Advanced Topics in Multicore Architecture & Software Systems

Serializing efficiently

Administration

- HW#3 out tonight, due **May 18**
  - 1 question on this lecture
  - 1 question on next lecture

Serializing efficiently

A part of the program is *serialized* if it cannot be run by multiple threads concurrently

- For example, a critical section

Serialization is the mechanism used to guarantee this property

- For example, a lock

Motivation

A serialized code section doesn't exploit parallelism, and so is "slow"

Why care about making it efficient?

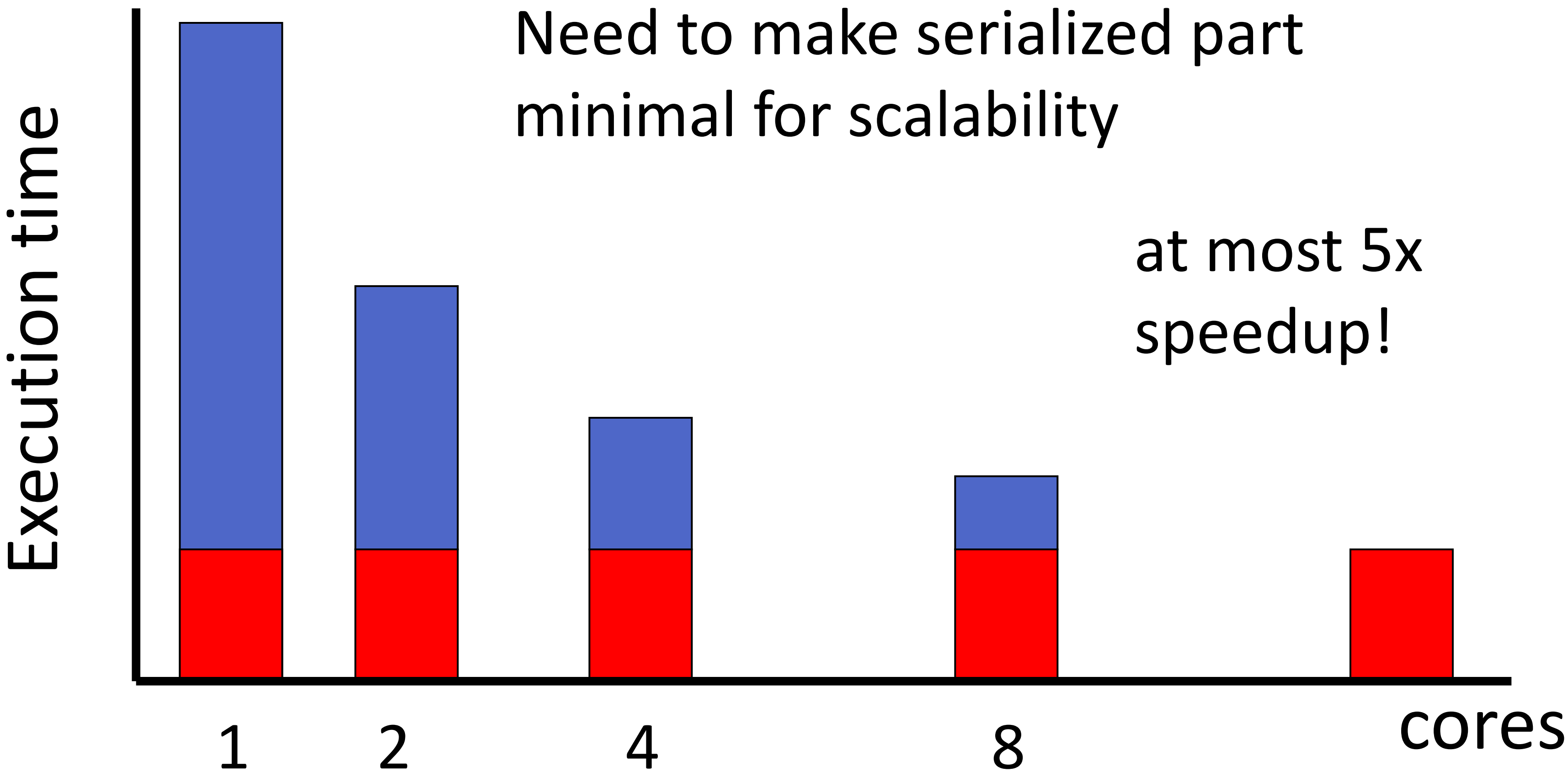
Motivation: Amdahl's law

What is the effect of speeding up ***p*** of the execution time by ***s***?

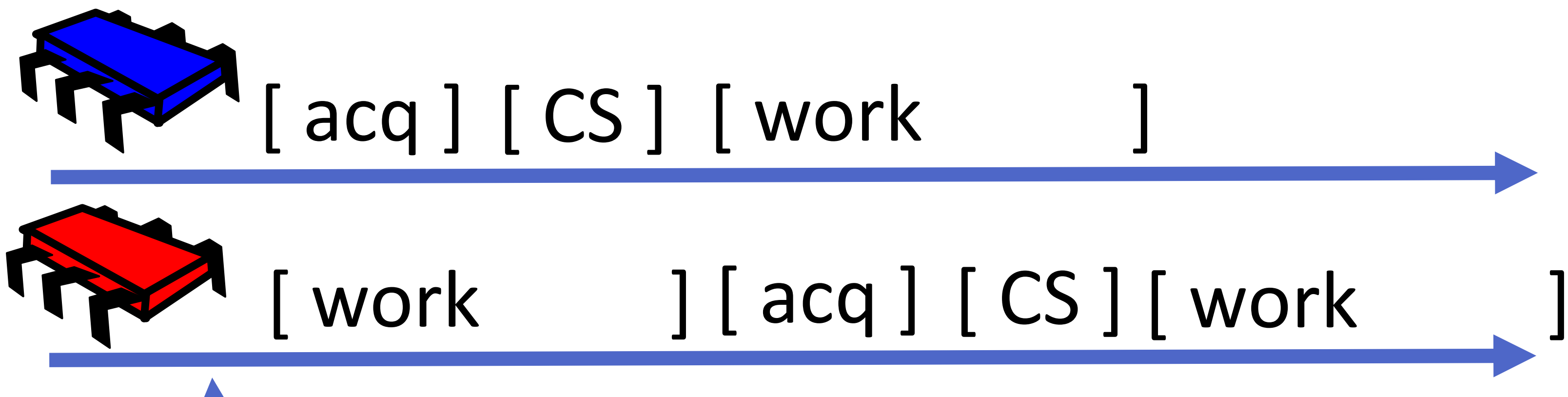
$$\text{speedup} = \frac{1}{(1-p) + p/s}$$

Amdahl's law

80% of execution parallelizable:

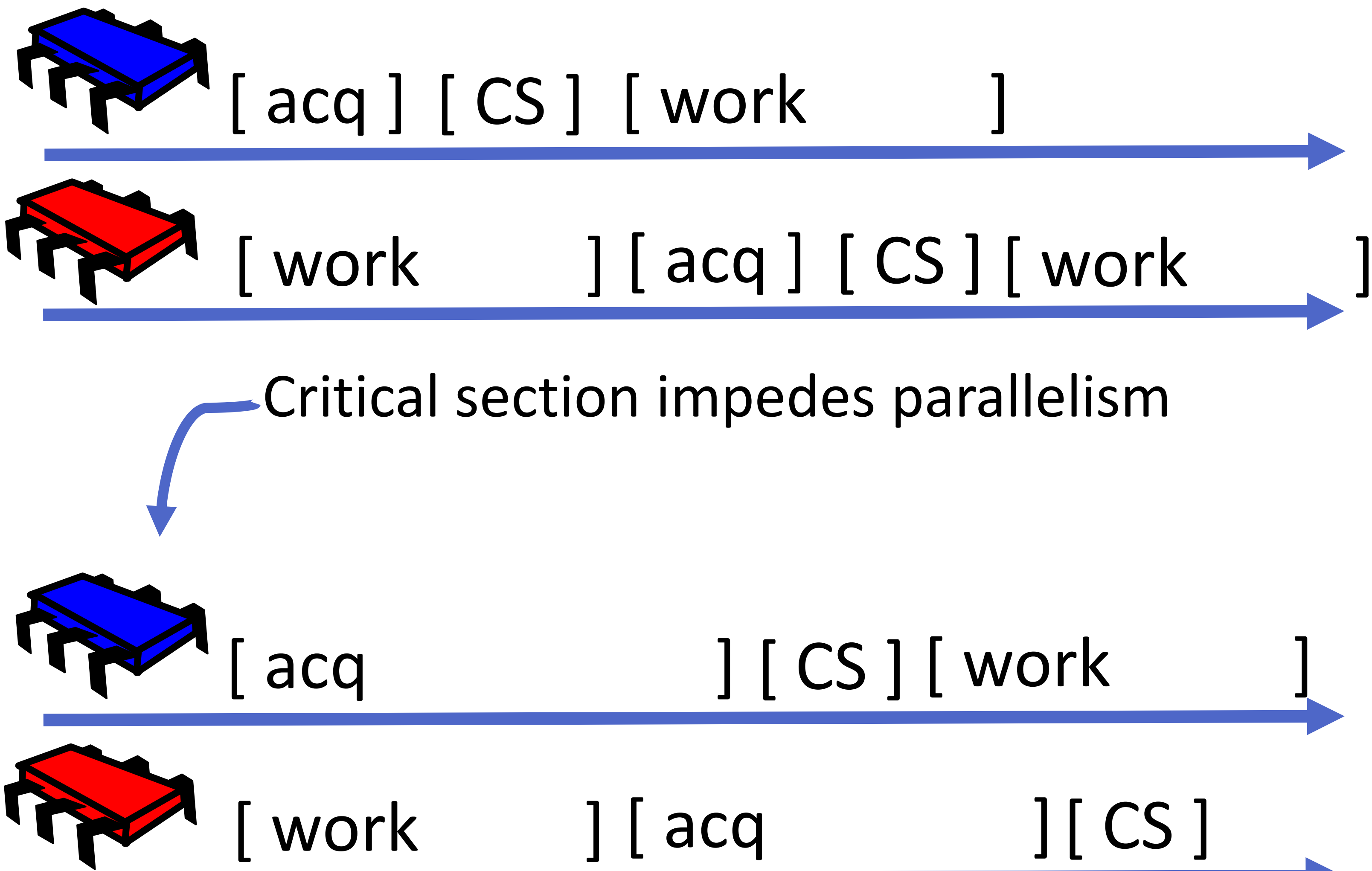


Another way of looking at it



Critical section overlaps with other work, doesn't impede parallelism

Another way of looking at it

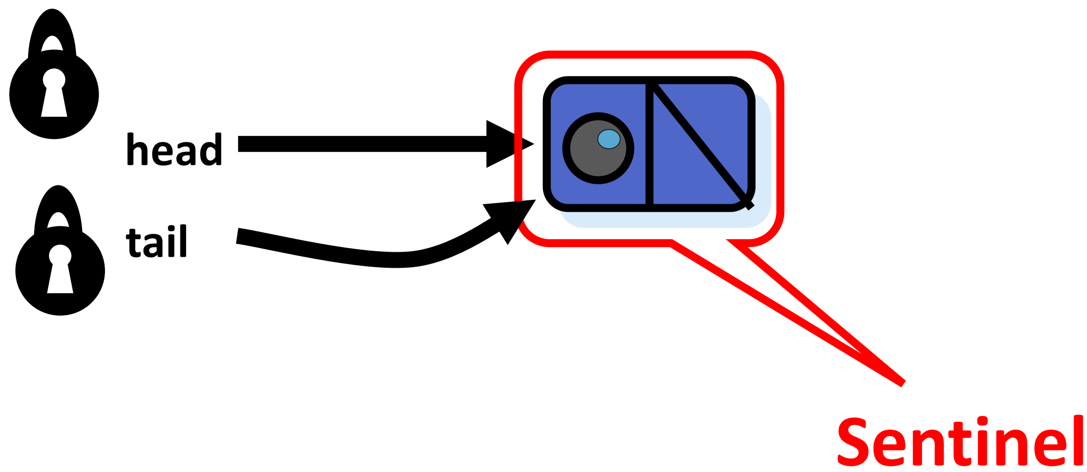


Today

- Lock implementation issues
- Delegation locking
- Lock-free synchronization and CAS failures

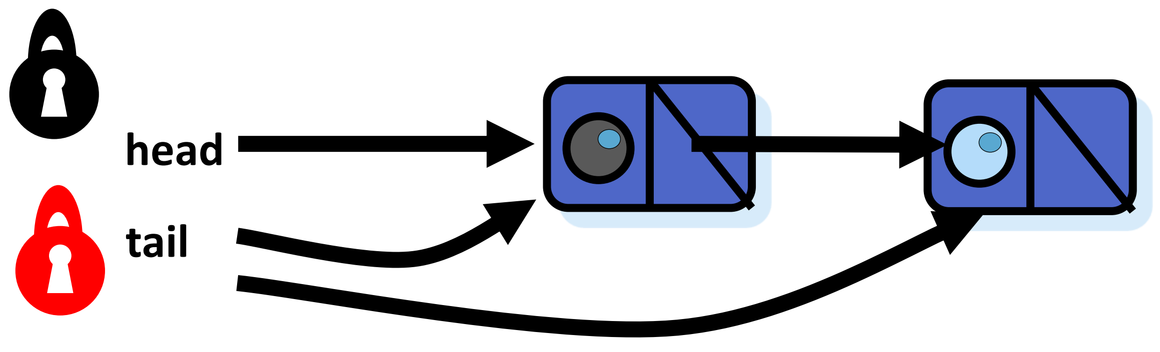
- Running example: a FIFO queue (unbounded)

Two-lock queue [Michael & Scott '96]

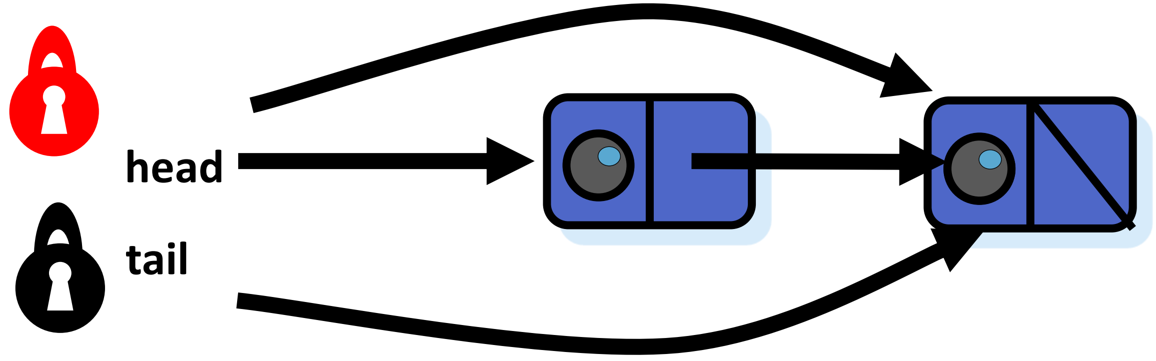


- Allows concurrent enqueue & dequeue operations in most situations

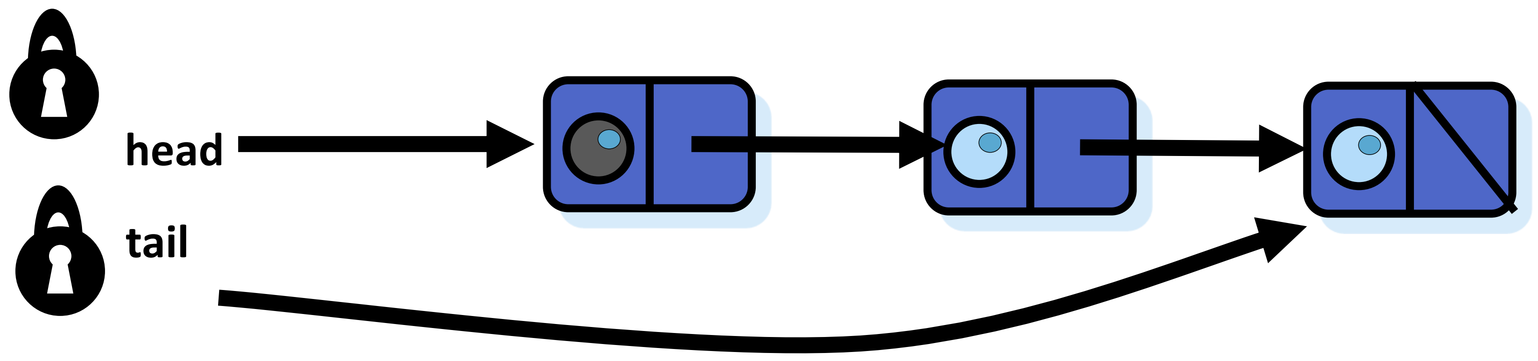
Two-lock queue: enqueue



Two-lock queue: dequeue



Concurrent enqueue & dequeue



Two-lock queue

```
Node { void* val; Node* next; };

Initially:
head = tail = new Node(NULL);
head->next = NULL;
```

Two-lock queue

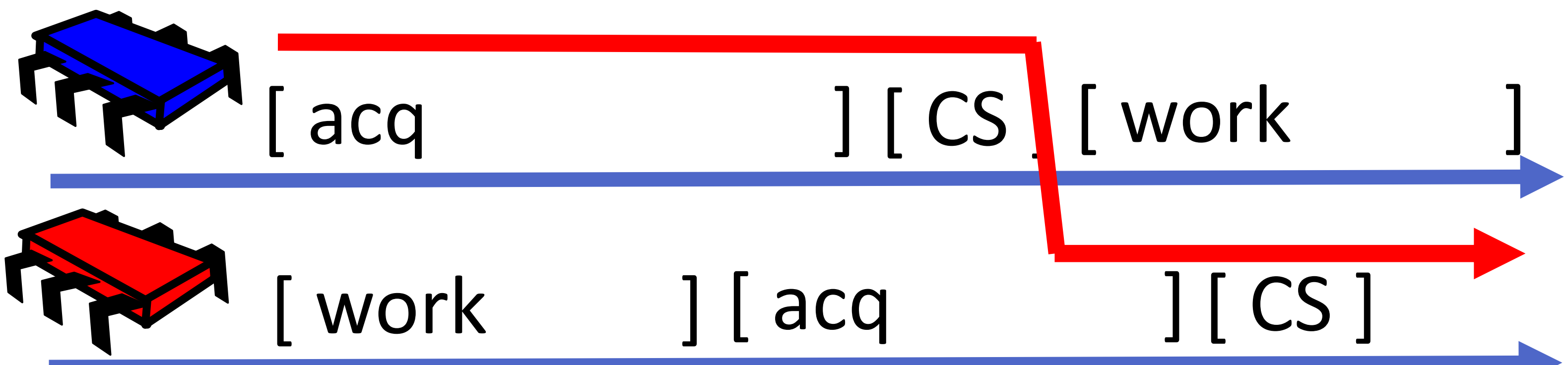
```
enqueue(void* v) {
  Node* node = new Node(v);
  node->next = NULL;
  lock(&tail_lock);
  tail->next = node;
  tail = node;
  unlock(&tail_lock);
}
```

Two-lock queue

```
void* dequeue() {
  lock(&head_lock);
  Node* node = head;
  Node* nh = head->next;
  if (!nh) { unlock(..); return 0; }
  head = nh;
  void* rv = nh->val;
  unlock(&head_lock);
  free(node);
  return rv;
}
```

Locks and critical sections

- Assumption: critical section is short
  - If not, the CS will be the bottleneck
- Goal: Minimize length of the *critical path*
  - Lock acquisition, CS & handoff

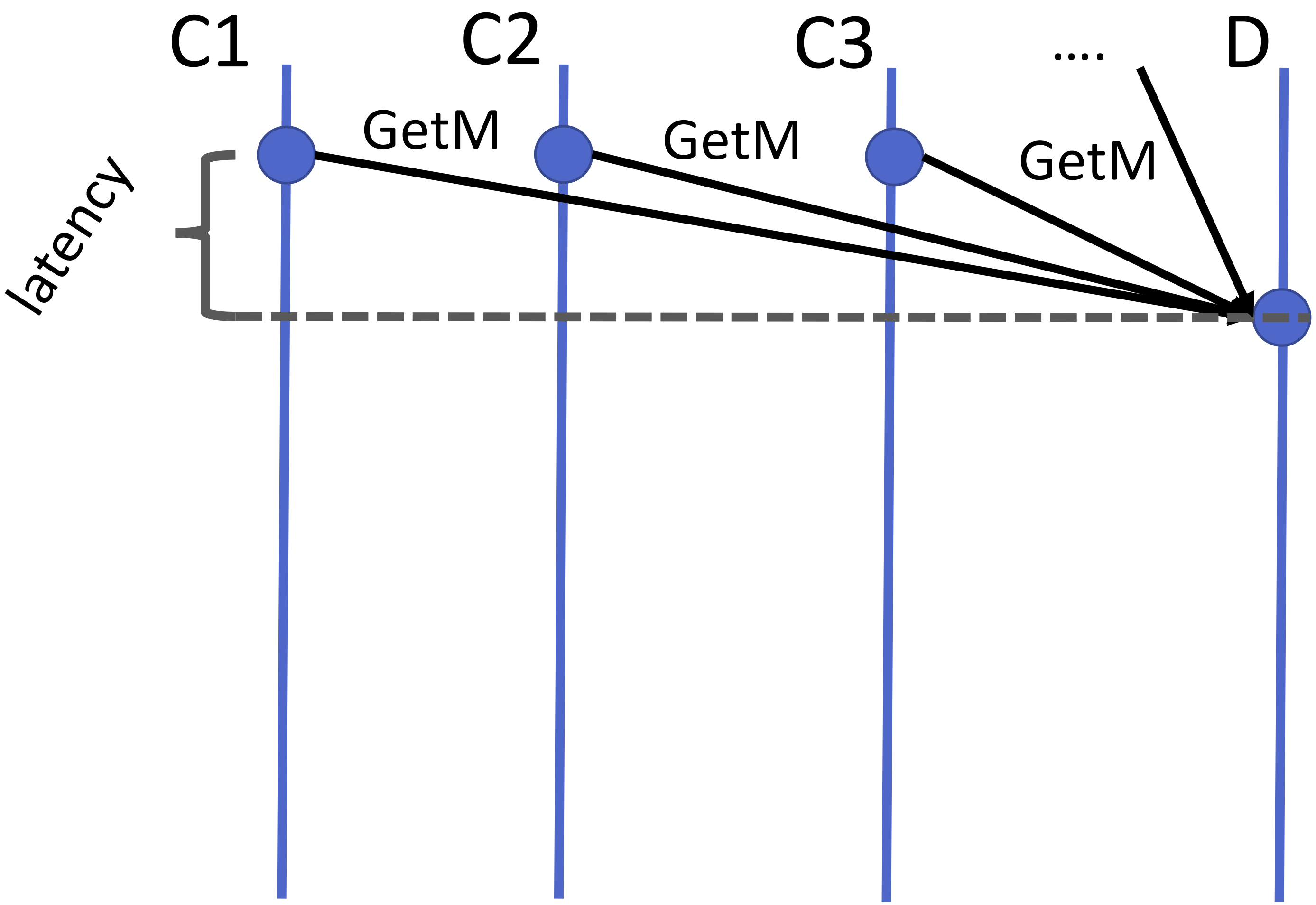


TAS lock

```
bool L;

lock() {
  while (!CAS(&L, 0, 1)) { }
}

unlock() {
  L = 0
}
```

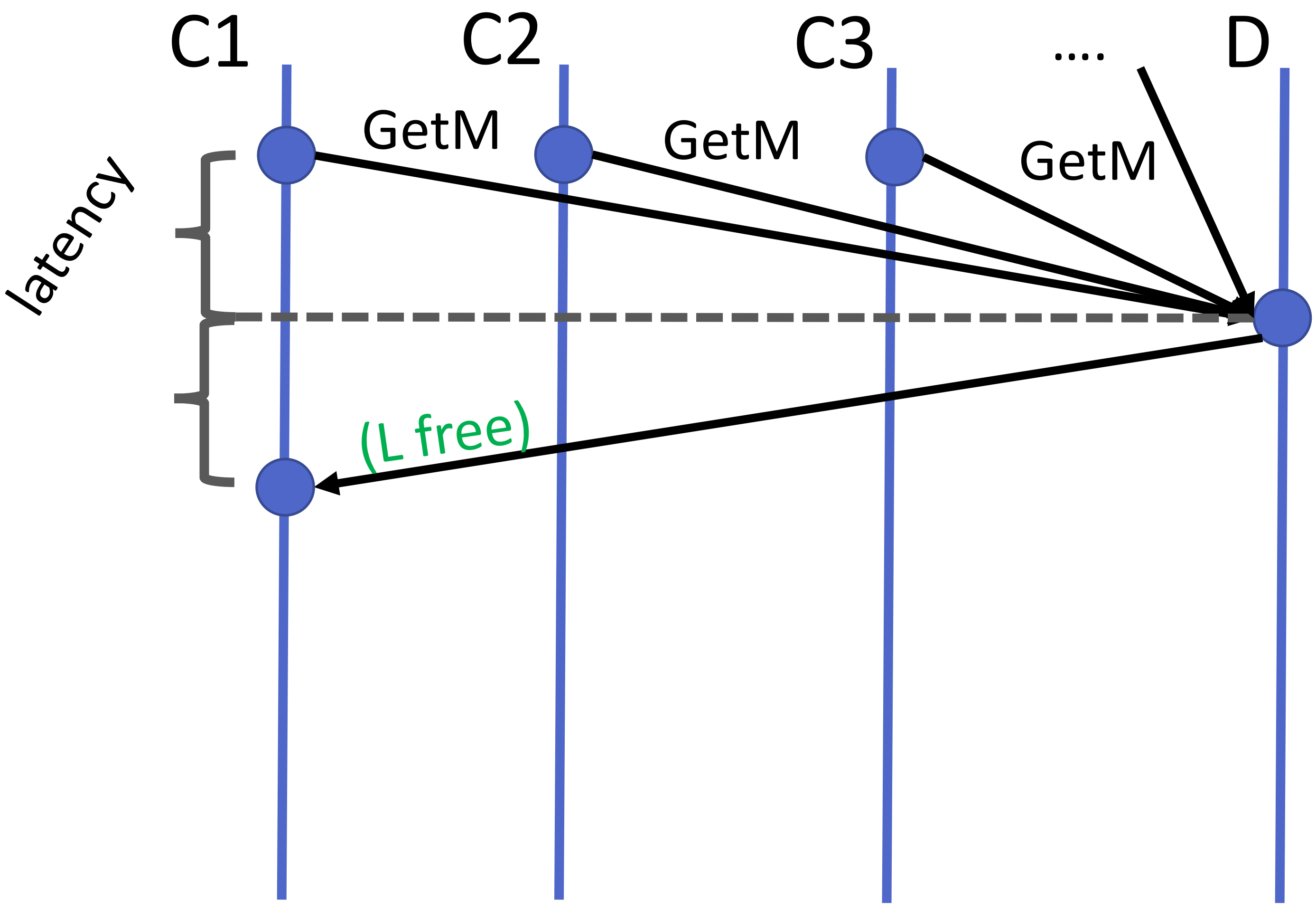
TAS analysis

Cores try to acquire concurrently



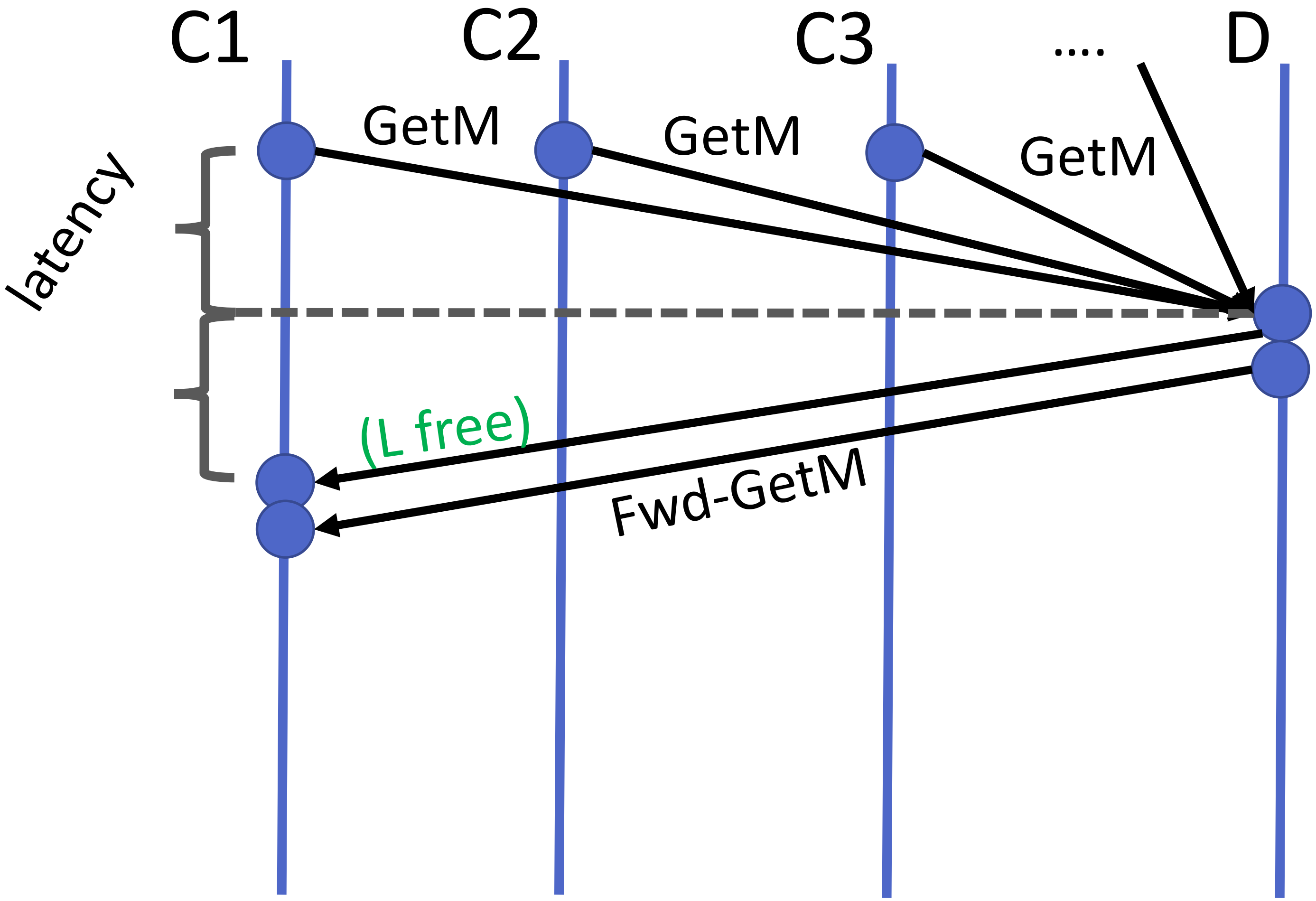
TAS analysis

Directory serializes requests



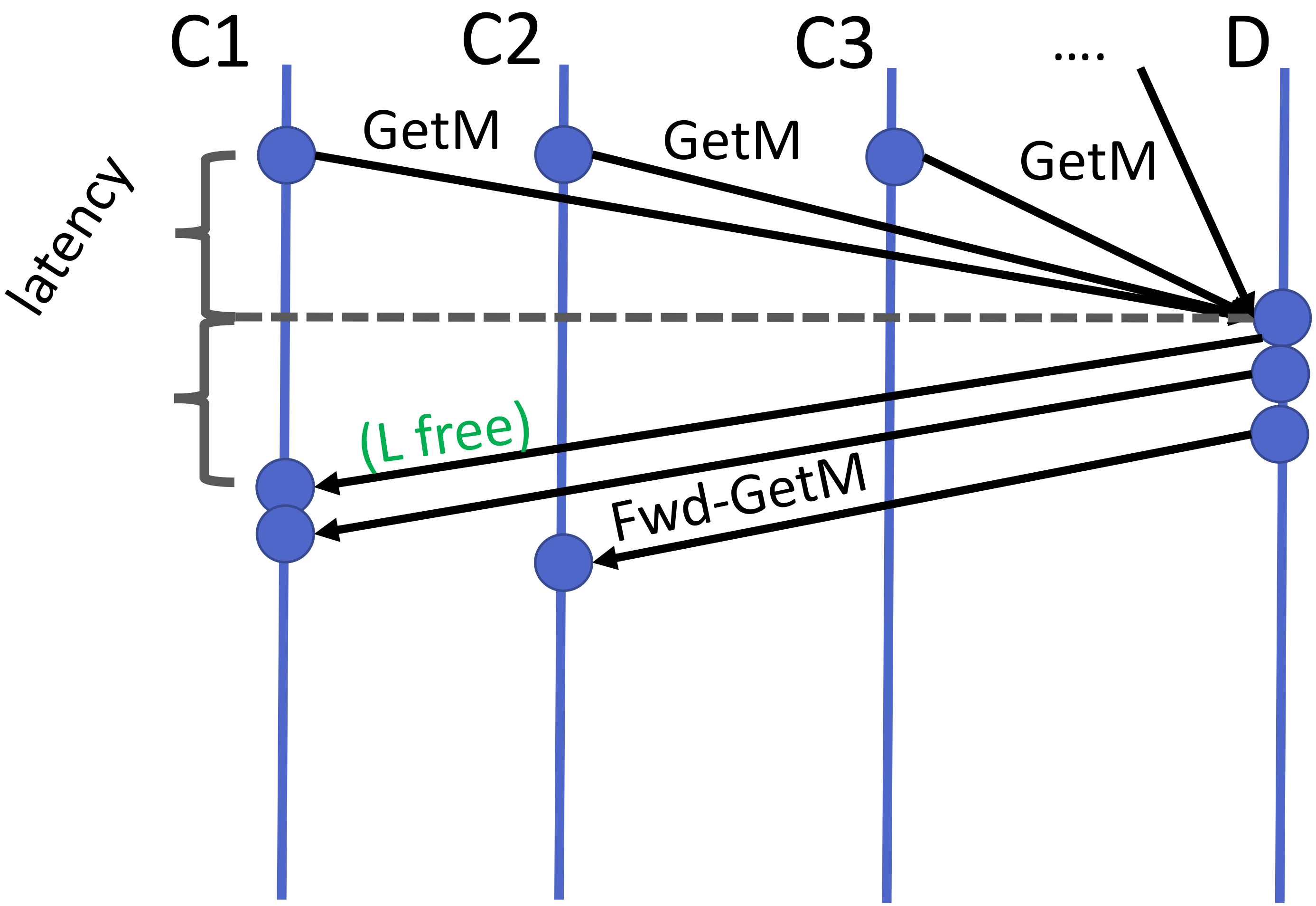
TAS analysis

Directory serializes requests



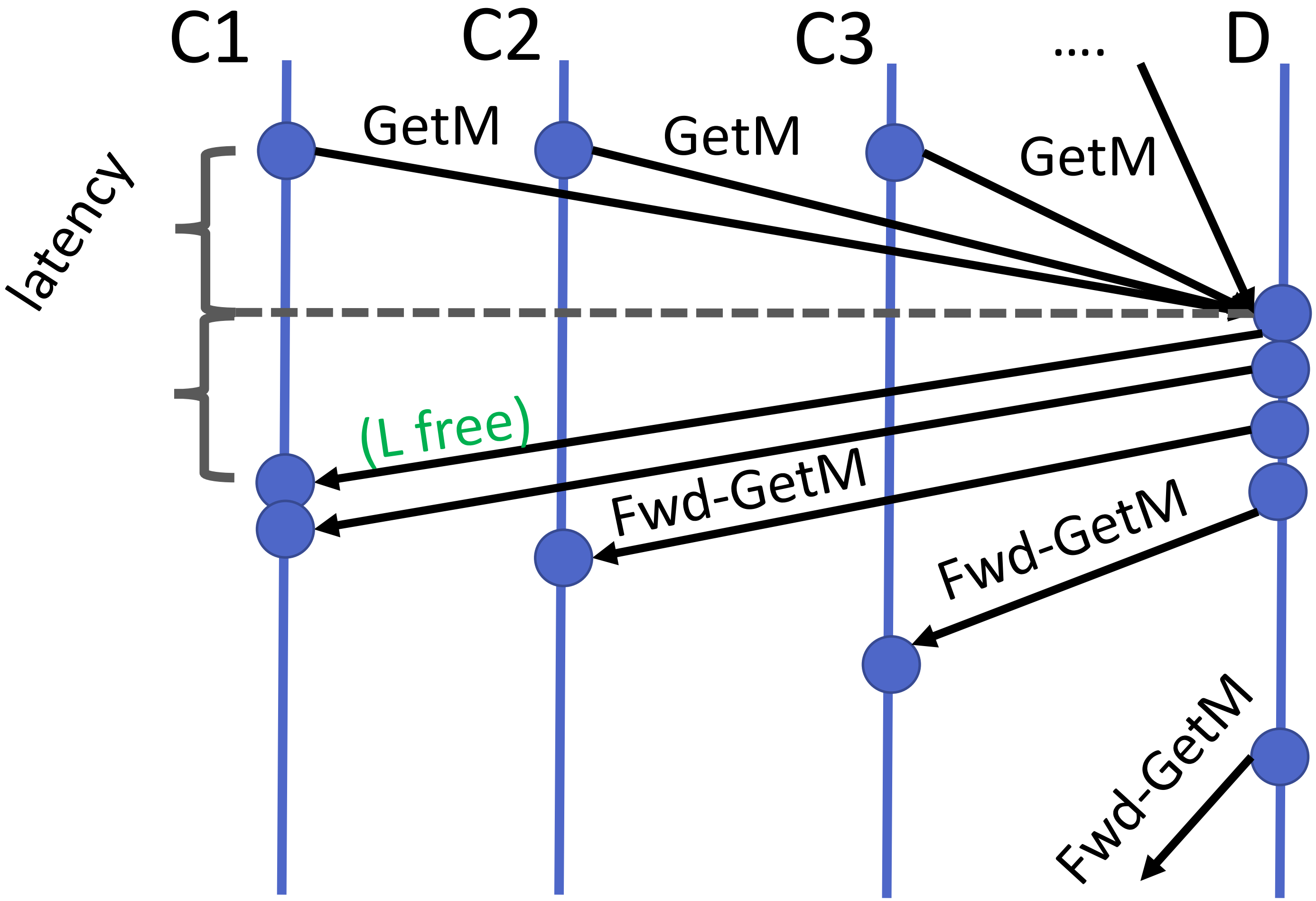
TAS analysis

Directory serializes requests



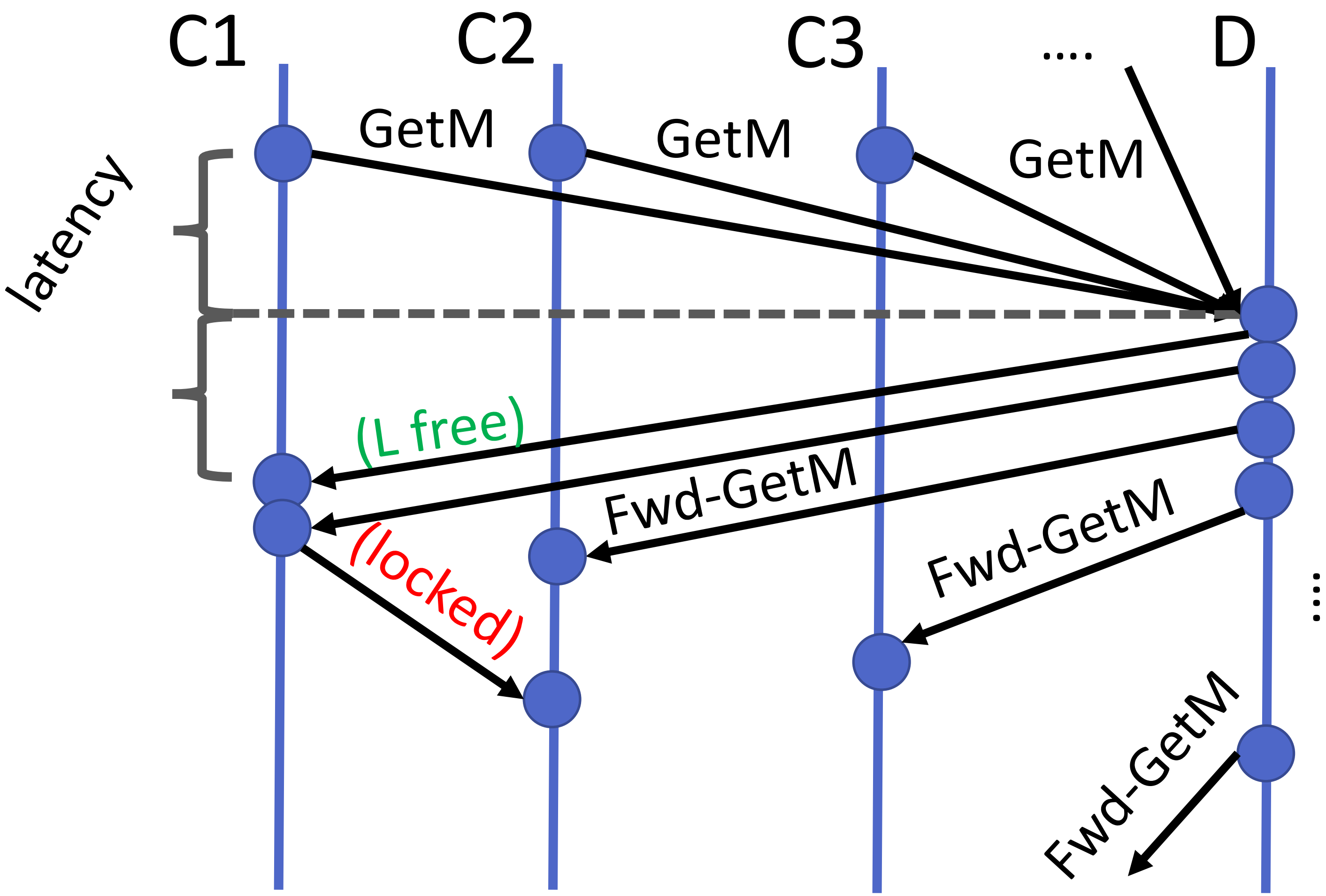
TAS analysis

Directory serializes requests



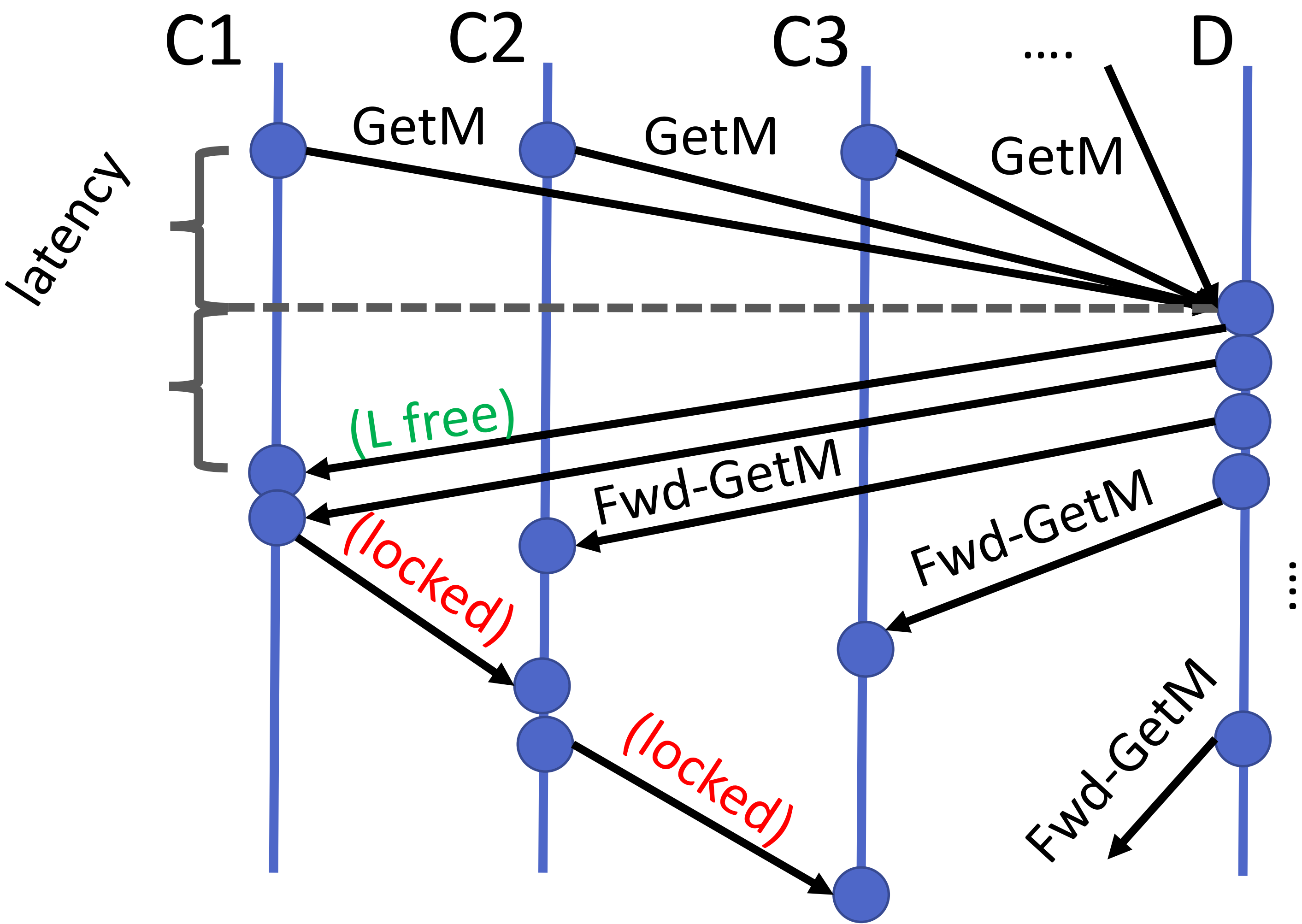
TAS analysis

Owner gets invalidated



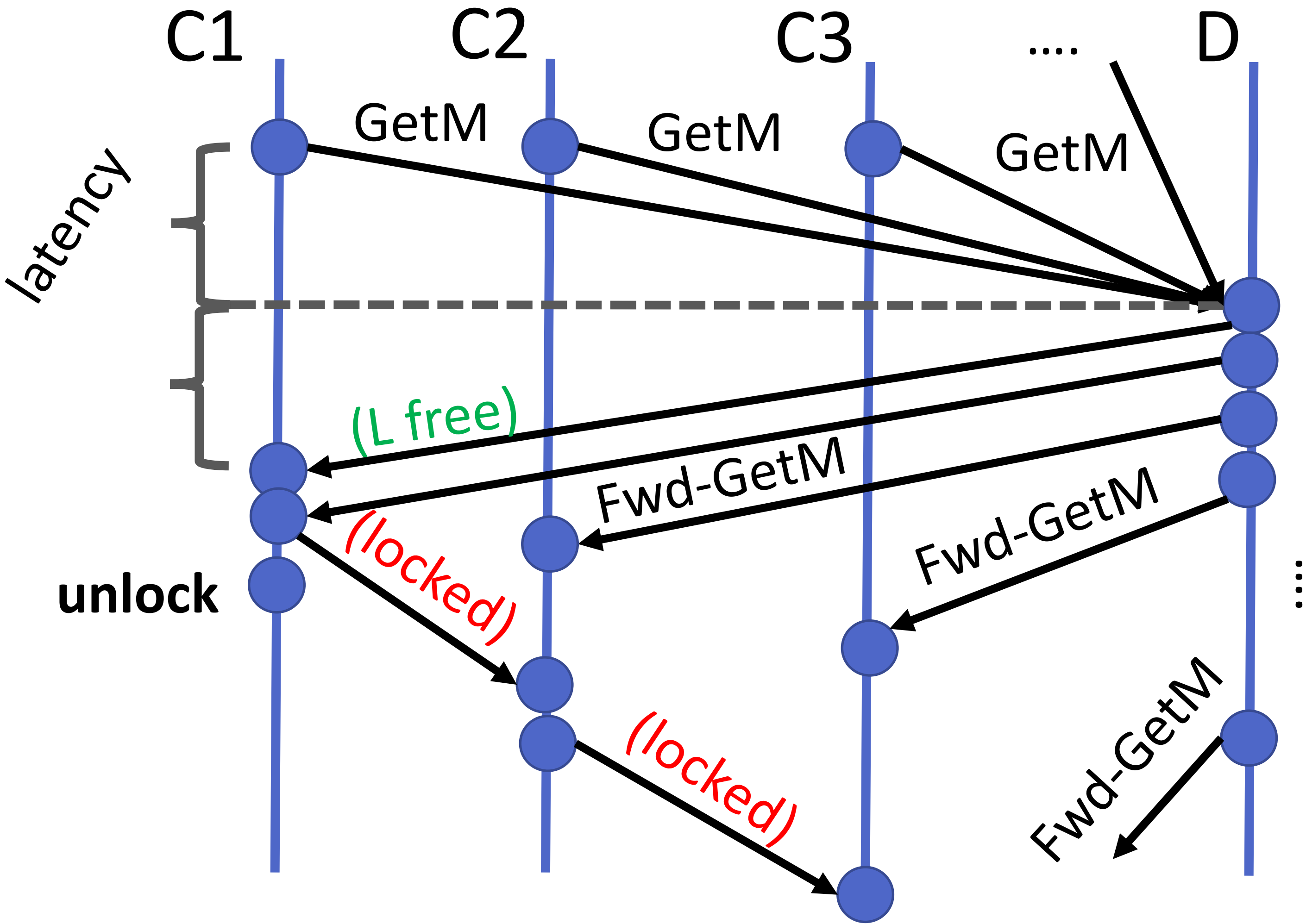
TAS analysis

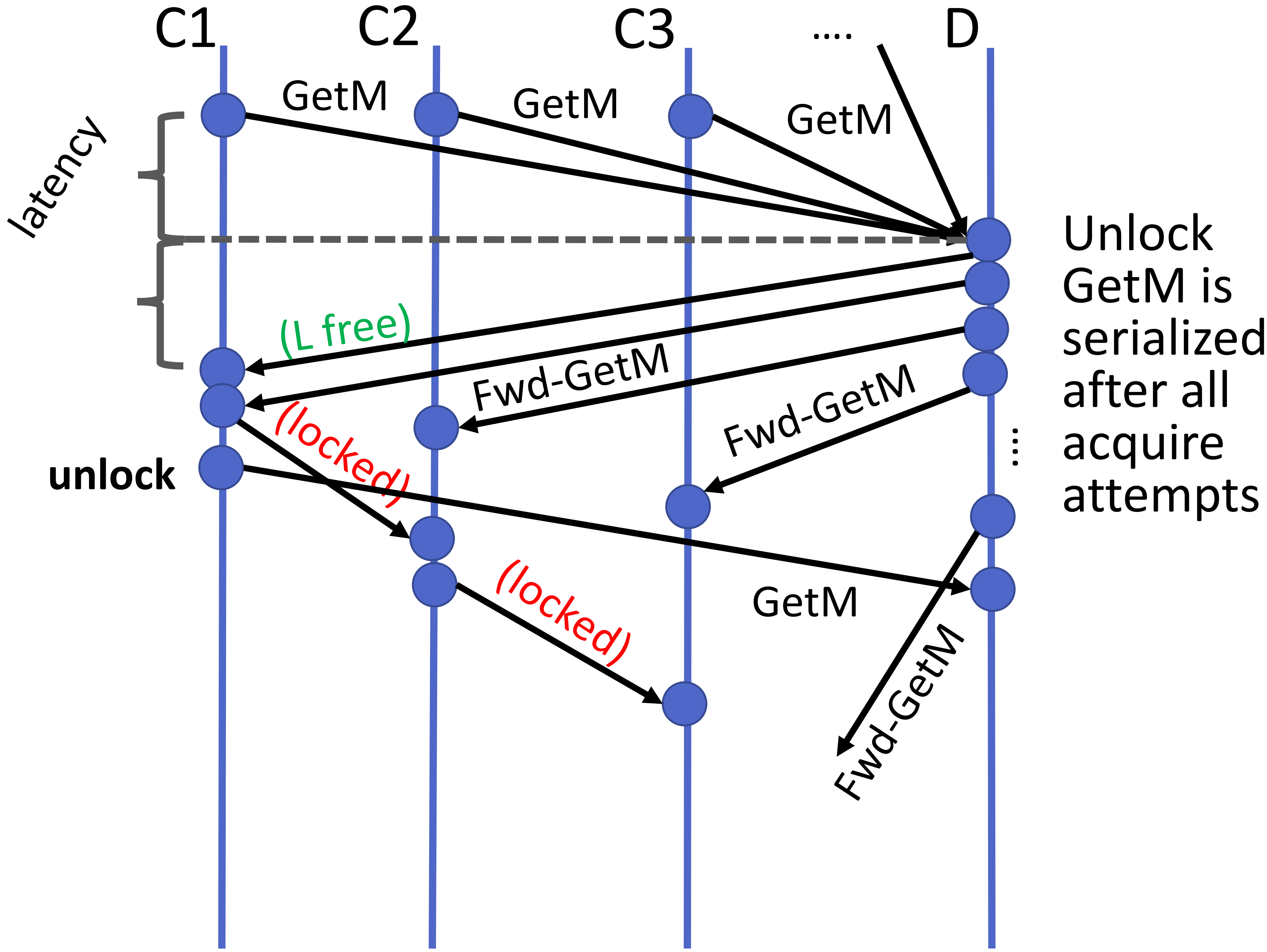
2nd in line gets invalidated

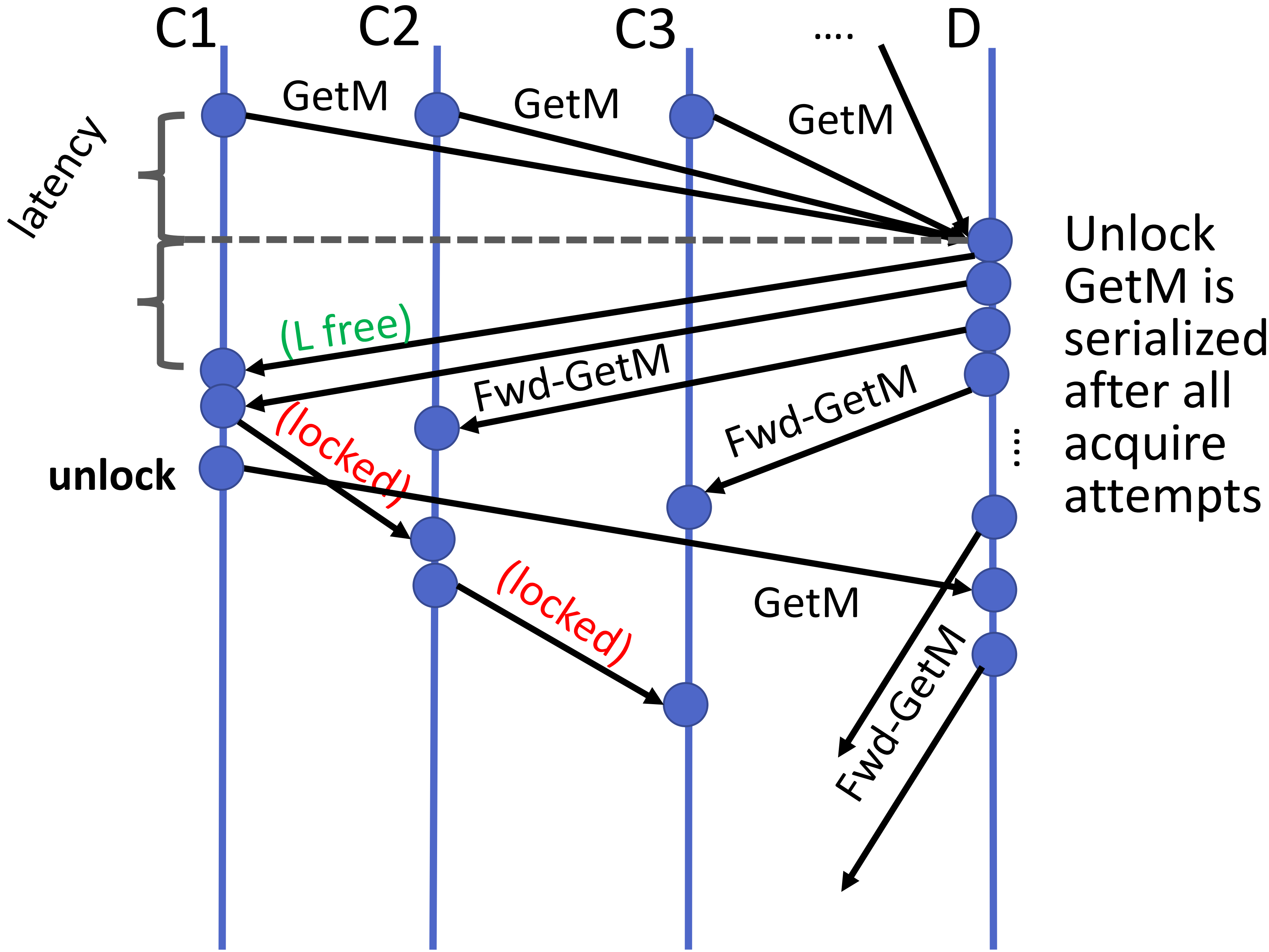


TAS analysis

Owner finishes CS, tries to unlock




C1
C2
C3
....
D
latency
GetM
GetM
GetM
Unlock
GetM is
serialized
after all
acquire
attempts
(L free)
Fwd-GetM
Fwd-GetM
(locked)
unlock
(locked)
GetM
Fwd-GetM


C1
C2
C3
....
D
GetM
GetM
GetM
latency
(L free)
Fwd-GetM
Fwd-GetM
(locked)
unlock
(locked)
GetM
Fwd-GetM
Unlock GetM is serialized after all acquire attempts

TAS analysis

- Coherence transactions by (failing) lock acquisitions create an *invalidation storm*, which forms an implicit queue of cores waiting for ownership of the line
- An unlock gets put at the end of the queue, and thus takes linear (N*L) time
  - N = # of contending cores
  - L = invalidation latency
  - Example multi-core case: N=20, L=20

TTAS lock

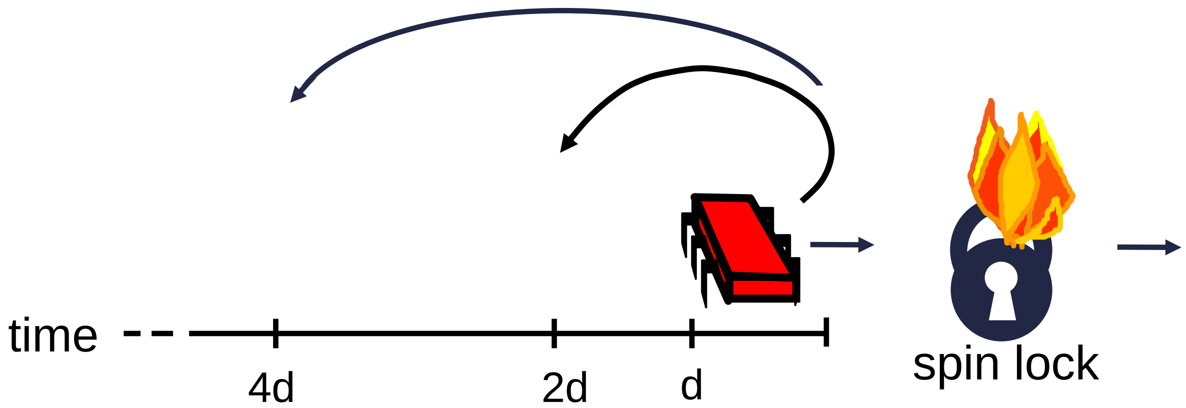
```
lock() {
  while (true) {
    while (L) { }
    if (CAS(&L, 0, 1)) return;
  }
}

unlock() {
  L = 0
}
```

TTAS analysis

- If all cores see lock free, all try to acquire it
  $\Rightarrow$ An invalidation storm as in TAS lock
- If CS is short (< length of N invalidations), the invalidation storm delays the lock release
- And thus the next acquisition
- And therefore increases the critical path length

Backoff

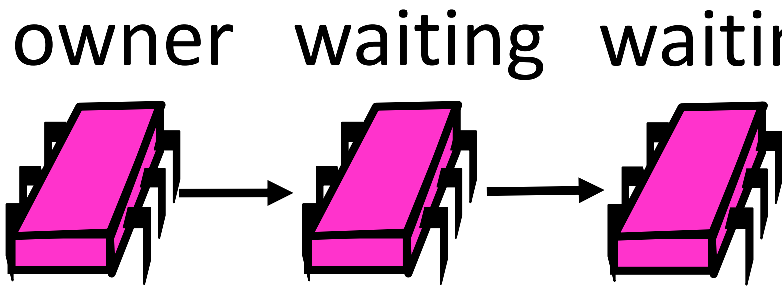
- Possible TTAS solution: exponential backoff
- If fail to lock, wait random duration before retry
- Fail again: double expected wait



- Very hard to get right in practice, requires tuning and values are machine-specific
- Doesn't avoid initial invalidation storm

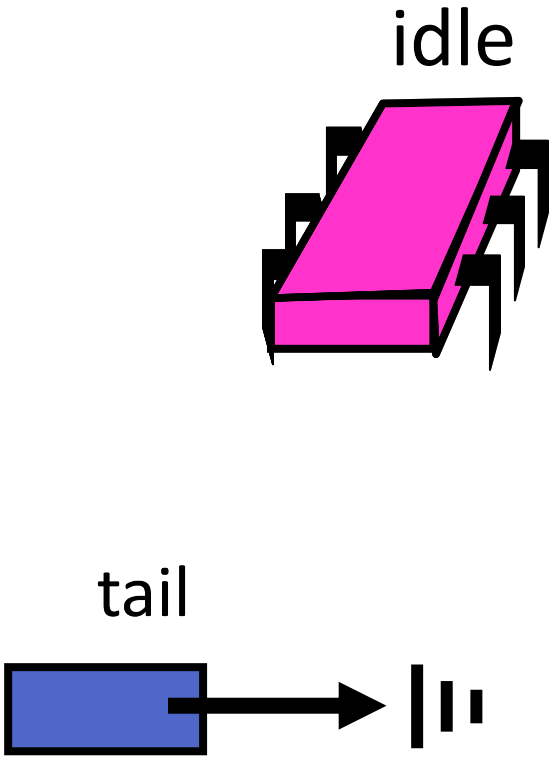
MCS lock [Mellor-Crummey & Scott '91]

- Goal: lock handoff in O(1) invalidations
- Put threads in explicit queue, so that unlock can transfer ownership to thread waiting at queue head

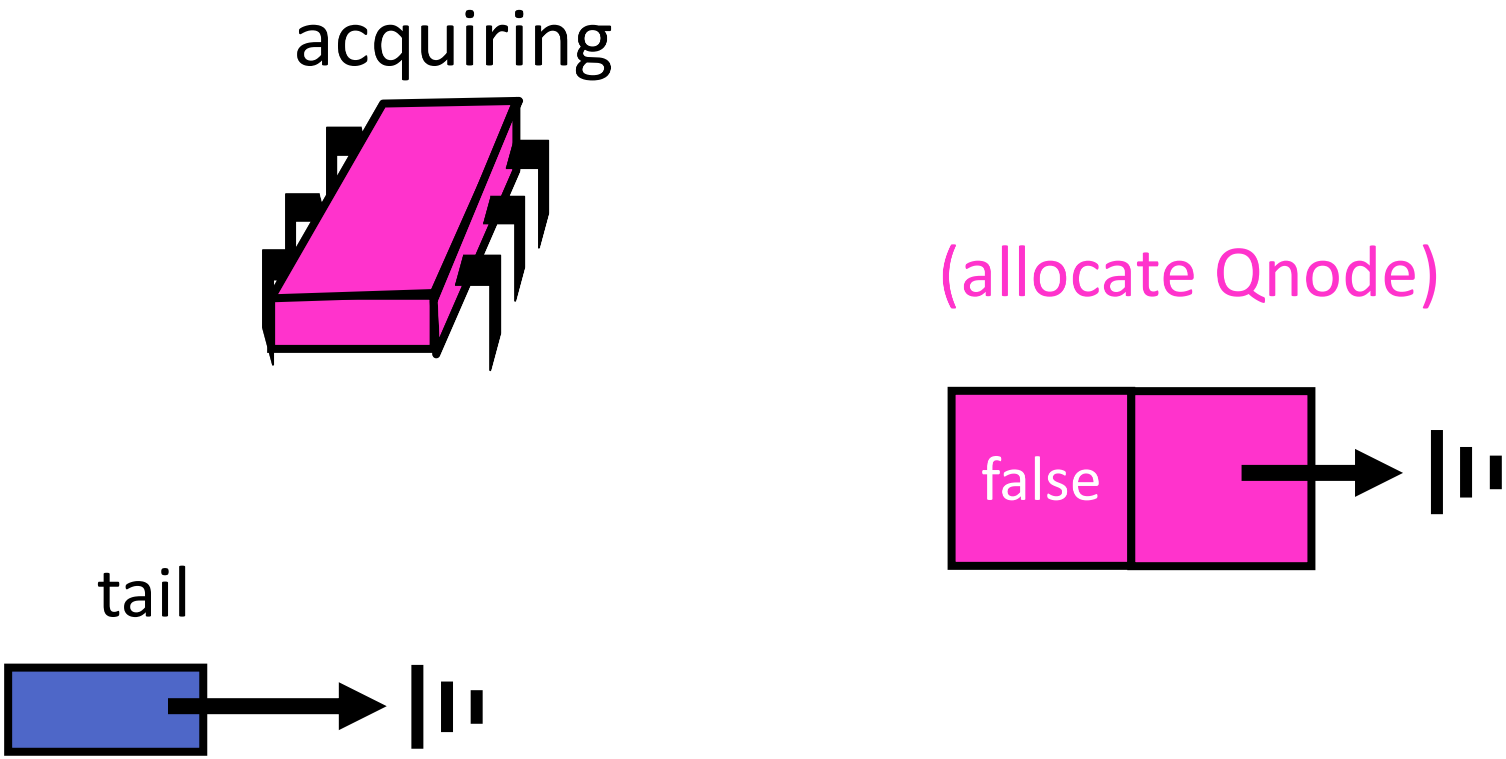


- Waiters do *local spinning* & don't get invalidated until reaching queue head

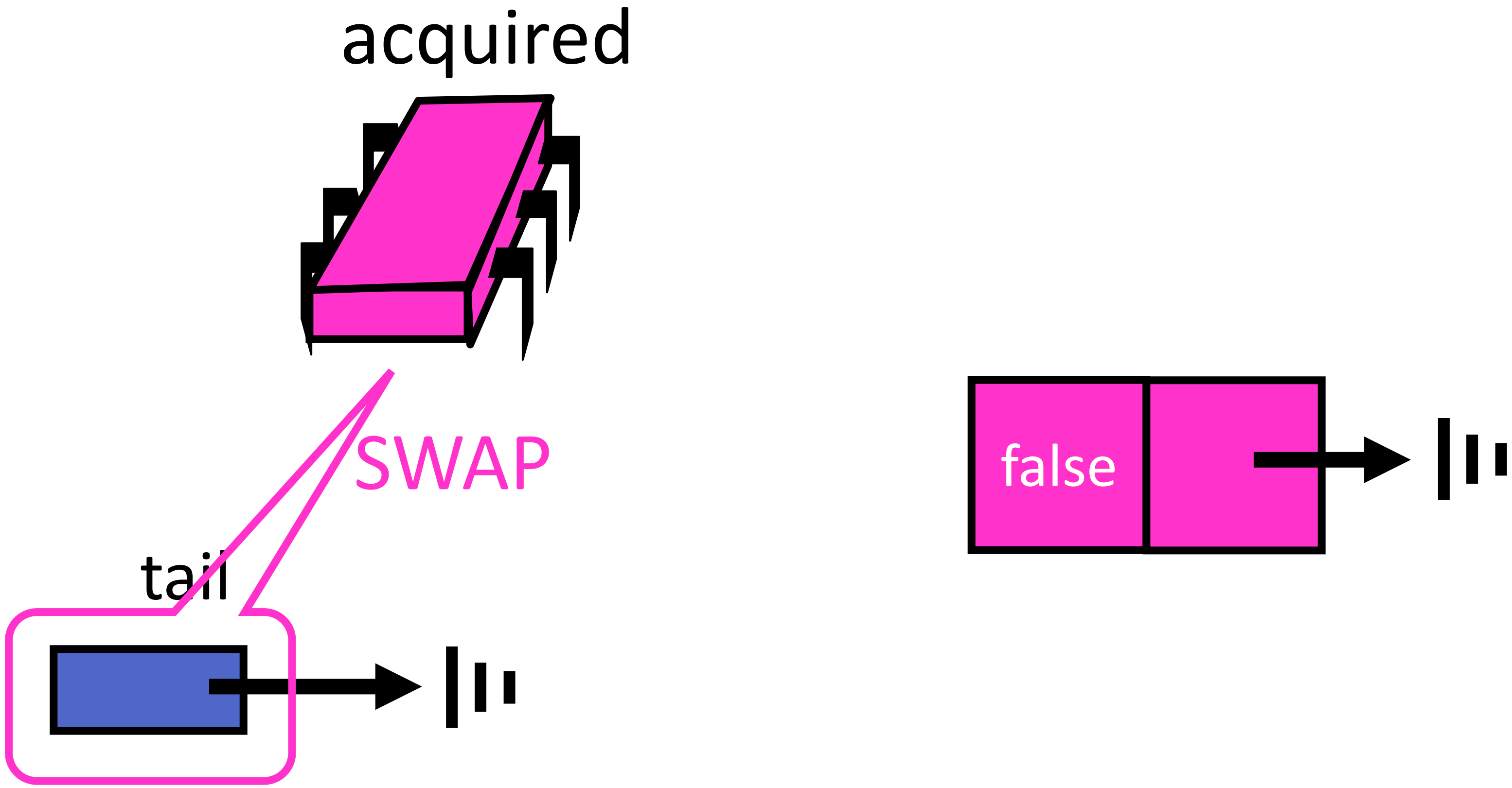
MCS lock [Mellor-Crummey & Scott '91]



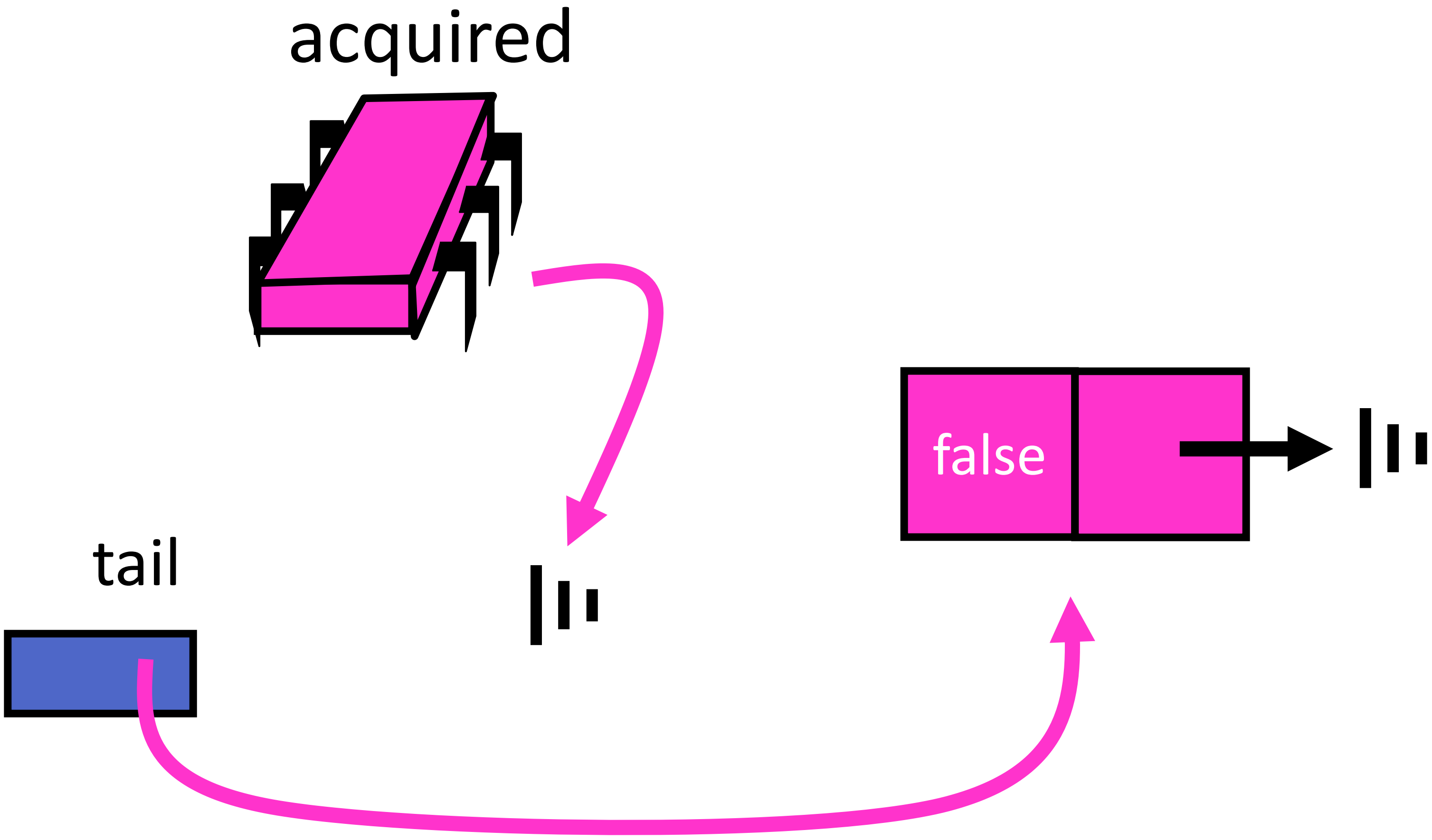
MCS lock [Mellor-Crummey & Scott ‘91]



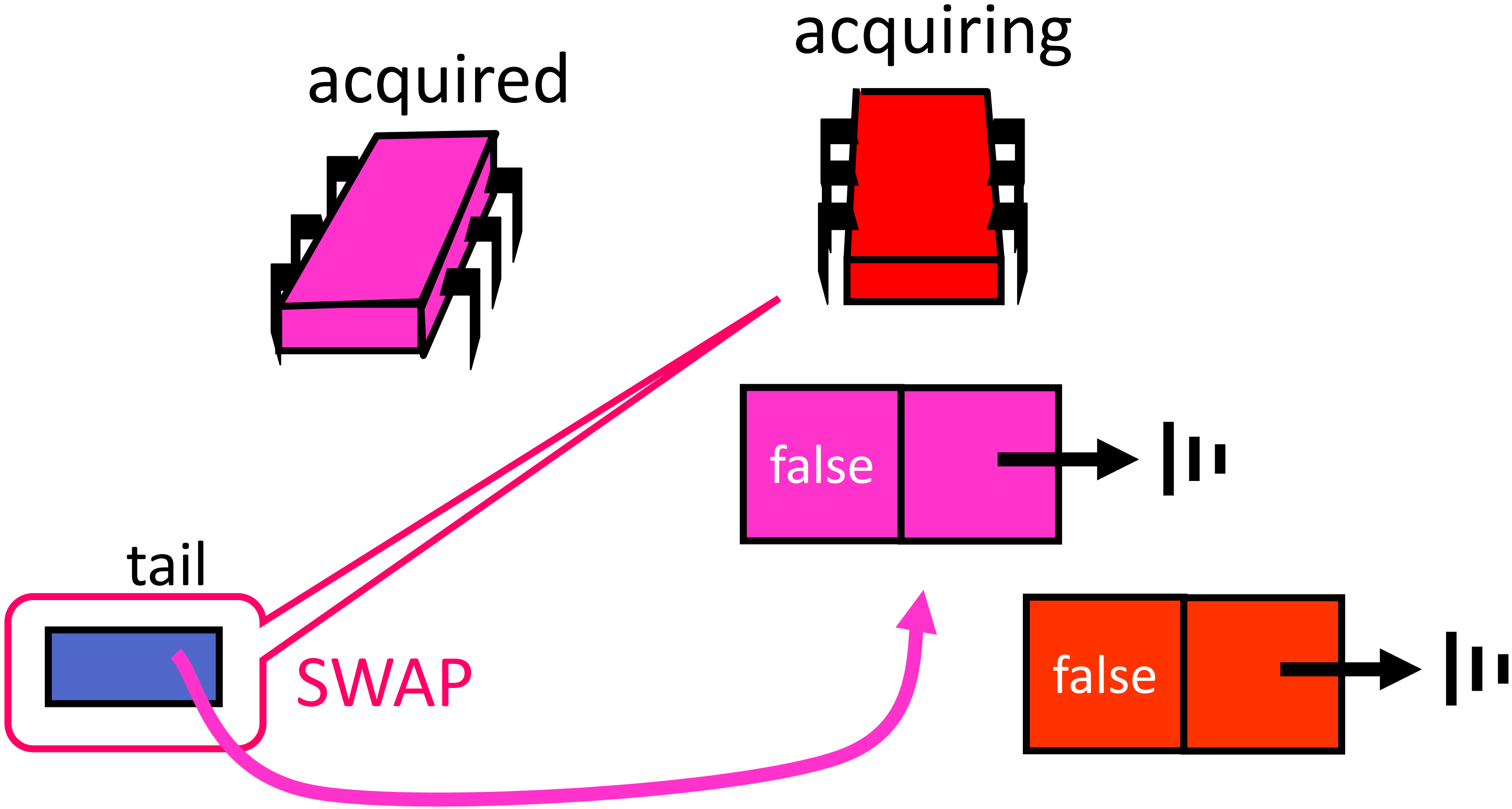
MCS lock [Mellor-Crummey & Scott '91]



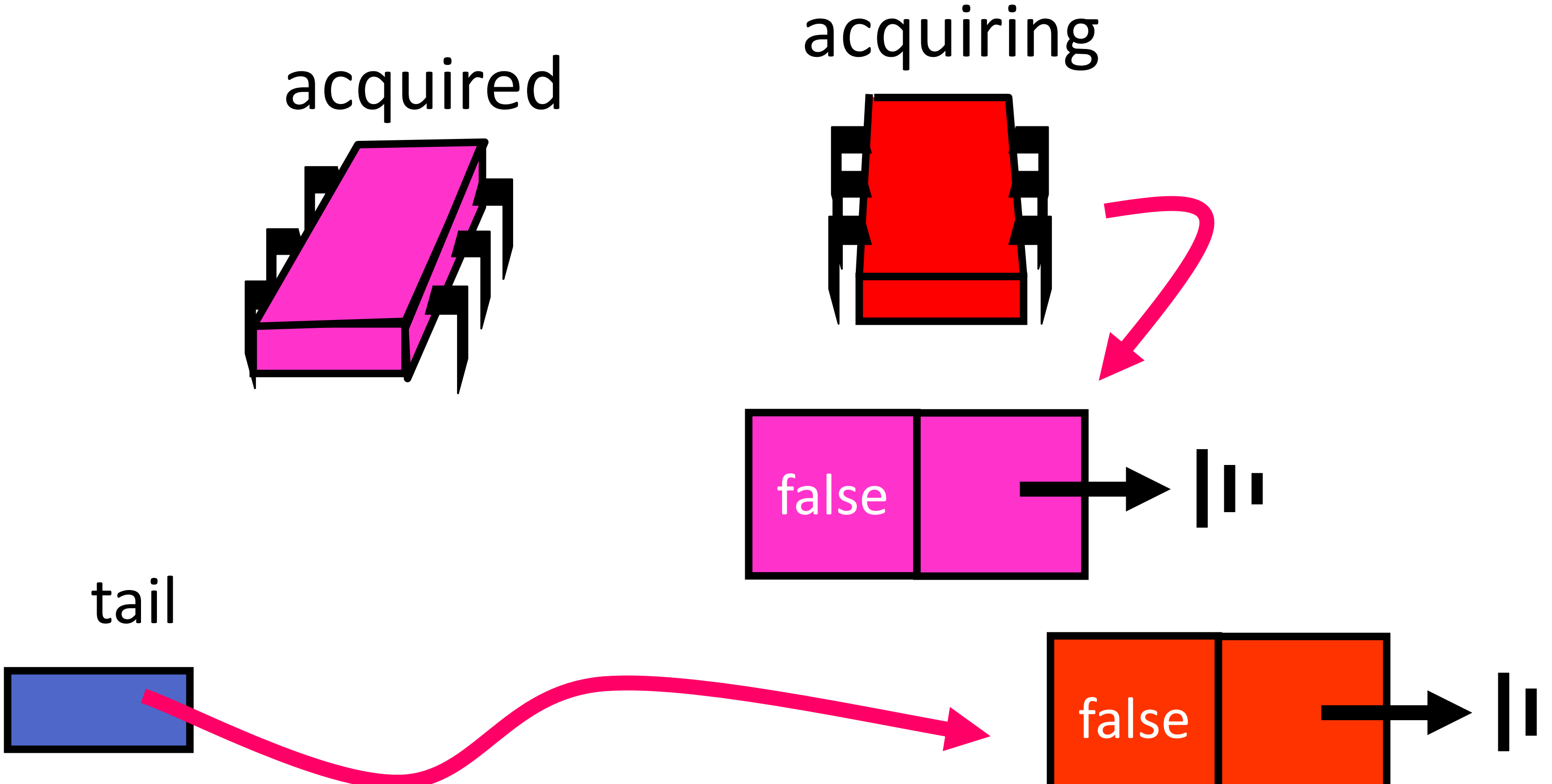
MCS lock [Mellor-Crummey & Scott '91]



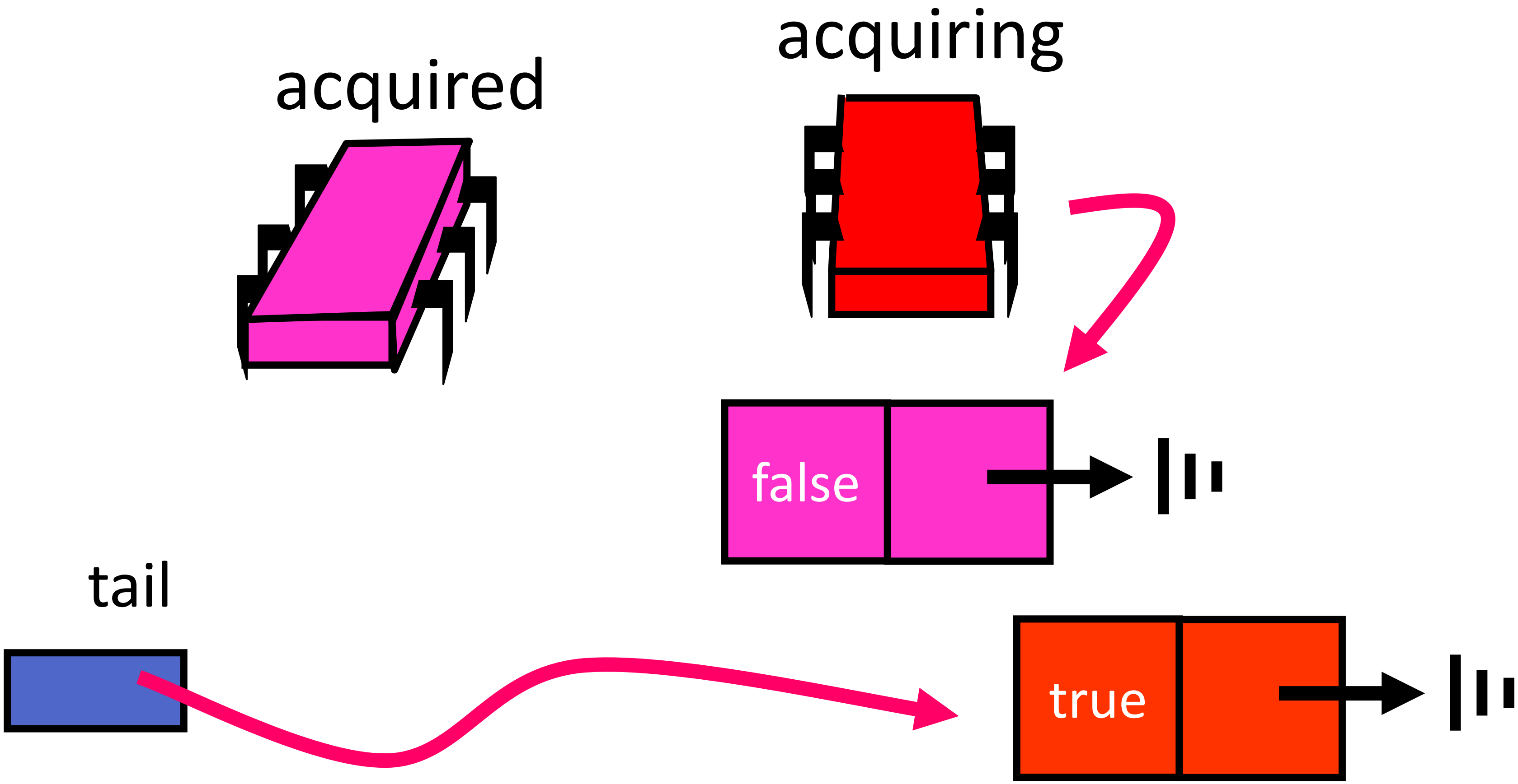
MCS lock [Mellor-Crummey & Scott '91]



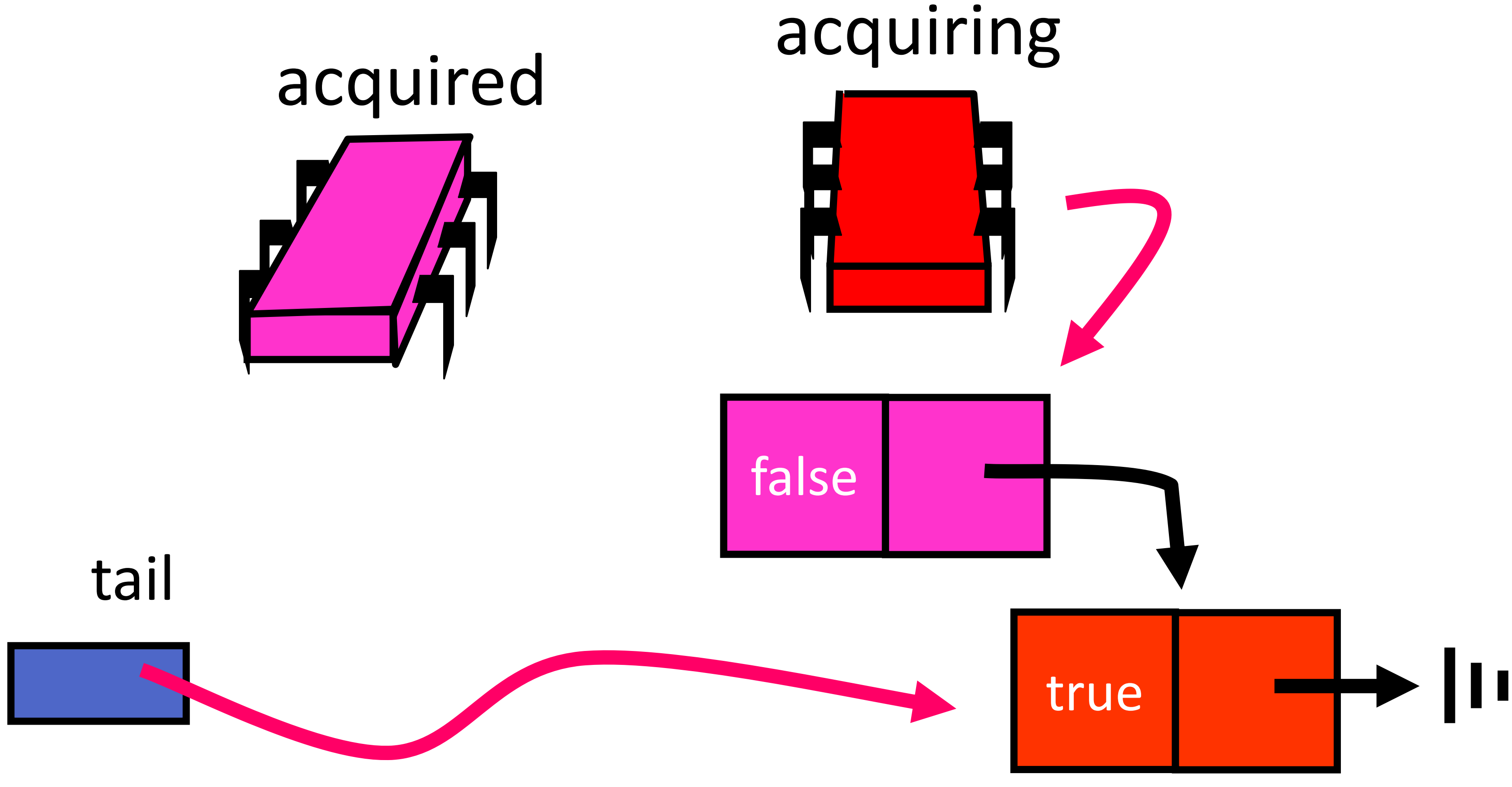
MCS lock [Mellor-Crummey & Scott '91]



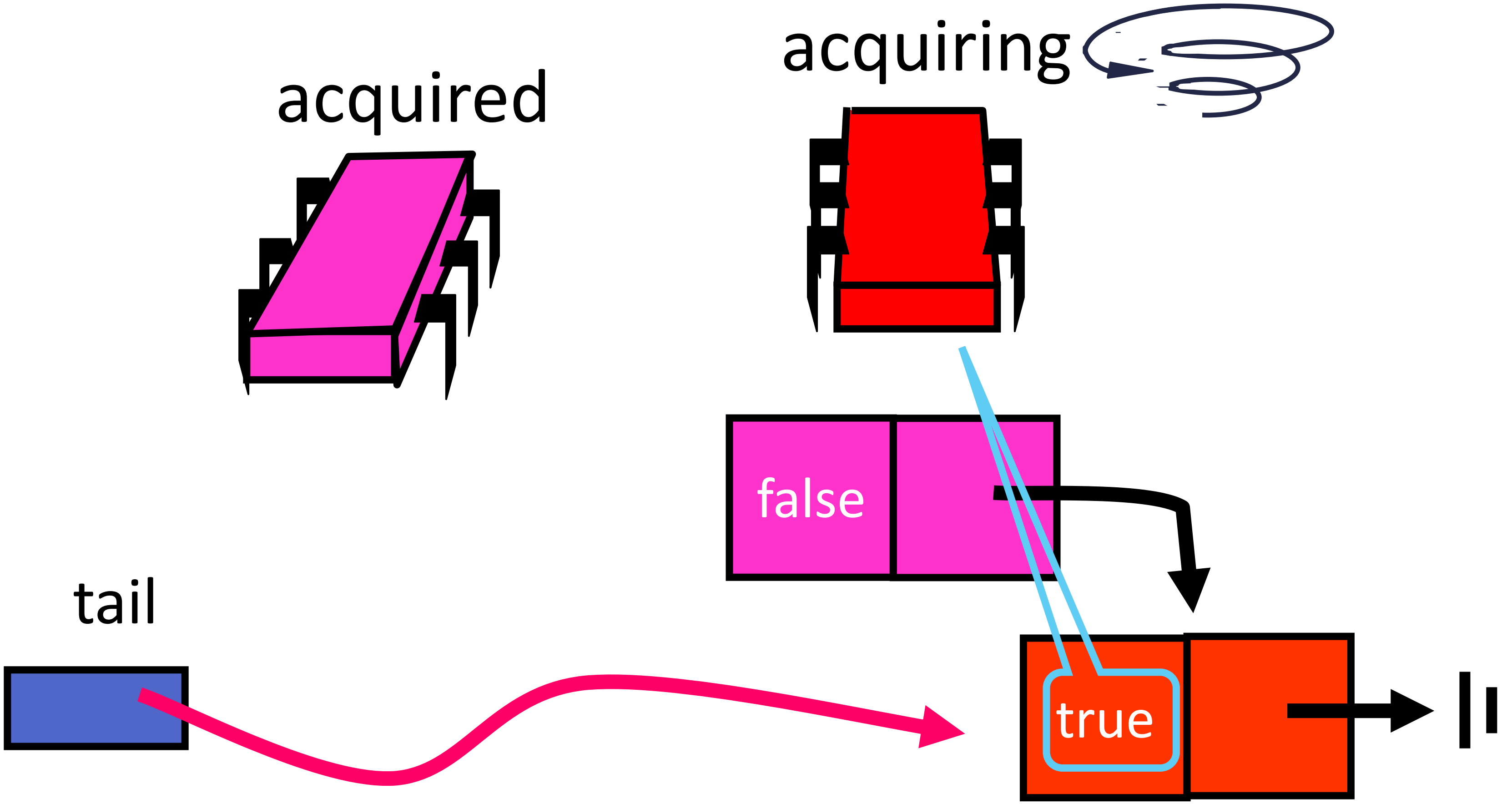
MCS lock [Mellor-Crummey & Scott '91]



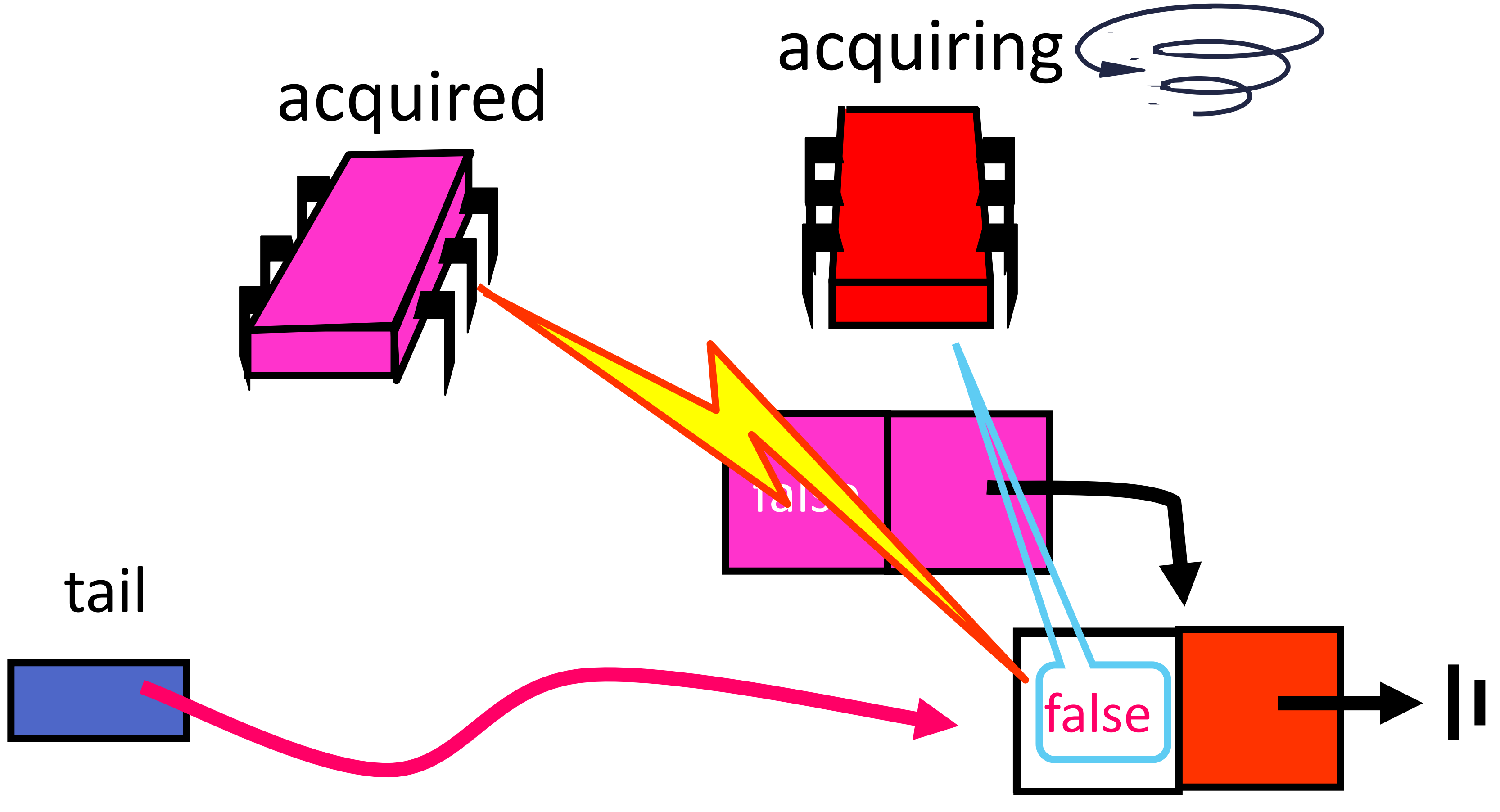
MCS lock [Mellor-Crummey & Scott '91]



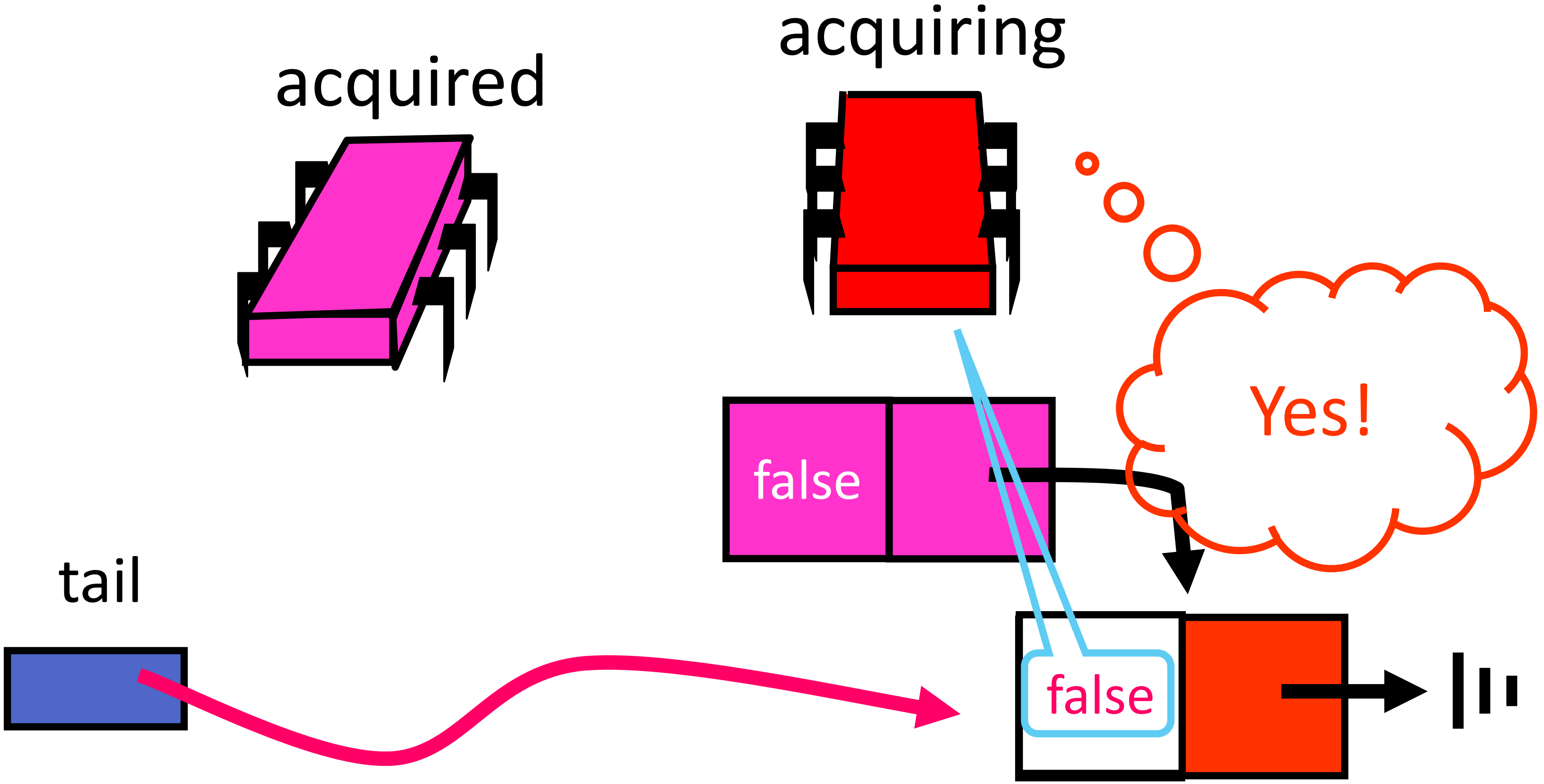
MCS lock [Mellor-Crummey & Scott '91]



MCS lock [Mellor-Crummey & Scott '91]



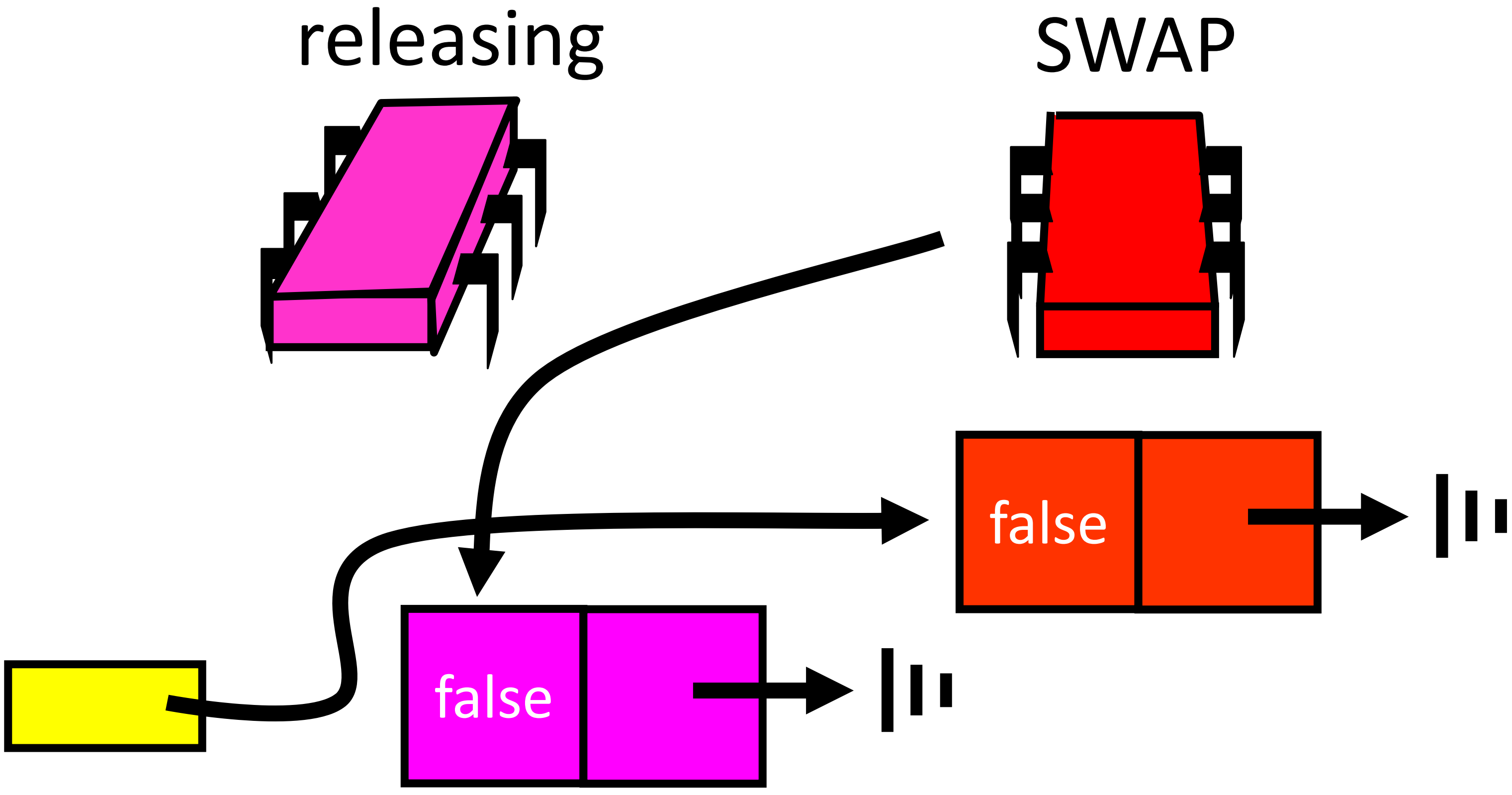
MCS lock [Mellor-Crummey & Scott '91]



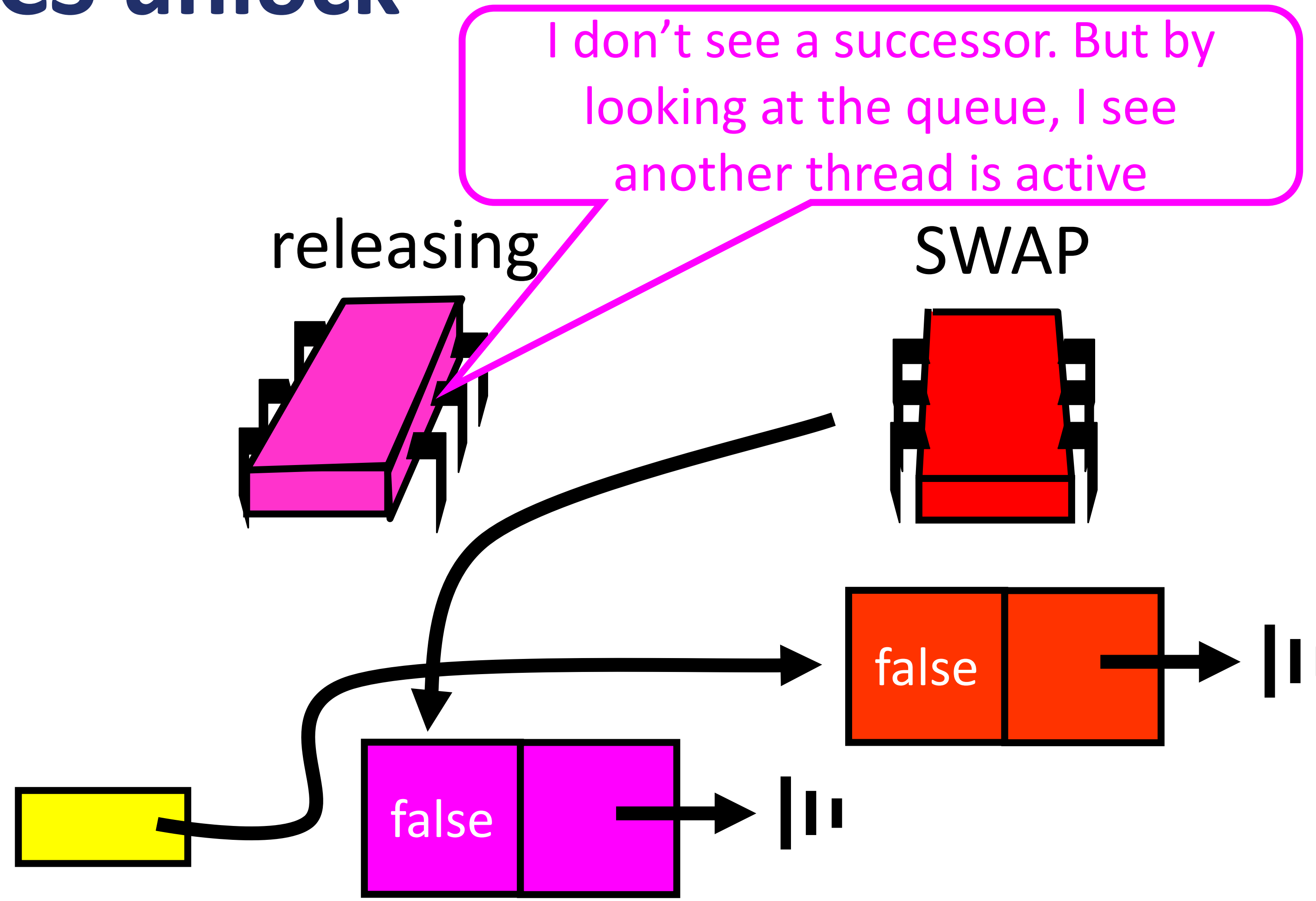
MCS lock

```
qnode : { qnode* next; bool lock; }

lock(qnode *n) {
  n->next = NULL
  pred = SWAP(&LOCK, n)
  if (pred) {
    n->lock = true
    pred->next = n
    while (n->lock) { }
} }
```

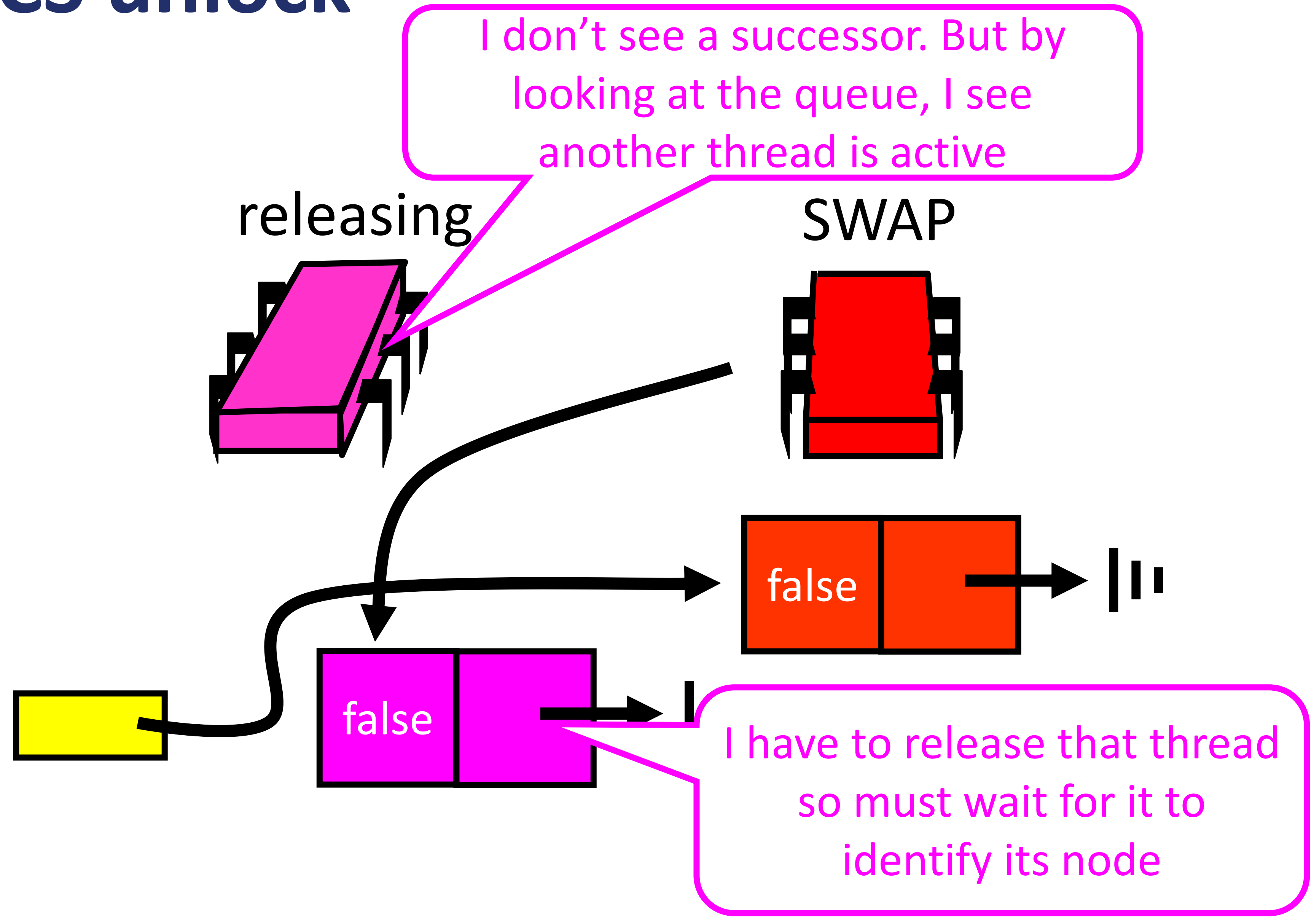
MCS unlock



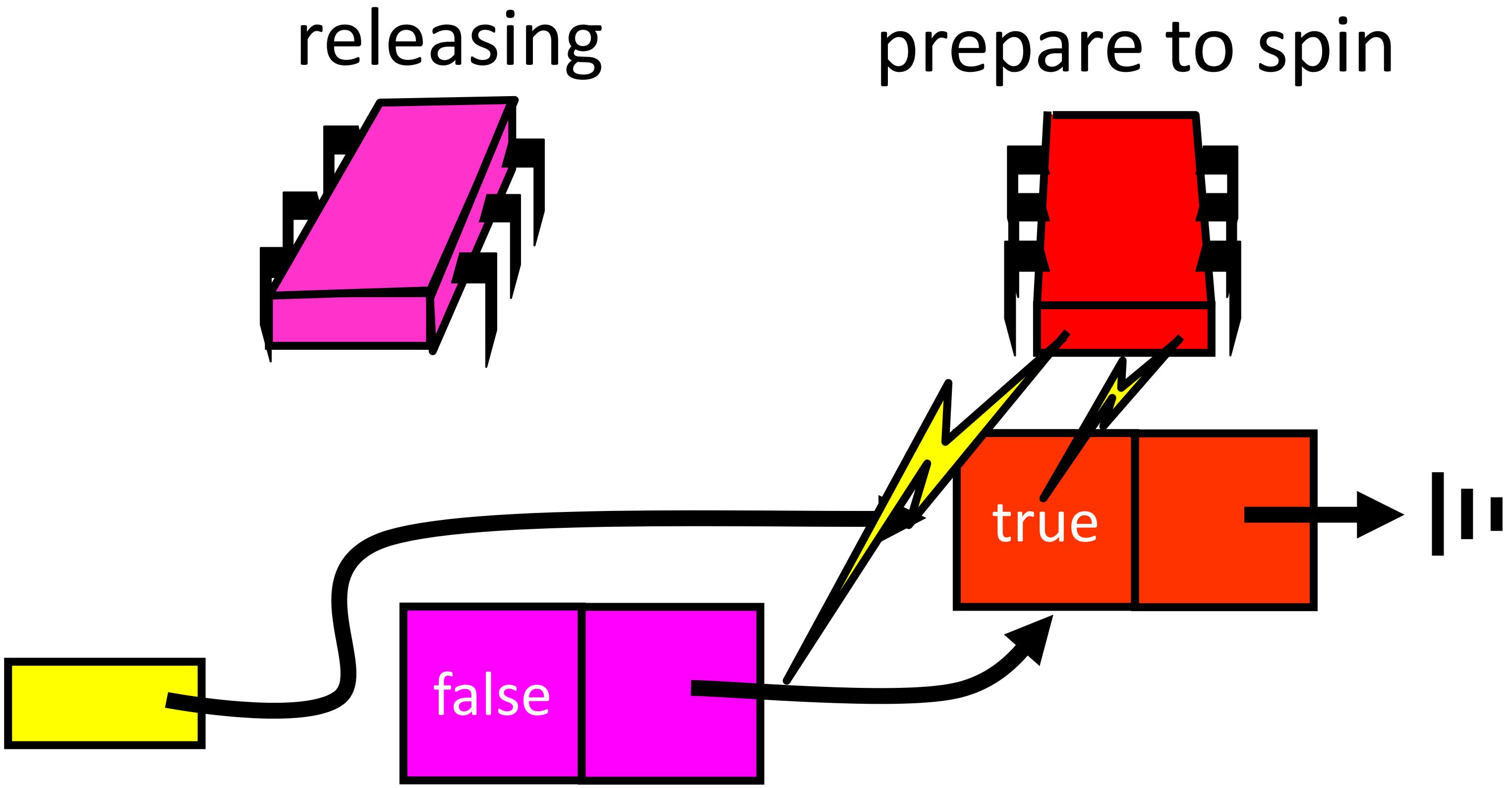
MCS unlock


I don't see a successor. But by looking at the queue, I see another thread is active
releasing
SWAP
false
false

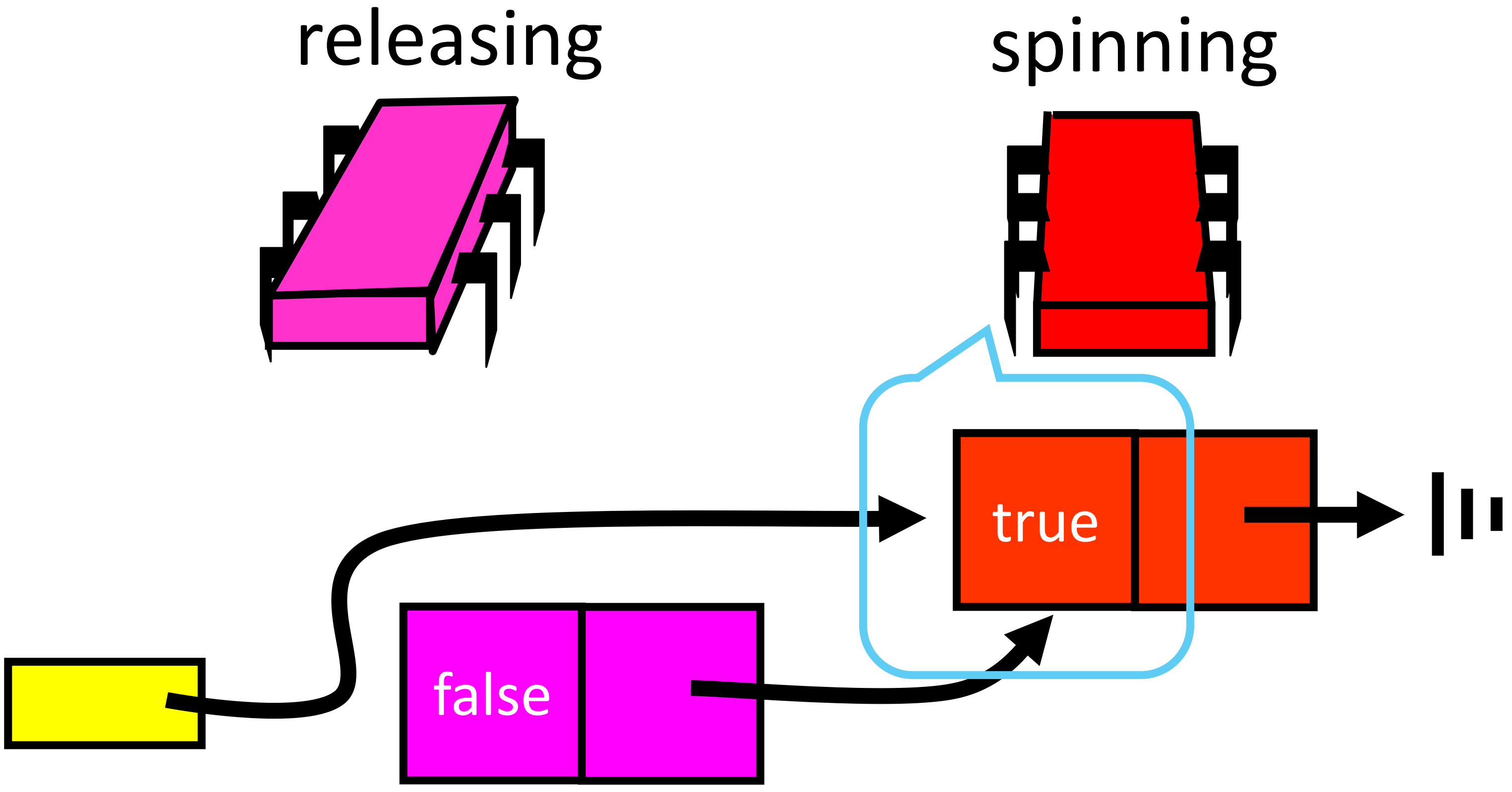
MCS unlock


I don’t see a successor. But by looking at the queue, I see another thread is active
releasing
SWAP
false
false
I have to release that thread so must wait for it to identify its node

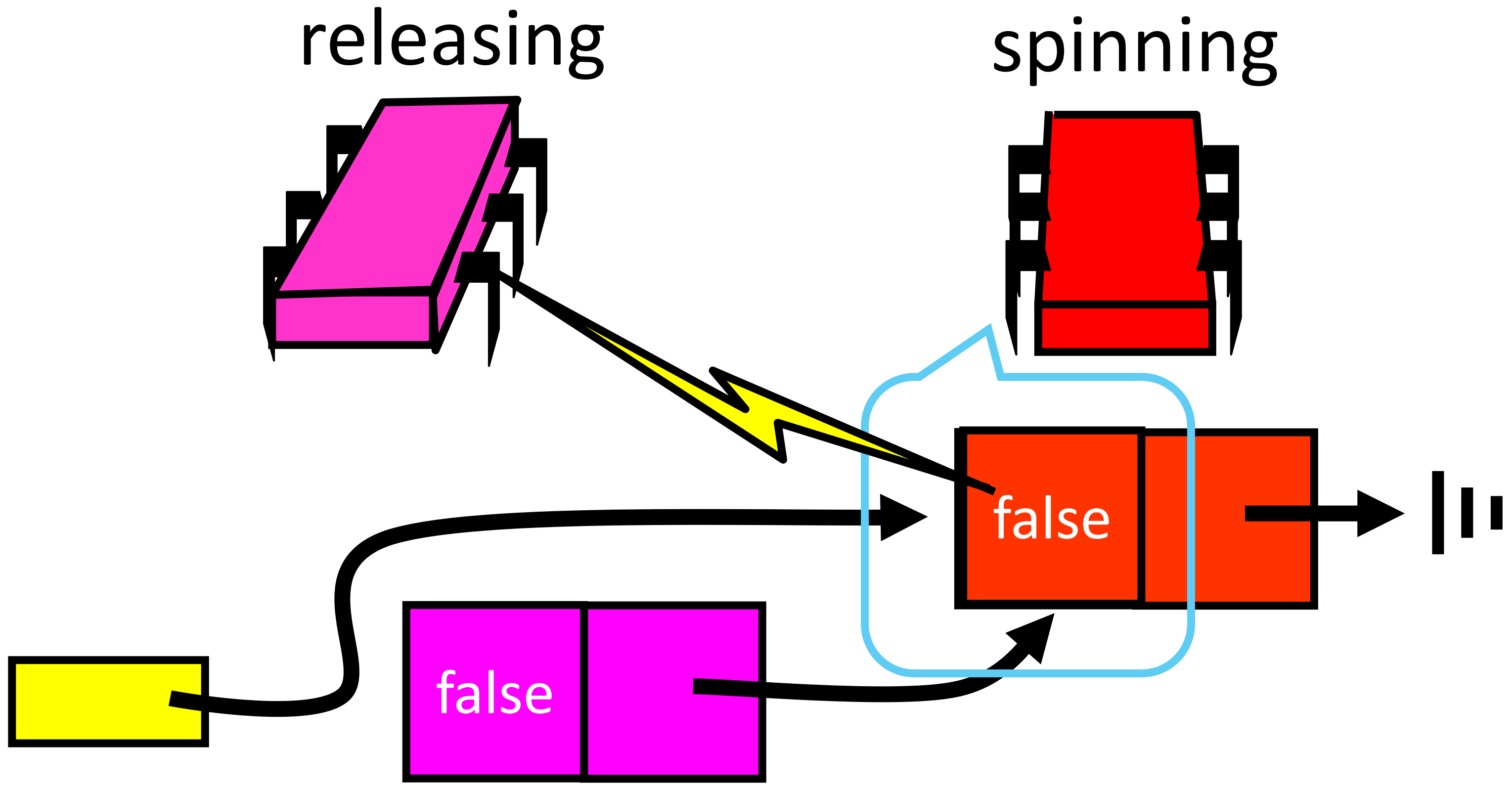
MCS unlock



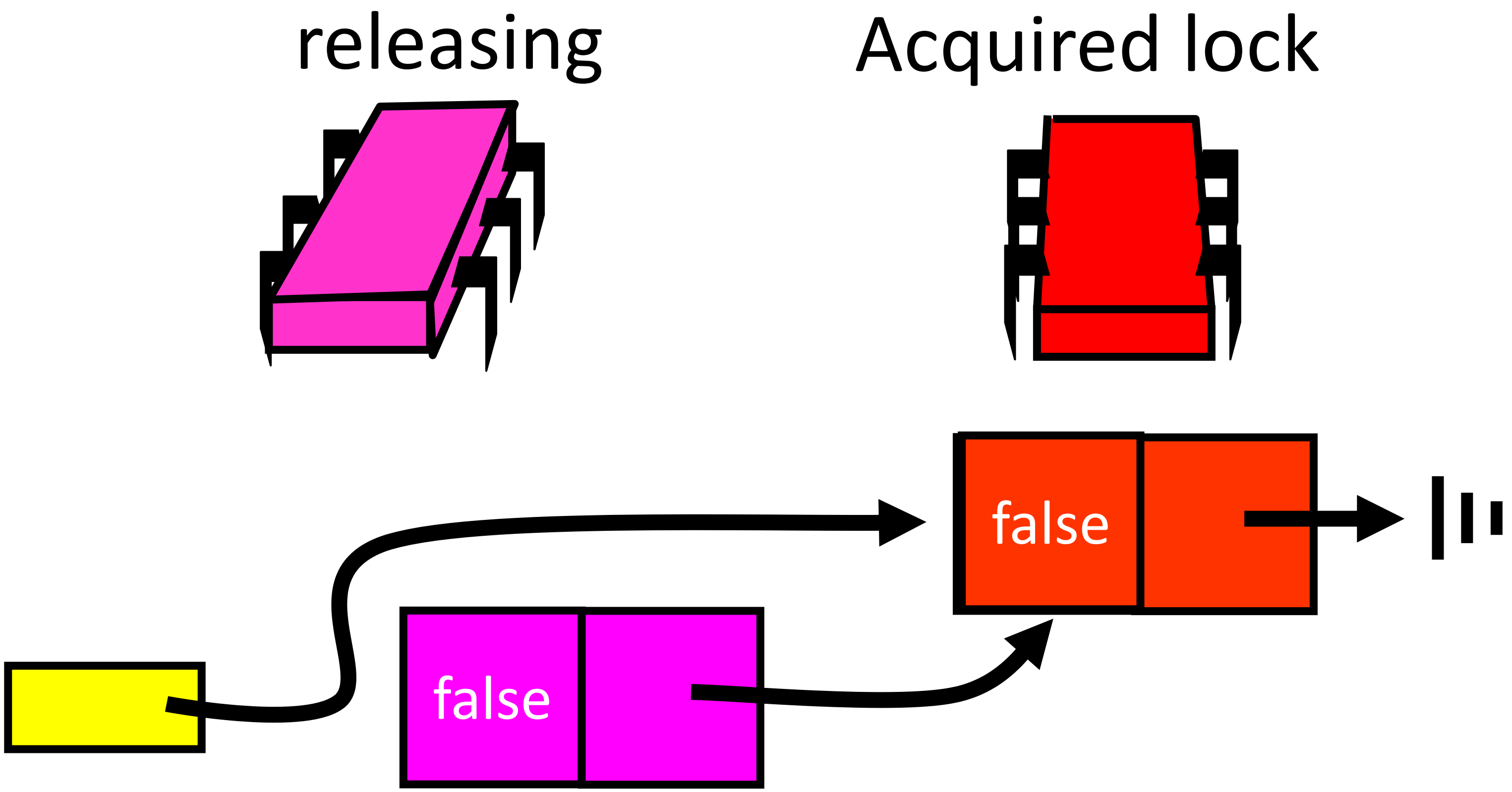
MCS unlock



MCS unlock



MCS unlock



MCS unlock

```
qnode : { qnode *next; bool lock; }

unlock(qnode *n) {
  if (n->next == NULL) {
    if (CAS(&LOCK, n, NULL)) return;
    while (n->next == NULL) { }
  }
  n->next->lock = false;
}
```

MCS lock questions

- Why isn't the contended SWAP in lock() a problem? (Invalidation storm, etc.)

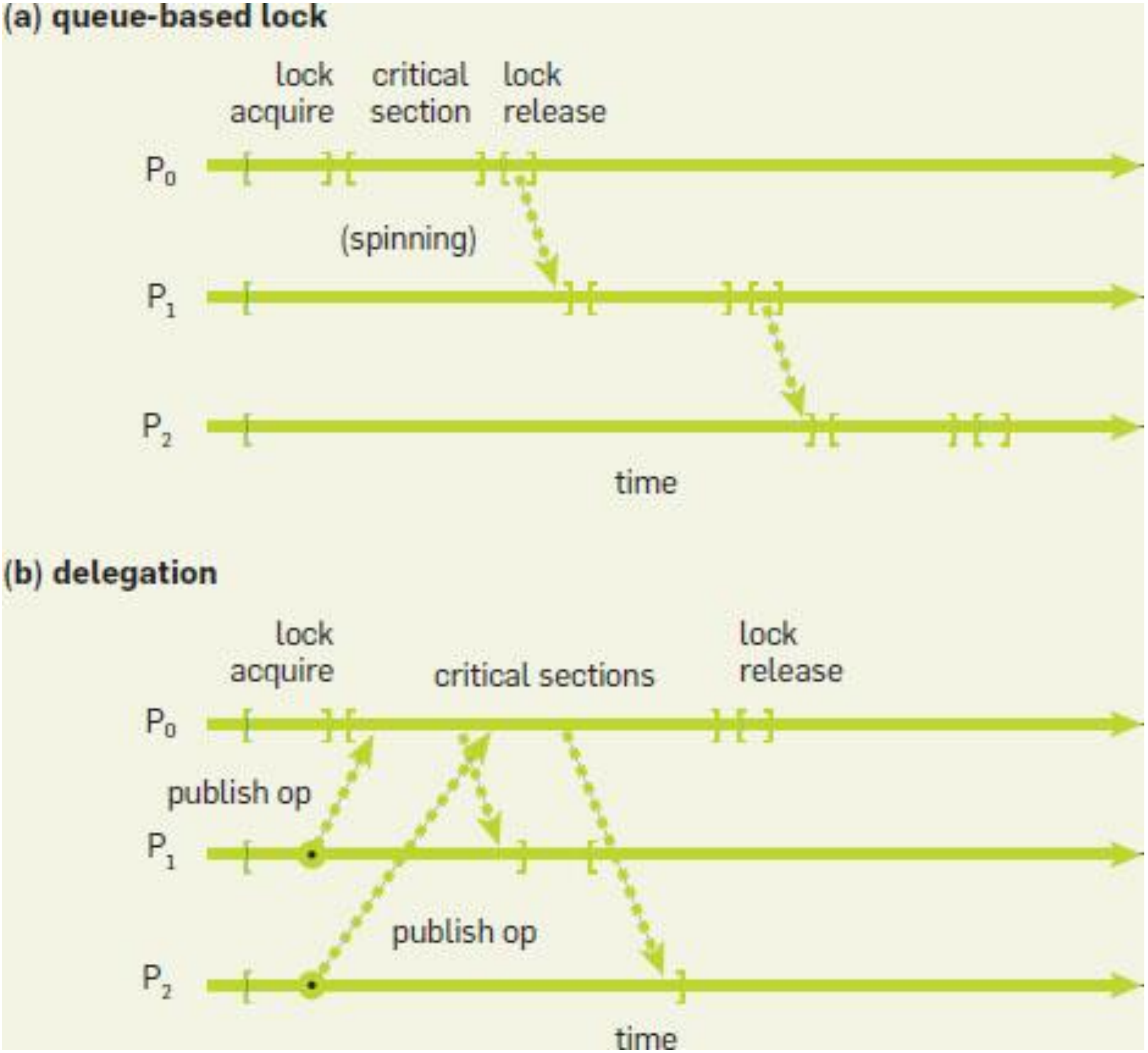
- Exercise: design a lock with queue links in reverse order
  - If B after A in queue, B spins on A's node
  - Problem with this approach: B doesn't spin on its own memory
  - Problem in **DSM** (distributed shared memory) model, where there are no caches and each memory location is either local or remote
    - Modern HW doesn't work this way

Delegation

Delegation

- Idea: Since CS are serialized anyway, just let a single thread run them
  - Thread that gets lock turns into a *server* and executes the operations of the waiting threads on their behalf
- Benefits:
  - Remove many lock acquisition/release from critical path
  - Might speed up CS (data hot in server's cache)
  - Enables *semantic optimizations*

Delegation intuition

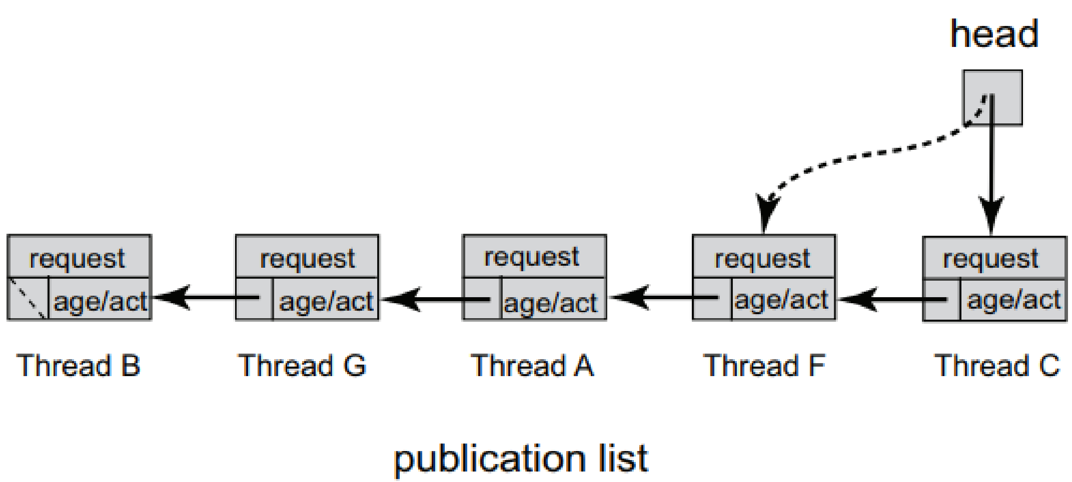


Flat combining (FC) [Hendler, Incze, Shavit, Tzafrir ‘10]

- Maintains *publication list*
  - Nodes contain *request* and *response* fields

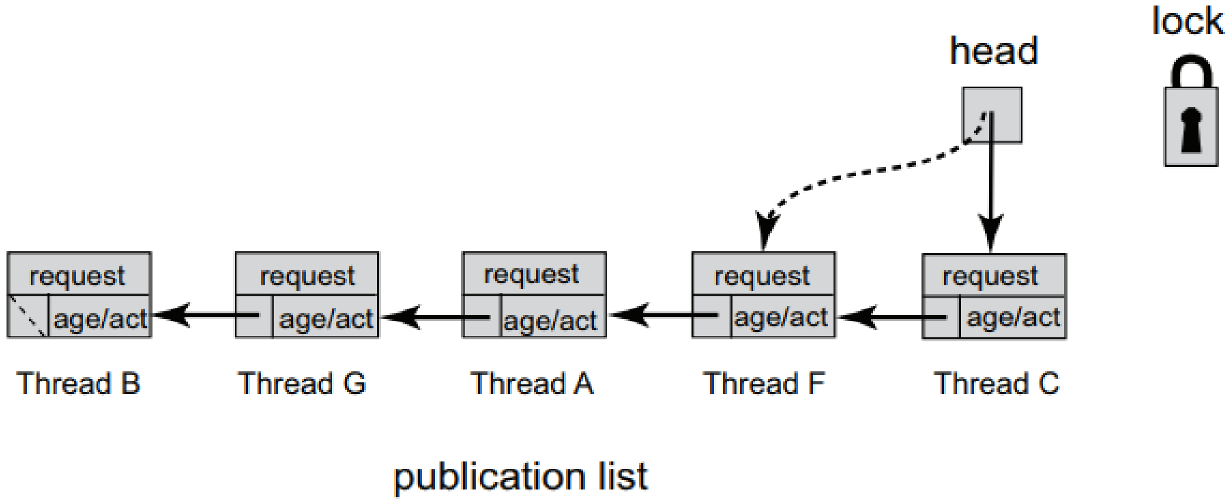
Flat combining

- Lock:
  1) Add node to publication list (Treiber-stack)



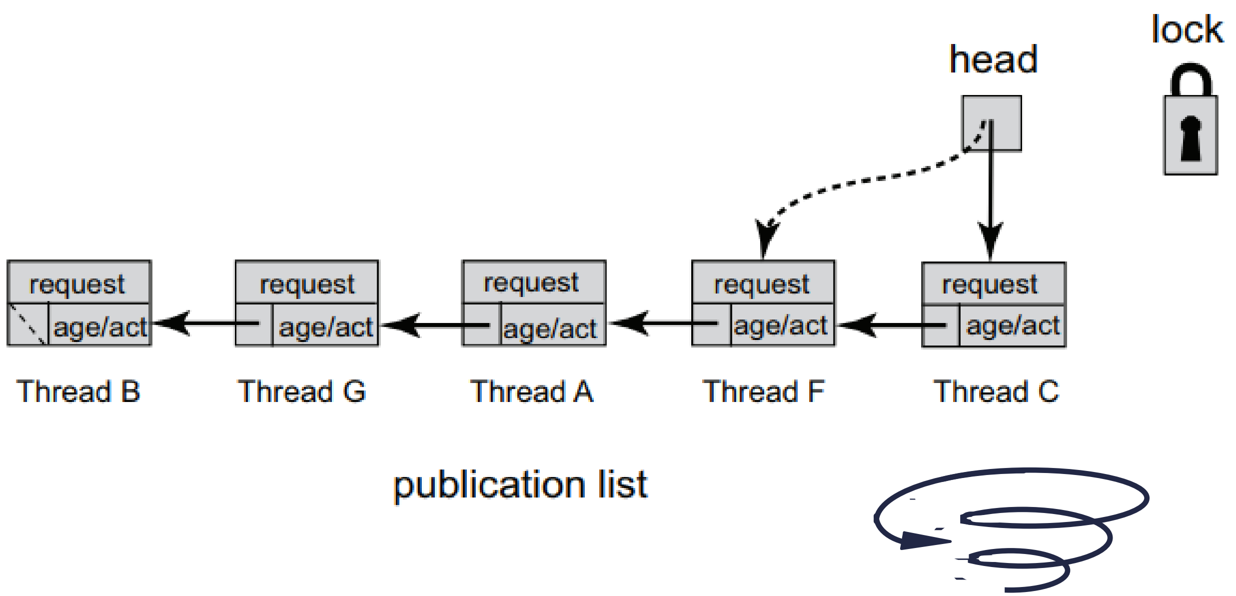
Flat combining

- Lock:
  2) Check if lock taken



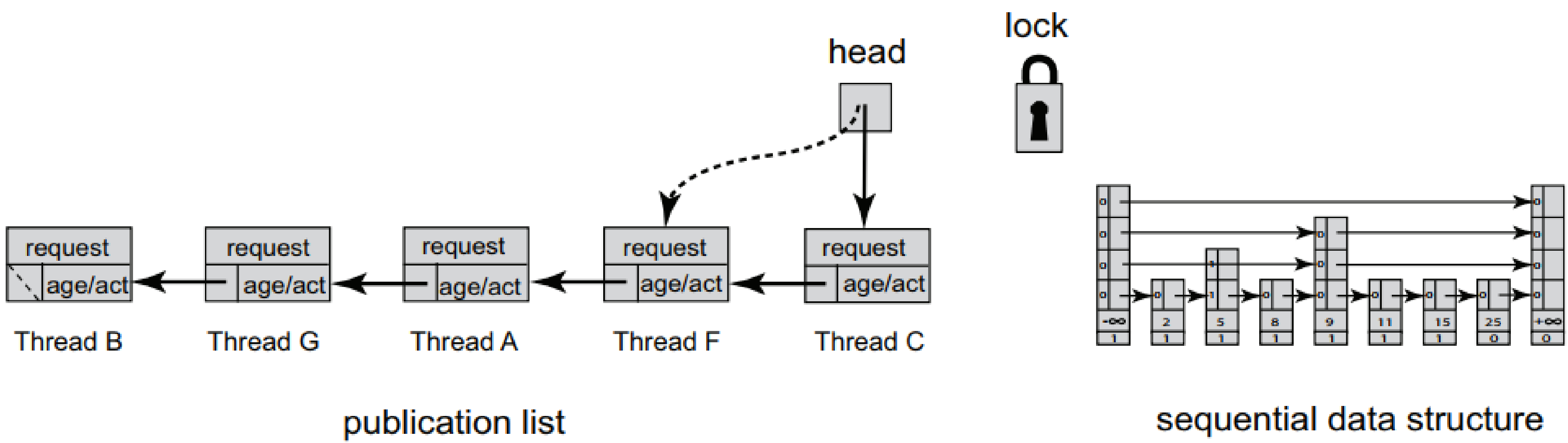
Flat combining

- Lock:
  3) If lock taken, spin waiting for answer



Flat combining

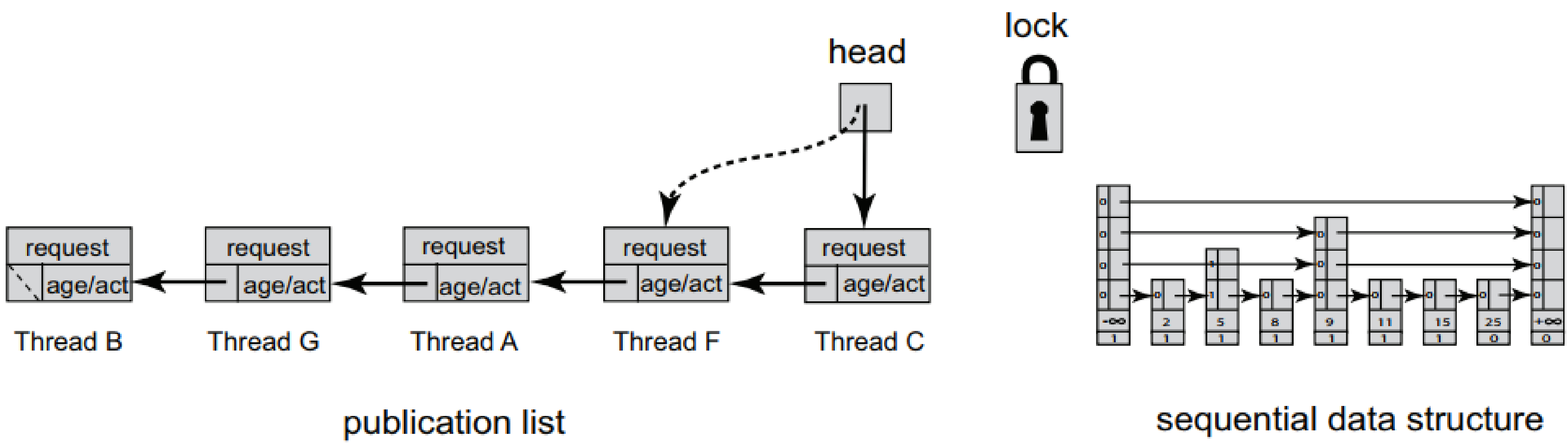
- Lock:
  4) If lock not taken, acquire it



Flat combining

- Lock:

  5) Scan publication list, applying each operation request, and writing the response

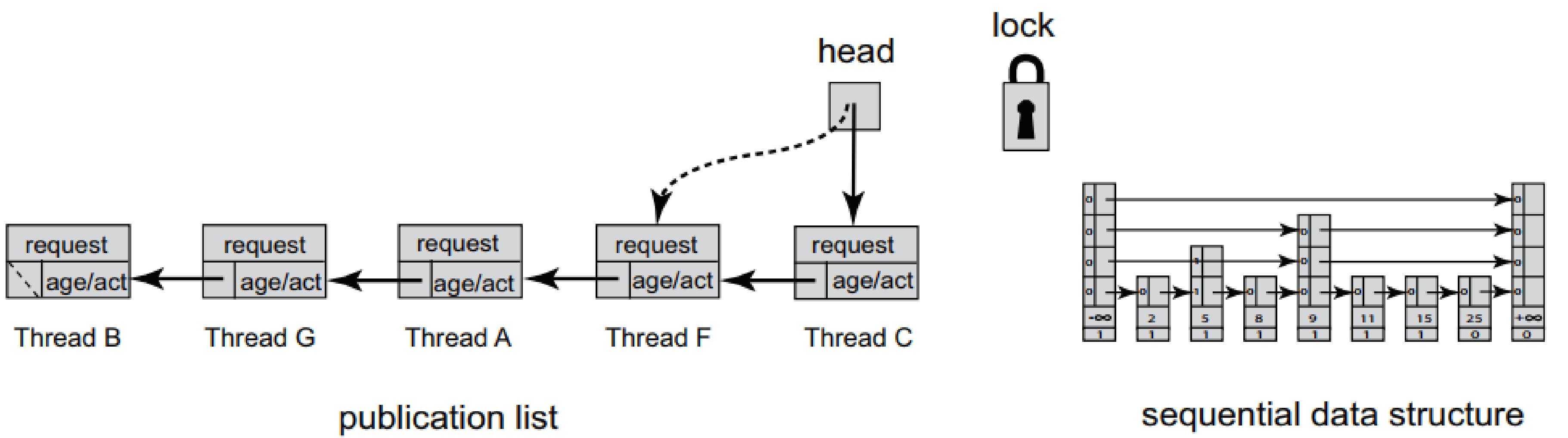


Flat combining

- CAS contention on publication list head?

  6) When done, don't remove your record;

     Next time just reuse it



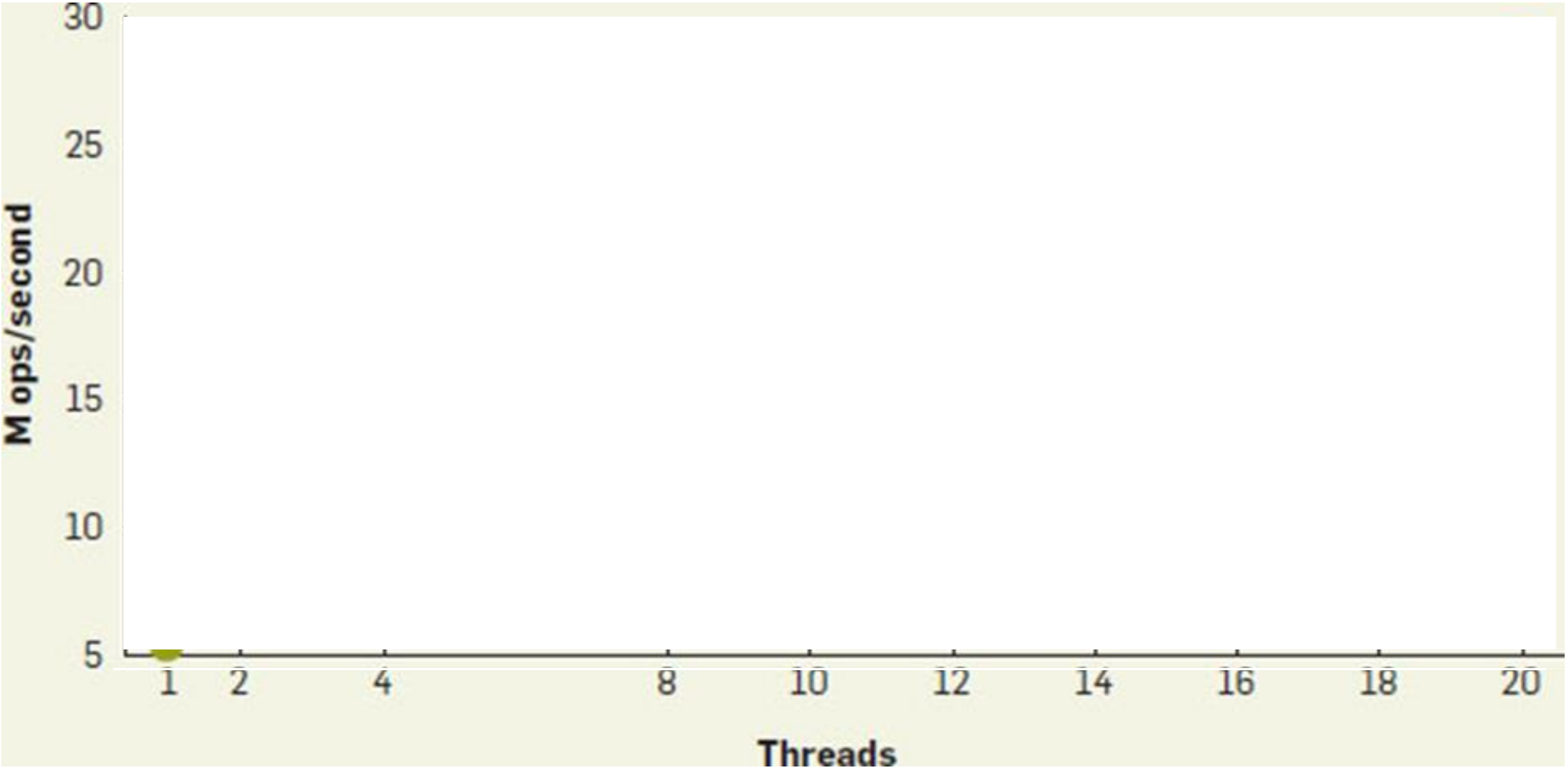
Flat combining issues

- Starvation
  - Combiner:
    - Stop helping after *K* operations
  - Waiting when combiner has left:
    - Check lock every once in a while, if free try to become combiner
- GC publication list:
  - Combiner does it every once in a while

Two-lock queue example

- 10-core x 2 hyperthreads CPU
- Enqueue/dequeuer benchmark
- Repeatedly
  - Enqueue
  - Do random work
  - Dequeue
  - Do random work
- Measure *throughput* (sum of queue ops per second over all threads)

Two-lock queue example



Semantic optimizations

- Combiner's global view allows it to optimize by leveraging data structure semantics
- *Combining*: apply many ops as one
  - inc() inc() inc() -> add(3)
- *Elimination:* allow opposing operations to cancel out
  - add(x) remove(x) => no memory change required
- Deferring

Deferring

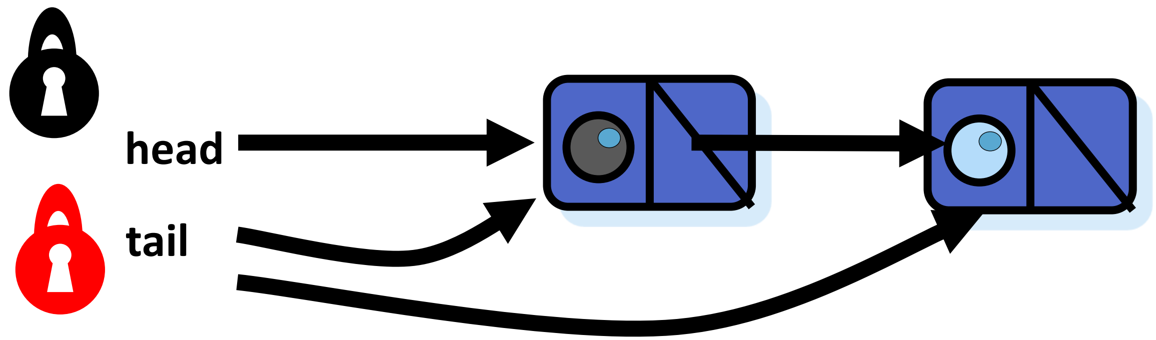
- **Idea:** Operation with void response (like enqueuer) can deposit request *and not wait for response*
  - A.k.a. *asynchronous critical section*

Lock-free synchronization

- *Lock-free* algorithm
  - Some operation must eventually complete
- Rules out use of locks
  - If lock holder stops taking steps, nobody can make progress
- Lock-free is often used as proxy for performance
  - Lock holders don't really die, but can be delayed => If algorithm doesn't have to wait for them, it should perform better

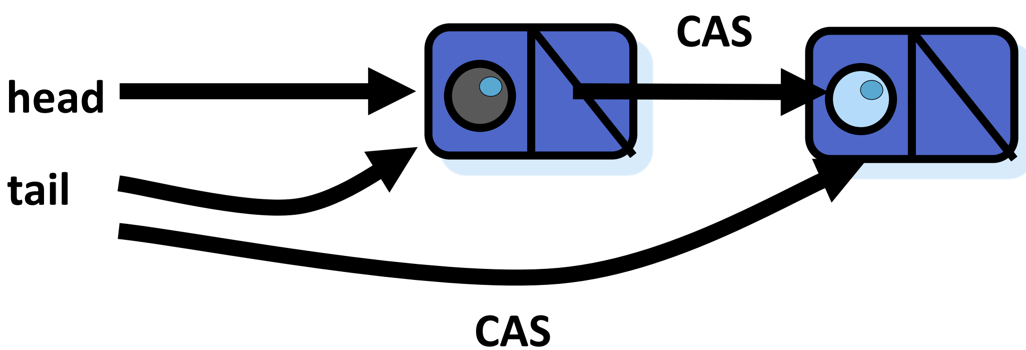
Lock-freeing the two-lock queue

- Enqueue with locks (reminder):



Lock-freeing the two-lock queue

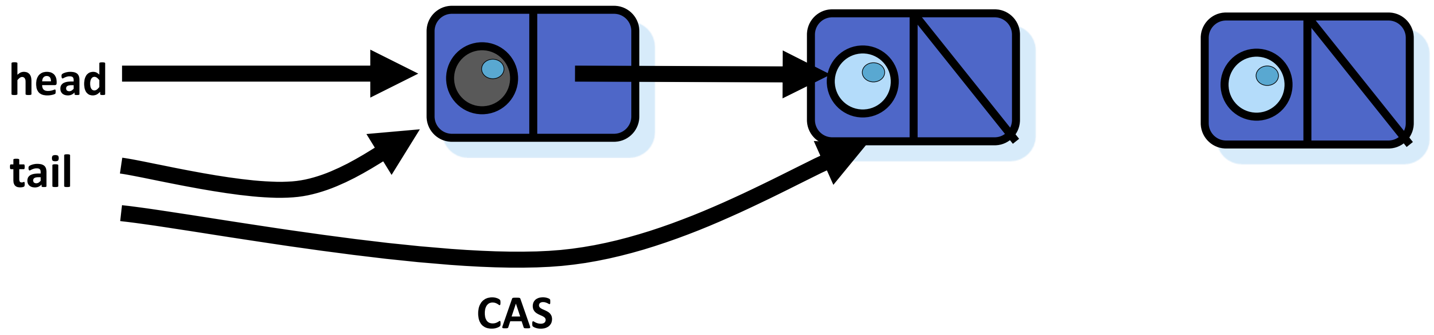
- Enqueue:



- Logical enqueue (linearization point)
  - Dequeues can now see node
- Physical enqueue

Lock-freeing the two-lock queue

- Enqueue in inconsistent state:



- Fix it, and then retry

MS queue [Michael & Scott '96]

```
Node* Head;
Node* Tail;

Initially:

Head = Tail = new Node(0);
Head->next = 0;
```

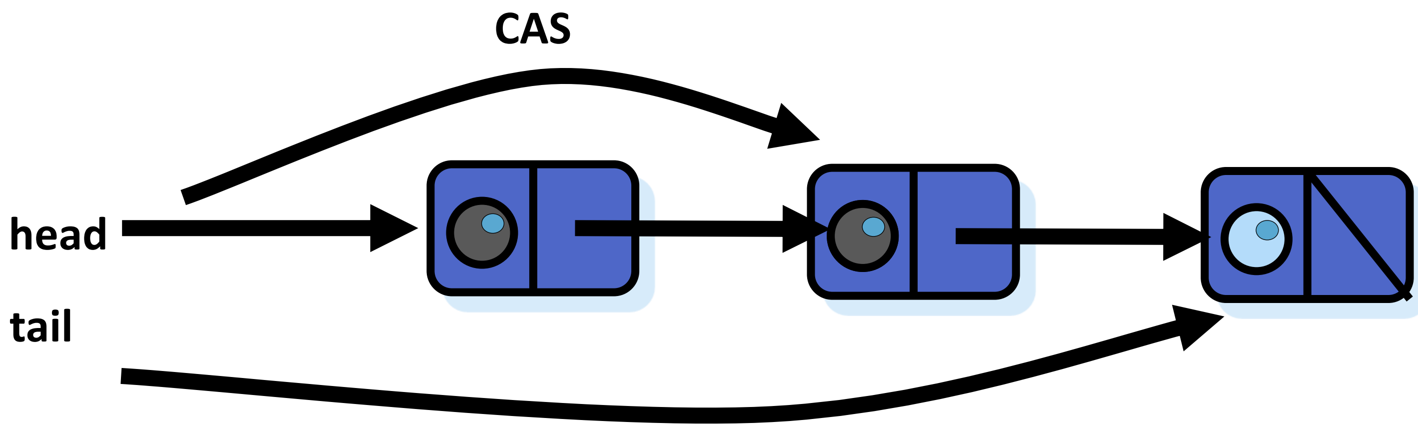
MS queue [Michael & Scott '96]

```
void enqueue(void *v) {
  Node *n = new Node(v); // next=NULL
  while (1) {
    Node *tail = Tail;
    Next *next = tail->next;

    if (next == NULL) {
      if (CAS(&tail->next, 0, n))
        break;
    } else {
      CAS(&Tail, tail, next);
    }
  }
  CAS(&Tail, tail, n); }
```

linearization point

Lock-freeing the two-lock queue

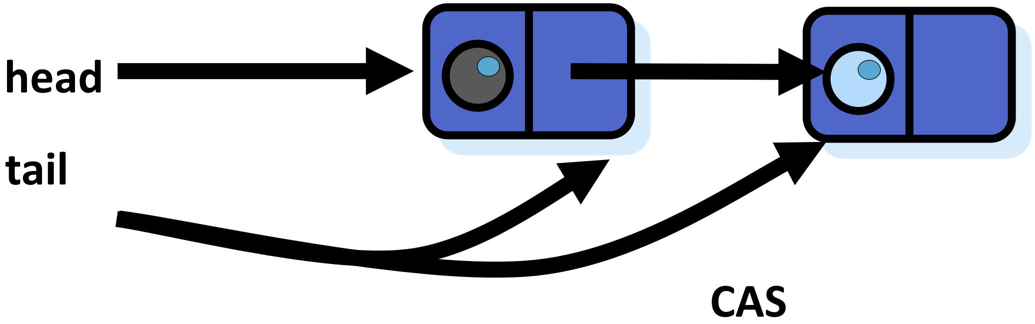
- Dequeue:



- Makes first node the new sentinel
- Old sentinel can be freed
  - (Assume GC)

Lock-freeing the two-lock queue

- Dequeue:



- Also fix inconsistent states

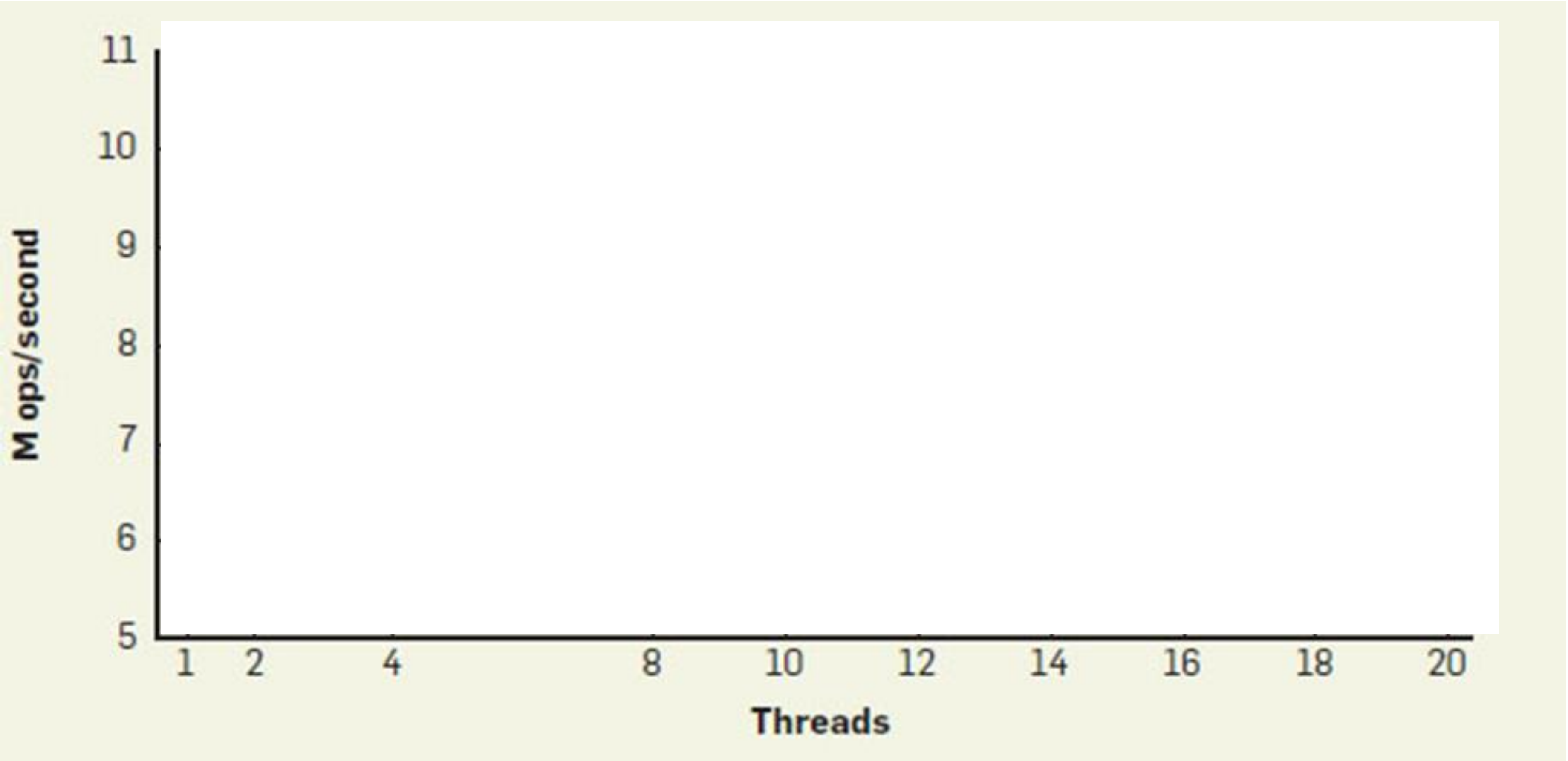
MS queue [Michael & Scott '96]

```
void* dequeue() {
  while (1) {
    Node* head = Head;
    Node* tail = Tail;
    Next* next = head->next;

    if (head == tail) {
      if (next == NULL) return EMPTY;
      CAS(&Tail, tail, next);
    } else {
      void* rv = next->value;
      if (CAS(&Head, head, next)) return rv;
    }
  }
}
```

Is this necessary?

assume GC

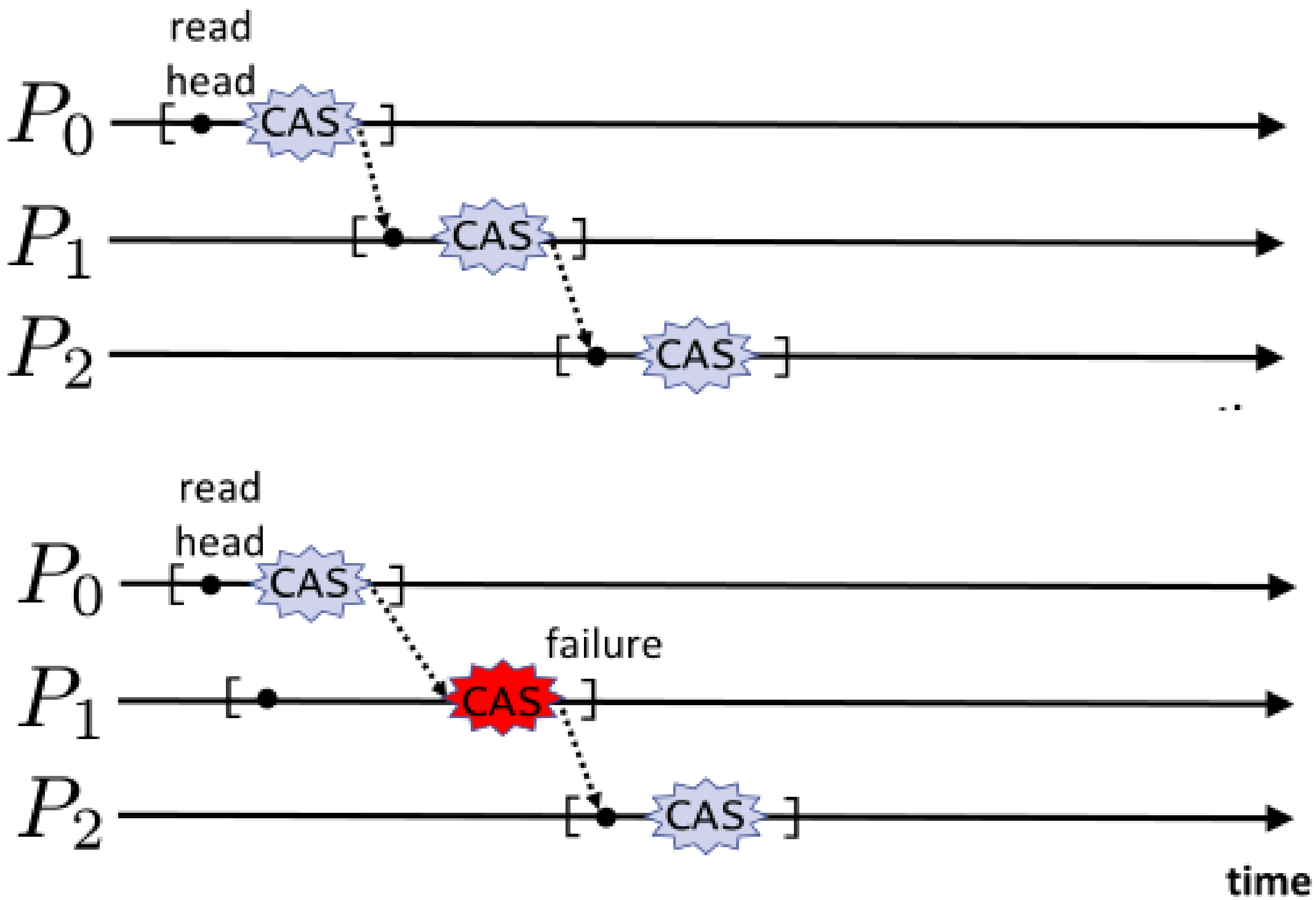
Lock-freedom to the rescue?



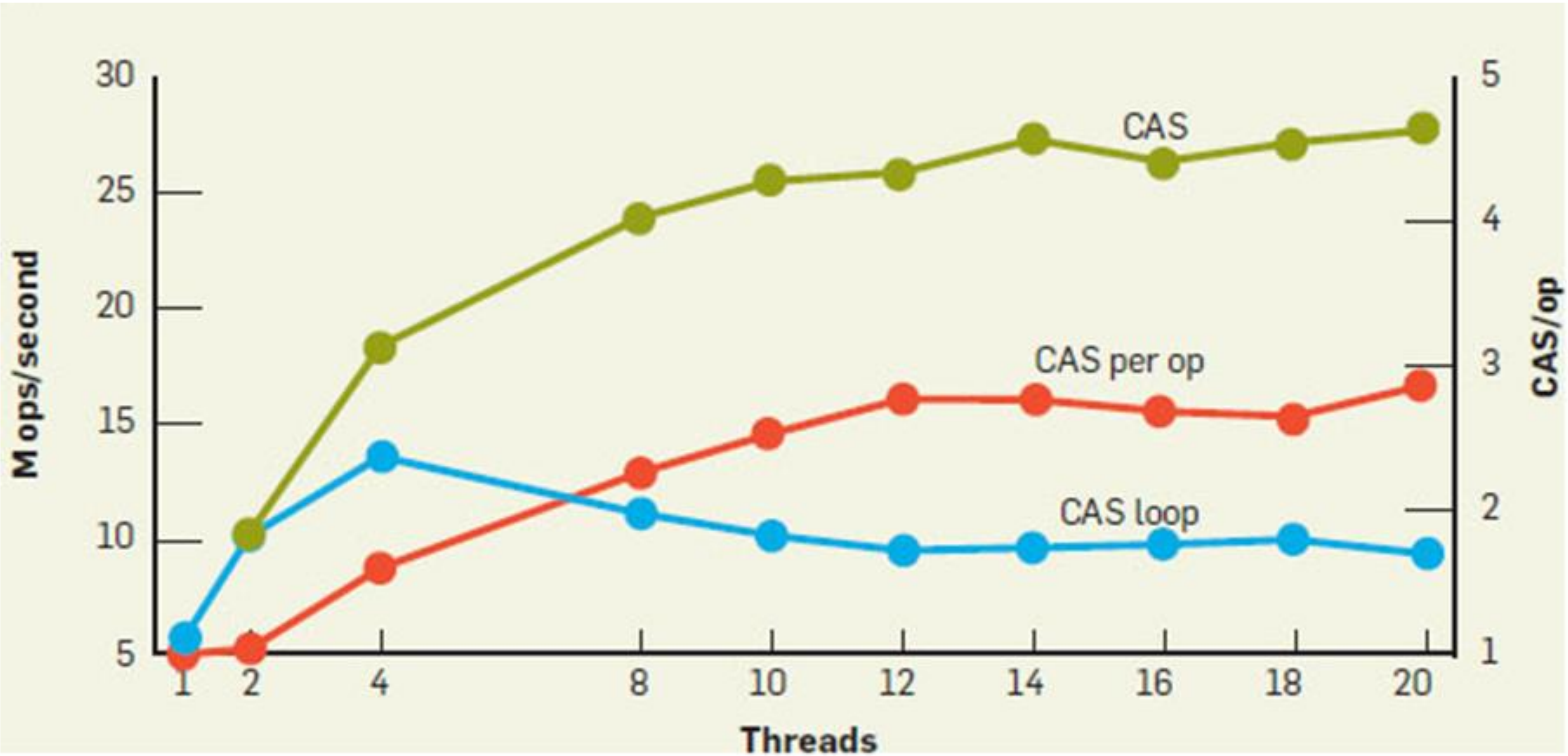
CAS failures

- MS queue poor performance is due to *CAS failures*, which add *useless work* to the critical path

CAS failures



CAS failures



Avoiding CAS failures

- Idea: Try to replace CAS with an atomic instruction that doesn’t fail, so that every atomic op contributes to useful work
  - SWAP, Fetch-And-Add

FAA queue

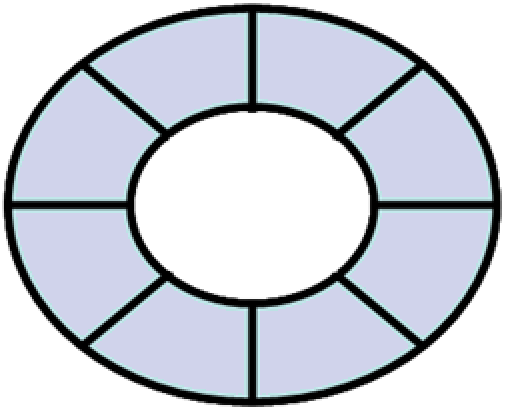
```
cell : { void *value; }
cell Q[] = ⊥ .....; uint head = 0, tail = 0;
```

```
enqueue(x) {
  while (true) {
    t = F&A(&tail, 1)
    if (CAS(&Q[t],⊥, x))
      return
  }
}
```

```
dequeue(x) {
  while (true) {
    h = F&A(&head, 1)
    if (!CAS(&Q[h],⊥ ,⊤ ))
      return Q[h]
    if (tail ≤ h+1) return EMPTY
} }
```

Bounding the queue

- Use cyclic array



R nodes

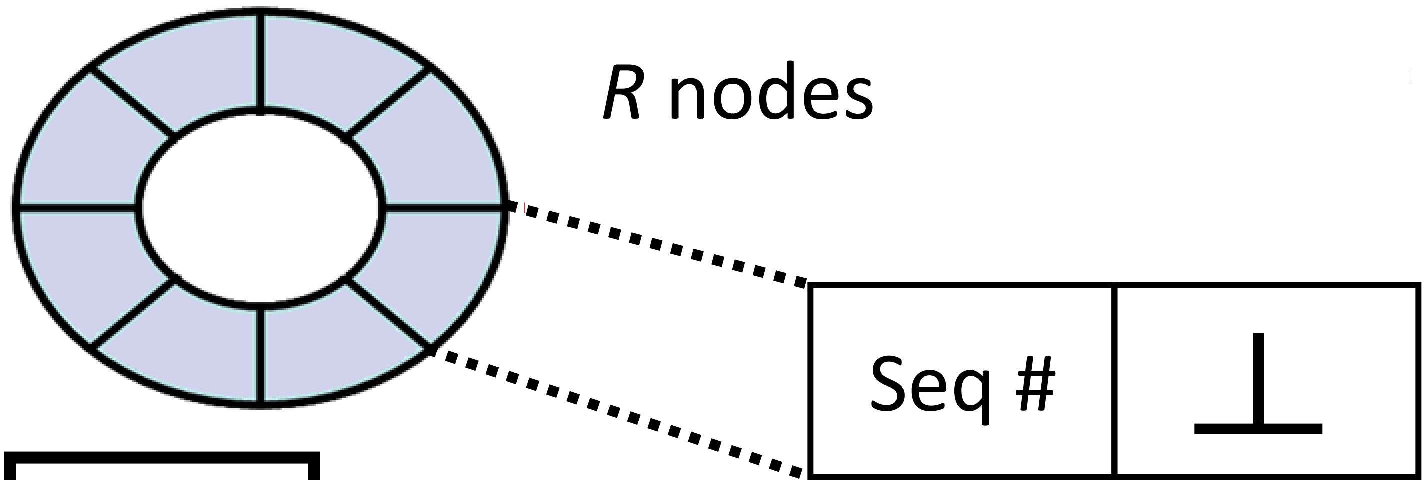
Tail:
0

Head:
0

Bounding the queue

- Use cyclic array

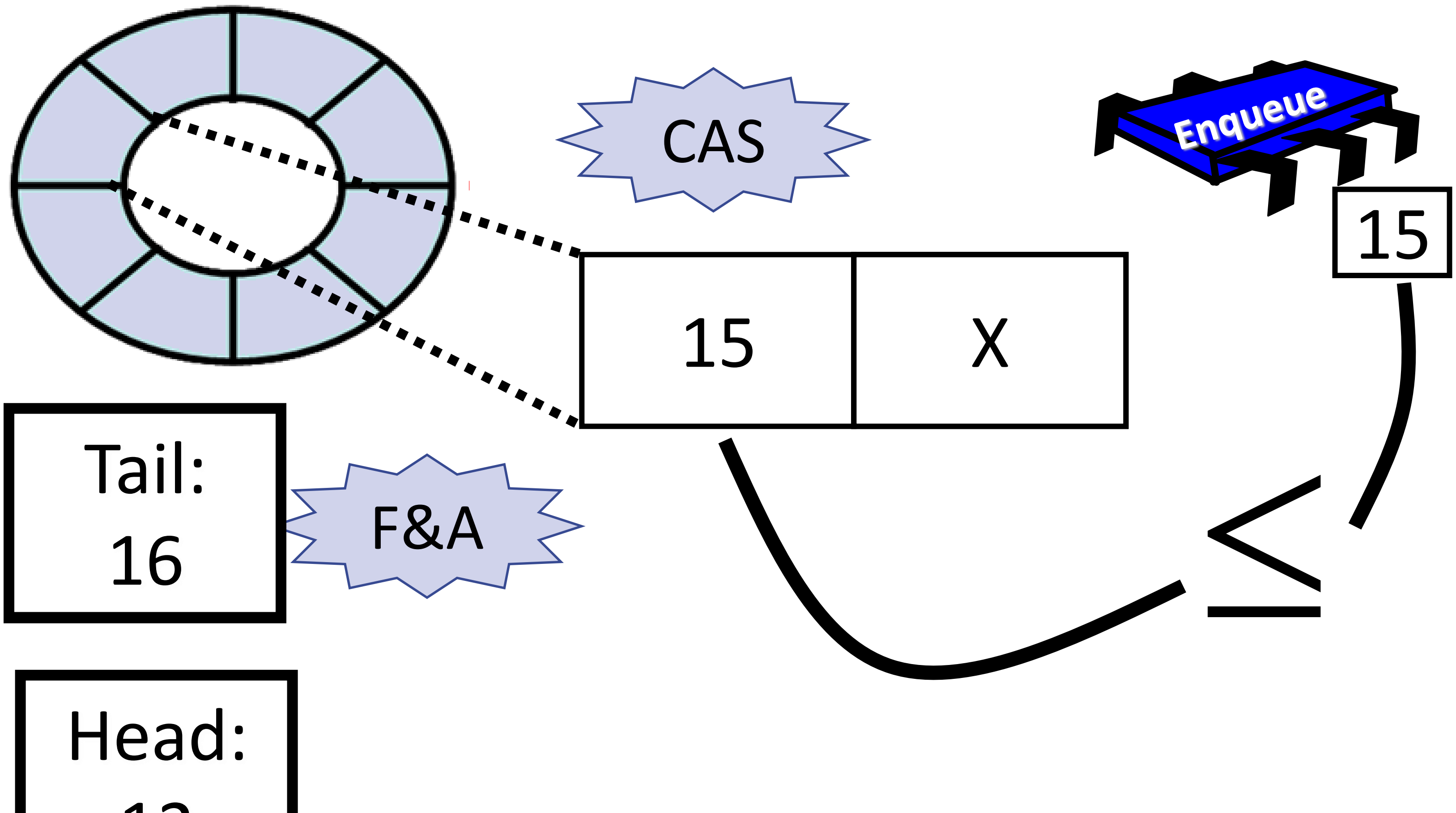
R nodes



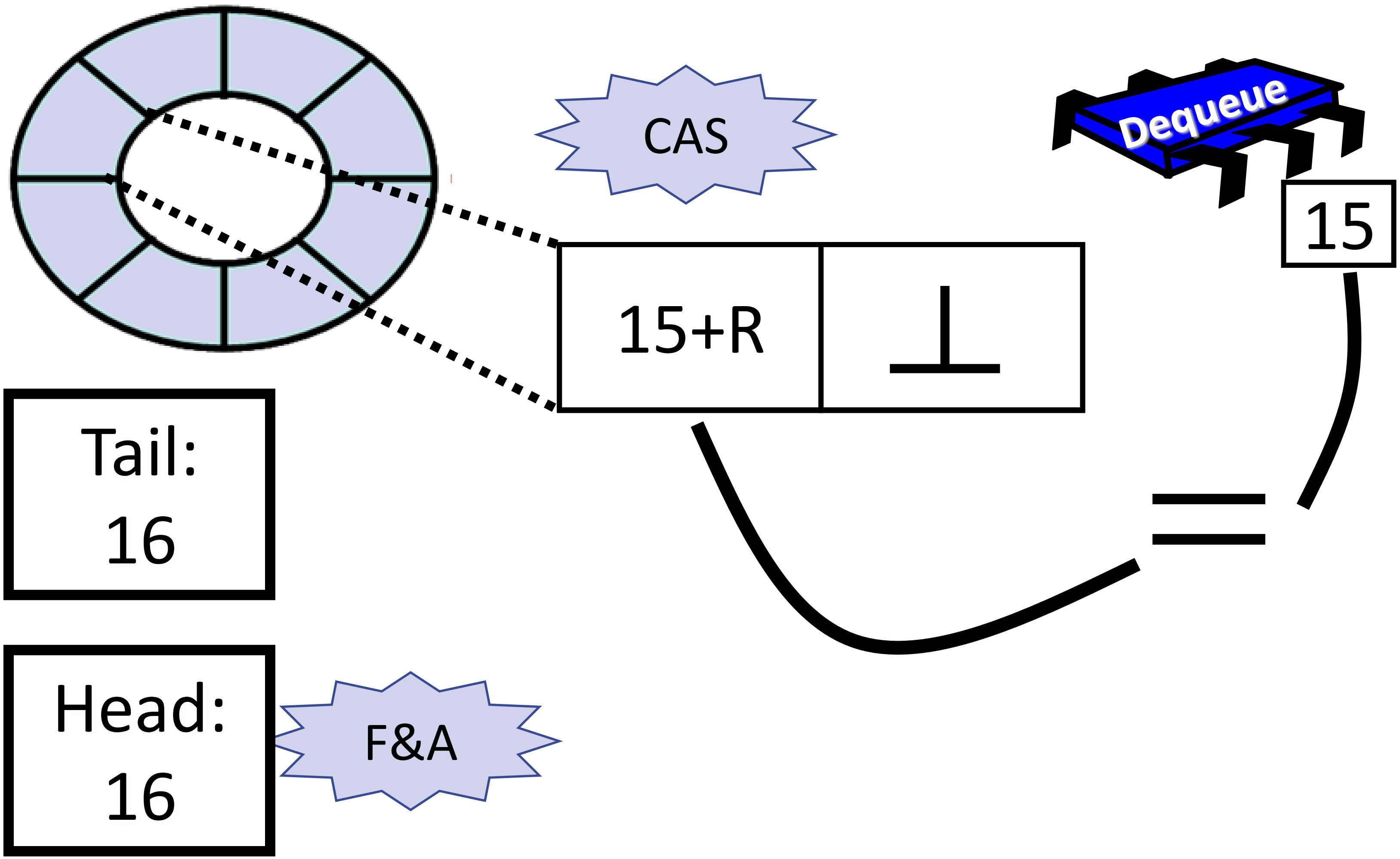
Tail:
0

Head:
0

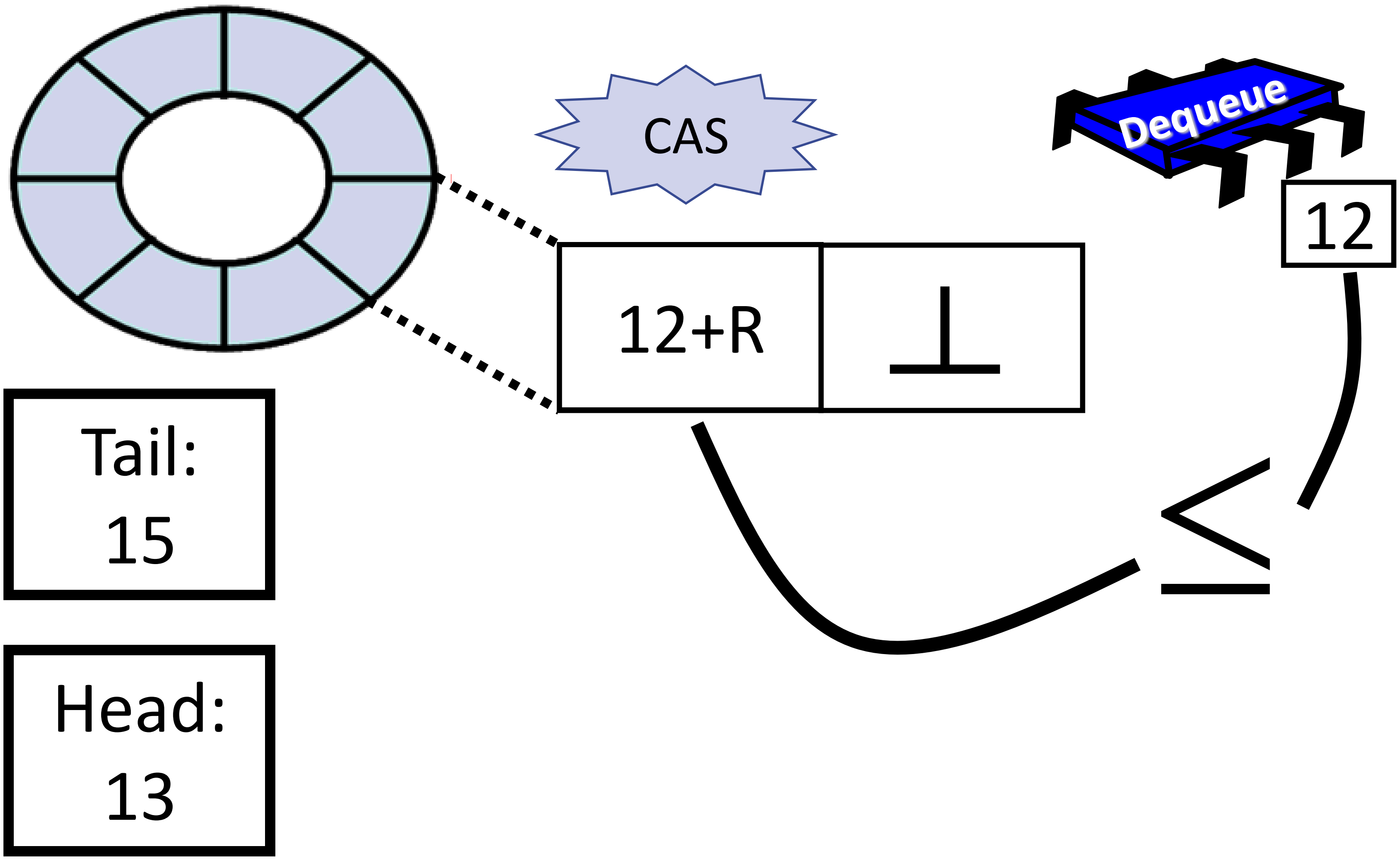
Enqueue



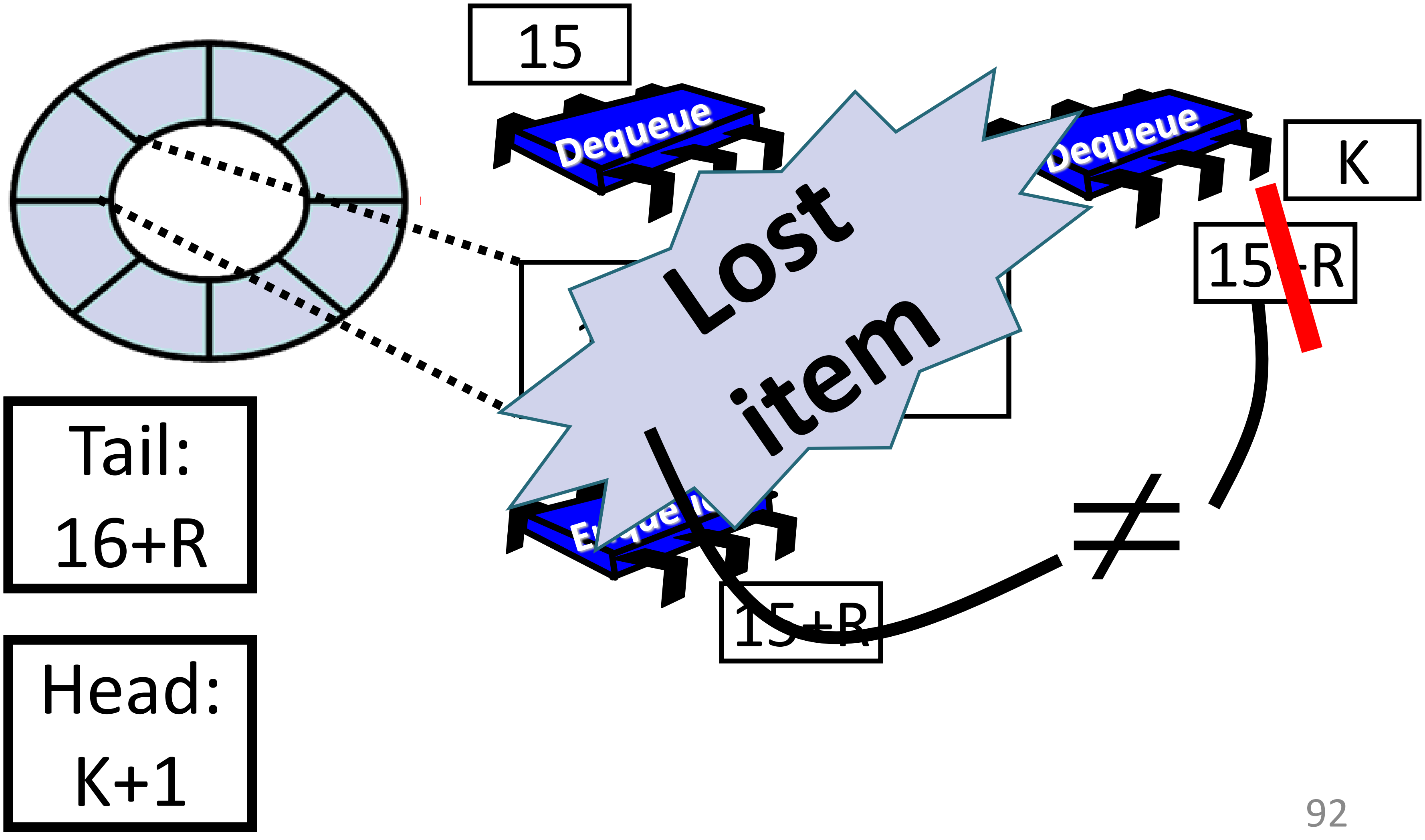
Dequeue returning a value



Dequeue before enqueue



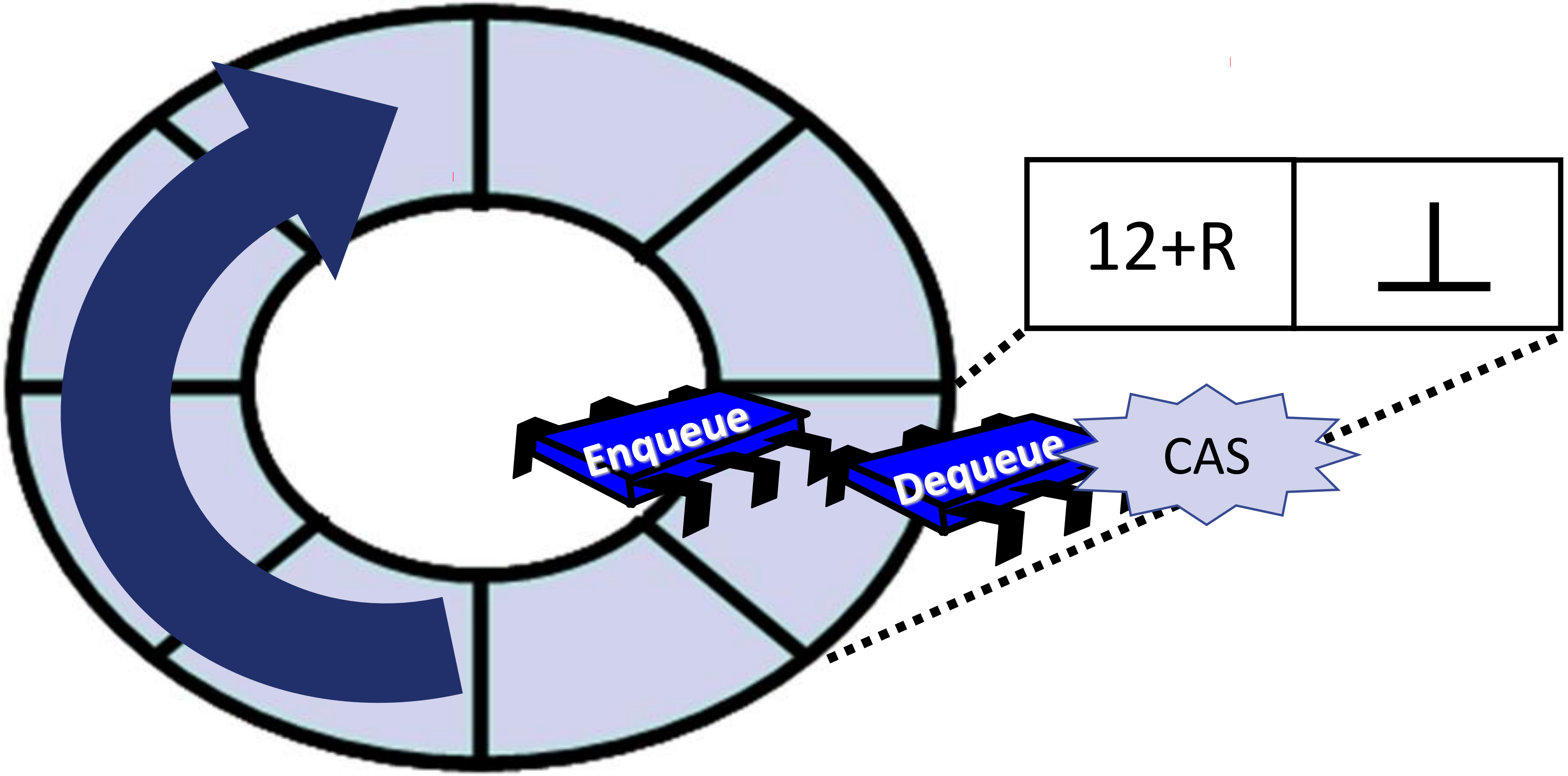
Deq/Enq mismatch



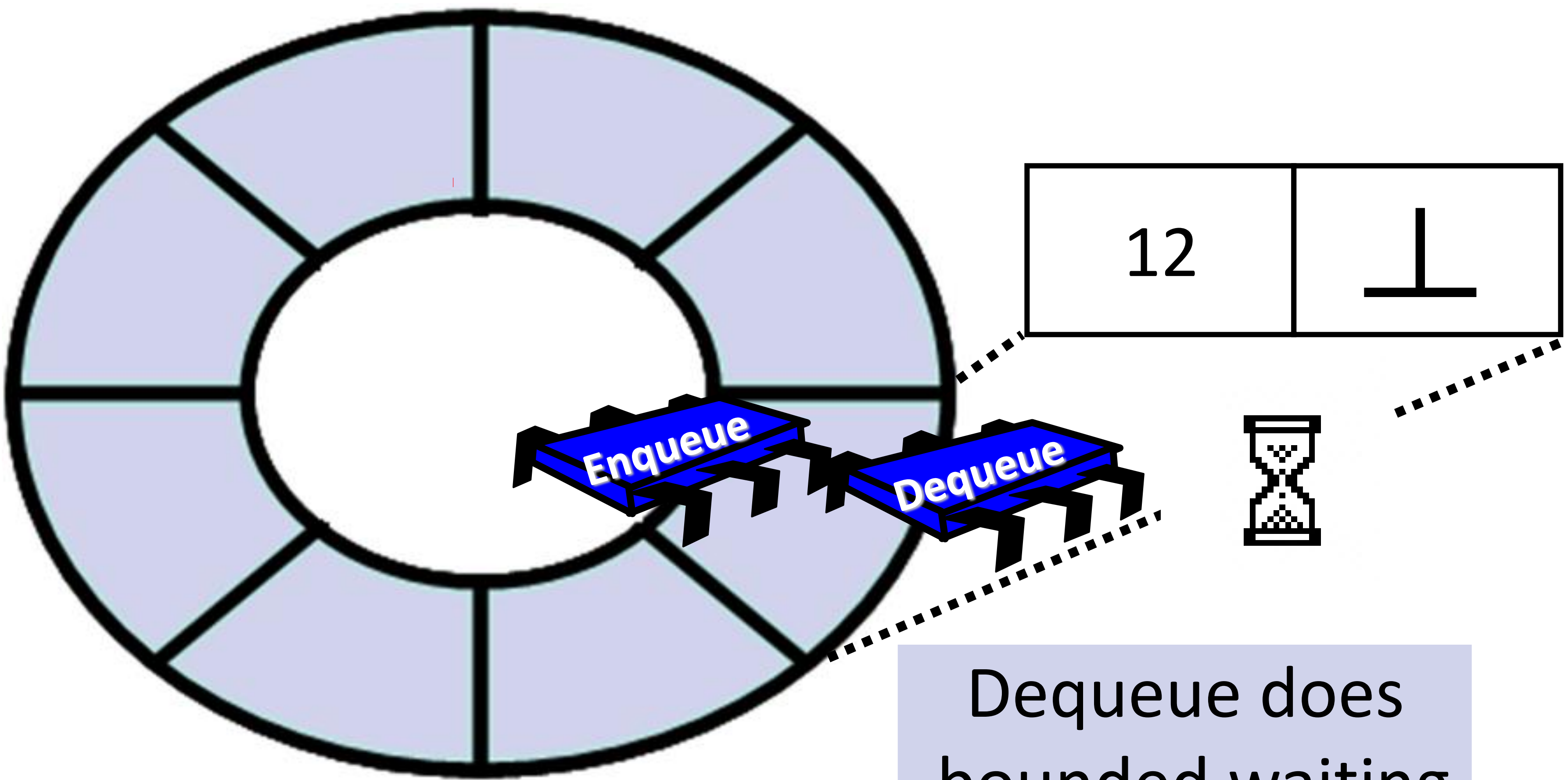
Deq/Enq mismatch

- Simple solution: if dequeue with index *y* sees <*val*, *x*> for $x < y$ in its cell, it waits:
  - Eventually, dequeuer with index *x* will remove *val*
- But this isn't lock-free
- Lock-free solution more complex, see the paper

Algorithm livelocks

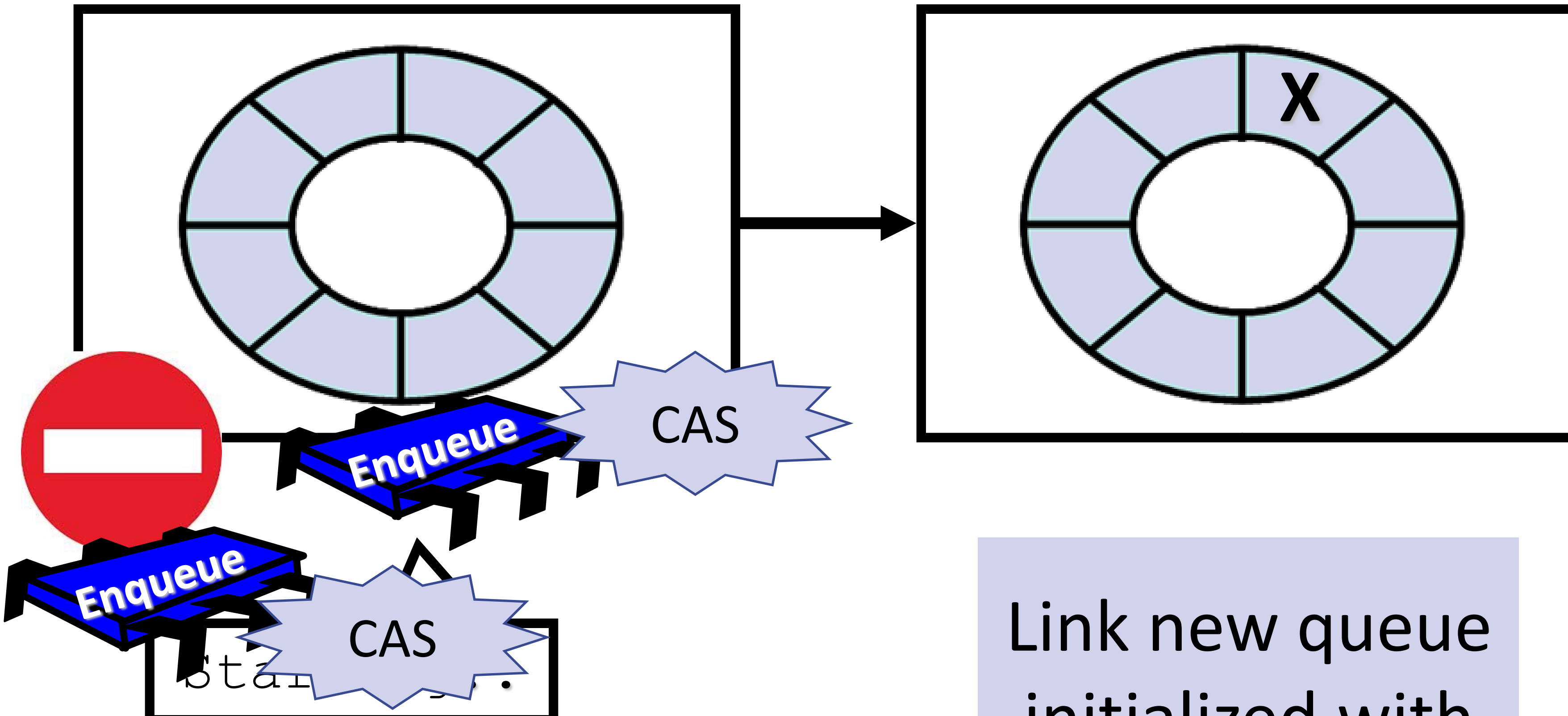


Practical livelock solution



Dequeue does bounded waiting

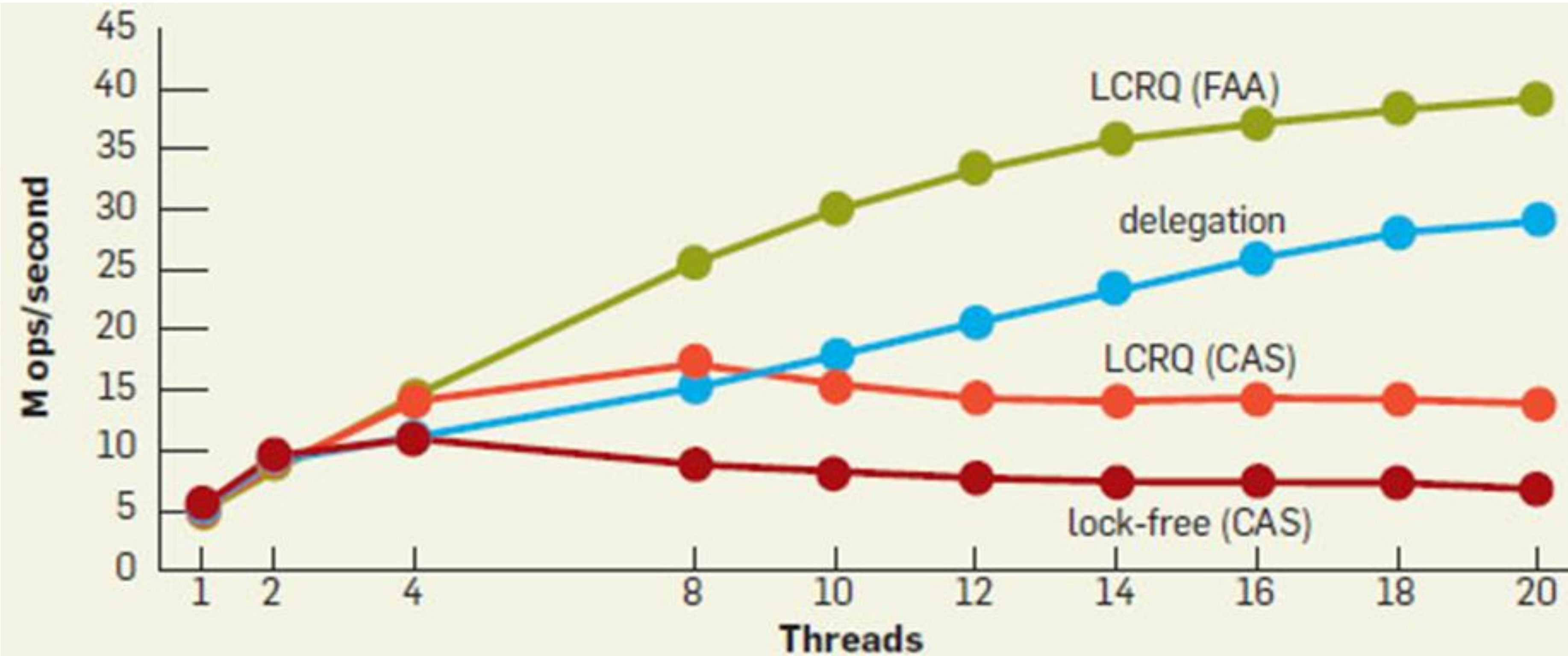
Solution: list of nodes



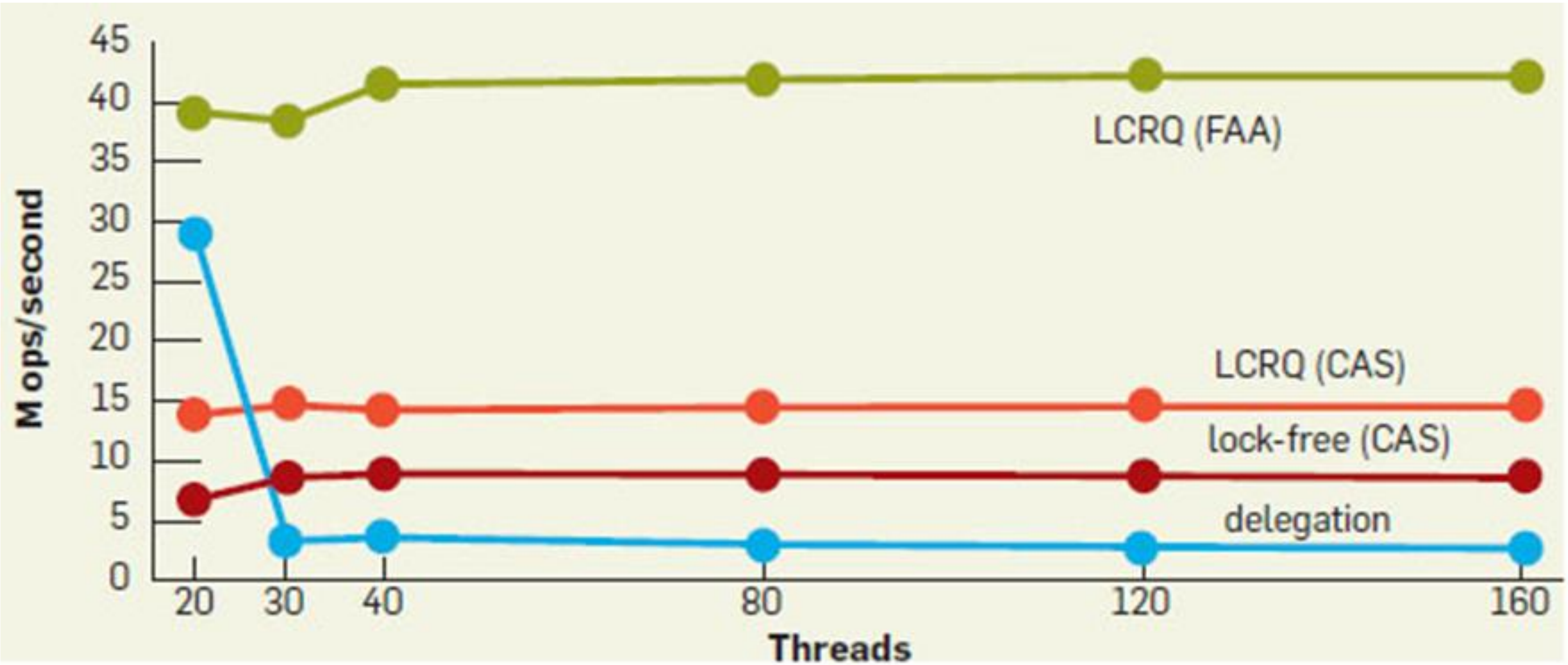
Close queue

Link new queue initialized with element

Queue example



Queue with preemption



Summary

- Importance of serializing efficiently
- Efficient lock algorithms
- Delegation + optimizations
- Avoiding CAS failures in lock-free algorithms