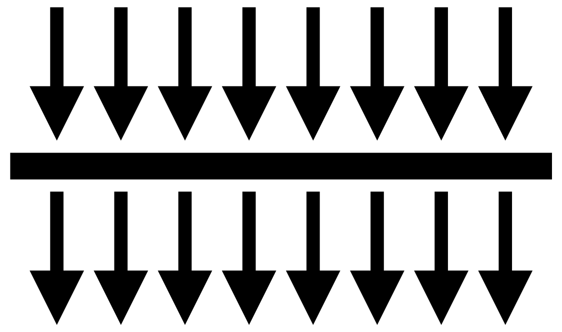
- Mutual exclusion (critical sections)
  - Lock & unlock semantics



- Event notification
  - Point-to-point (producer-consumer, flags)
  - I/O, interrupts, exceptions

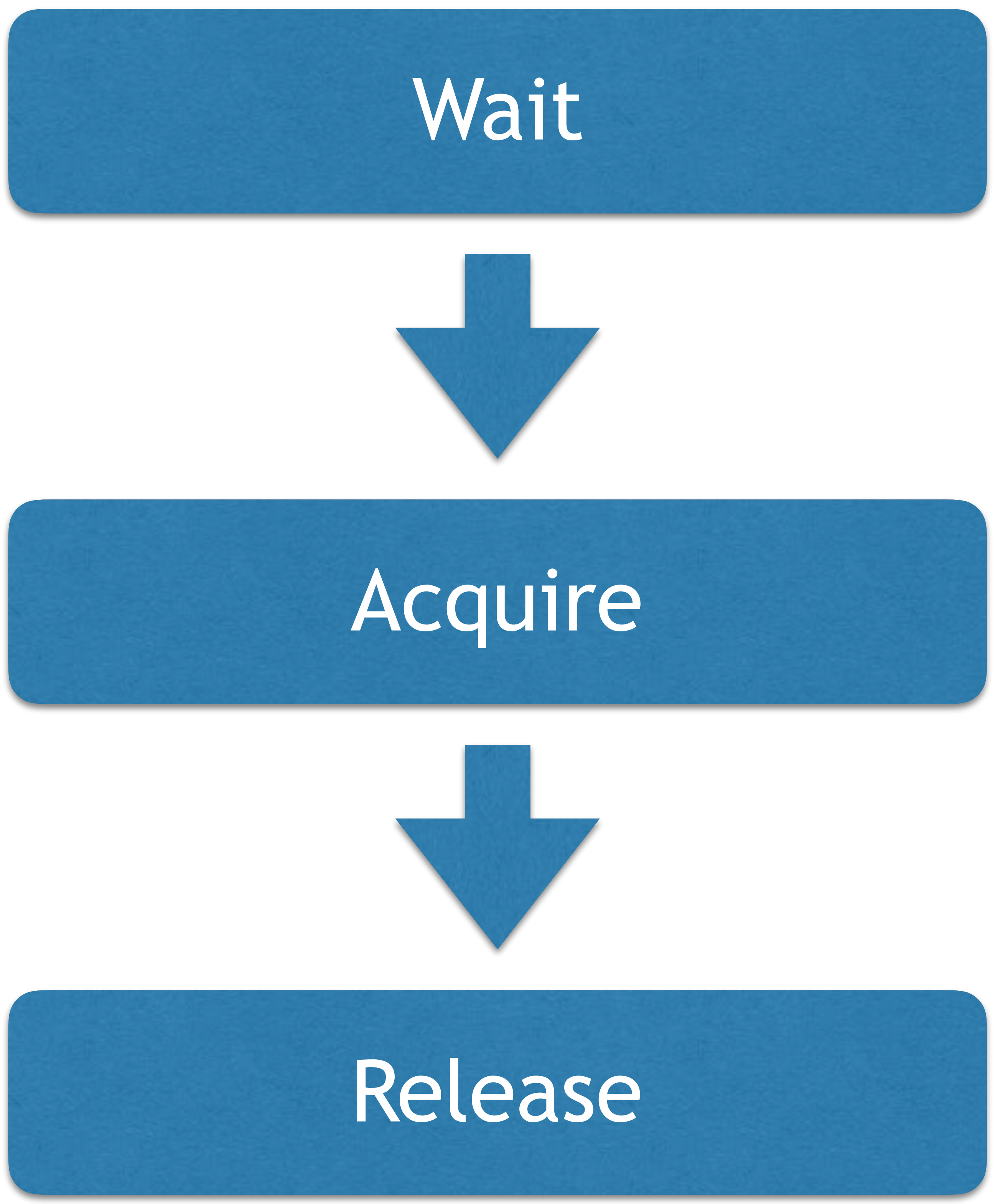


- Barrier synchronization
- Higher level constructs
  - Queues, software pipelines, (virtual) time, counters, ...



SYNCHRONIZATION IN DETAIL

1. Waiting algorithm: spin or block
   - Spin (aka busy wait)
     - Waiting process repeatedly tests a location until it changes
     - Releasing process updates this location
     - Low overhead, but high CPU utilization (and maybe bus traffic)
   - Block (aka suspend or back-off)
     - Waiting process is de-scheduled
     - High overhead (scheduling), but no CPU cycles wasted
     - Exponential back-off
   - Hybrids (spin, then block)
2. Acquire method: how to obtain the lock or pass the barrier
   - Compete (tournament) or hand-over (specified)
3. Release method: how to allow other processes to proceed
   - Reciprocal to acquire method



IMPLEMENTATION TRADE-OFFS

User wants ease-of-programming, e.g.:

```
LOCK (lock_variable) & UNLOCK (lock_variable)
BARRIER (barrier_variable, num_of_procs)
```

SW advantages: flexibility, portability

HW advantages: speed, efficiency

Design objectives

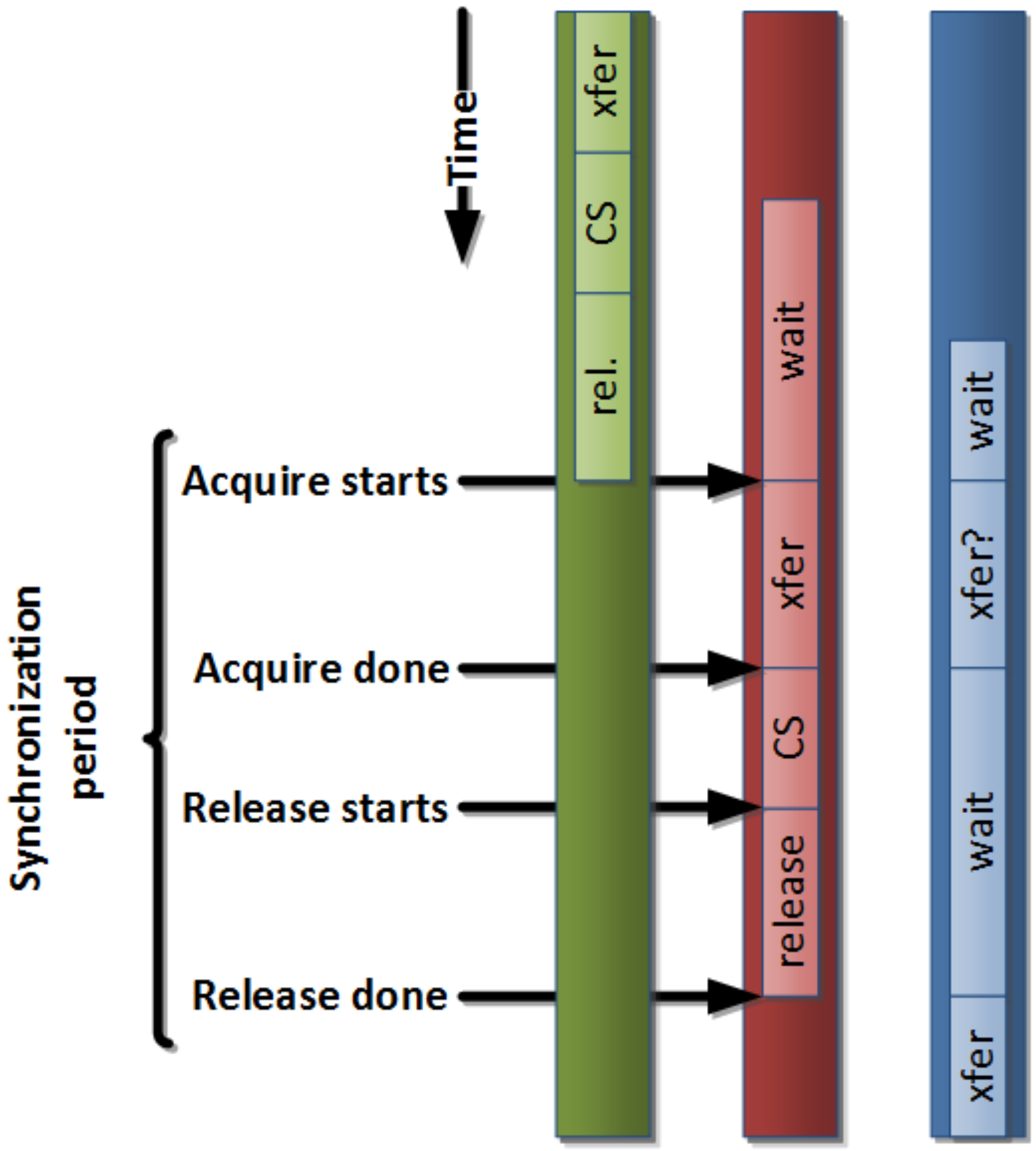
- Low latency
- Low traffic
- Low storage
- Scalability (minimizing wait times)
- Fairness (no starvation, FCFS, ...)

FUNDAMENTAL ISSUES

- Same synchronization may have different behavior at different times
  - Locks: high/low contention
  - Low contention => low latency; high contention => fairness
  - Different performance needs: low latency vs. high throughput
  - Different algorithms best for each, need different primitives
- Multiprogramming can change synchronization behavior
  - Process scheduling, resource interaction (devices)
  - May require algorithms that are worse in dedicated case
- Rich area of HW/SW interaction
  - Which primitives are available?
  - What communication pattern costs more/less?

SW LOCKS

LOCK-BASED MUTUAL EXCLUSION



In general: low overhead

Case of no contention

- Low latency

Case of contention

- Low period
- Low traffic
- Fairness

HOW NOT TO IMPLEMENT A LOCK

Read/Modify/Write cycle

Not atomic

Variable change simultaneously seen by many

Context switches, etc...

```
LOCK
  while (lock_var == 1);
  lock_var = 1;

UNLOCK
  lock_var = 0;
```

SOLUTION: ATOMIC READ-MODIFY-WRITE

Bus-based systems (UMA)

- Hold bus and issue load/store operations without any intervening process by other processors

Scalable systems (NUMA)

- Acquire exclusive ownership via cache coherence
- Perform load/store operations without allowing external coherence requests

```
//r: register
//m[]: memory location

Test&Set(r,x)
{if r==m[x] then m[x]=1;}

Fetch&Op(r1,r2,x,op)
{r1=m[x]; m[x]=op(r1,r2);}

Swap(r,x)
{temp=m[x]; m[x]=r; r=temp;}

Compare&Swap(r1,r2,x)
{temp=r2; r2=m[x]; if r1==r2 then m[x]=temp;}
```

TEST-AND-SET SPIN-LOCK (TASLOCK)

Performance problem

CAS is both read and write, thus spinning causes lots of invalidations

```
// wait-and-acquire
acquire (lock_ptr) {
  while (true) {
    //perform test-and-set
    old = cas (lock_ptr, UNLOCKED, LOCKED);
    if (old == UNLOCKED) break; //got lock!
    // otherwise: keep spinning
  }
}

// release
release (lock_ptr) {
  lock_ptr == UNLOCKED;
}
```

TEST-AND-TEST-AND-SET SPIN-LOCK (TTASLOCK)

Most of the spinning is now read-only, i.e. on local cache copy

```
// wait-and-acquire
acquire (lock_ptr) {
  while (true) {
    //perform test
    if (lock_ptr == UNLOCKED) {
      //perform test-and-set
      old = cas (lock_ptr, UNLOCKED, LOCKED);
      if (old == UNLOCKED) break; //got lock!
      // otherwise: keep spinning
    }
  }
}

// release
release (lock_ptr) {
  lock_ptr == UNLOCKED;
}
```

TTASLOCK ISSUES

Performance issues remain

- Every time the lock is released, all the processors load it and likely try to CAS the block
- Causes a storm of coherence traffic, clogs things up badly

One solution: back-off locks

- Instead of spinning constantly, rather check the lock frequently
- Exponential back-off works well in practice

Another problem with spinning

- Spinning threads can result in other threads starving (on the same core)
- Solution: x86 adds a „PAUSE“ instruction
- Tells the processor to suspend the thread for some time

No fairness

TICKET LOCKS

To ensure fairness and to reduce coherence storms

Ticket locks have two counters: `next_ticket` & `now_serving`

Summary of operations

- To "get in line", atomically increment `next_ticket`
- Spin on `now_serving` to wait for your call

```
acquire (lock_ptr) {
  my_ticket = fetch_and_inc(lock_ptr
                              ->next_ticket)
  while(lock_ptr->now_serving !=
        my_ticket); // spin
}

release (lock_ptr) {
  lock_ptr->now_serving ++;
}
```

TICKET LOCKS

Properties

- Reduced “coherence storm“ problem: To acquire, only need to read `now_serving`
- No CAS on critical path of lock handoff
- FIFO order, fair
- Problems might occur if threads are swapped out by OS

Padding to avoid false sharing

- Allocate `now_serving` and `next_ticket` on different cache blocks
- No interference anymore

Proportional back-off

- Estimate of wait time: `(my_ticket-now_serving)*avg_hold_time`

```
acquire (lock_ptr) {
  my_ticket = fetch_and_inc(lock_ptr
                            ->next_ticket)
  while(lock_ptr->now_serving !=
        my_ticket); // spin
}

release (lock_ptr) {
  lock_ptr->now_serving ++;
}
```

ARRAY-BASED QUEUE LOCKS

One location per thread might be beneficial: Completely avoid coherence storms!

Basic idea: array of size N (for N threads), either `go_ahead` or `must_wait`

- Initialization: first slot to `go_ahead`, others to `must_wait`
- Padded to one slot per cache block
- Maintain a `next_slot` counter, similar to `next_ticket`

Acquire:

- `my_slot = fetch_and_inc (next_slot) % N`
- Spin while `slots[my_slot]` contains `must_wait`, wait for `go_ahead`
- Reset `slots[my_slot]` back to `must_wait`

Release:

- Set successor `slots [ (my_slot+1) % N ]` to `go_ahead`

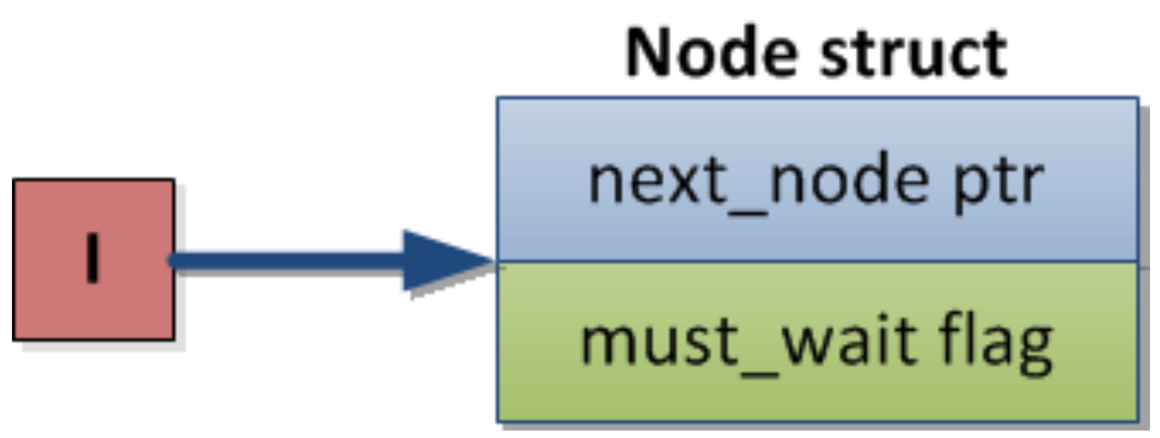
ARRAY-BASED QUEUE LOCKS

- Variants: Anderson 1990, Graunke and Thakkar 1990
- Desirable properties
  - Threads spin on dedicated locations
    - Only two coherence misses per handoff
    - Traffic independent of number of waiters
  - FIFO & fair (same as ticket lock)
- Undesirable properties
  - Higher uncontended overhead than a TTASLock
  - Storage requirements of O(N) for each lock
  - 128 threads at 64B padding: 8kB per lock!
  - N might change during run time
- List-based locks address the O(N) storage problem
  - Variants: MCS lock (1991), CLH lock (1993/1994)

LIST-BASED QUEUE LOCKS (MCS)

A lock is a pointer to a linked list node

- `next_node` pointer
- Boolean `must_wait`



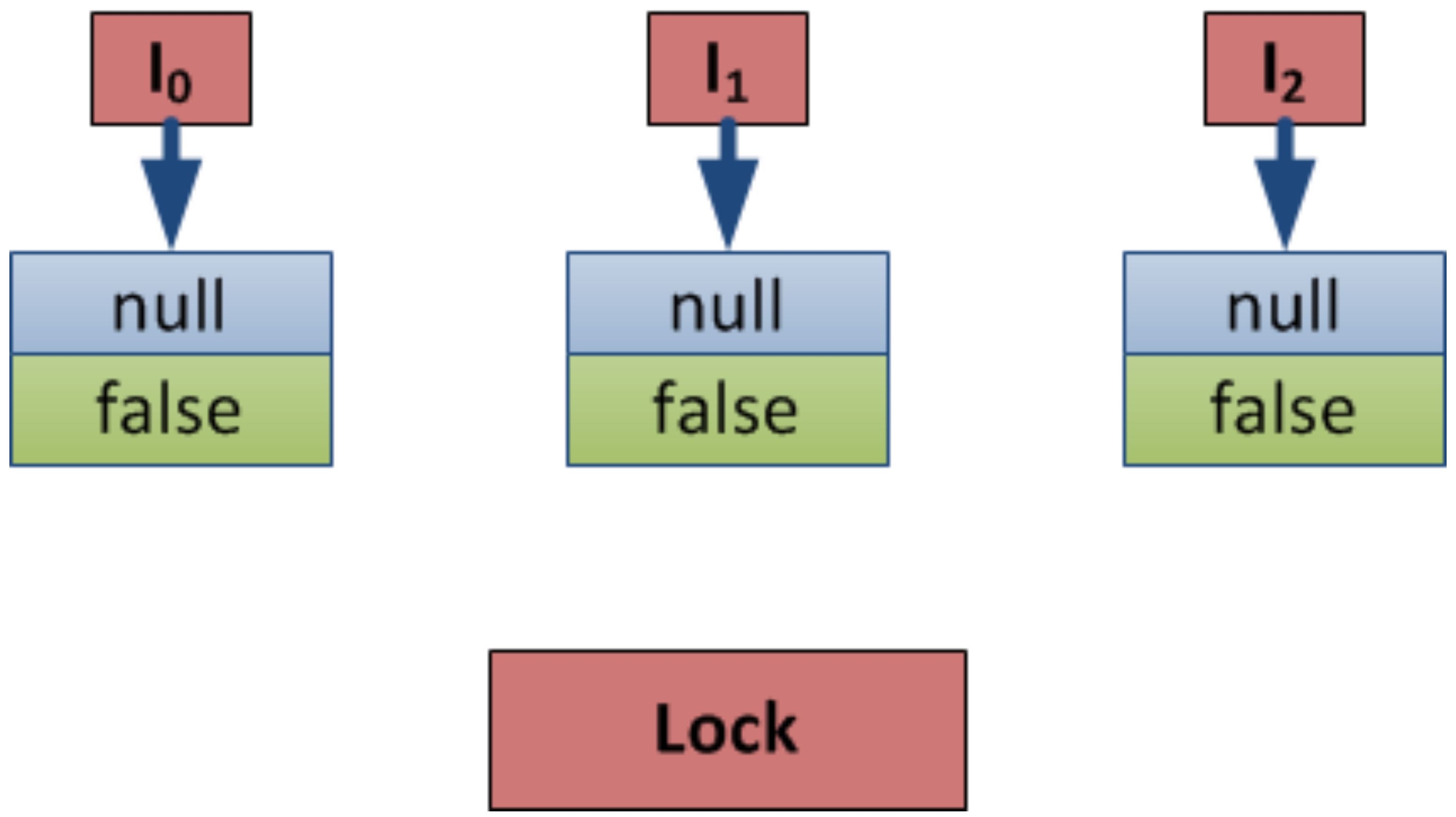
Each thread has its own local pointer to a node “I”

```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    I->must_wait = true;
    //predecessor must wake us up
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

fetch_and_store (lock, I) atomically writes I to lock and returns previous value of lock

MCS LOCK EXAMPLE: TIME 0

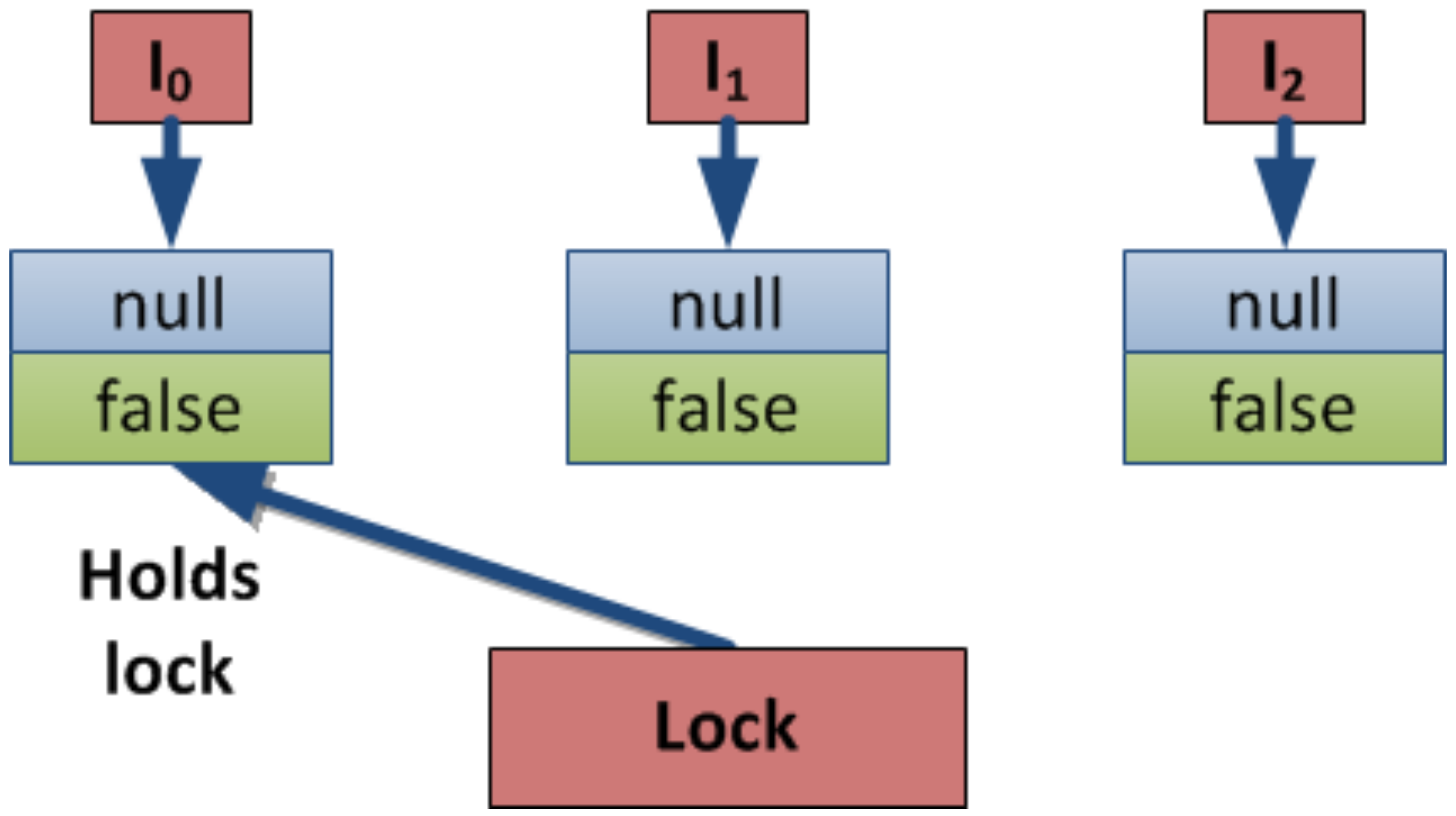



```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

MCS LOCK EXAMPLE: TIME 1

t1: acquire (lock)



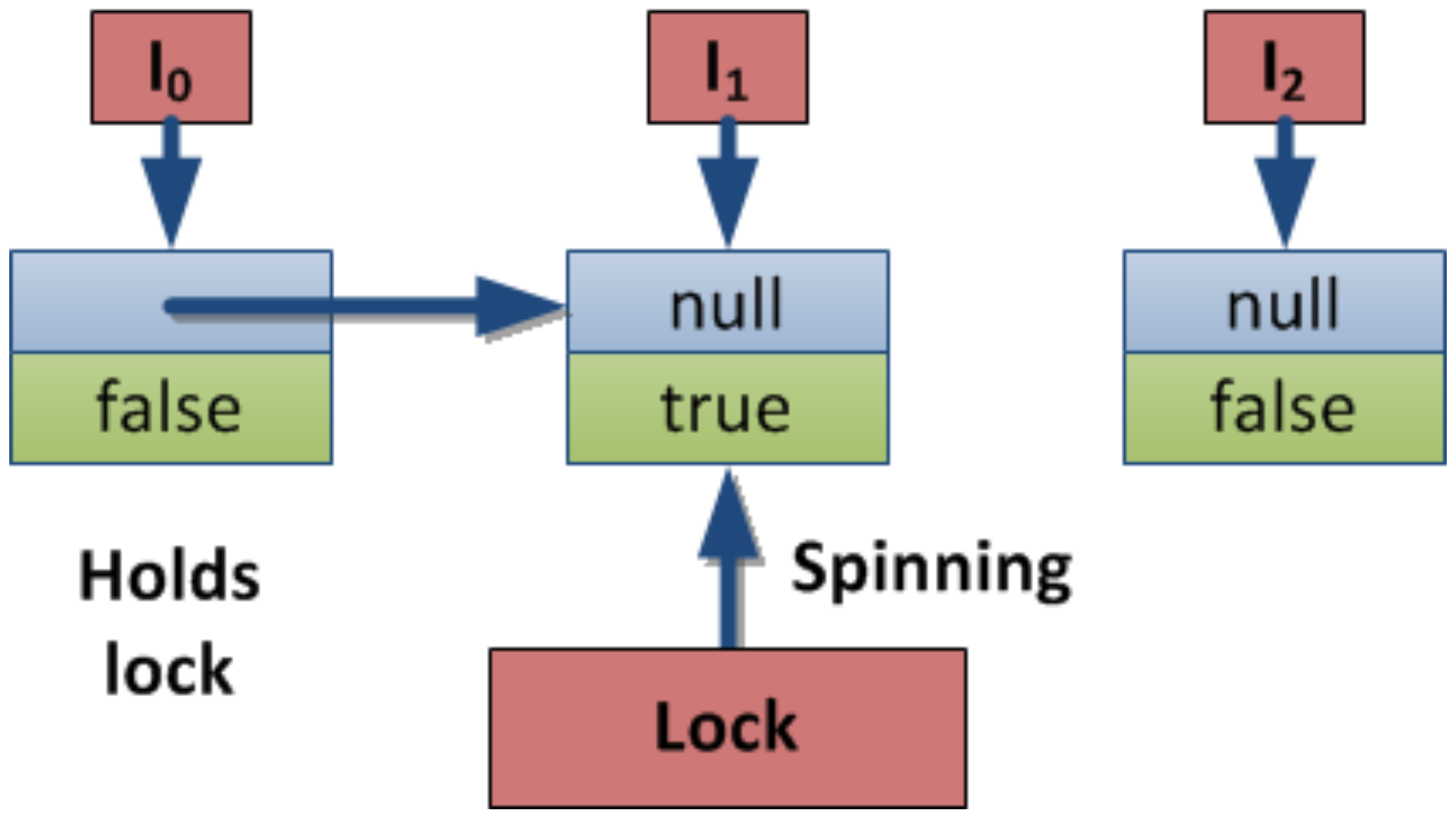

```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

MCS LOCK EXAMPLE: TIME 2

t1: acquire (lock)

t2: acquire (lock)




```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

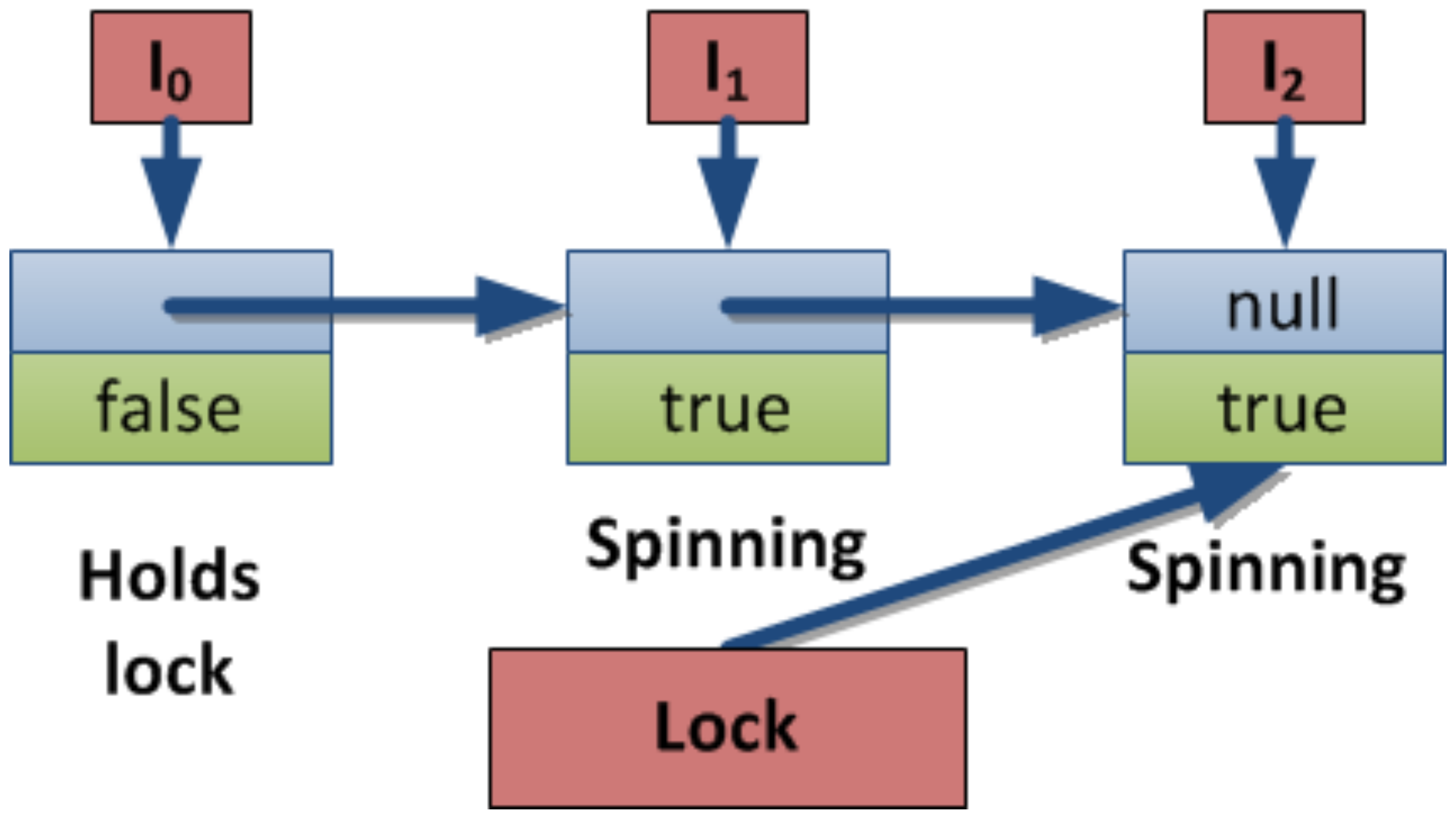
```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

MCS LOCK EXAMPLE: TIME 3

t1: acquire (lock)

t2: acquire (lock)

t3: acquire (lock)




```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

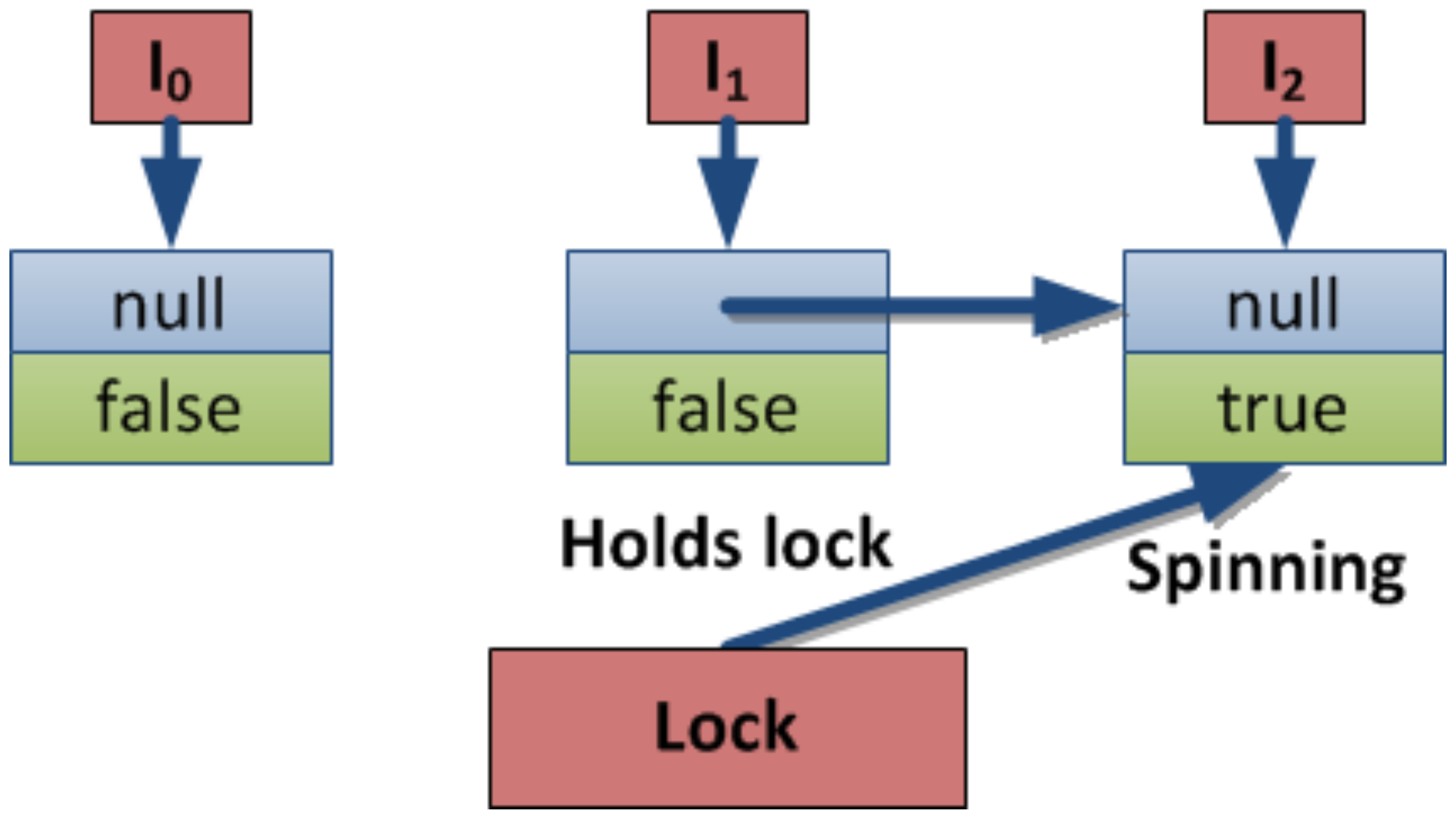
MCS LOCK EXAMPLE: TIME 4

t1: acquire (lock)

t2: acquire (lock)

t3: acquire (lock)

t1: release (lock)

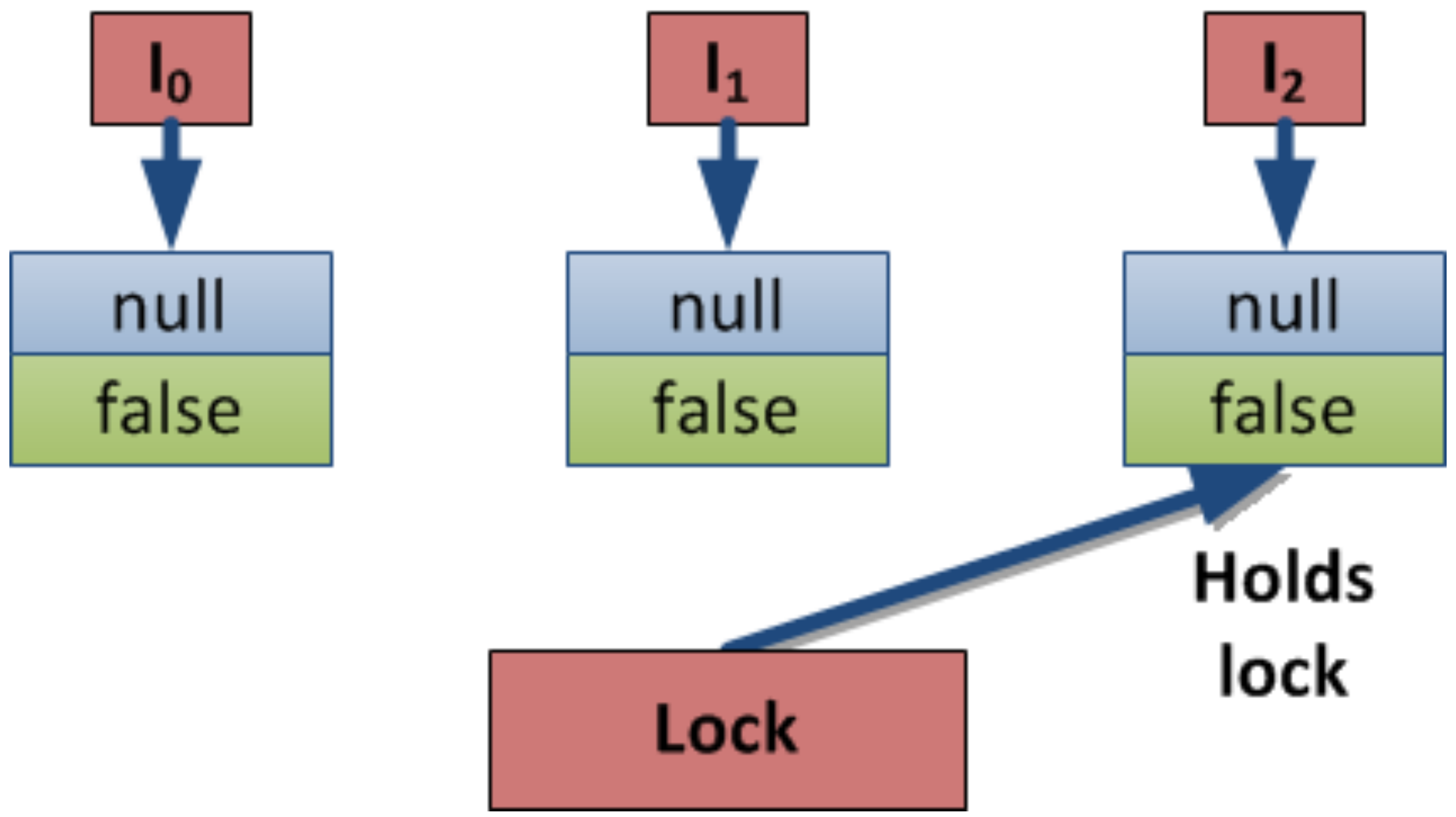



```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

MCS LOCK EXAMPLE: TIME 5

t1: acquire (lock)
t2: acquire (lock)
t3: acquire (lock)
t1: release (lock)
t2: release (lock)




```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

MCS LOCK EXAMPLE: TIME 6

t1: acquire (lock)

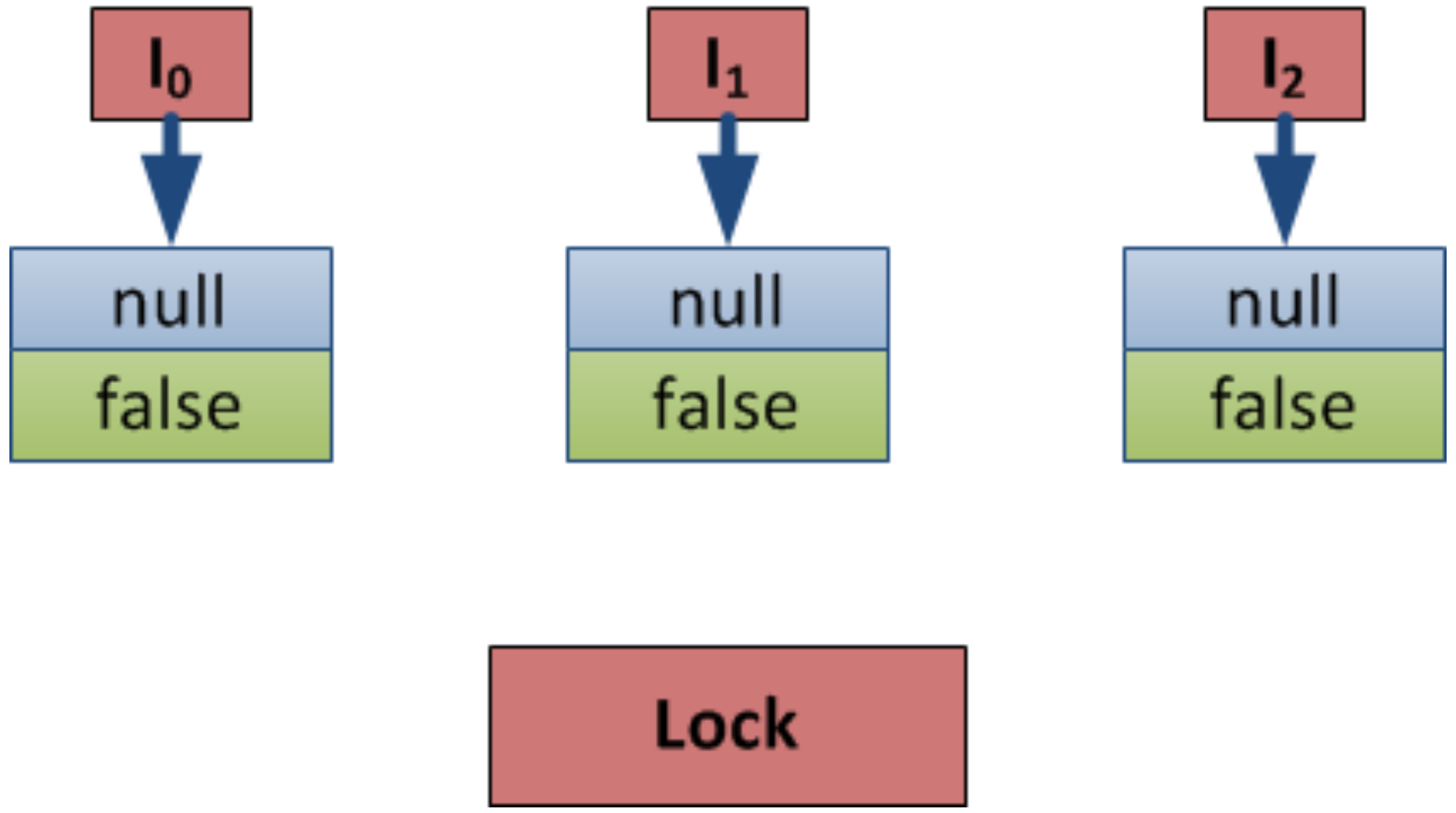
t2: acquire (lock)

t3: acquire (lock)

t1: release (lock)

t2: release (lock)

t3: release (lock)




```
acquire (lock) {
  I->next = null; //(re-)init
  predecessor = fetch_and_store (lock, I);
  if (predecessor != null) {
    i->must_wait = true;
    //predecessor must wake us
    predecessor->next = I;
    //spin until lock is released
    while (i->must_wait);
  }
}
```

```
release (lock) {
  if (I->next == null) {
    //no known successor - really?
    if ( CAS ( lock, I, null) )
      // CAS succeeded, lock freed
      return;
    //spin to learn successor
    while (I->next == null);
  }
  I->next->must_wait = false;
}
```

HW LOCKS

HW SOLUTION: QUEUE ON LOCK BIT (QOLB)

HW maintains double-linked list between requesters

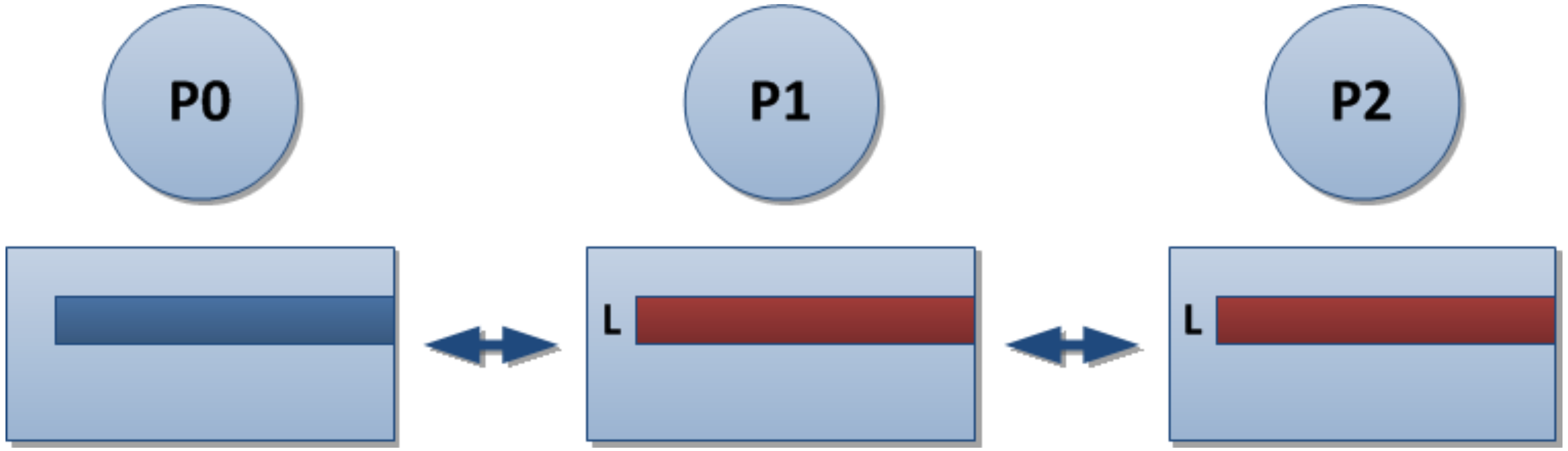
- Key idea from Scalable Coherent Interface (SCI)

Augment cache with locked bit

- Waiting caches spin on local locked cache line

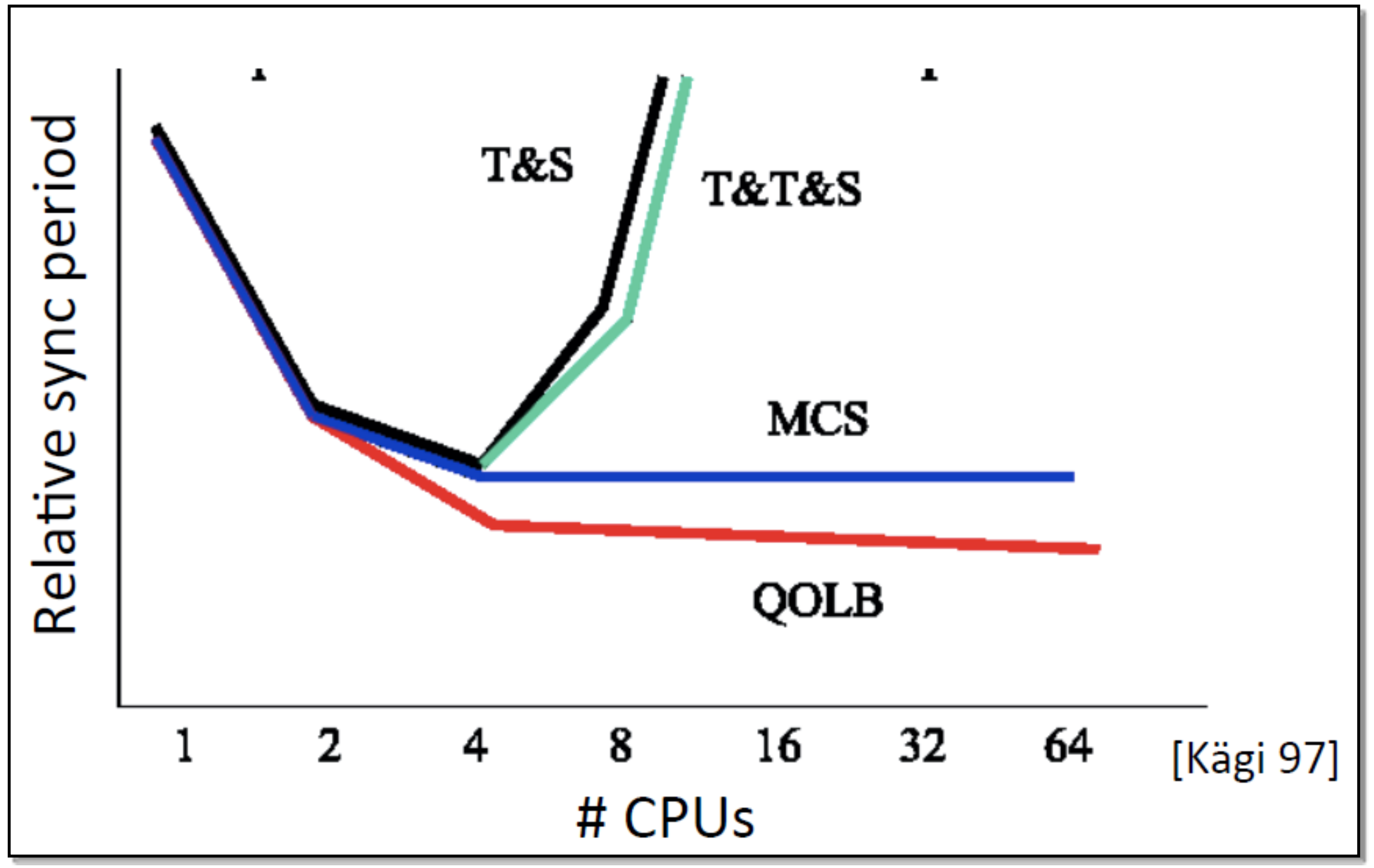
Upon release, holder sends cache line to 1st requester

- Only requires one message on interconnect



HW SOLUTION: QUEUE ON LOCK BIT (QOLB)

	Local Spin	Queue	Collocation	Prefetch
TAS	No	No	Optional	No
TTAS	Yes	No	Optional	No
MCS	Yes	Yes	Partial	No
QOLB	Yes	Yes	Optional	Yes



Local spinning: allows a requesting node to spin on a local copy of the lock: minimal traffic, local coherent spinning

Queue-based locking: eliminates arbitration overhead and reduces lock transfer time

Collocation: protected data is transferred with the transfer of the lock itself (partial for MCS because each requester is spinning on a different address)

Synchronous prefetch: processor can issue a lock request in advance of the critical section. The memory system will effect the transfer of the lock from the holder to the prefetching requester only when the holder releases the lock.

Alain Kägi, Doug Burger, and James R. Goodman. 1997. Efficient synchronization: let them eat QOLB. ISCA '97. DOI=http://dx.doi.org/10.1145/264107.264166

PERFORMANCE OF LOCKS

- Contention vs. no contention
  - TASLock is best in the absence of contention
    - Minimal overhead, single RMW operation
  - Queue-based is best with medium contention
  - Another idea: switch implementation based on lock behavior
    - Reactive synchronization: Lim & Agarwal – 1994
    - SmartLocks: Eastep et al – 2009
- High contention can be an indicator for a poorly written program
  - Need better algorithms or data structures
- Mind the coherency storms
  - Locks are a perfect example for this