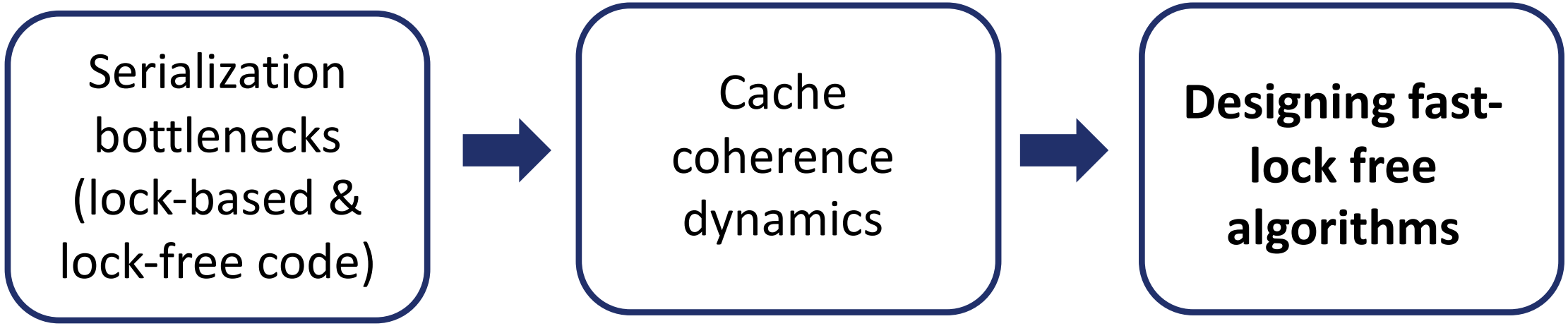
Designing Fast Lock-Free Algorithms by Understanding Cache Coherence Dynamics

Adam Morrison

Tel Aviv University

The Blavatnik School of Computer Science
The Raymond and Beverly Sackler Faculty of Exact Sciences
Tel Aviv University

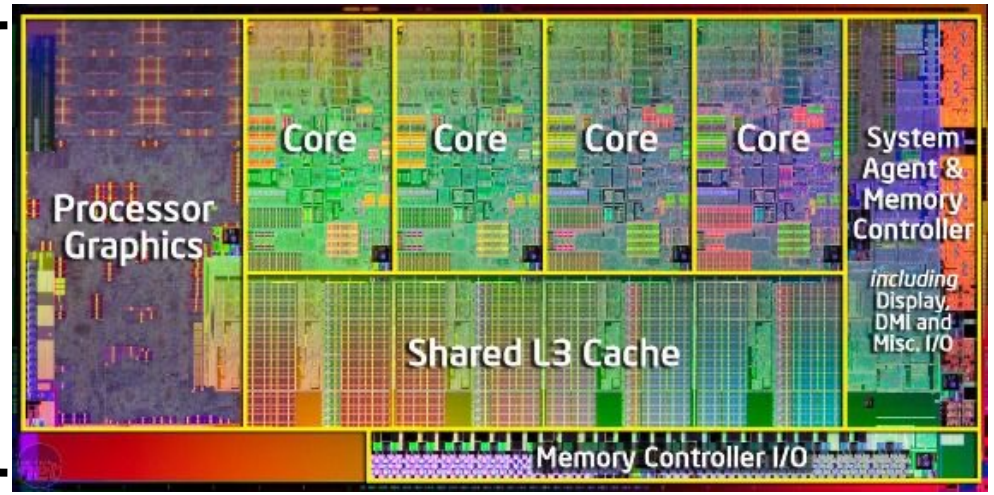
Outline

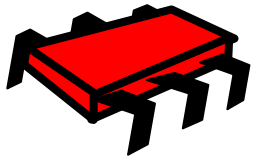

Serialization bottlenecks (lock-based & lock-free code)
Cache coherence dynamics
Designing fast-lock free algorithms

Context

Multi-core and/or **multi-processor** shared-memory system

processor



sequential code =  = thread/core/processor

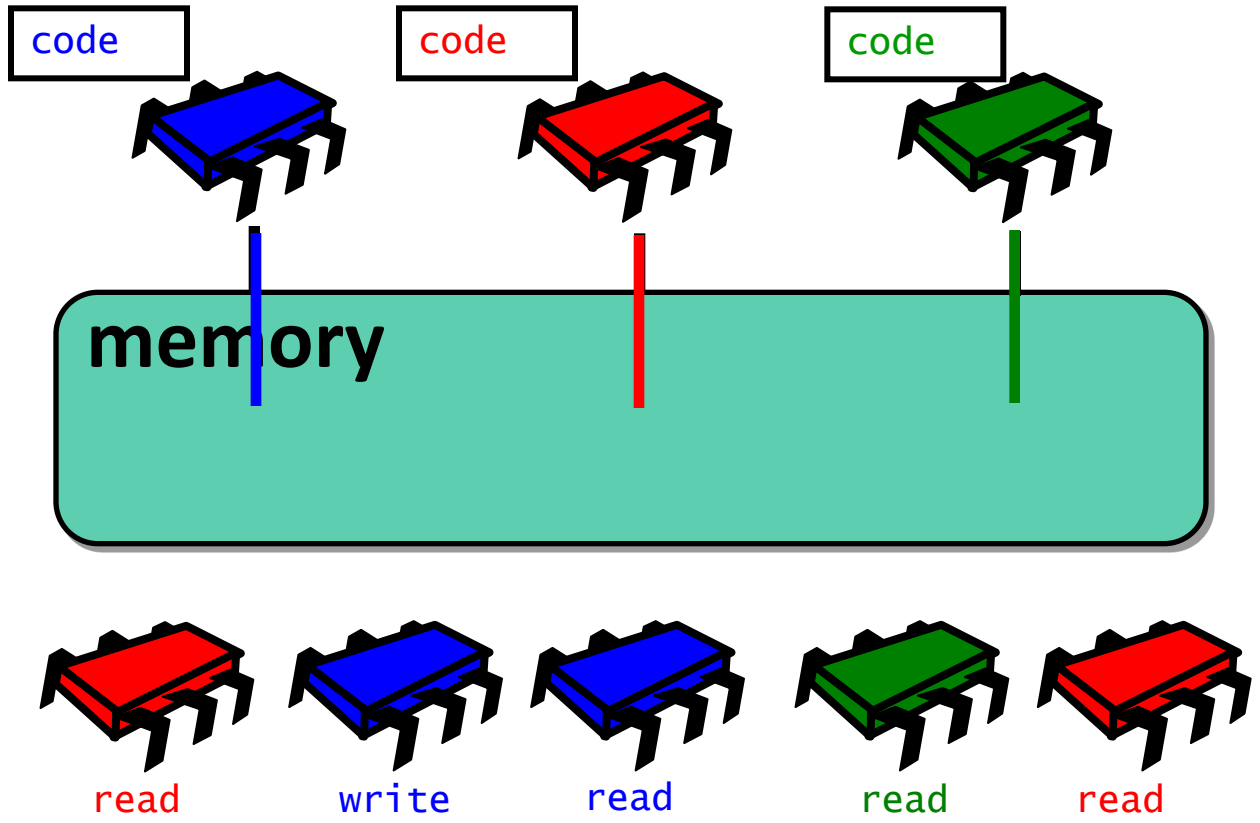
Setting

A **simplified** model of the system

Ignoring memory consistency model (of hardware / programming language), memory fences, etc.

Examples in pseudo code

(C/Java inspired)

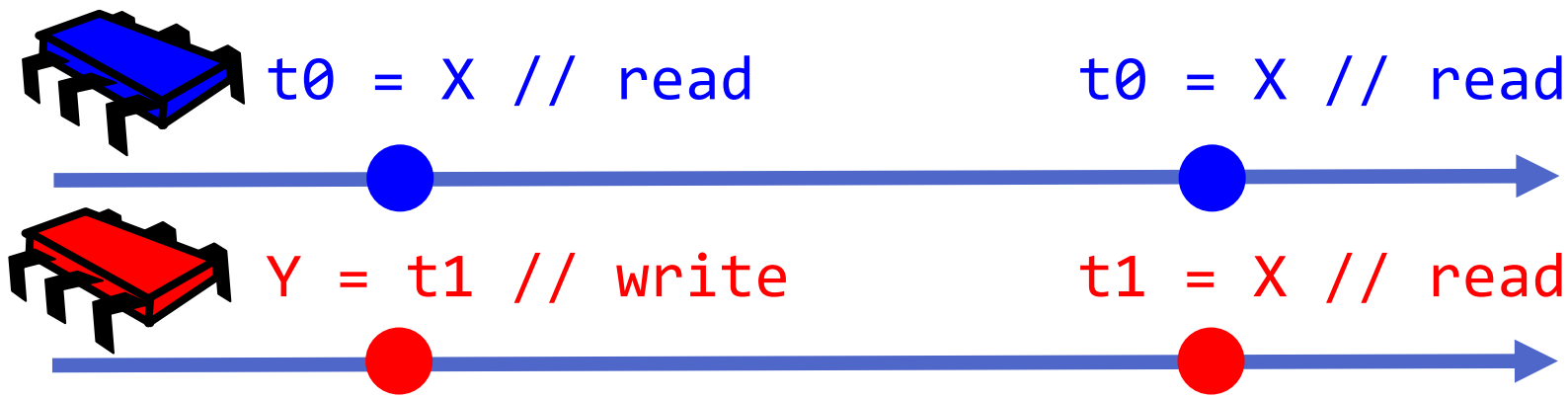


any interleaving is possible

Connection of model to reality

Modeling parallelism with a **sequential** execution model?!

Intuition (1/2): Non-conflicting parallel memory operations **commute**

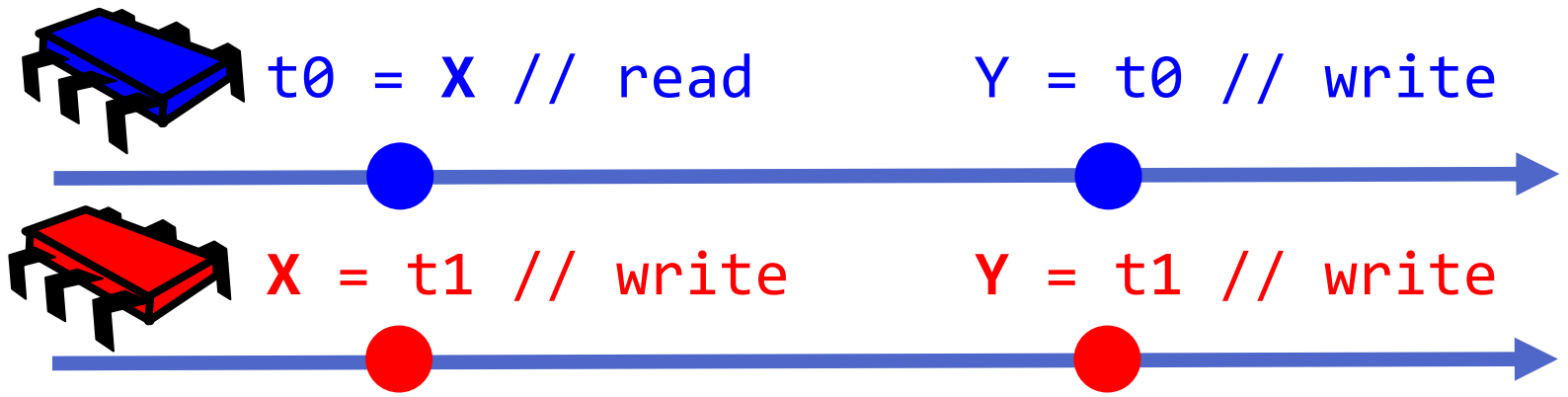


⇒ Pretend they happen in some sequential order; it doesn't matter

Connection of model to reality

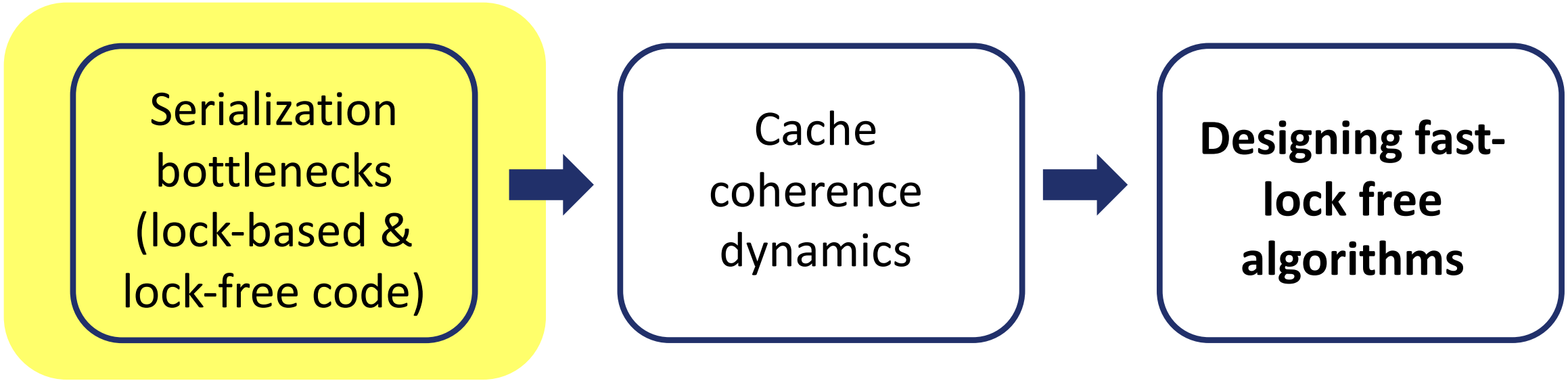
Modeling parallelism with a **sequential** process?!

Intuition (2/2): Conflicting memory operations **serialize** in hardware



⇒ They **do** happen in some sequential order

Part 1


Serialization bottlenecks (lock-based & lock-free code)
Cache coherence dynamics
Designing fast-lock free algorithms

Approach

Show general principles via a concrete **running example**

Goal: Design a concurrent FIFO queue
Multi-producer, multi-consumer, unbounded

$\Rightarrow$**Data structure:** Linked list with Head and Tail pointers

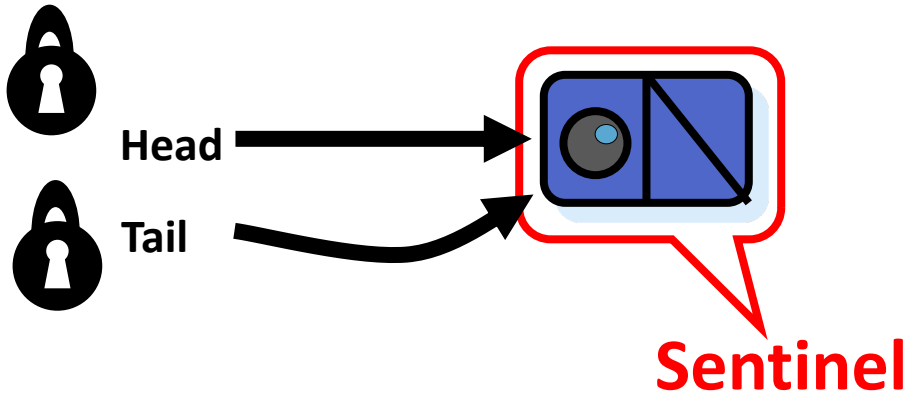
Challenge: Synchronize concurrent operations

Two-lock queue [Michael & Scott '96]

First approach:

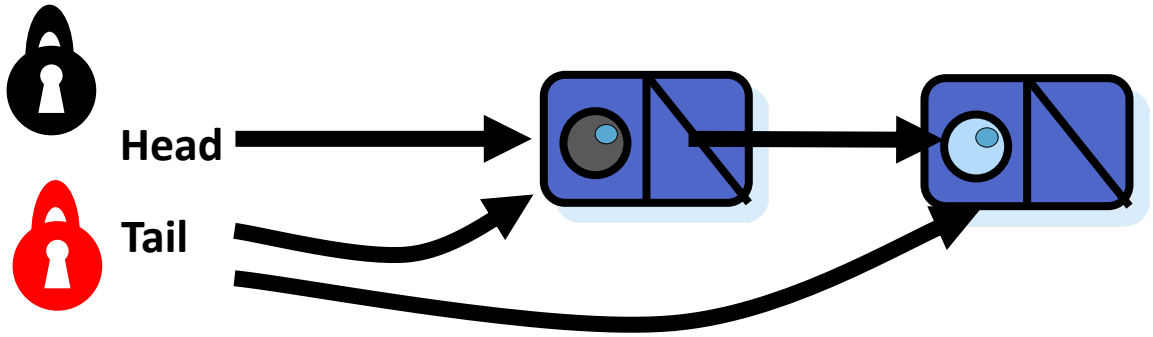
Lock-based algorithm

Allows concurrent enqueue & dequeue



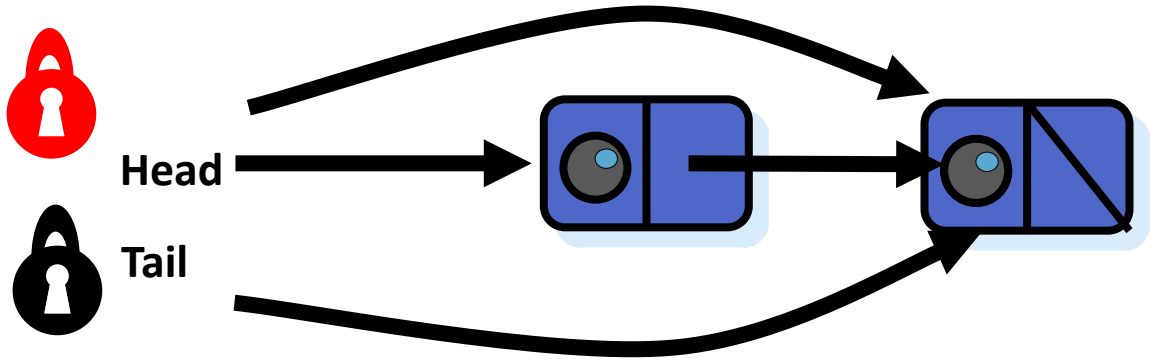

```
Node { T val; Node* next; };

Initially:
Head = Tail = new Node(NULL);
Head->next = NULL;
```

Two-lock queue: enqueue()




```
enqueue(T val) {
  Node *n = new Node(val);
  lock(&tail_lock);
  Tail->next = n;
  Tail = n;
  unlock(&tail_lock); }
```

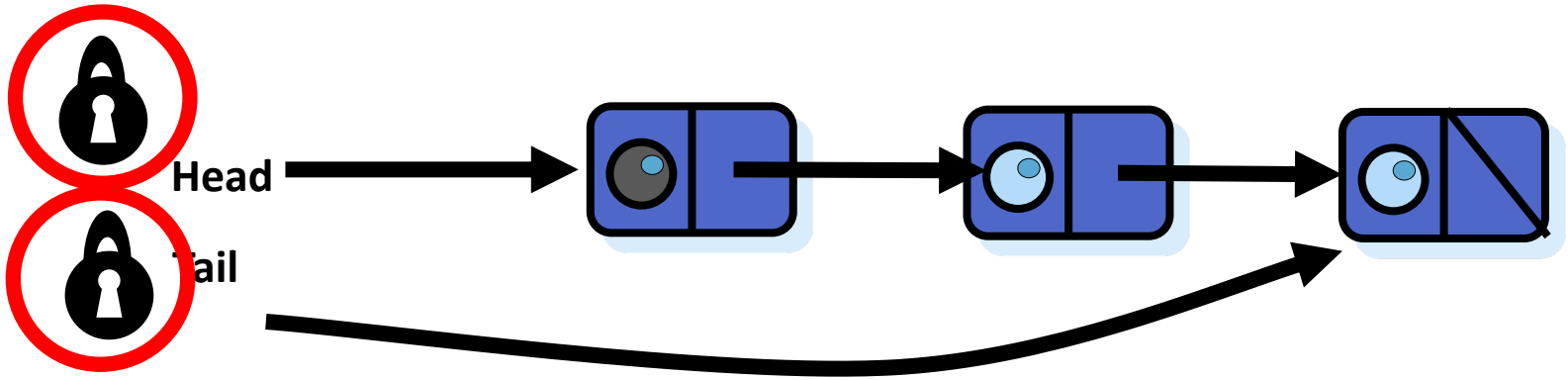
Two-lock queue: dequeue()




```
dequeue(T val) {
  lock(&head_lock);
  Node *n = Head->next;
  if (n) {
    rv = n->val;
    Head = n->next;
  } else {
    rv = EMPTY;
  }
  unlock(&tail_lock);
  return rv;
}
```

Two-lock queue scalability

At most **two** cores can be doing useful queue work



Lock-free synchronization

Lock-free (or **nonblocking**) algorithm

Formal definition: **Some** executing method must eventually complete

(But individual method invocations might starve!)

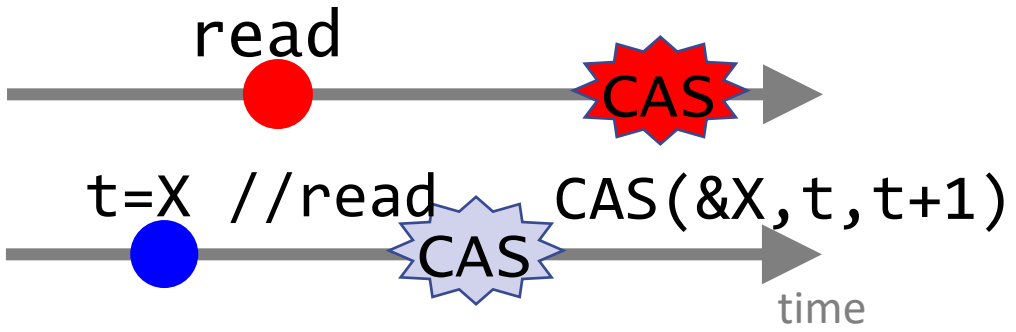
$\Rightarrow$ Rules out use of locks

If lock holder stops taking steps, nobody can make progress

Instead rely on **atomic read-modify-write** instructions, such as `compare-and-set` (CAS)

CAS

```
CAS(T *addr, T old, T new) {
    atomic {
      if (*addr != old)
         return false;
      *addr = new;
      return true;
    }
}
```



read-to-CAS segment

Lock-freedom to the rescue!

Lock-freedom is often used as **proxy** for performance

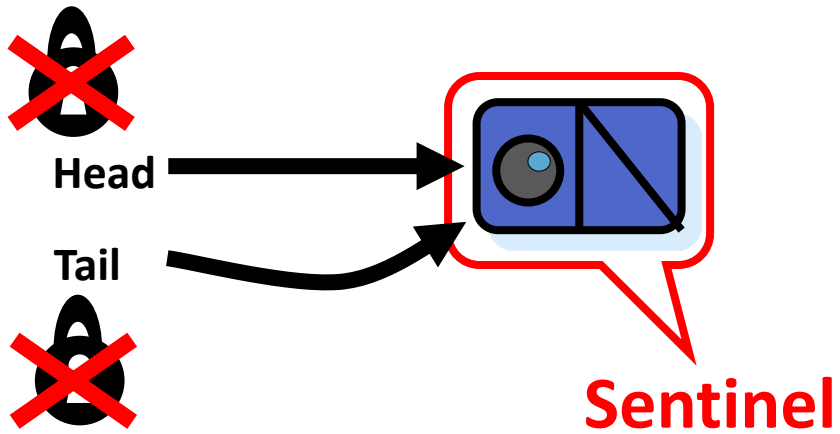
In practice, lock holders don’t **die**, but can be **delayed**

$\Rightarrow$ **Intuition:** Not waiting for delayed lock holders improves performance

Lock-free queue [Michael & Scott ‘96]

Second queue approach:

Lock-freeing the two-lock queue




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

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

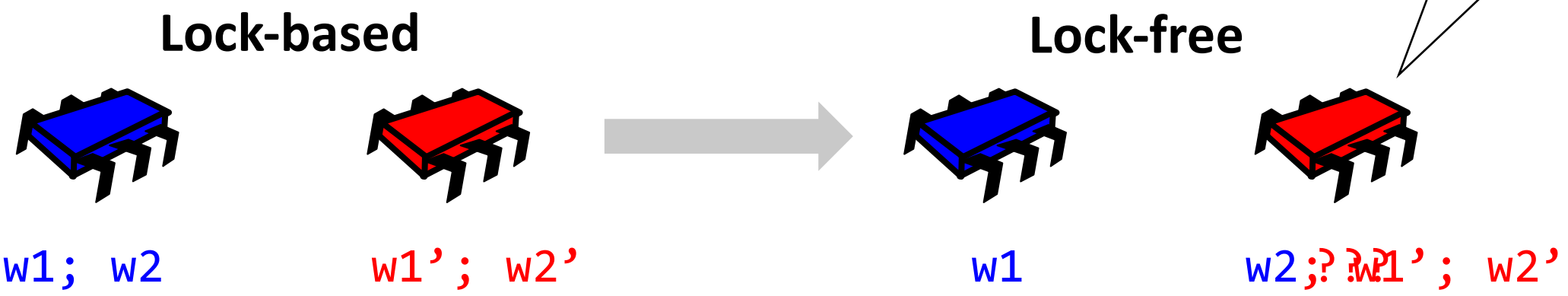
“Recipe” for lock-freeing code

Replace each write **w** in the critical section with a **CAS**

Intuition: Guarantees write isn’t lost (overwritten)

Challenge: Dealing with **inconsistent state** of the data structure
Occurs when a thread stalls mid-operation

Typical solution: **Helping**

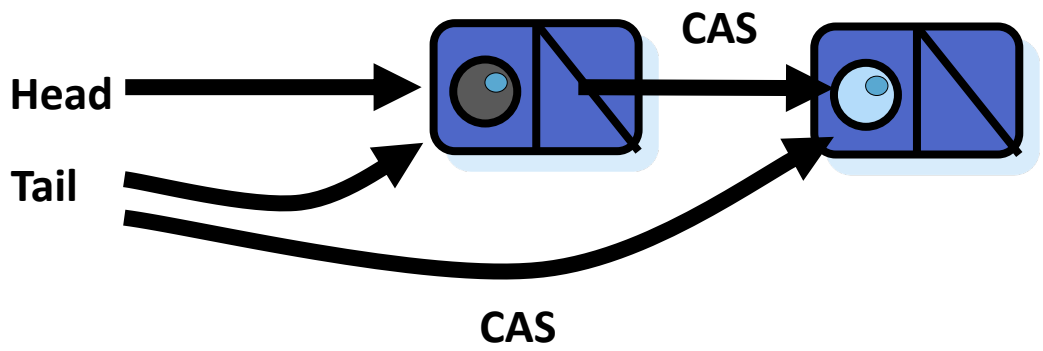


Lock-free enqueue [Michael & Scott '96]

Logical enqueue happens on node connection

⇒ Dequeue can now see node

Physical enqueue follows

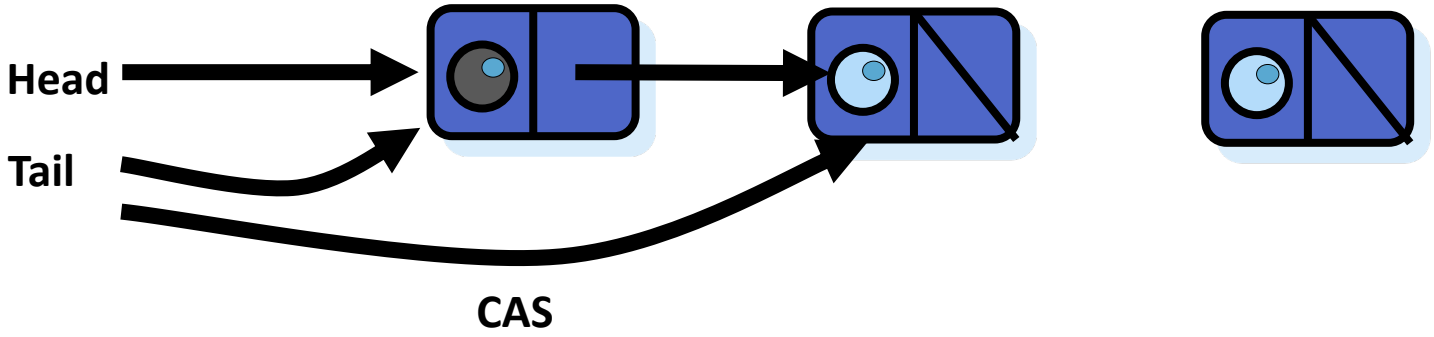



```
void enqueue(T v) {
 Node *n = new Node(v);
 while (true) {
  Node *tail = Tail;
  Node *next = tail->next;

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

Lock-free enqueue [Michael & Scott '96]

Helping:

If observe **inconsistent state**, fix it & retry

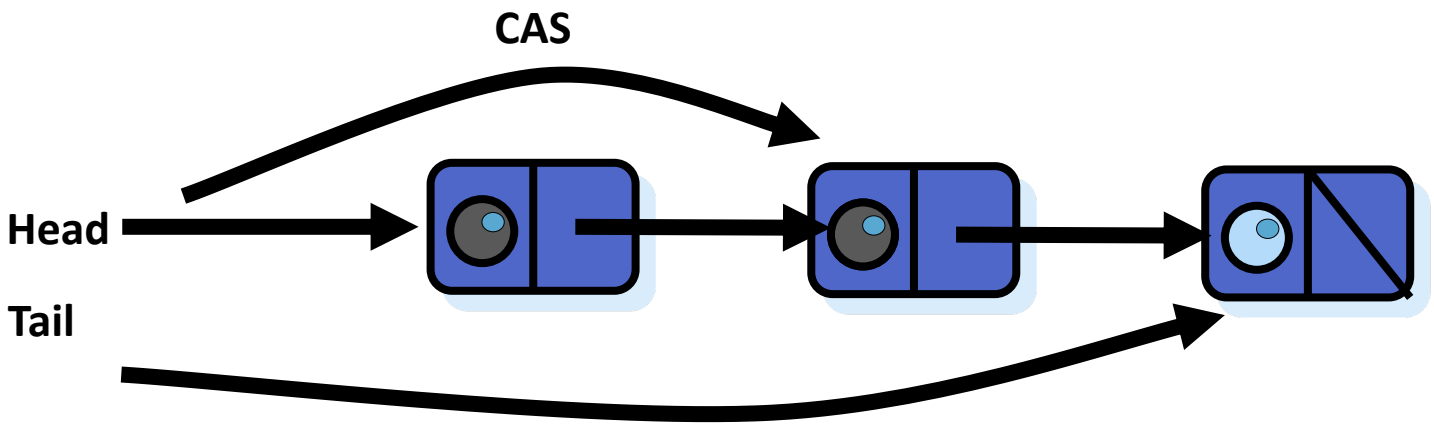



```
void enqueue(T v) {
 Node *n = new Node(v);
 while (true) {
  Node *tail = Tail;
  Next *next = tail->next;

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

Lock-free dequeue [Michael & Scott '96]

Make first node the new sentinel

Old sentinel can be reclaimed

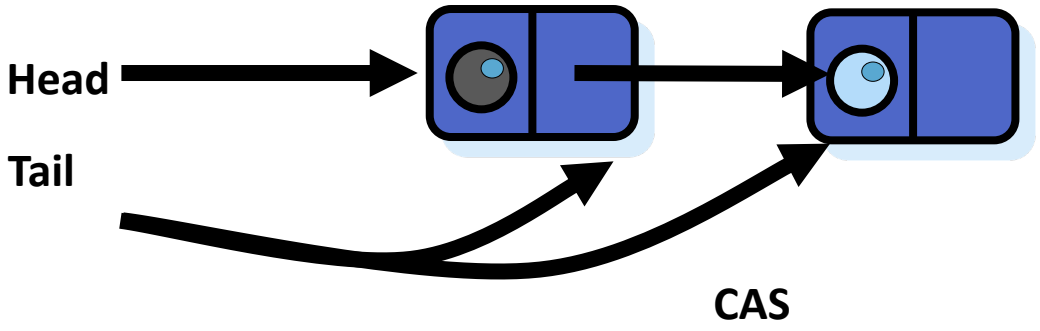



```
T dequeue() {
 while (true) {
  Node* head = Head;
  Node* tail = Tail;
  Next* next = head->next;

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

Lock-free dequeue [Michael & Scott '96]

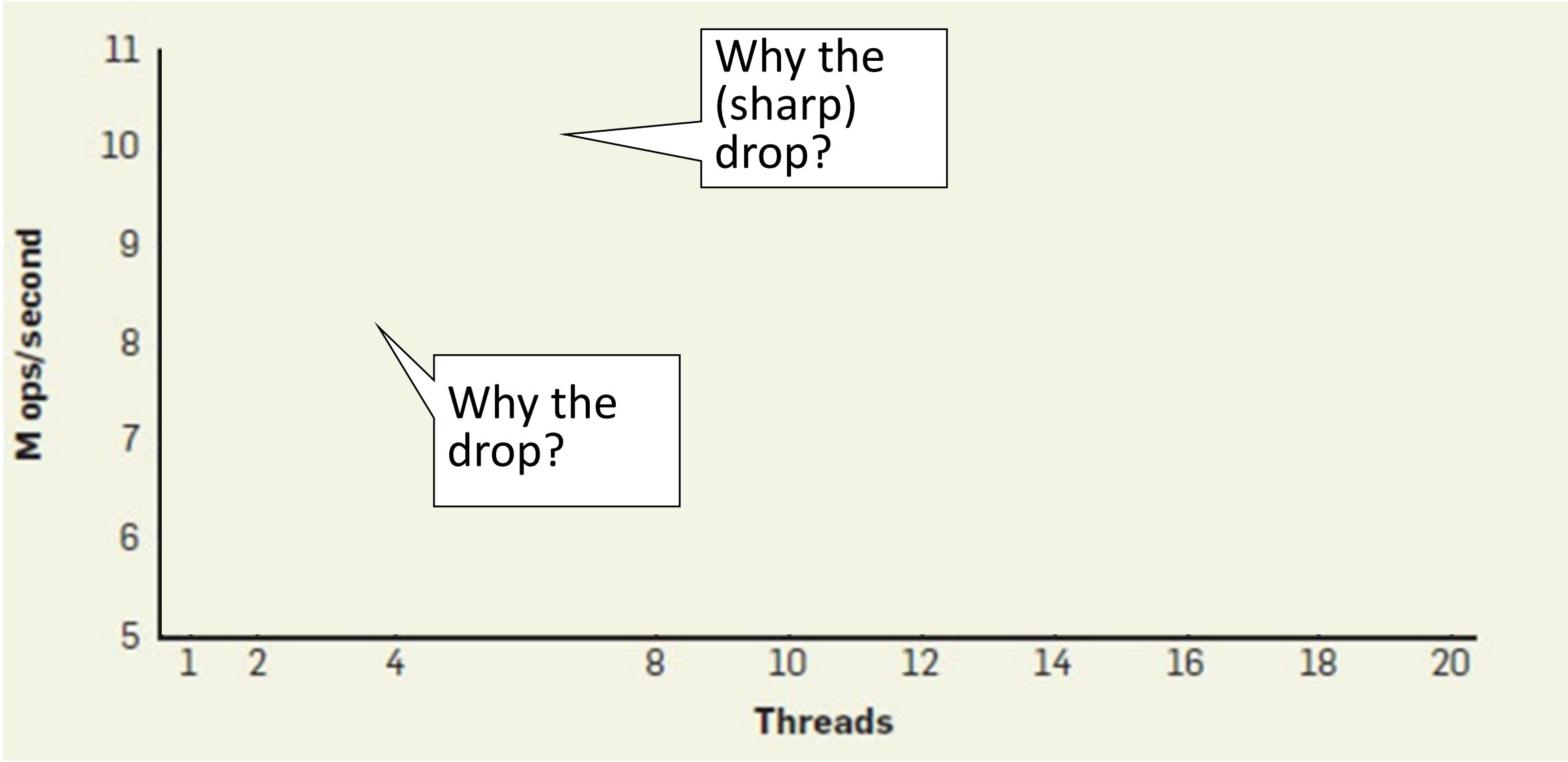
Helping:

Fix up inconsistent state, if observed

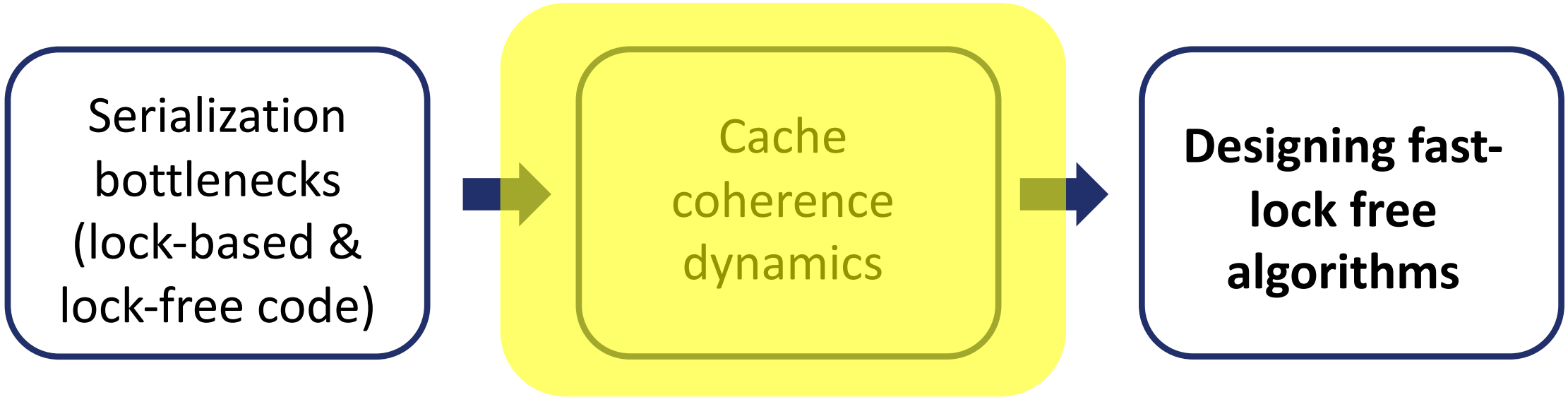



```
T dequeue() {
 while (true) {
  Node* head = Head;
  Node* tail = Tail;
  Next* next = head->next;

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

Lock-freedom to the rescue?

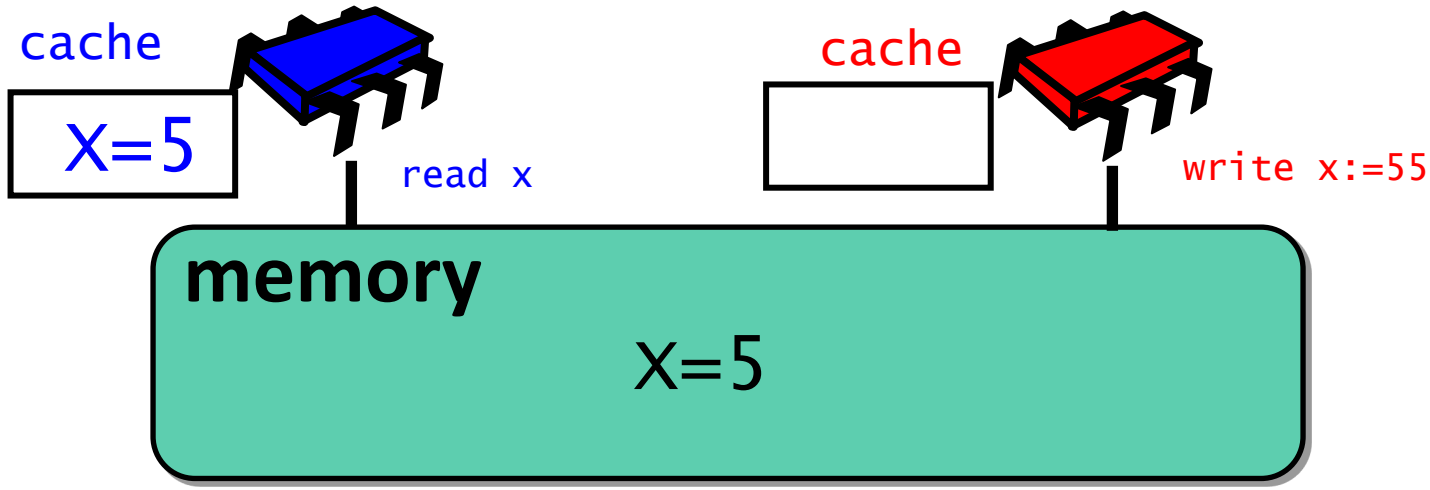


Part 2


Serialization bottlenecks (lock-based & lock-free code)
Cache coherence dynamics
Designing fast-lock free algorithms

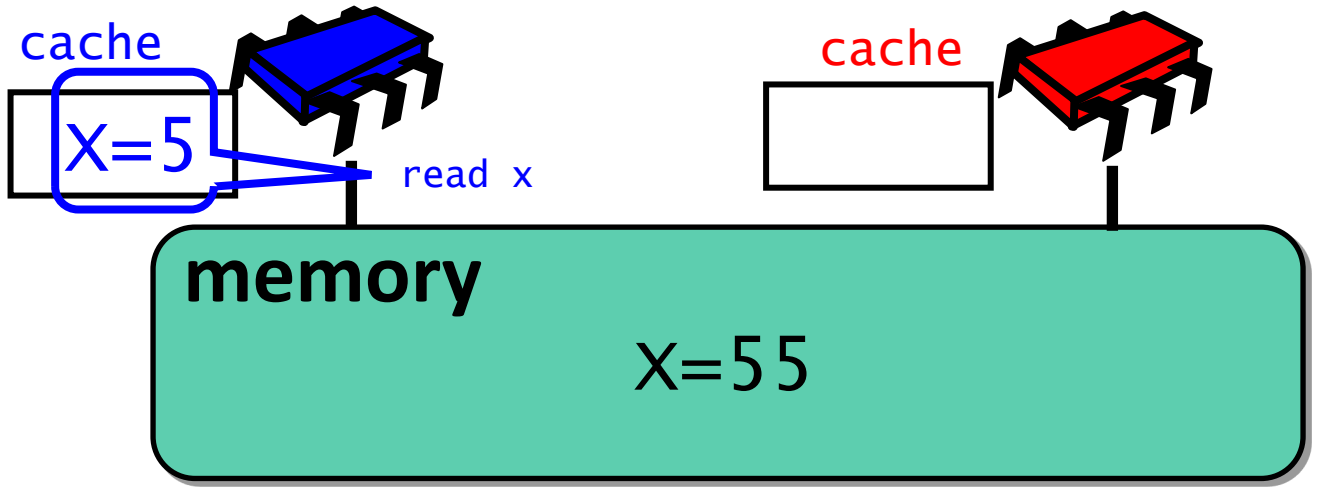
Cache coherence

The **cache incoherence** problem: core can read **stale** data



Cache coherence

The **cache incoherence** problem: core can read **stale** data



Cache coherence

The **cache incoherence** problem: core can read **stale** data

Solution: Caches use a **distributed protocol** to guarantee fresh data

$\Rightarrow$ In a cache coherent system, caches are **invisible**

Any execution observed could also occur without caches

(Hence, our simplified model!)

MSI protocol

Common denominator of commercial protocols (e.g.: MESIF, MOESI)

Idea: Single-writer/multi-reader protocol

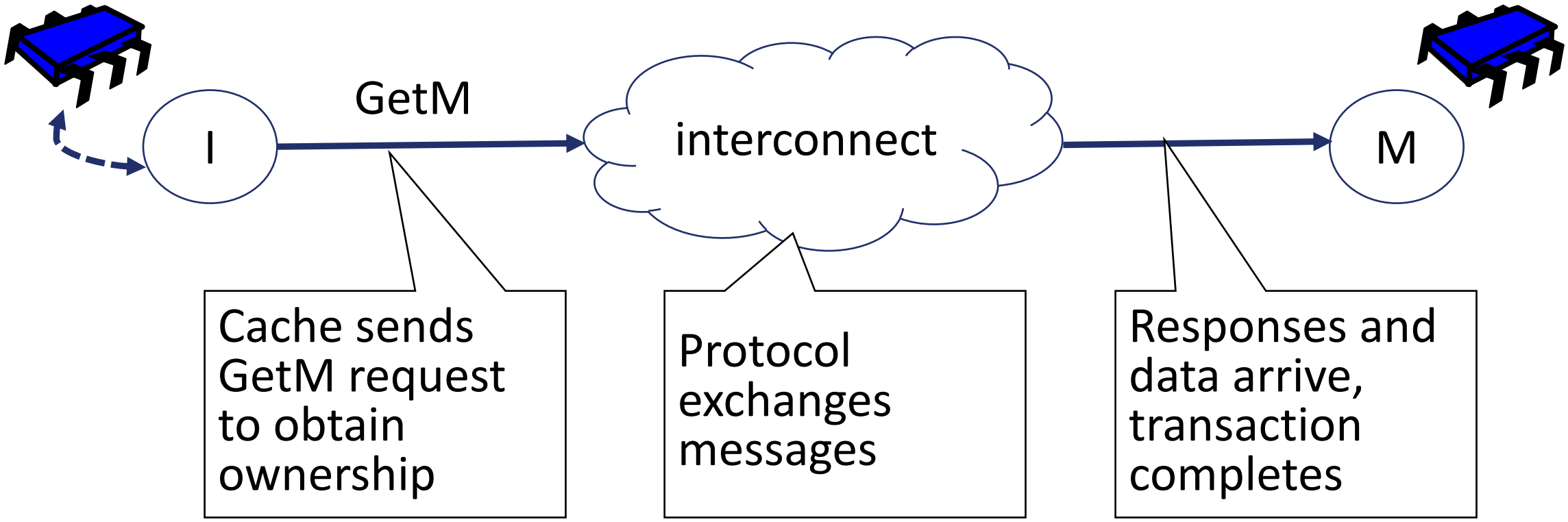
For any cache line, at any time, there is either a single core that may write it (and read it), or some number of cores that may read it

A core's cache has a **state** for every line:

- **M**(odified): core allowed to write/read
- **S**(hared): core allowed to read
- **I**(nvalid): core can't read nor write

MSI protocol: obtaining write permission

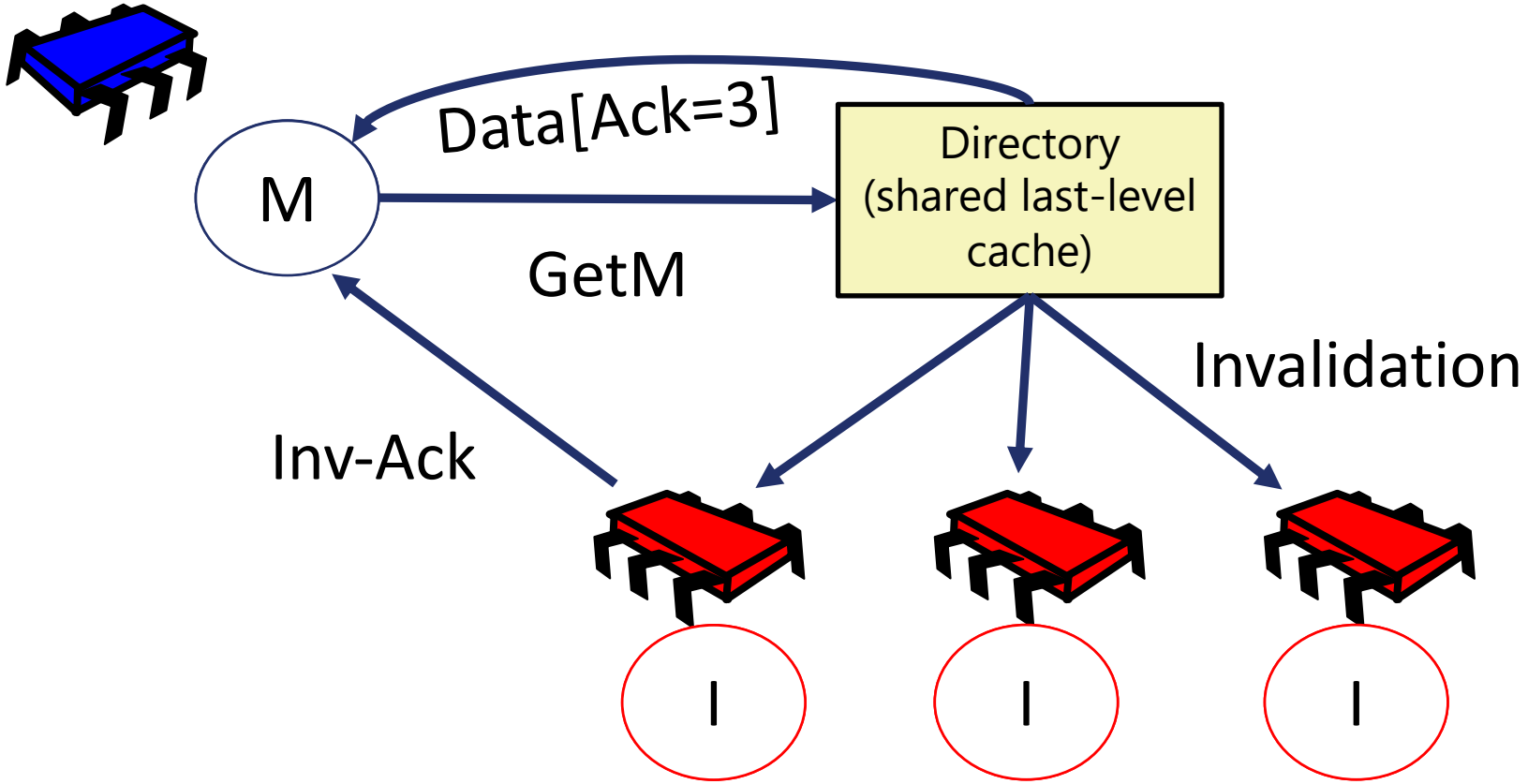
Core attempts to write a cache line



MSI protocol: obtaining write permission

Modern servers use **directory-based** protocols

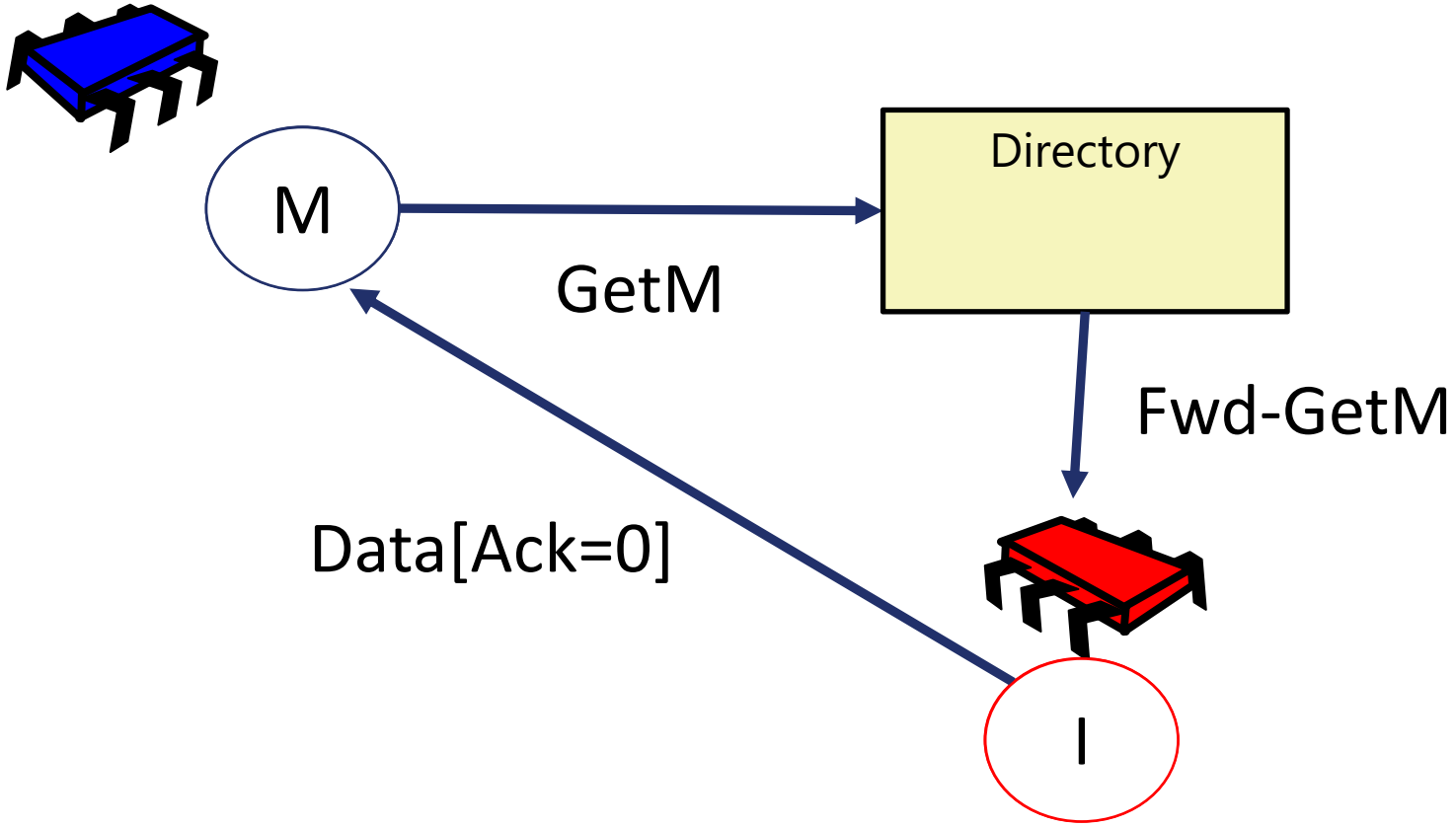
Directory holds the state of each line & manages ownership transfers



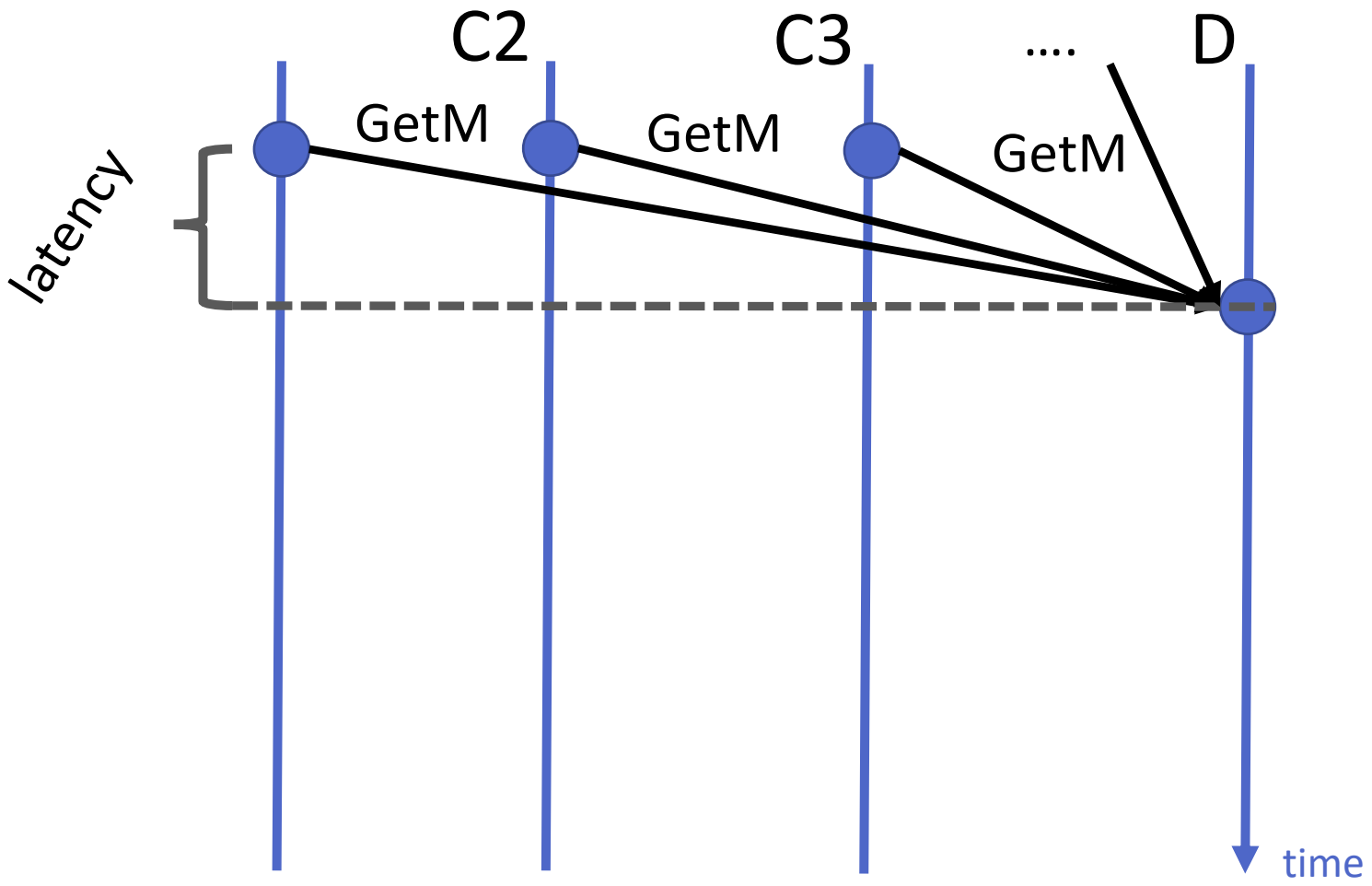
MSI protocol: obtaining write permission

Servers use **directory-based** protocols

Directory holds the state of each line & manages ownership transfers

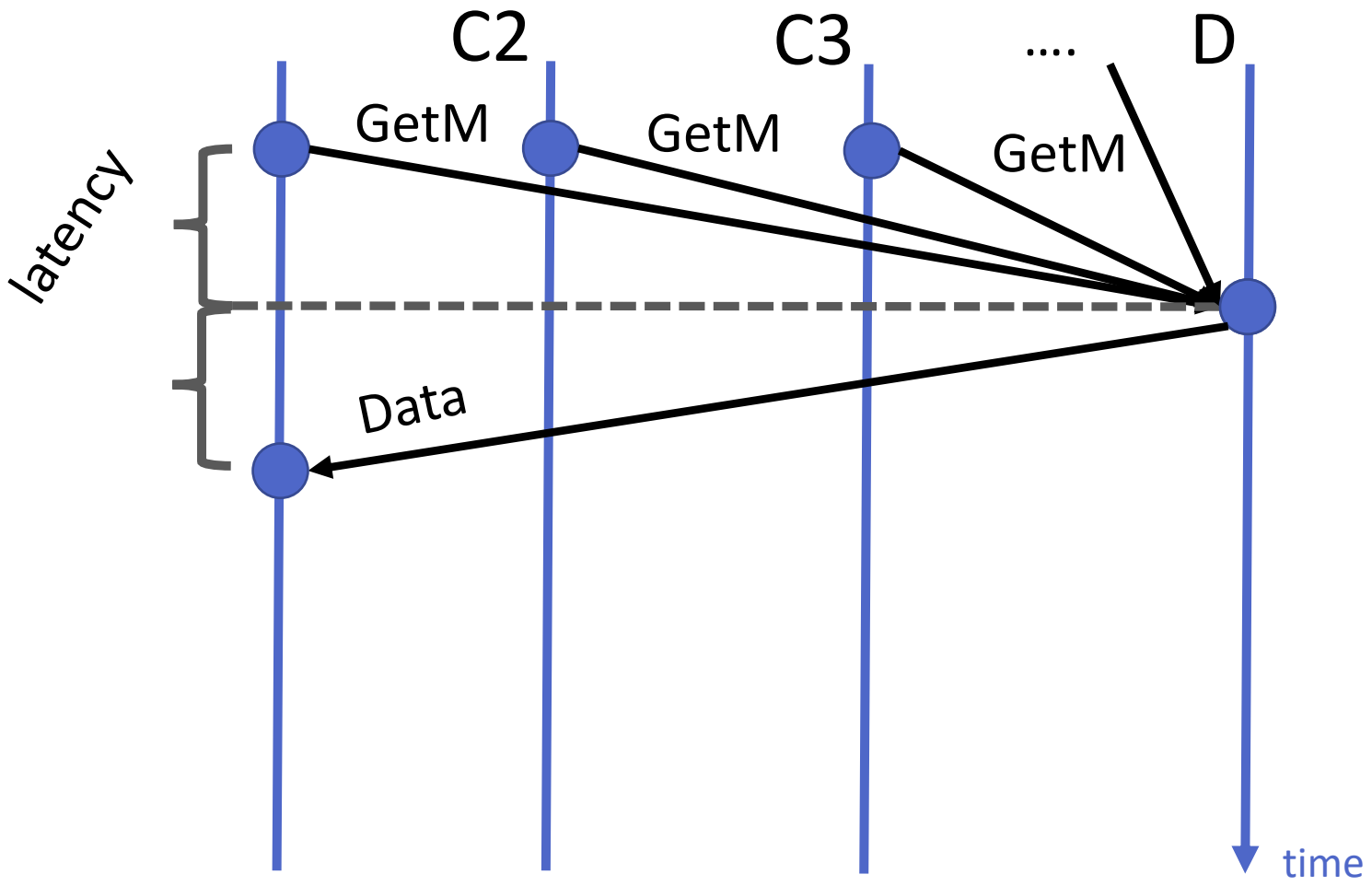


Cache coherence serialization



What happens when many cores write to a cache line concurrently?

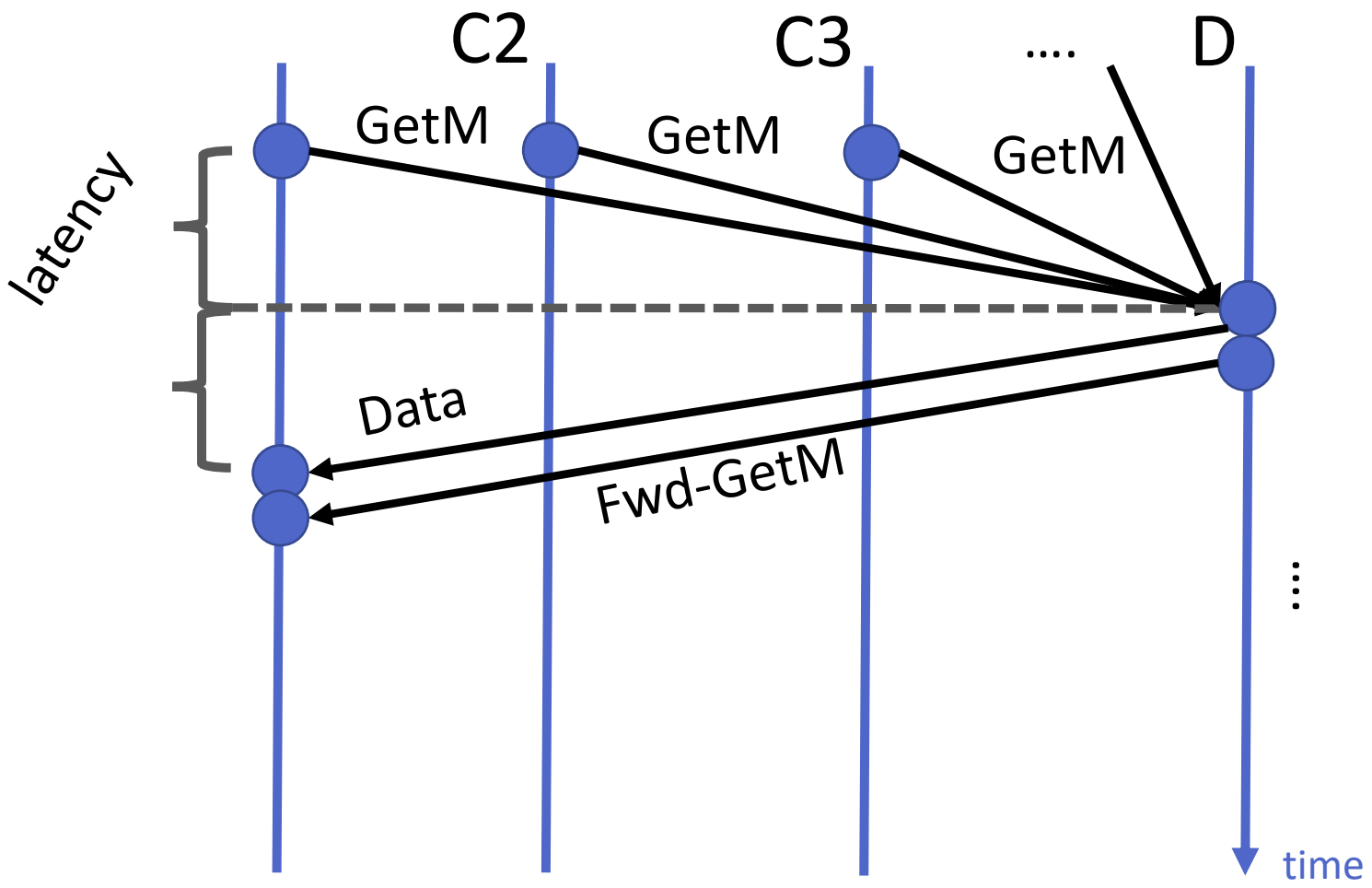
Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

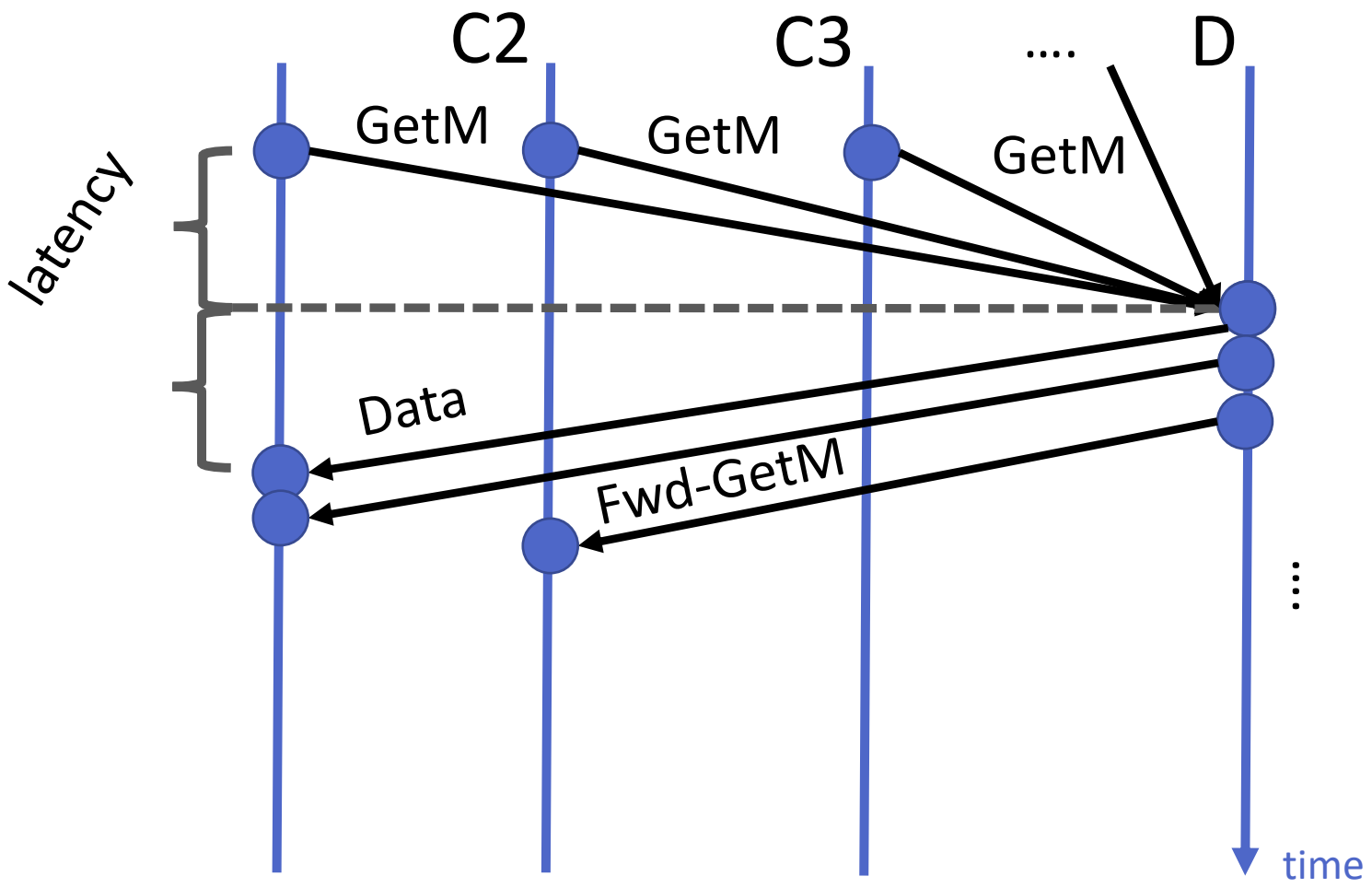
Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

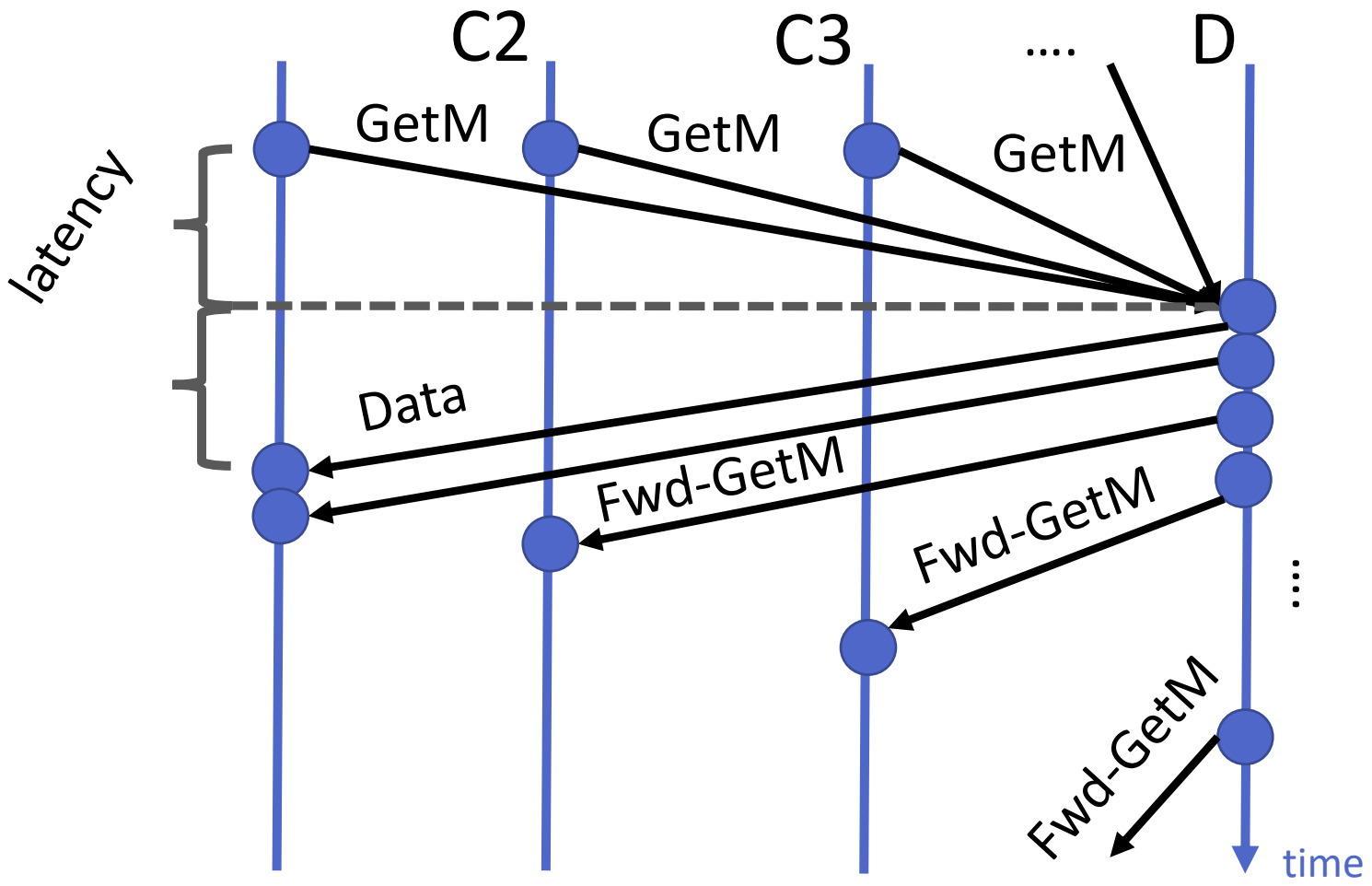
Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

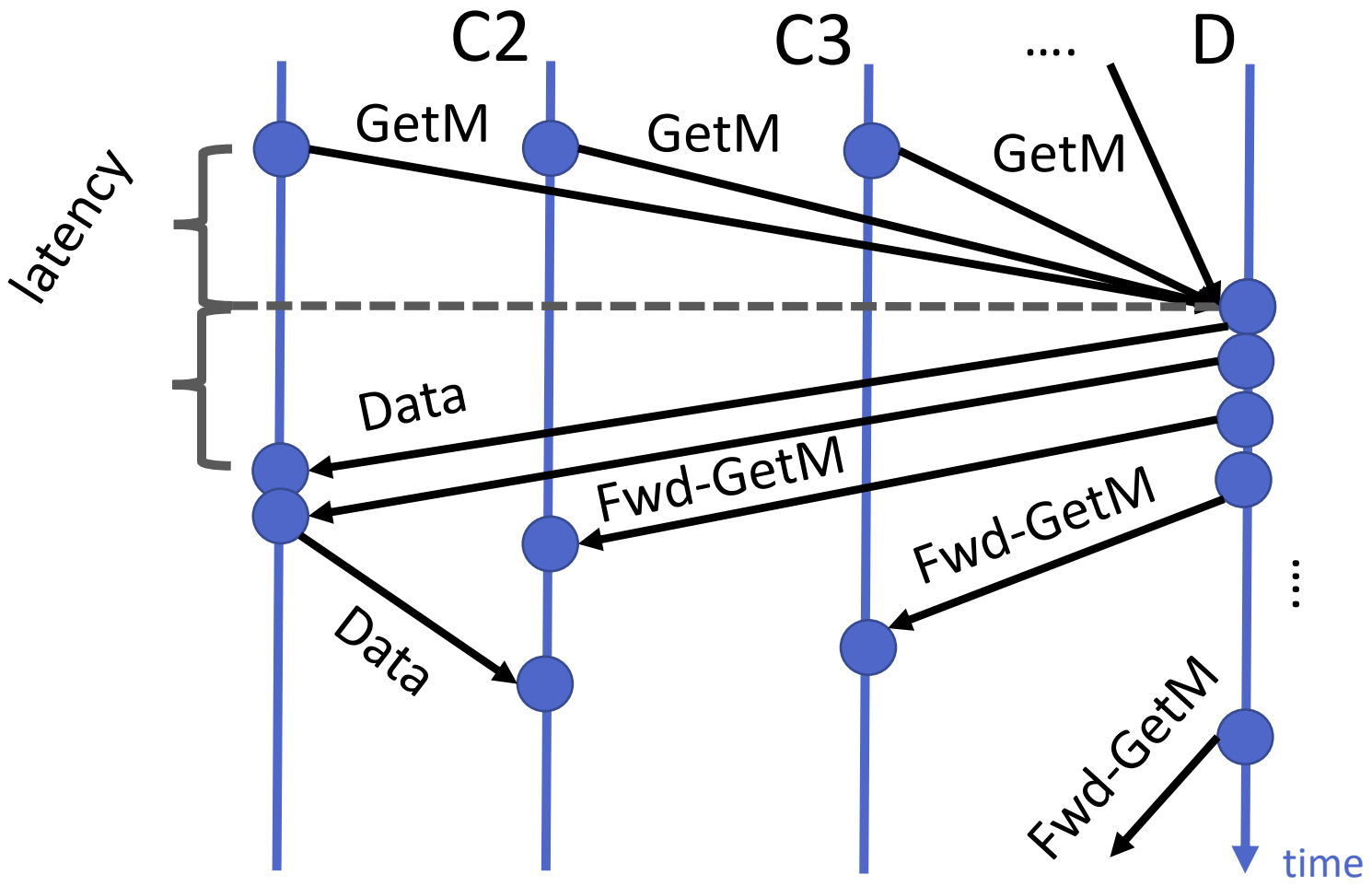
Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

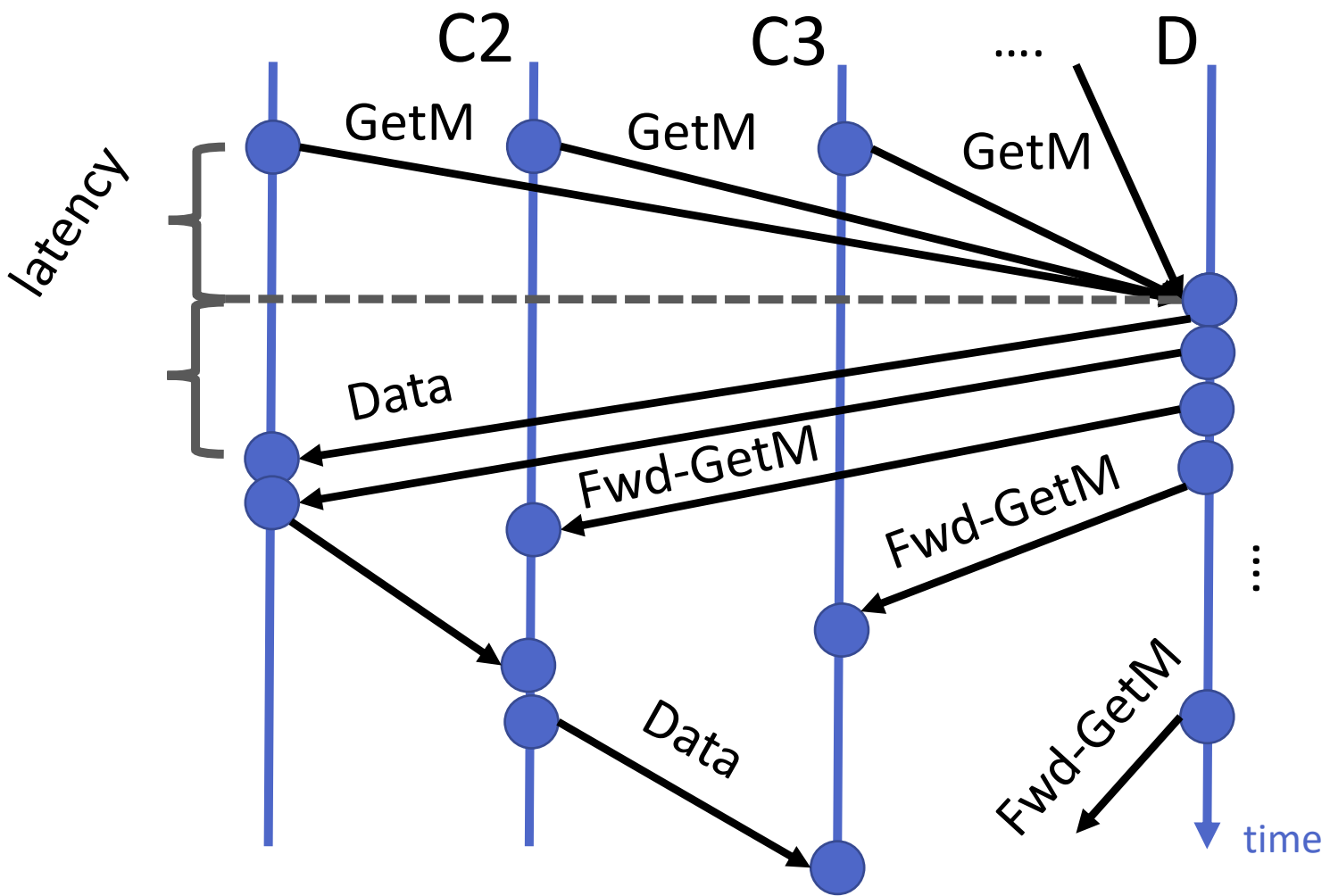
Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

Cache coherence serialization



What happens when all cores write to a cache line concurrently?

Directory **serializes** requests

⇒Time to acquire write permission increases linearly with # of acquiring cores!

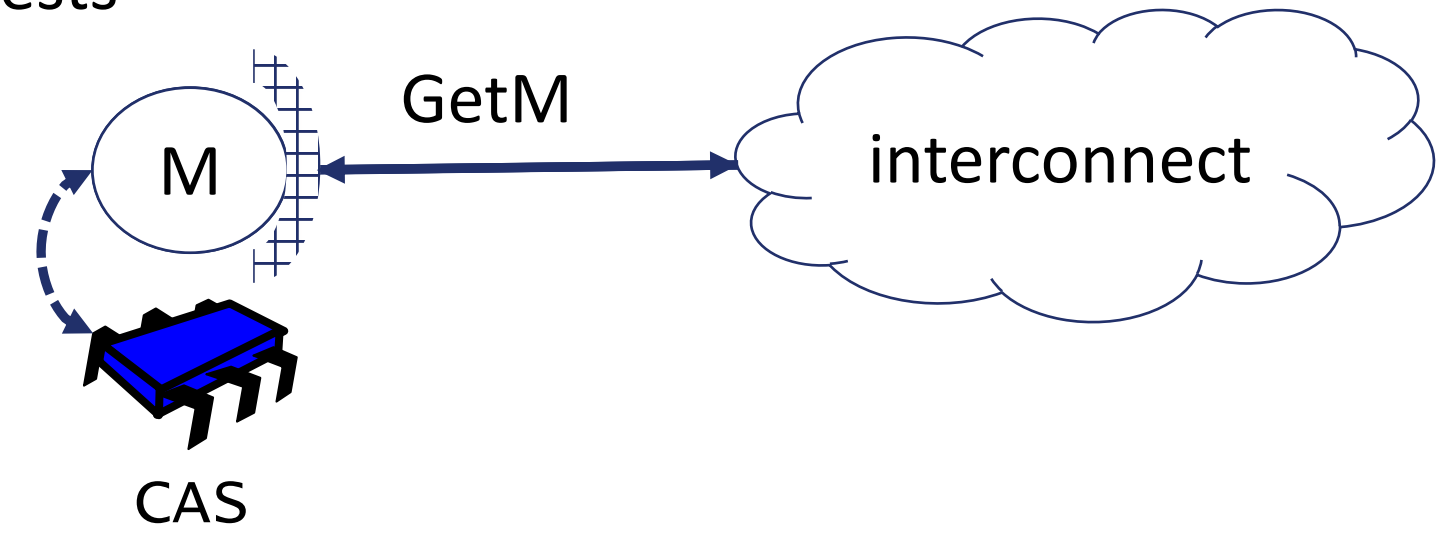
Cache line contention

Atomic read-modify-write instructions

Atomicity guaranteed if cache line isn't **invalidated** between R and W

No need to "lock the bus"

- Get line in M state
- Cache stalls incoming requests for line
- Cores performs RMW instruction (read+write)
- Cache resumes processing incoming requests

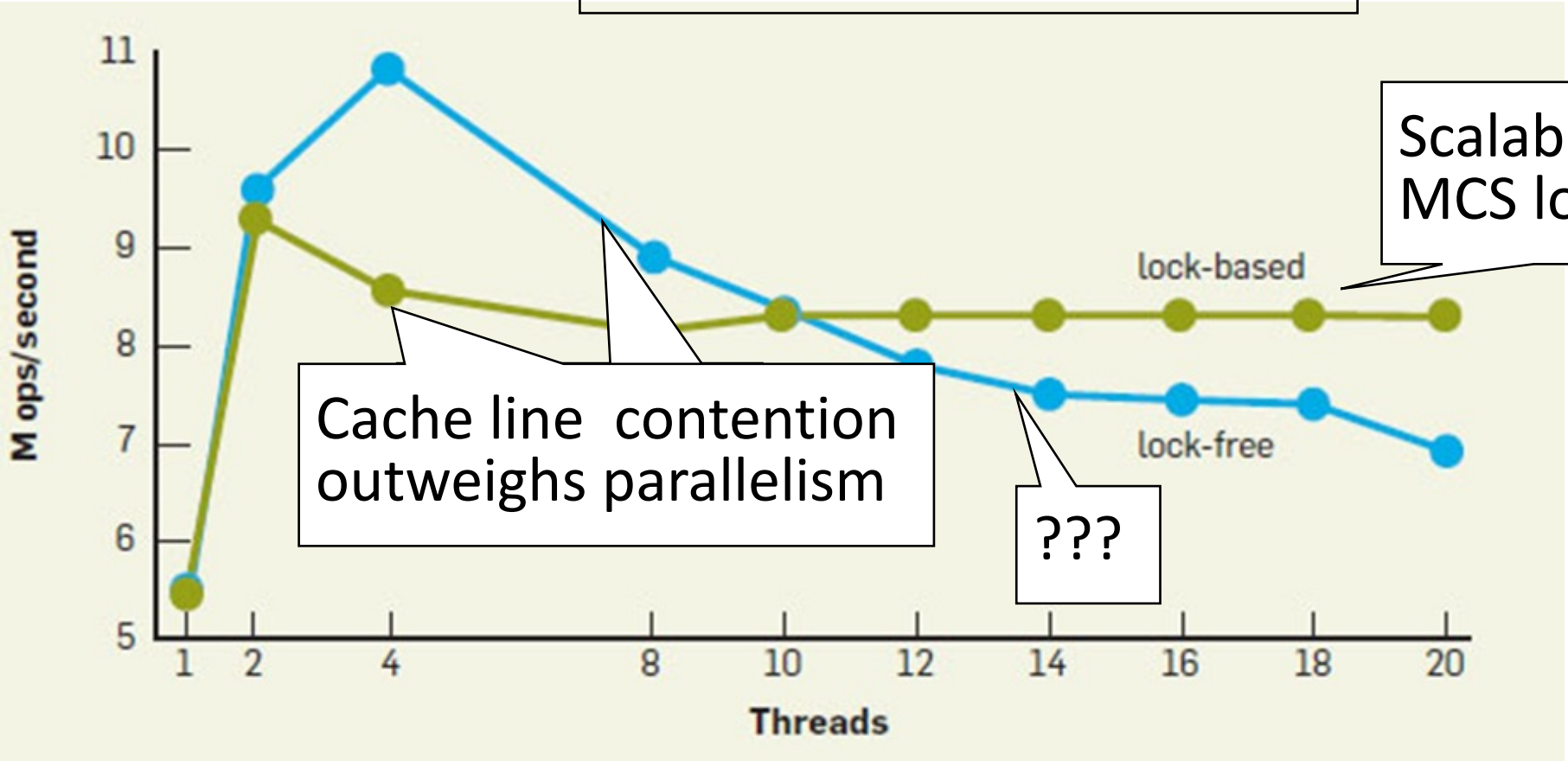


Contended atomics → cache line contention

Acquire time = **N** * **L** cycles

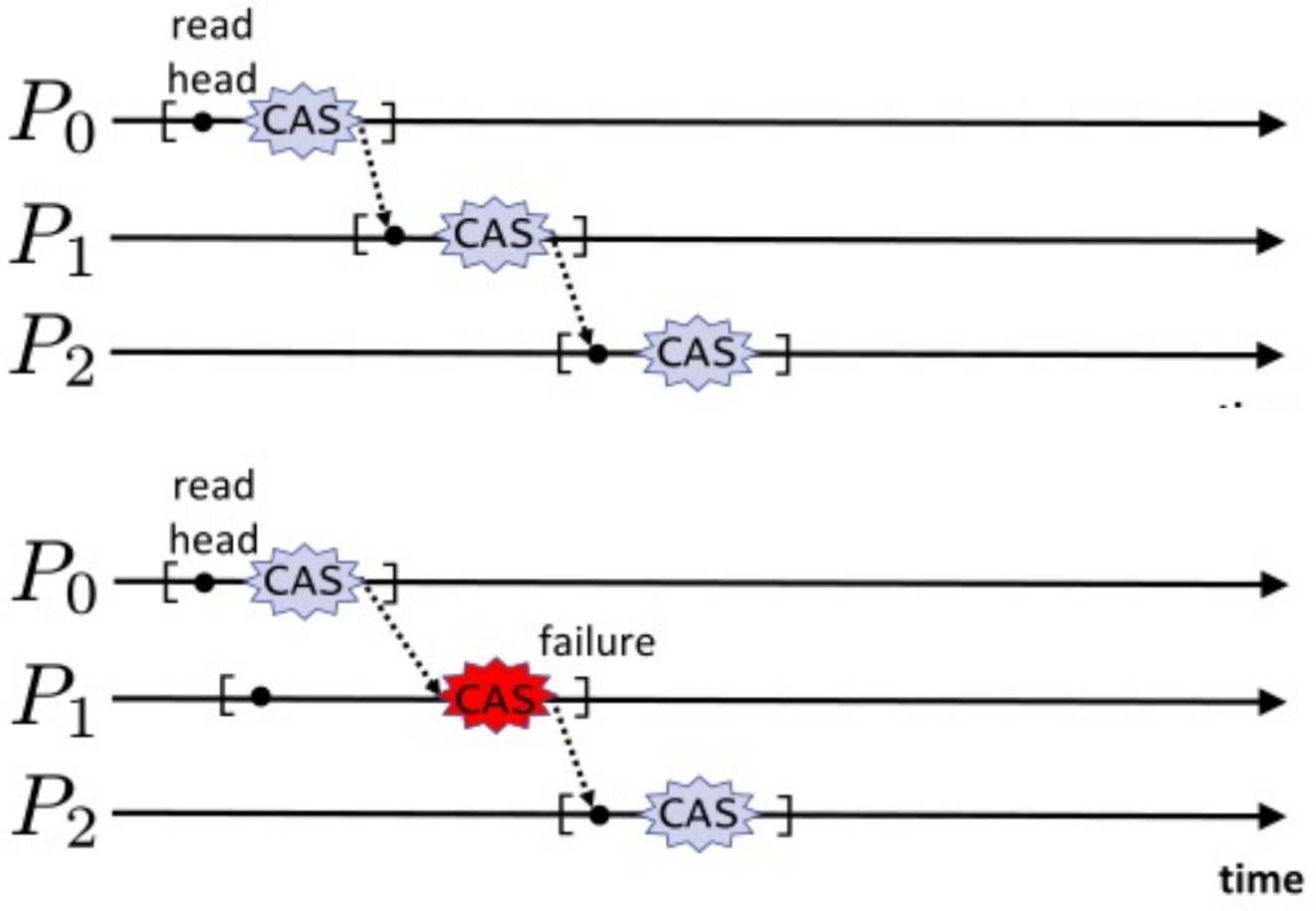
N = # of contending cores

L = invalidation latency



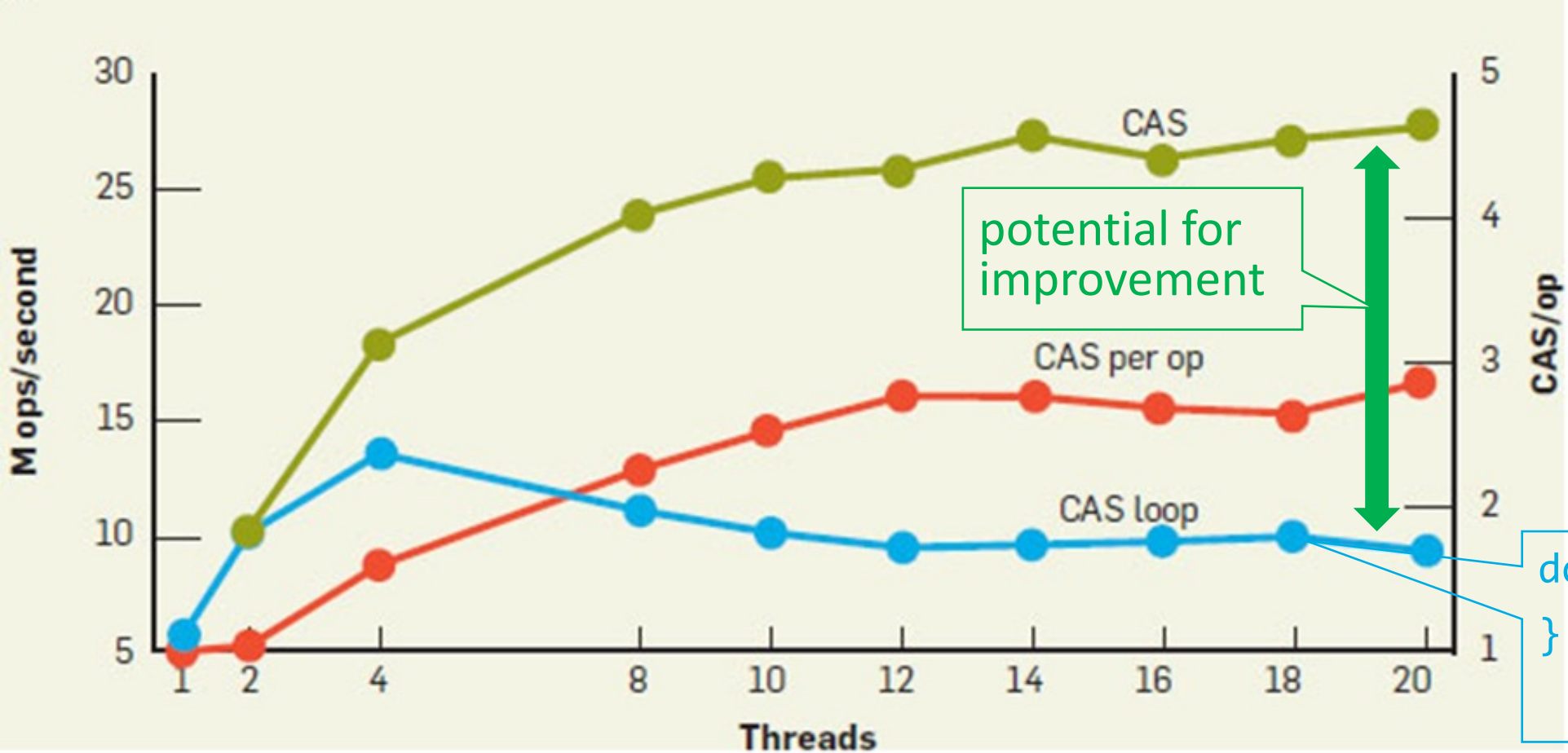
CAS failures

A failed CAS **delays** all cores but doesn't complete **useful work**

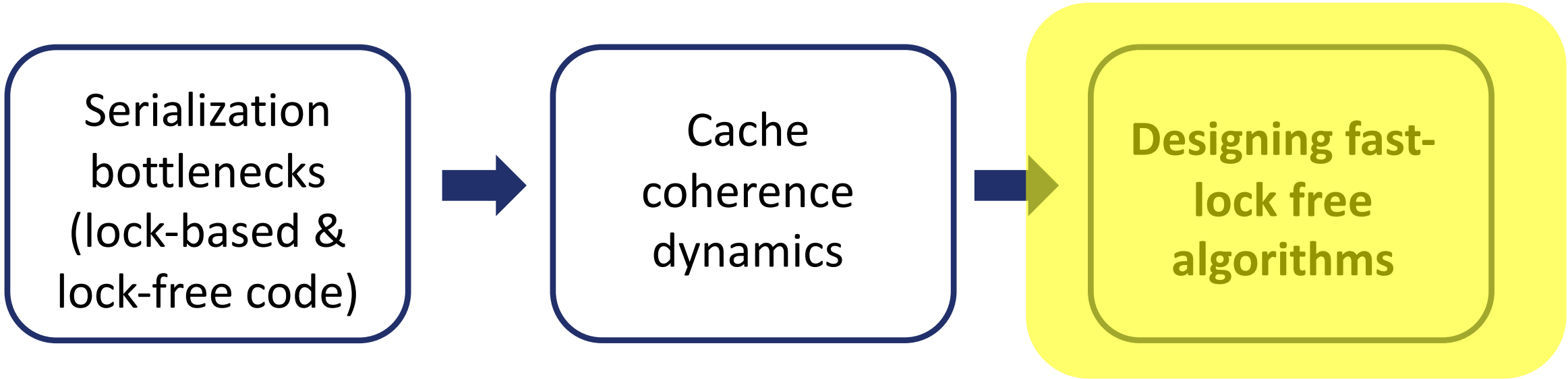


CAS failures

Raw CAS throughput is higher than CAS loop (queue) throughput



Part 3


Serialization bottlenecks (lock-based & lock-free code)
Cache coherence dynamics
Designing fast-lock free algorithms

Avoiding CAS failures

Idea: Try to replace CAS with an atomic instruction that **doesn't fail**, so that every atomic operation contributes to **useful work**

Will use **fetch-and-add** (FAA)

```
FAA(int *addr, int x) {
    atomic {
      int old = *addr;
      *addr += x;
      return old;
    }
}
```

Avoiding CAS failures

Plan:

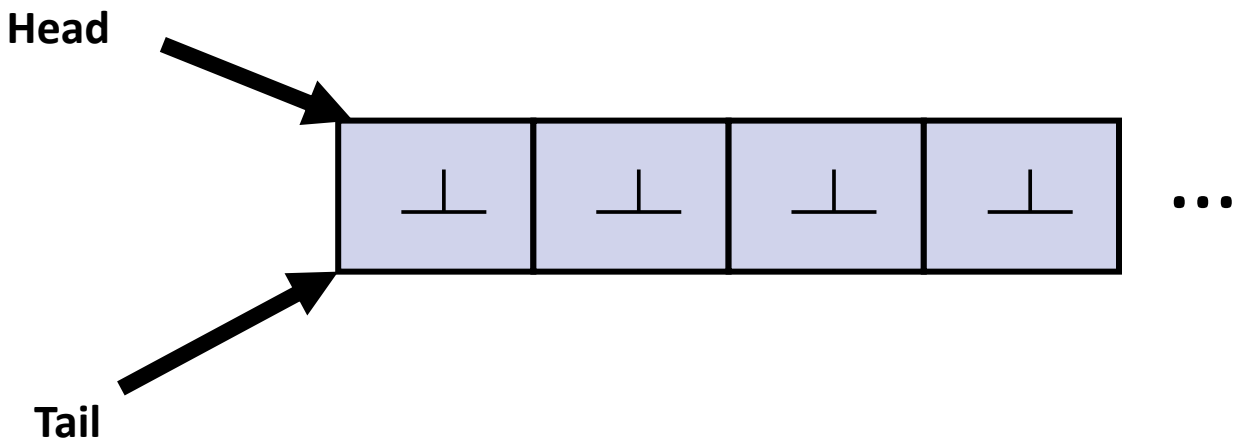
Simple but unrealistic algorithm to illustrate idea

Convert it to a practical algorithm

(Unrealistic) FAA queue

Data structure is **infinite array**

Head and **Tail** are indices (conceptually, pointers) into array

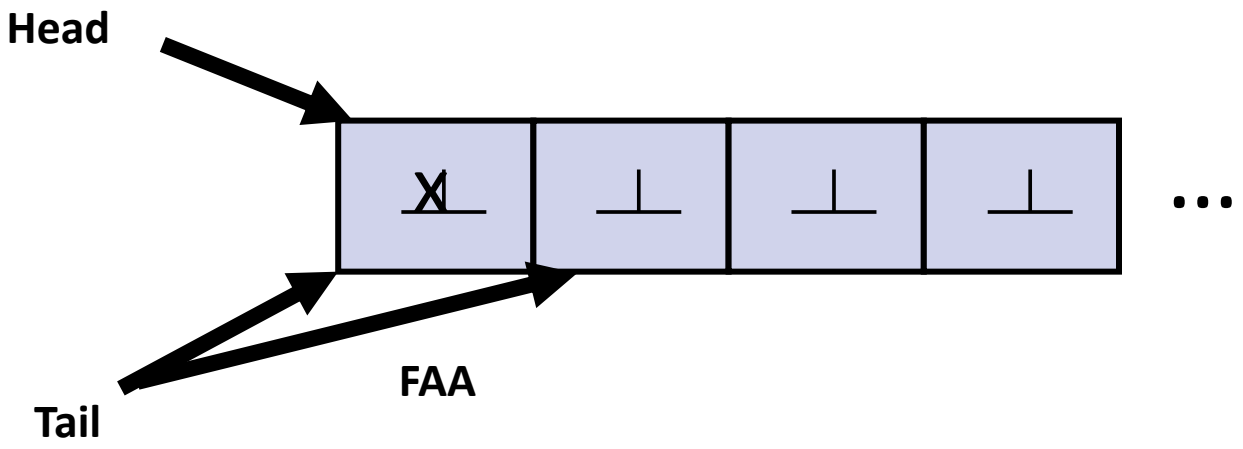



```
Cell { T val; };

Initially:
Head = Tail = 0;
Q = Cell[]
All cells have value ⊥
```

(Unrealistic) FAA queue enqueue

Obtain **unique** cell index (**contended** operation)

CAS value into cell (**not** contended)



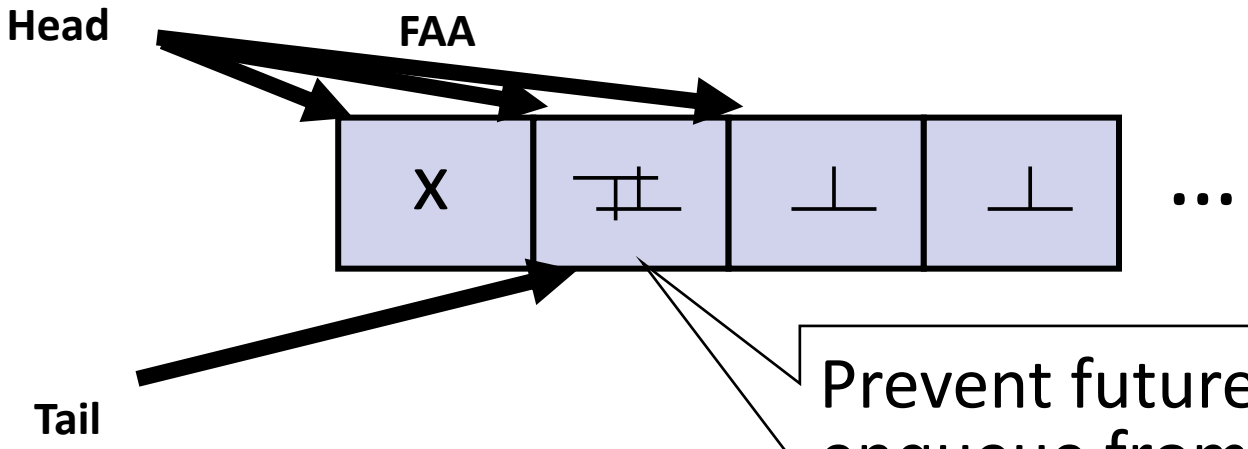

```
enqueue(x) {
 while (true) {
  t = F&A(&Tail, 1)
  if (CAS(&Q[t], ⊥, x))
    return
  }
}
```

(Unrealistic) FAA queue dequeue

Obtain **unique** cell index (**contended** operation)

If cell not empty: return value

If cell empty:




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

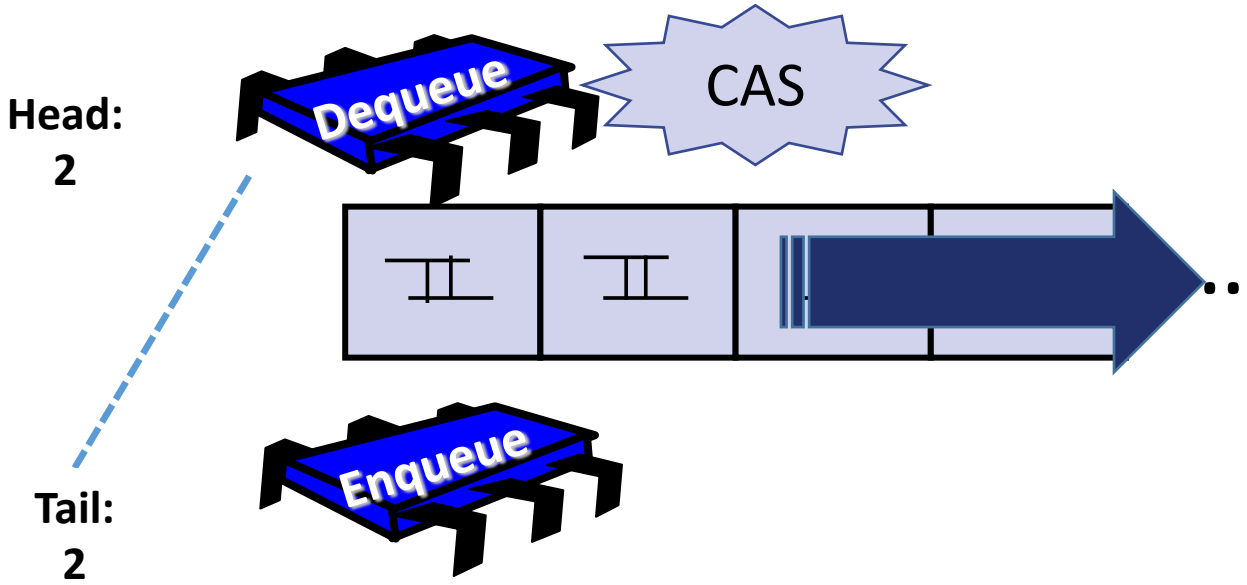
FAA queue problems

1) I**nfinite arrays don’t exist**

FAA queue problems

1) Infinite arrays don’t exist

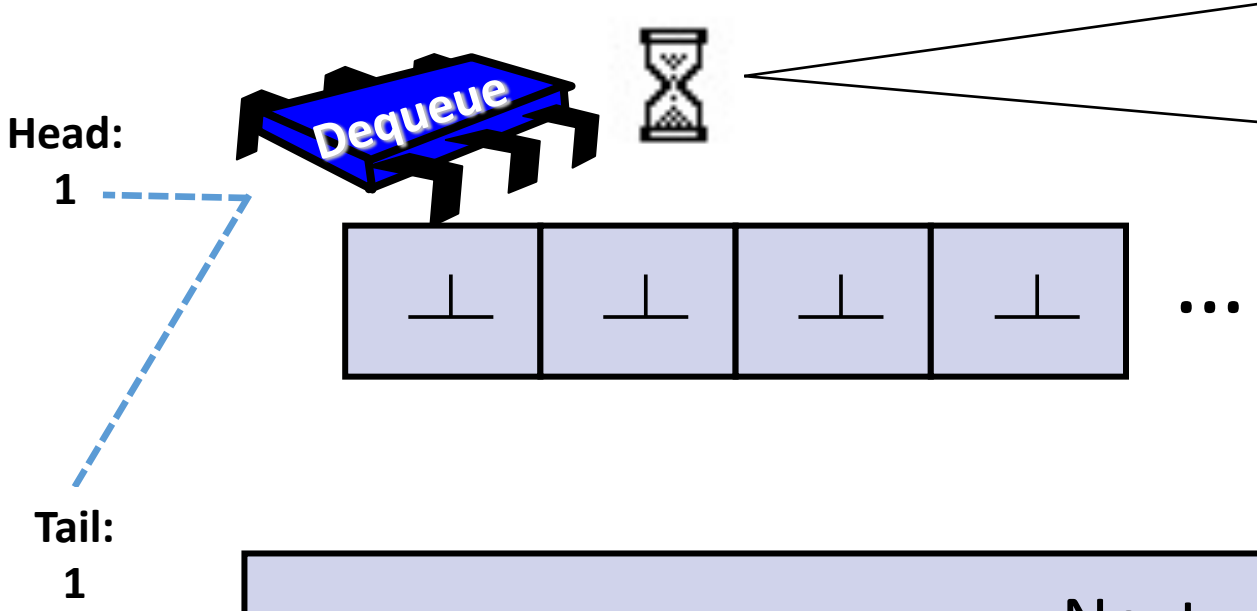
2) **Algorithm isn’t lock-free!**



FAA queue problems

1) Infinite arrays don’t exist

2) Algorithm isn’t lock-free!



Heuristic solution:

Dequeue **waits** for enqueue to make progress

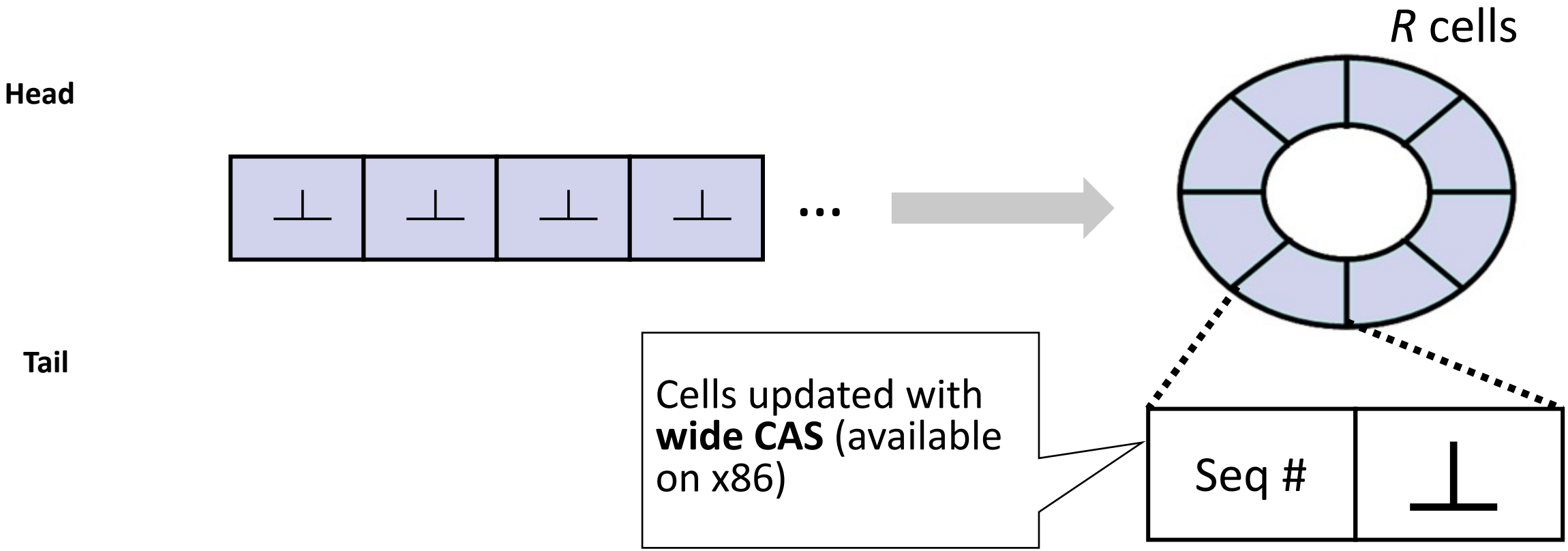
Still not **formally** lock-free

Next:

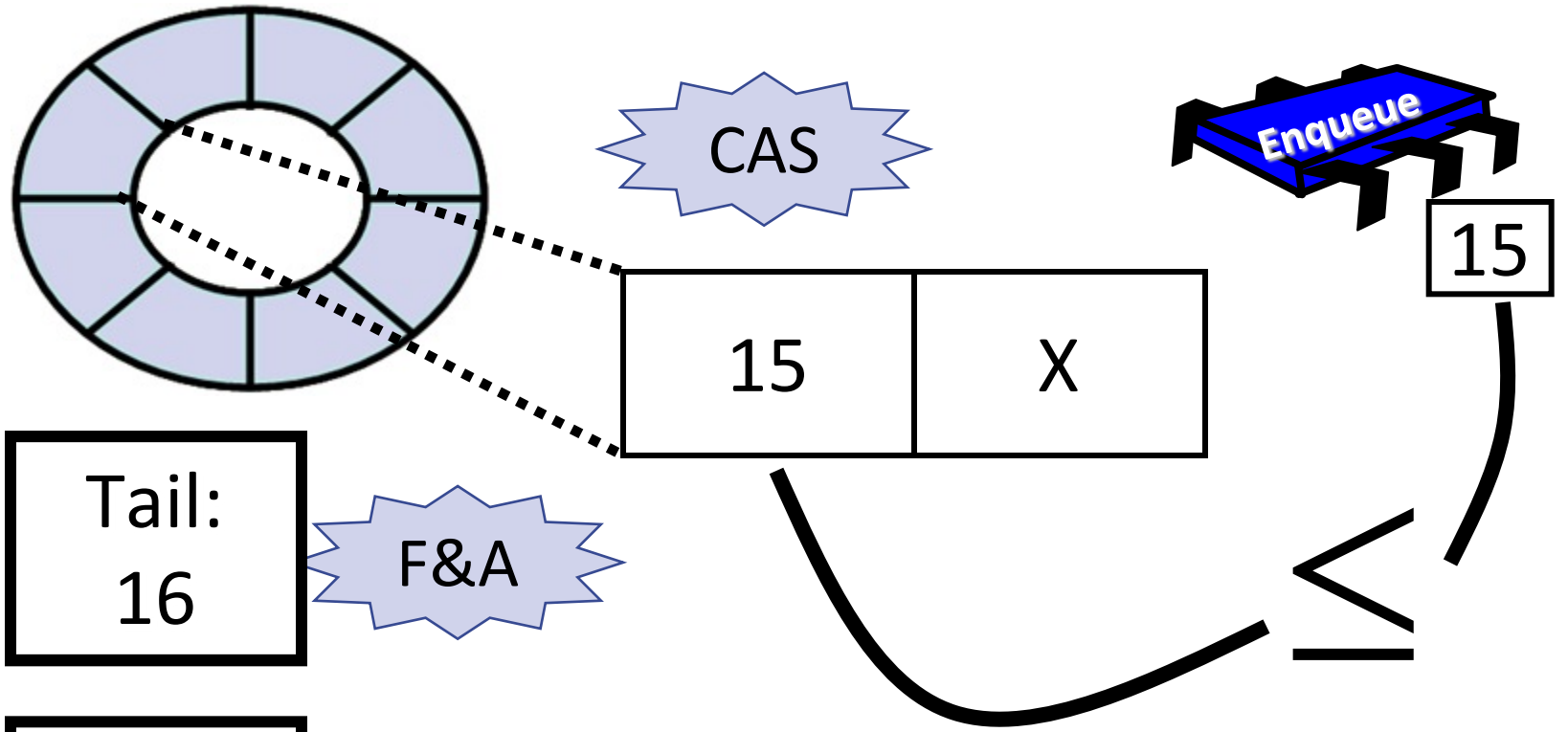
LCRQ algorithm, solves both problems

Cyclic ring queue (CRQ)

1) Don’t have infinite arrays →Represent queue as **cyclic** array



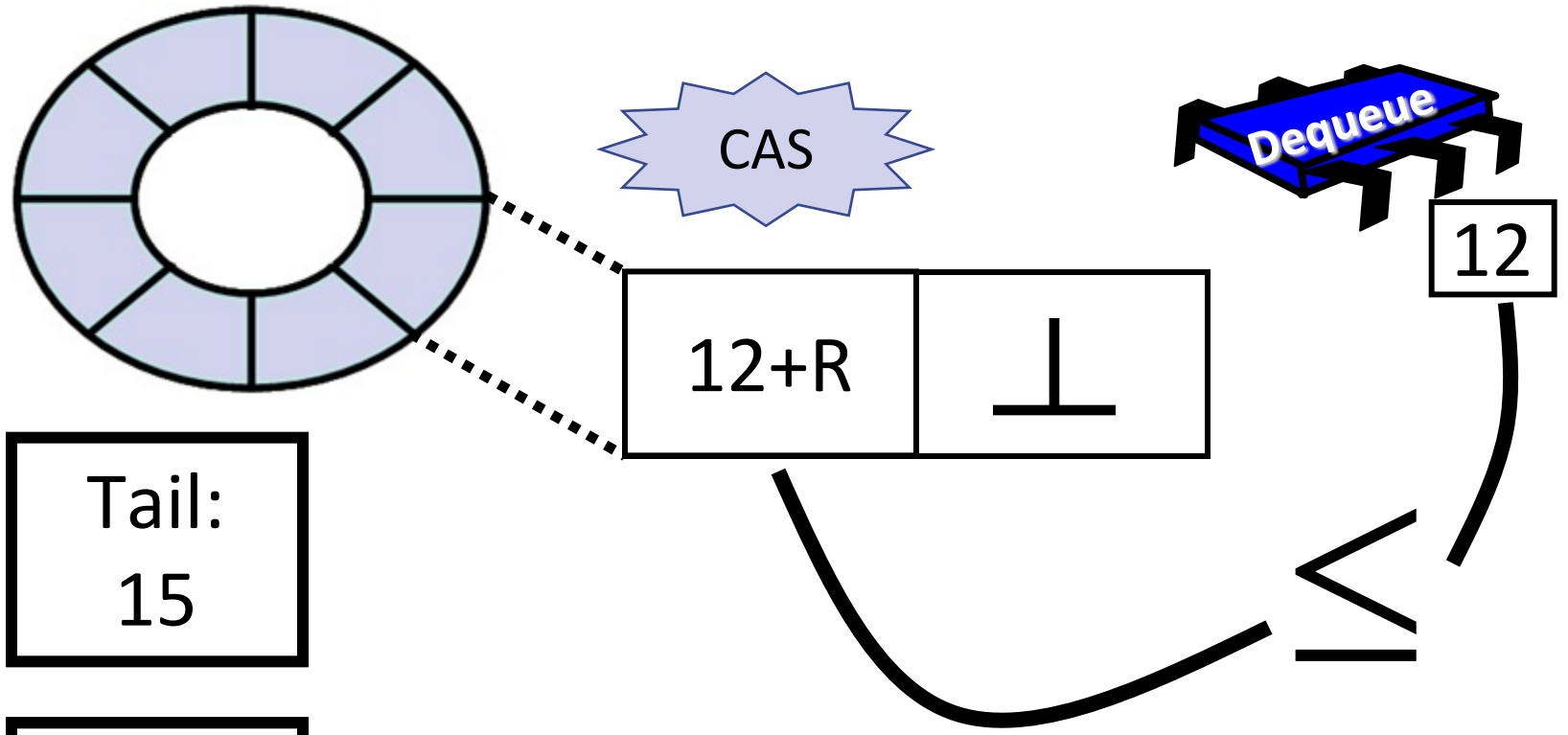
CRQ enqueue



If cell's seq# not greater than mine, deposit value and update seq# to mine.

Otherwise, retry

CRQ dequeue



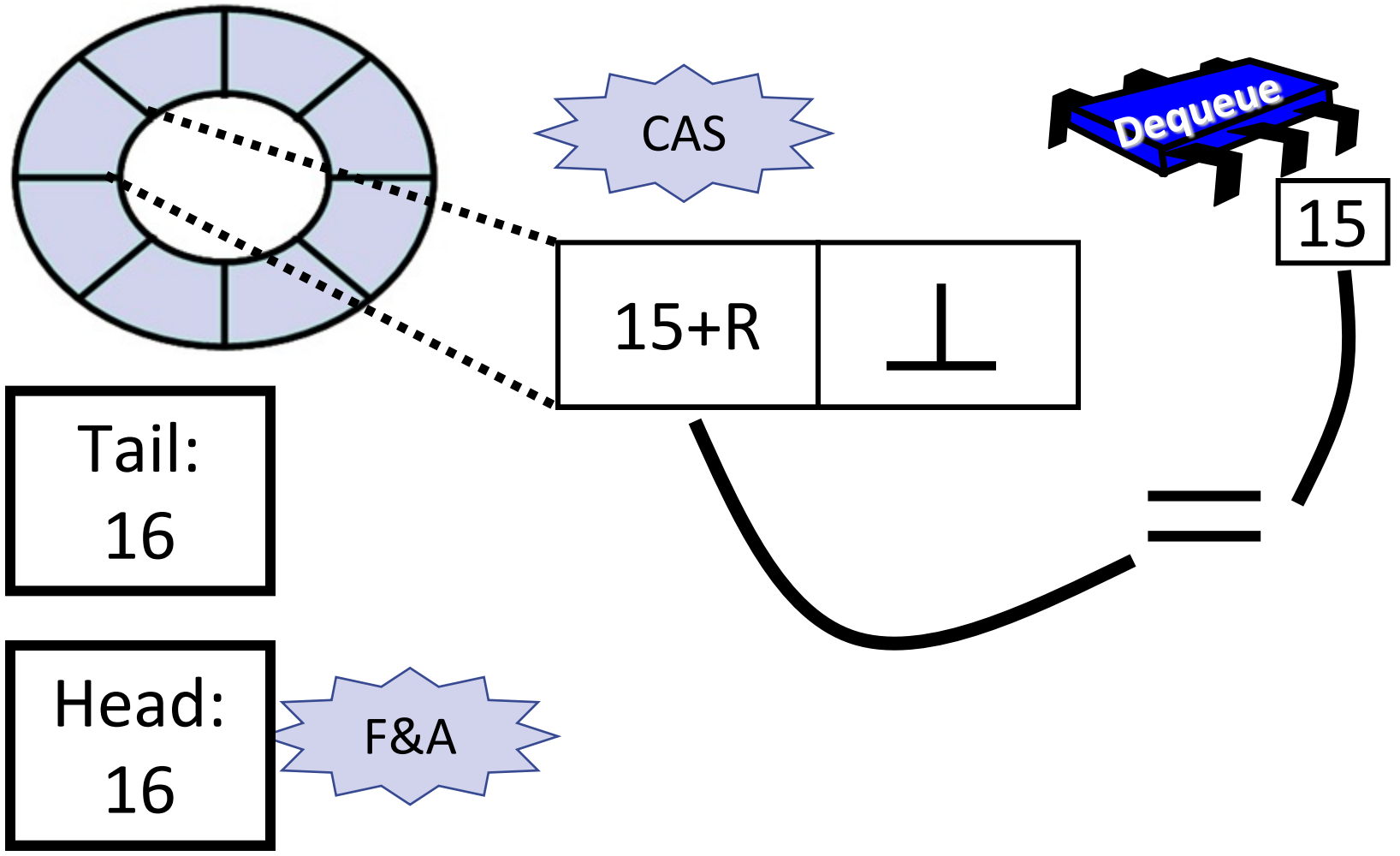
Tail:
15

Head:
13

Cell empty:

If seq# not greater than mine, **prevent future enqueue**.

CRQ dequeue



Cell empty:

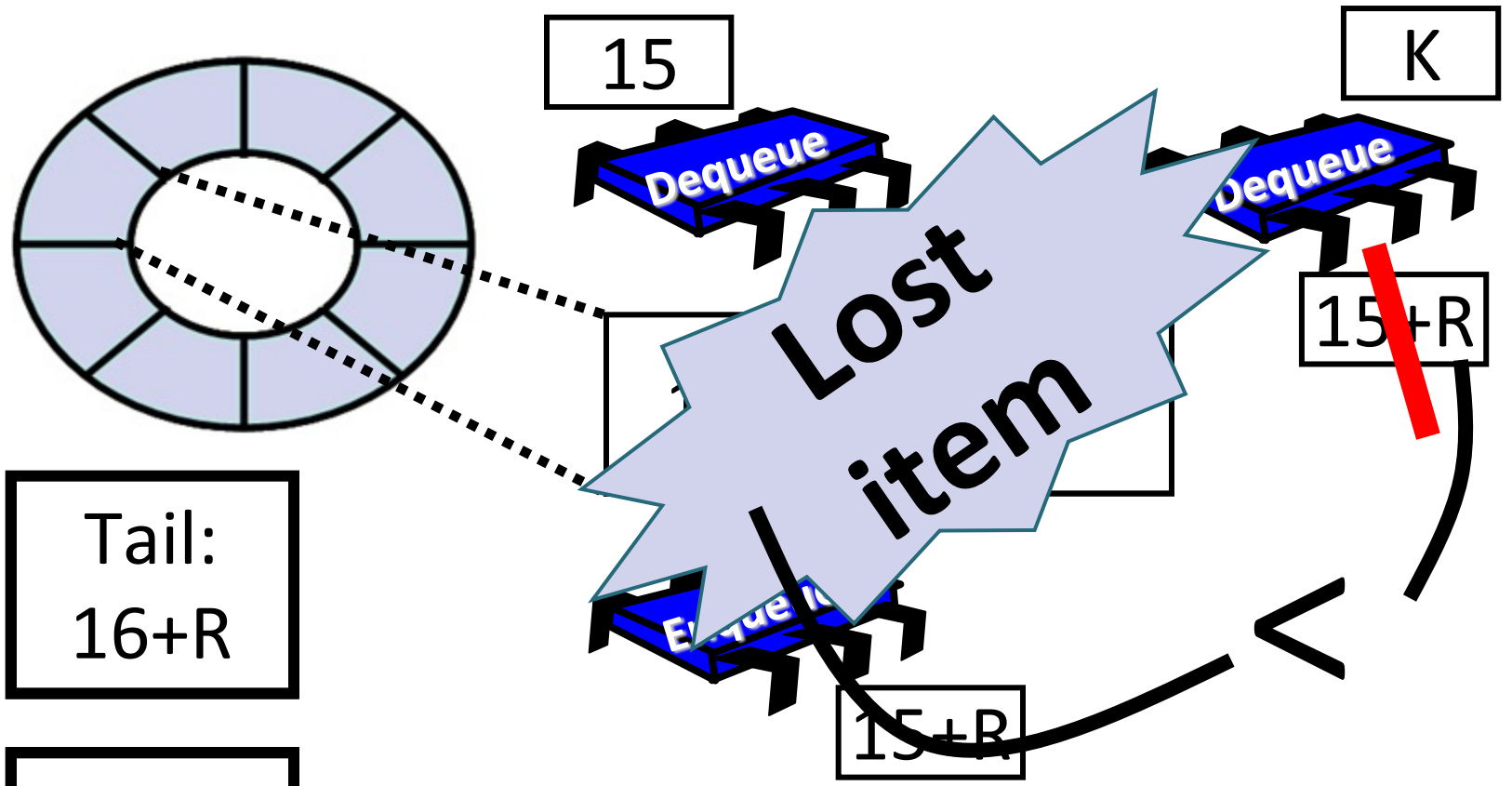
If seq# not greater than mine, **prevent future enqueue**.

Cell not empty:

If seq# = mine, return value and bump seq# to next iteration

If seq# > mine, retry

CRQ dequeue



Tail: 16+R

Head: K+1

Dequeue arrives before enqueue while cell is occupied by previous iteration – **no analogy in infinite array case**.

Cell empty:

If seq# not greater than mine, **prevent future enqueue**.

Cell not empty:

If seq# = mine, return value and bump seq# to next iteration

If seq# > mine, retry

Otherwise: retry???

CRQ dequeue

Solution:

Bump seq# + **mark** the cell as **unsafe** for future enqueues (1 bit extra state)

Mark **removed** by later enqueue *E* if *E* detects that the corresponding dequeue **hasn't started** yet (by checking Head ≤ *E*'s index)

(full details in paper: https://www.cs.tau.ac.il/~mad/publications/ppopp2013-x86queues.pdf)

Cell empty:

If seq# not greater than mine, **prevent future enqueue**.

Cell not empty:

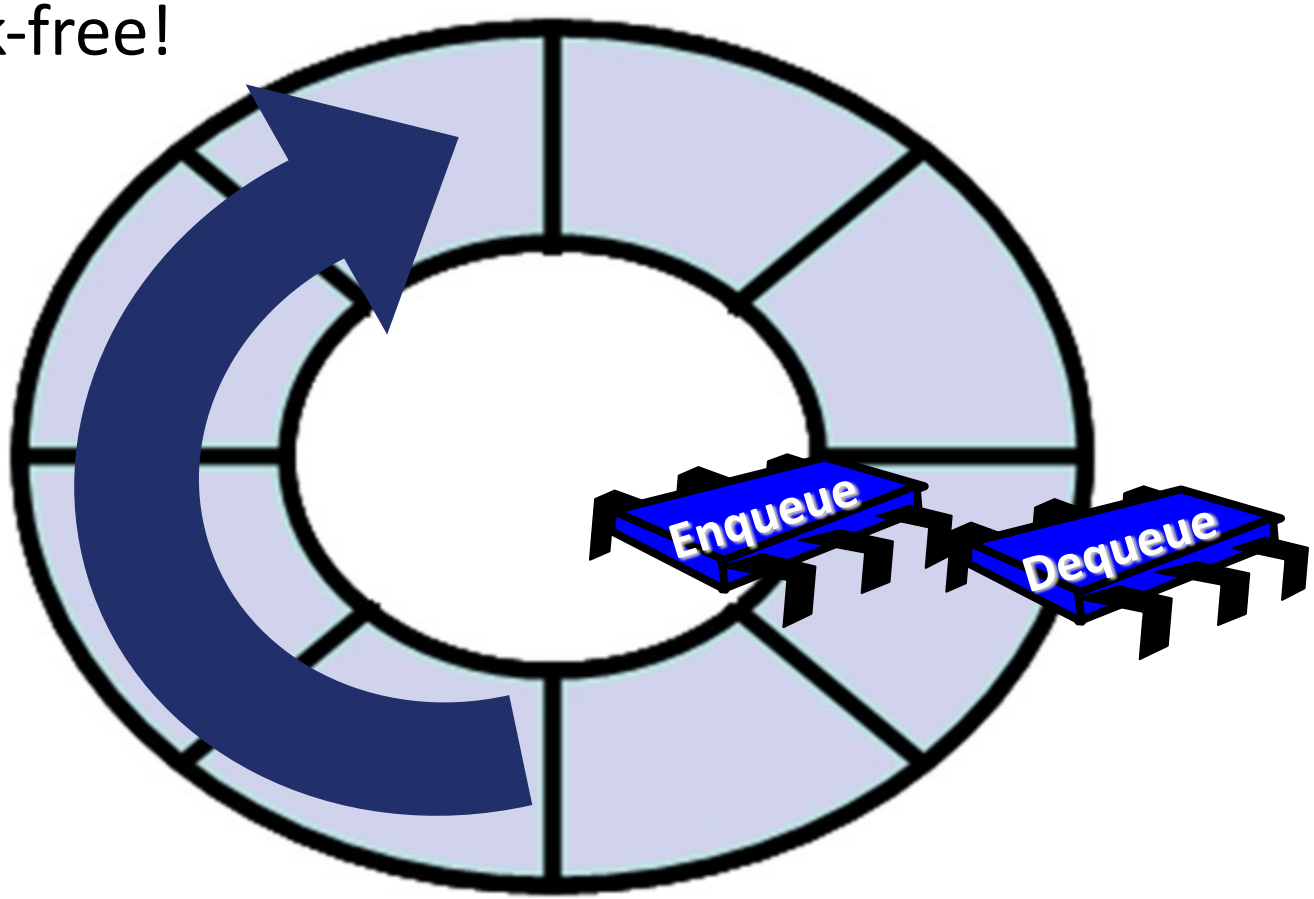
If seq# = mine, return value and bump seq# to next iteration

If seq# > mine, retry

Otherwise: retry???

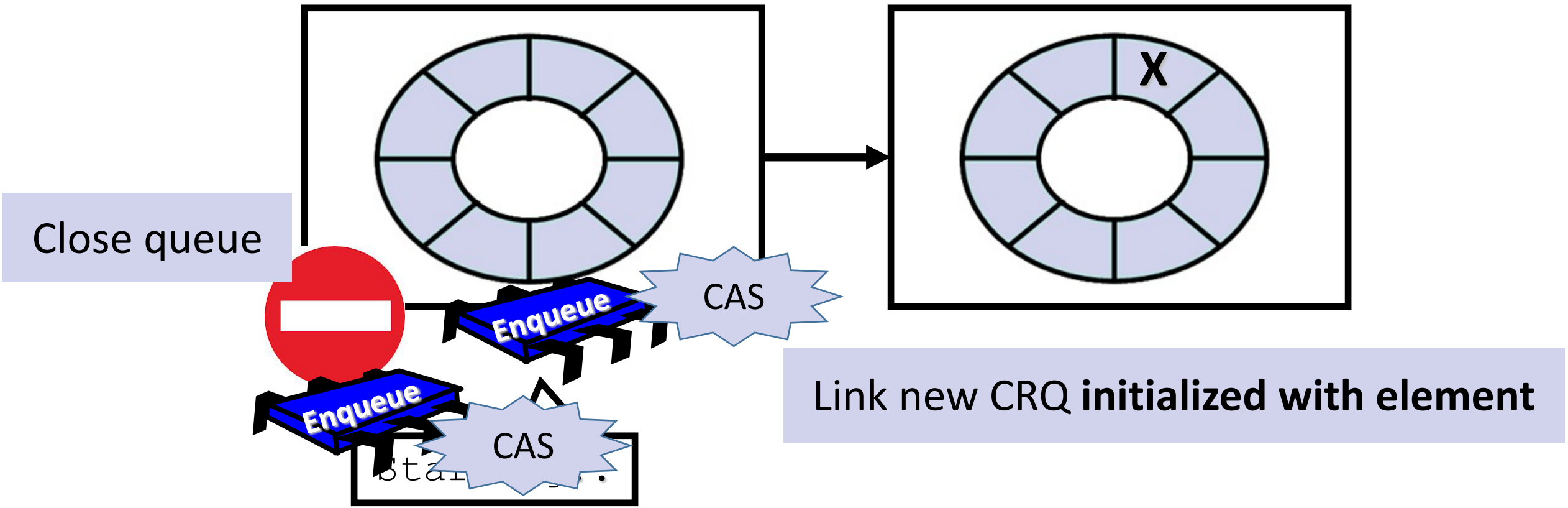
CRQ livelocks

1) Don’t have infinite arrays →Represent queue as **cyclic** array
2) Algorithm (still) isn’t lock-free!

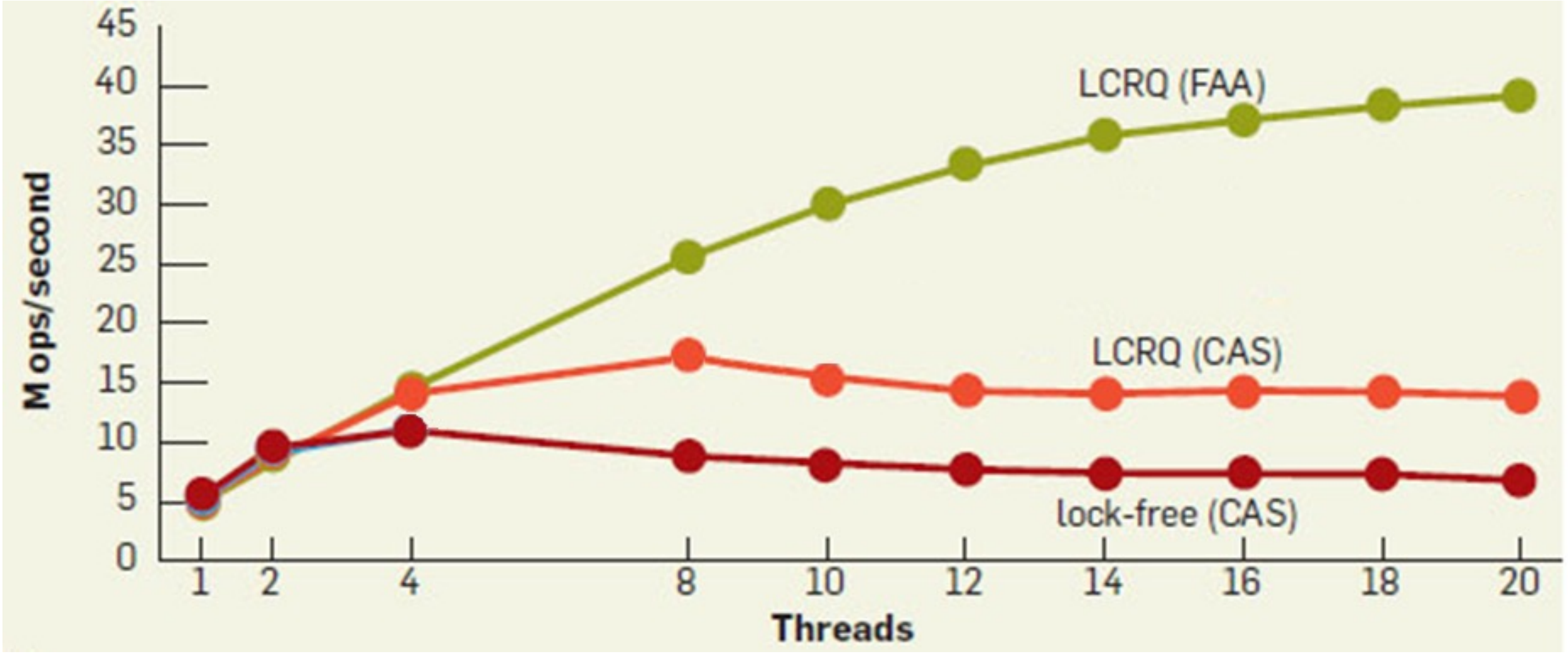


LCRQ [Morrison and Afek PPoPP'13]

1) Don't have infinite arrays →Represent queue as **cyclic** array
2) Algorithm isn't lock-free! → **List of CRQs**



Evaluation


45
40
35
30
25
20
15
10
5
0
M ops/second
LCRQ (FAA)
LCRQ (CAS)
lock-free (CAS)
1
2
4
8
10
12
14
16
18
20
Threads

Conclusion & takeaways

Lock-freedom **doesn’t imply** being faster than locking

$\Rightarrow$ Need to consider problem domain and workload

Contended CAS failures are very **wasteful** due to **coherence serialization**

$\Rightarrow$ Design to avoid them

Better yet: design to **avoid contention** in the first place

(Probably not possible for every data structure)