Concurrent Data Structures

Concurrent Algorithms 2015

Vasileios Trigonakis

Data Structures (DSs)

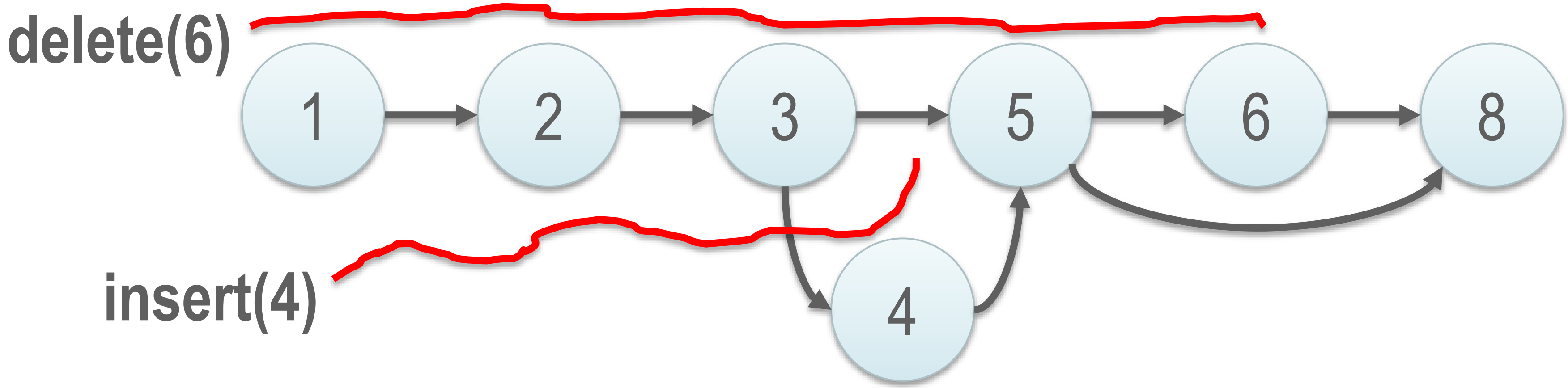
- Software constructs for efficiently storing data
  - **Different types**: lists, hash tables, trees, queues, …
- Accessed through the DS interface
  - Depends on the DS type, but always includes
  - Store an element
  - Retrieve an element
- **Element**
  - Set: just one value
  - Map: key/value pair

Concurrent Data Structures (CDSs)

- Concurrently accessed by multiple threads
  - Through the CDS interface → linearizable operations!

- Really important on multi-cores
- Used in most software systems

DS Example: Linked List



- A sequence of elements (nodes)
- **Interface**
  - search (aka contains)
  - insert
  - remove (aka delete)

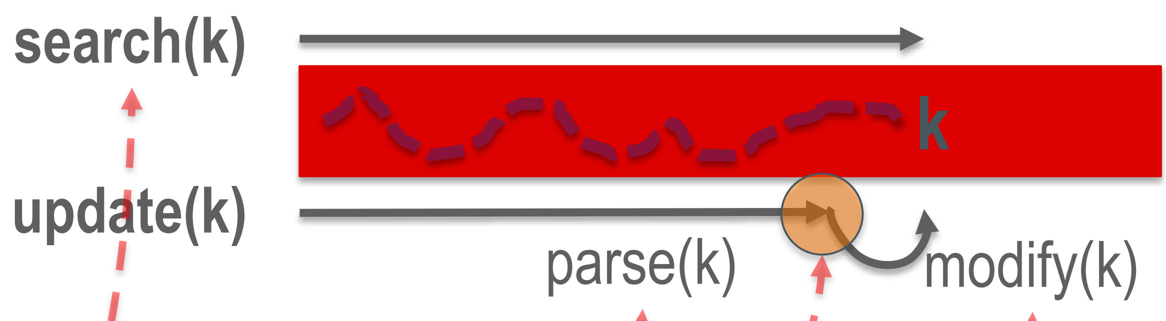
```
struct node
{
  value_t value;
  struct node* next;
};
```

Search Data Structures

- Interface
  1. search
  2. insert
  3. remove

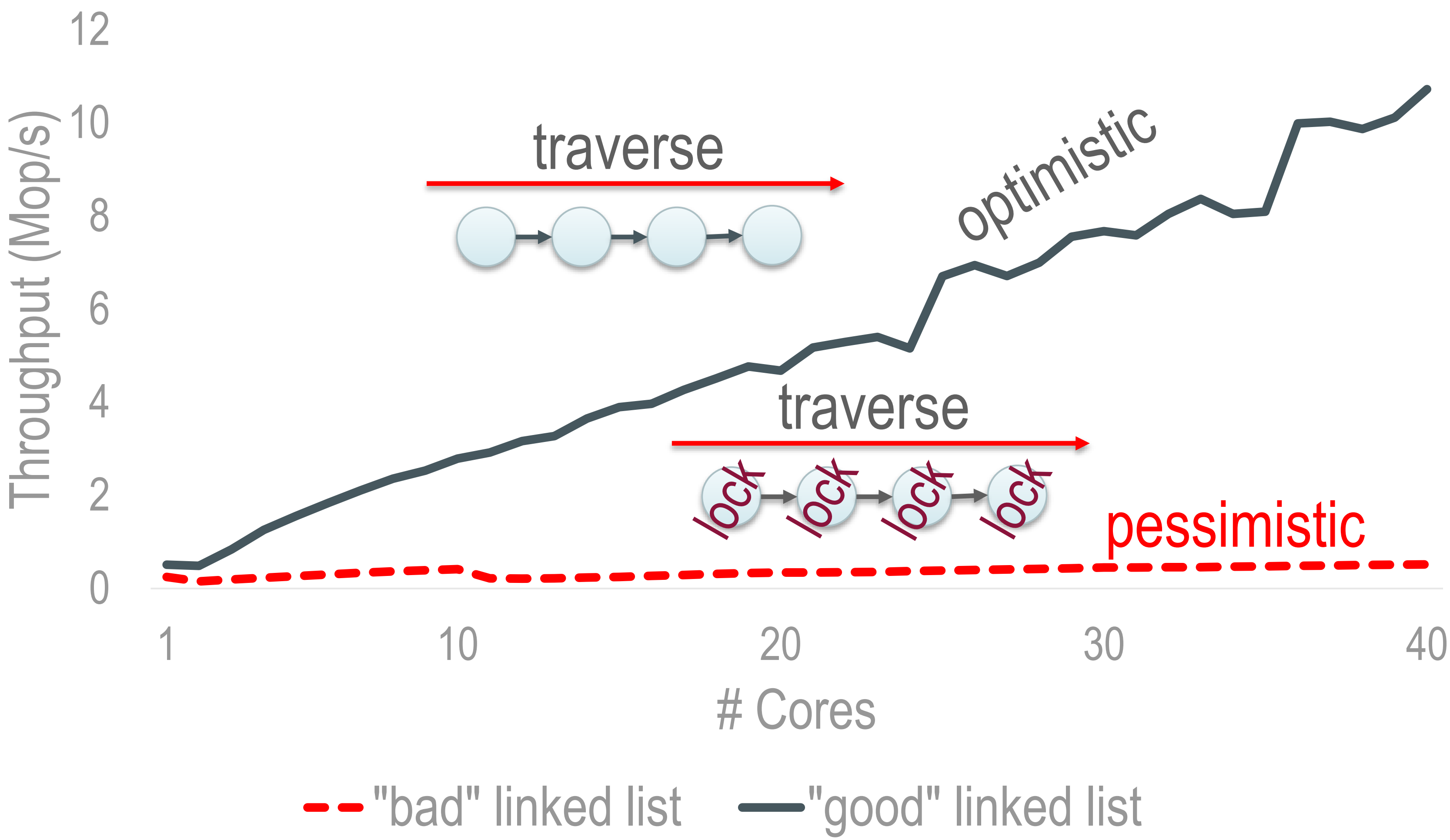
  updates

- Semantics
  1. read-only
  2. read-only
  3. read-only
  4. read-write



Optimistic vs. Pessimistic Concurrency

20-core Xeon
1024 elements



(**Lesson$_1$**) Optimistic concurrency is the only way to get scalability

The Two Problems in Optimistic Concurrency

- **Concurrency Control**
  How threads synchronize their writes to the shared memory (e.g., nodes)
  - Locks
  - CAS
  - Transactional memory

- **Memory Reclamation**
  How and when threads free and reuse the shared memory (e.g., nodes)
  - Garbage collectors
  - Hazard pointers
  - RCU
  - Quiescent states

Tools for Optimistic Concurrency Control (OCC)

- RCU: slow in the presence of updates
  - (also a memory reclamation scheme)
- STM: slow in general
- HTM: not ubiquitous, not very fast (yet)

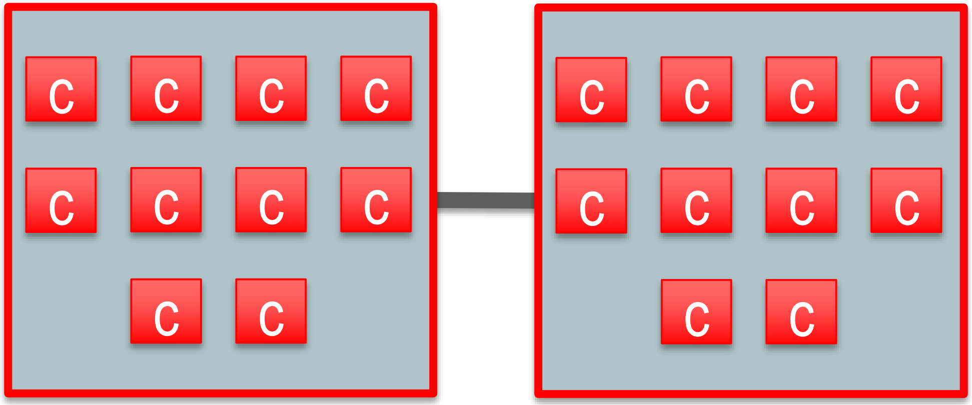
- Wait-free algorithms: slow in general
- (Optimistic) Lock-free algorithms: ☺
- Optimistic lock-based algorithms: ☺

We either need a lock-free or an optimistic lock-based algorithm

Parenthesis: Target platform

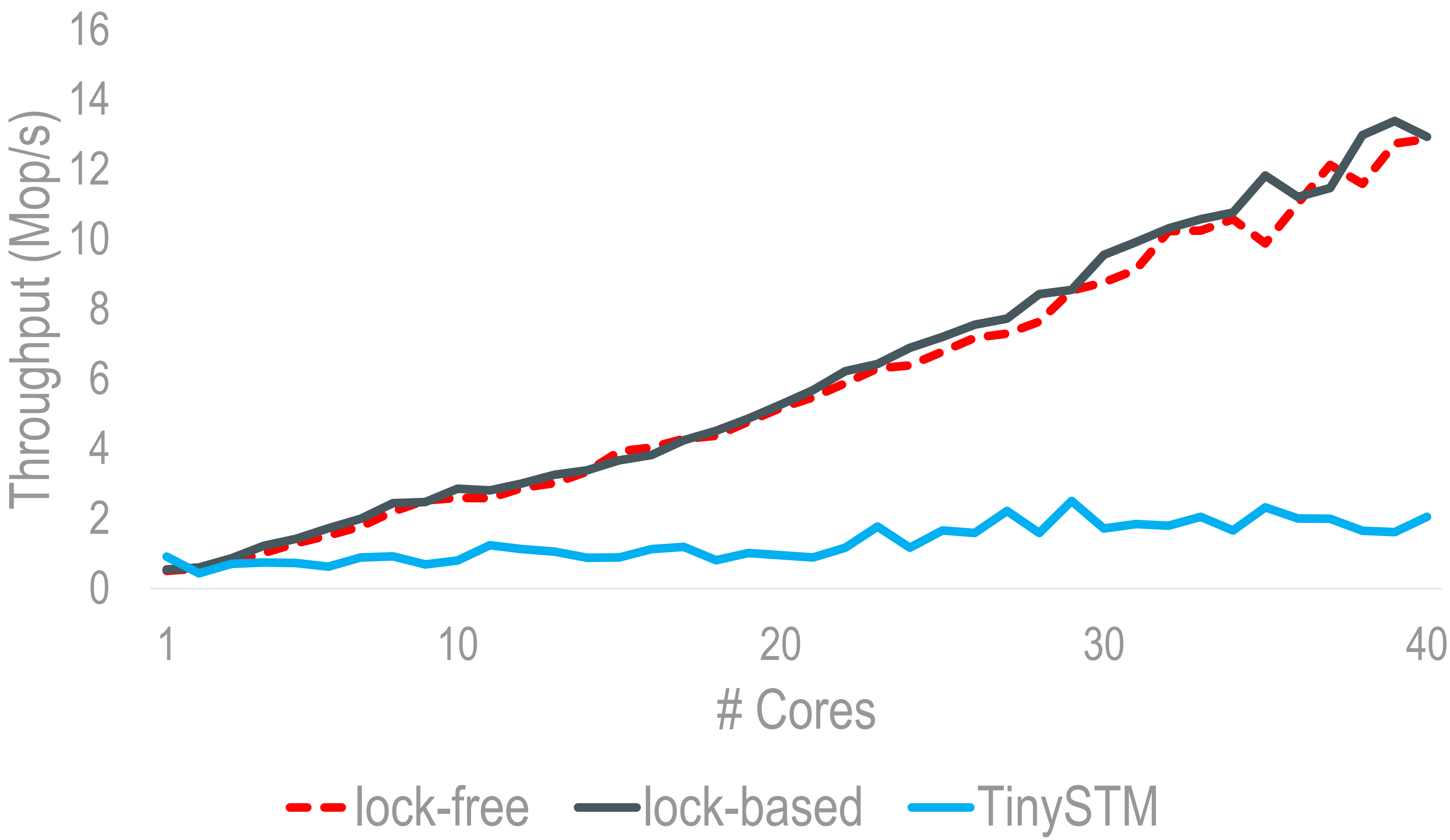
2-socket Intel Xeon E5-2680 v2 Ivy Bridge

- 20 cores @ 2.8 GHz, 40 hyper-threads
- 25 MB LLC (per socket)
- 256GB RAM



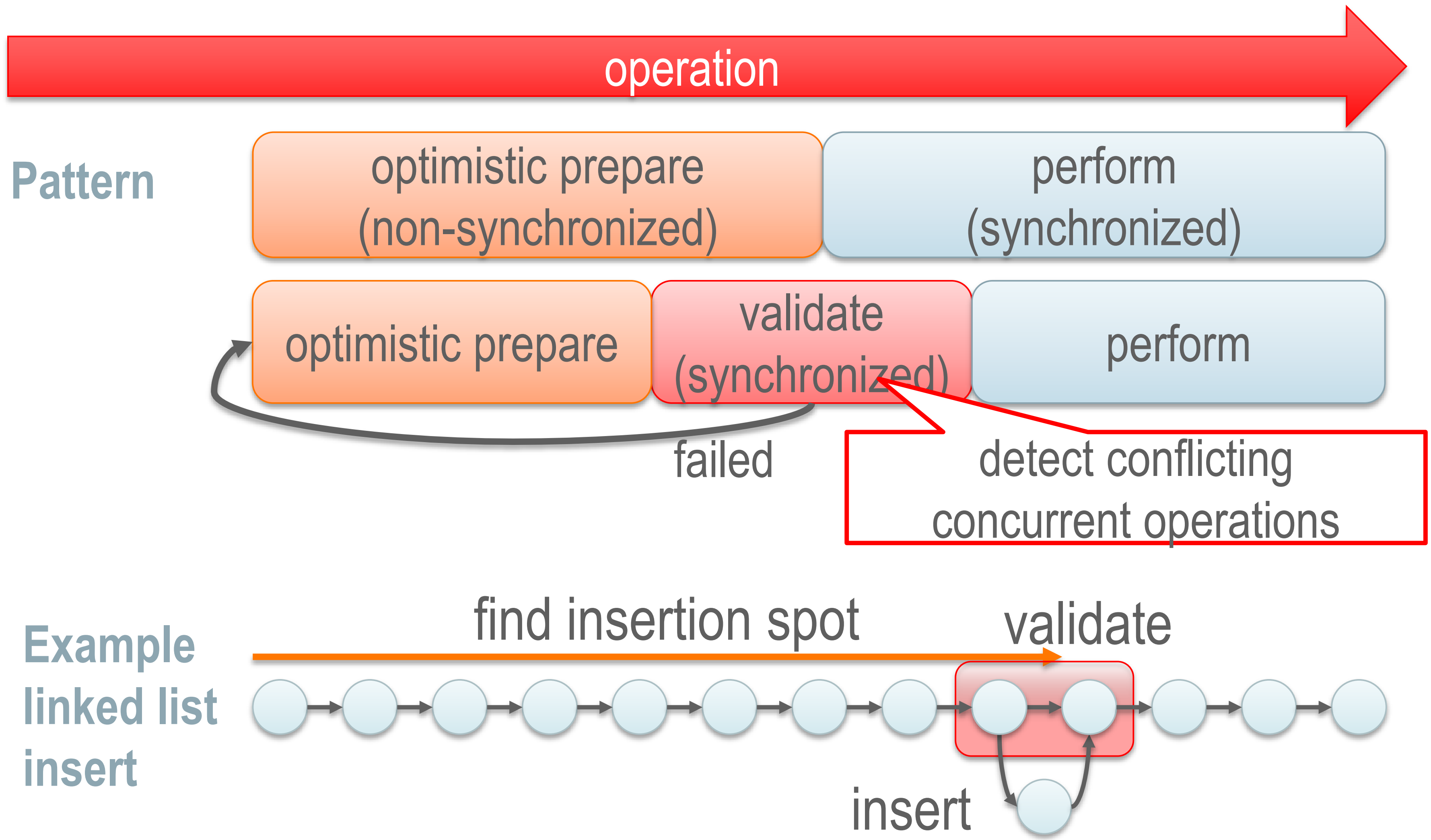
Concurrent Linked Lists – 1% Updates

1024 elements
1% updates



STM is slow and non-scalable ☹

Optimistic Concurrency in Data Structures

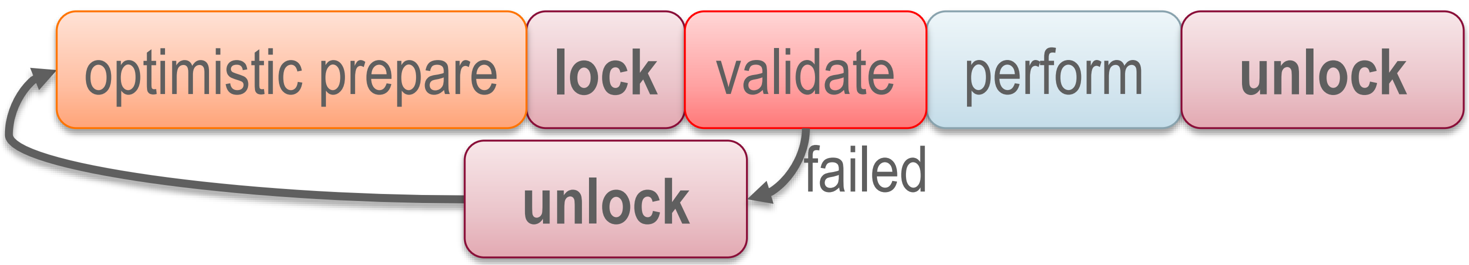


Validation plays a key role in concurrent data structures

Validation in concurrent data structures

- Lock-free: atomic operations

  - marking pointers, flags, helping, …
- Lock-based: lock → validate



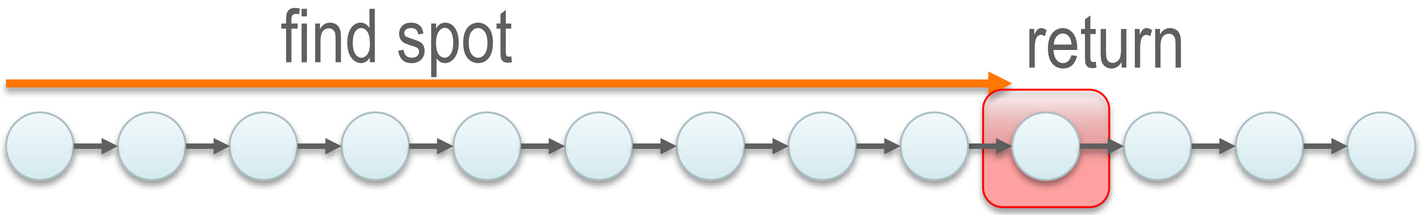
  - flags, pointer reversal, parsing twice, …

Validation is what differentiates algorithms

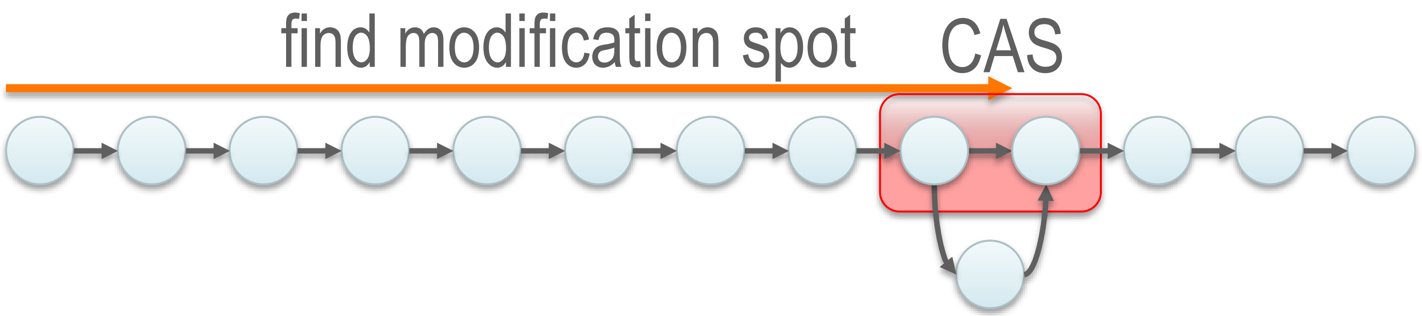
Let's design two concurrent linked lists:
A **lock-free** and a **lock-based**

Lock-free Sorted Linked List: Naïve

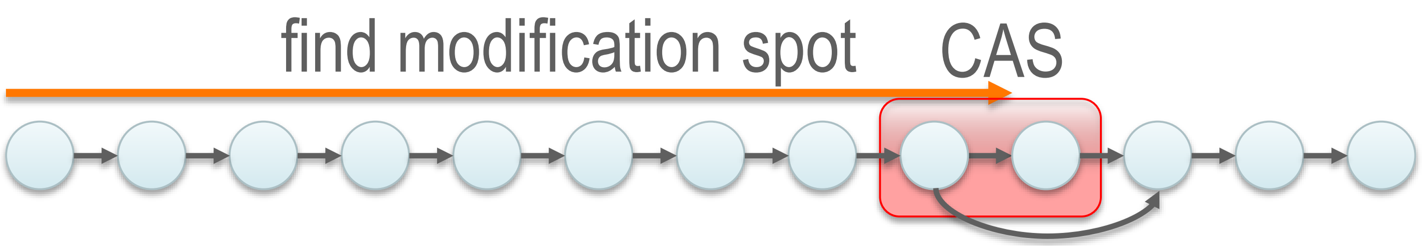
Search



Insert

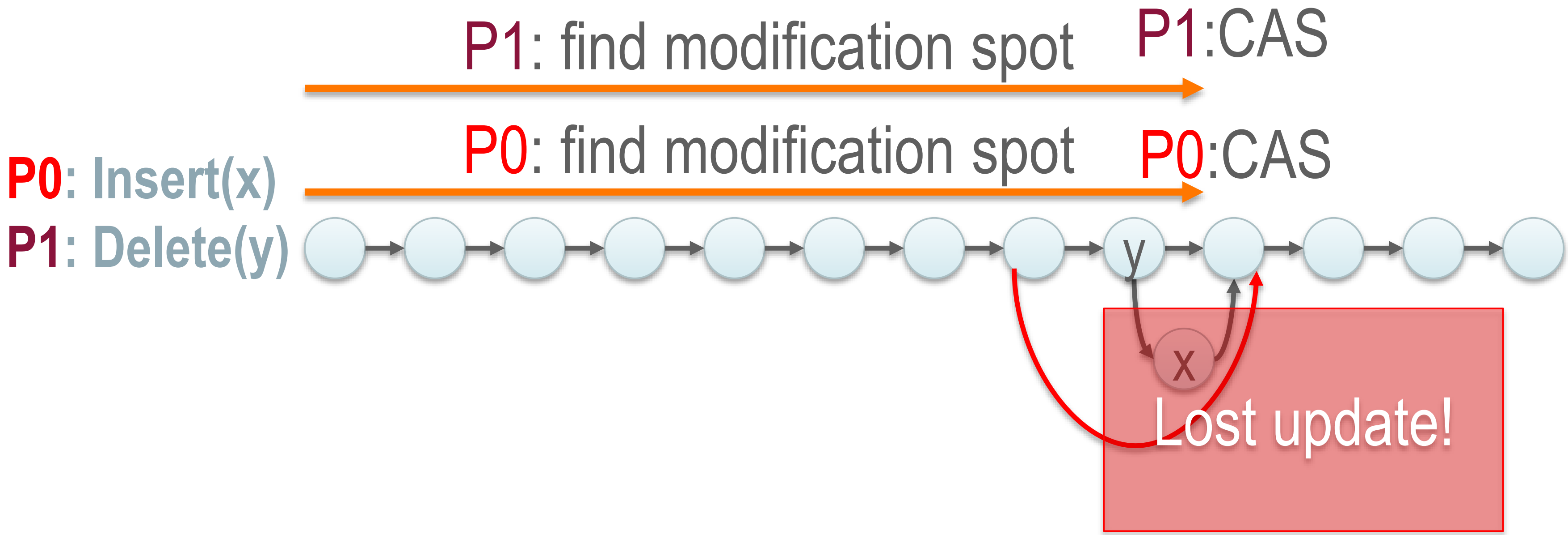


Delete



Is this a correct (linearizable) linked list?

Lock-free Sorted Linked List: Naïve – Incorrect



- **What is the problem?**
  - Insert involves one existing node;
  - Delete involves two existing nodes

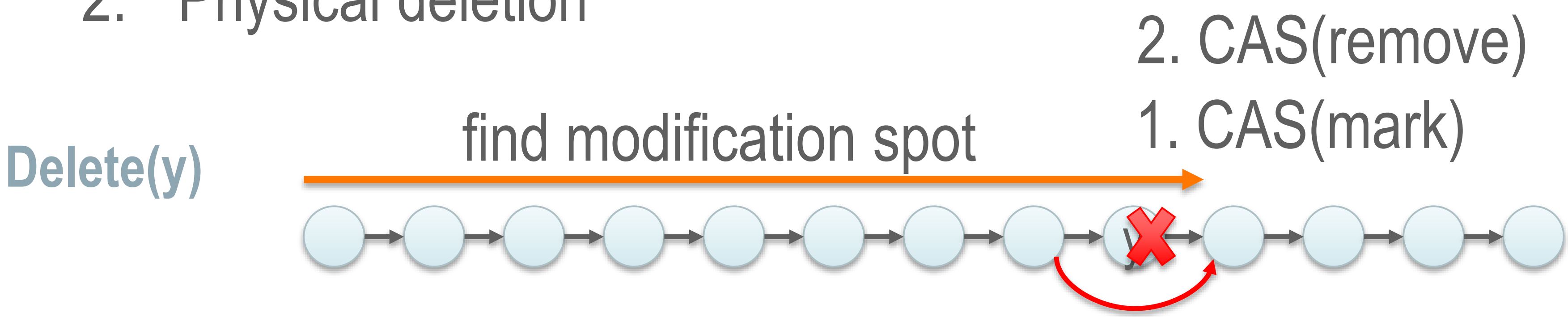
How can we fix the problem?

Lock-free Sorted Linked List: Fix

- **Idea**! To delete a node, make it unusable first…
  - **Mark it for deletion** so that
    1. You fail marking if someone changes next pointer;
    2. An insertion fails if the predecessor node is marked.

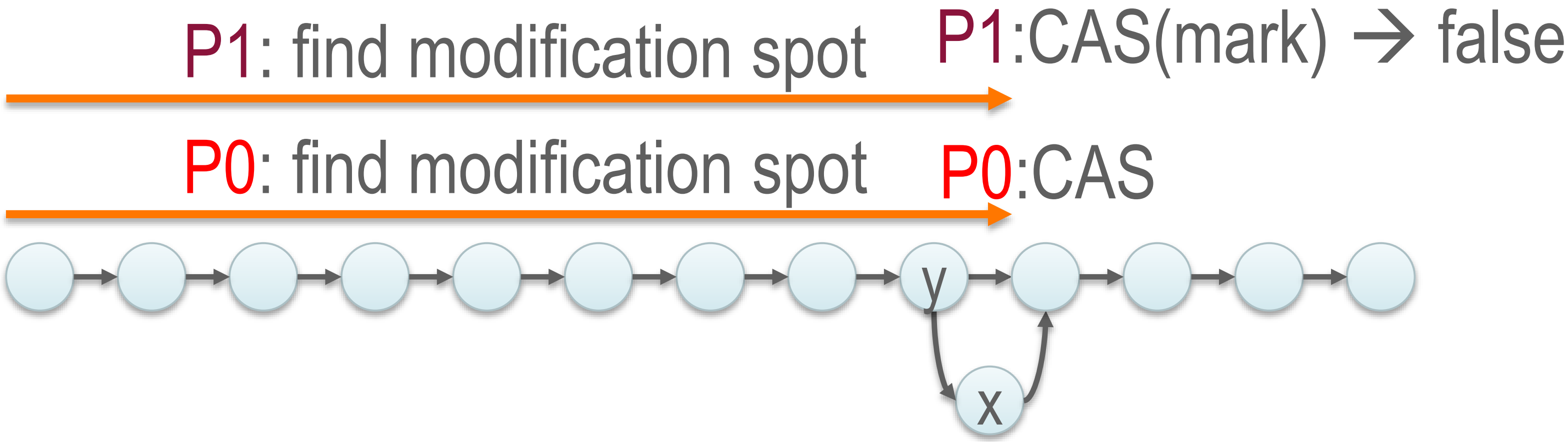
→ In other words: delete in two steps

1. Mark for deletion; and then
2. Physical deletion



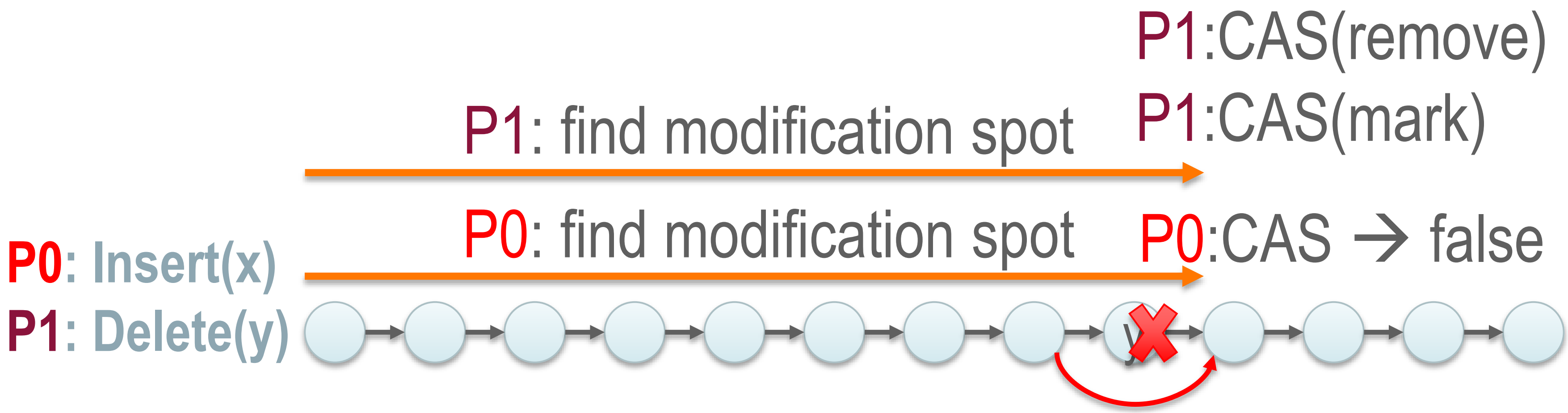
1. Failing Deletion (Marking)

P0: Insert(x)
P1: Delete(y)



- Upon failure → restart the operation
  - Restarting is part of "all" state-of-the-art-data structures

1. Failing Insertion due to Marked Node

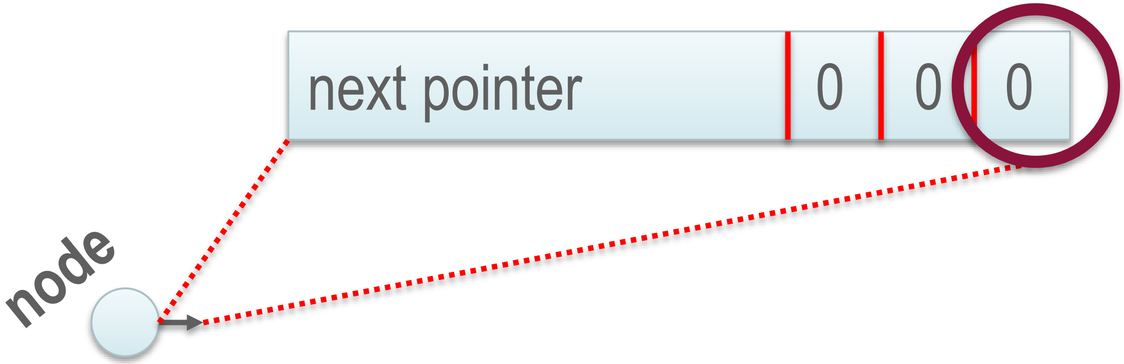


- Upon failure → restart the operation
  - Restarting is part of "all" state-of-the-art-data structures

How can we implement marking?

Implementing Marking (C Style)

- Pointers in 64 bit architectures
  - Word aligned - 8 bit aligned! (!! Remember TM class)




```
boolean mark(node_t* n)
    uintptr_t unmarked = n->next & ~0x1L;
    uintptr_t marked   = n->next | 0x1L;
    return CAS(&n->next, unmarked, marked) == unmarked;
```

Lock-free List: Putting Everything Together

- Traversal: traverse (requires unmarking nodes)
- Search: traverse
- Insert: traverse → CAS to insert
- Delete: traverse → CAS to mark → CAS to remove

What happers
if this CAS
fails??

- **Garbage (marked) nodes**
  – Cleanup while traversing (*helping* in this course's terms)

A pragmatic implementation of lock-free linked lists

What is not Perfect with the Lock-free List?

1. **Garbage nodes**
   - Increase path length; and
   - Increase complexity

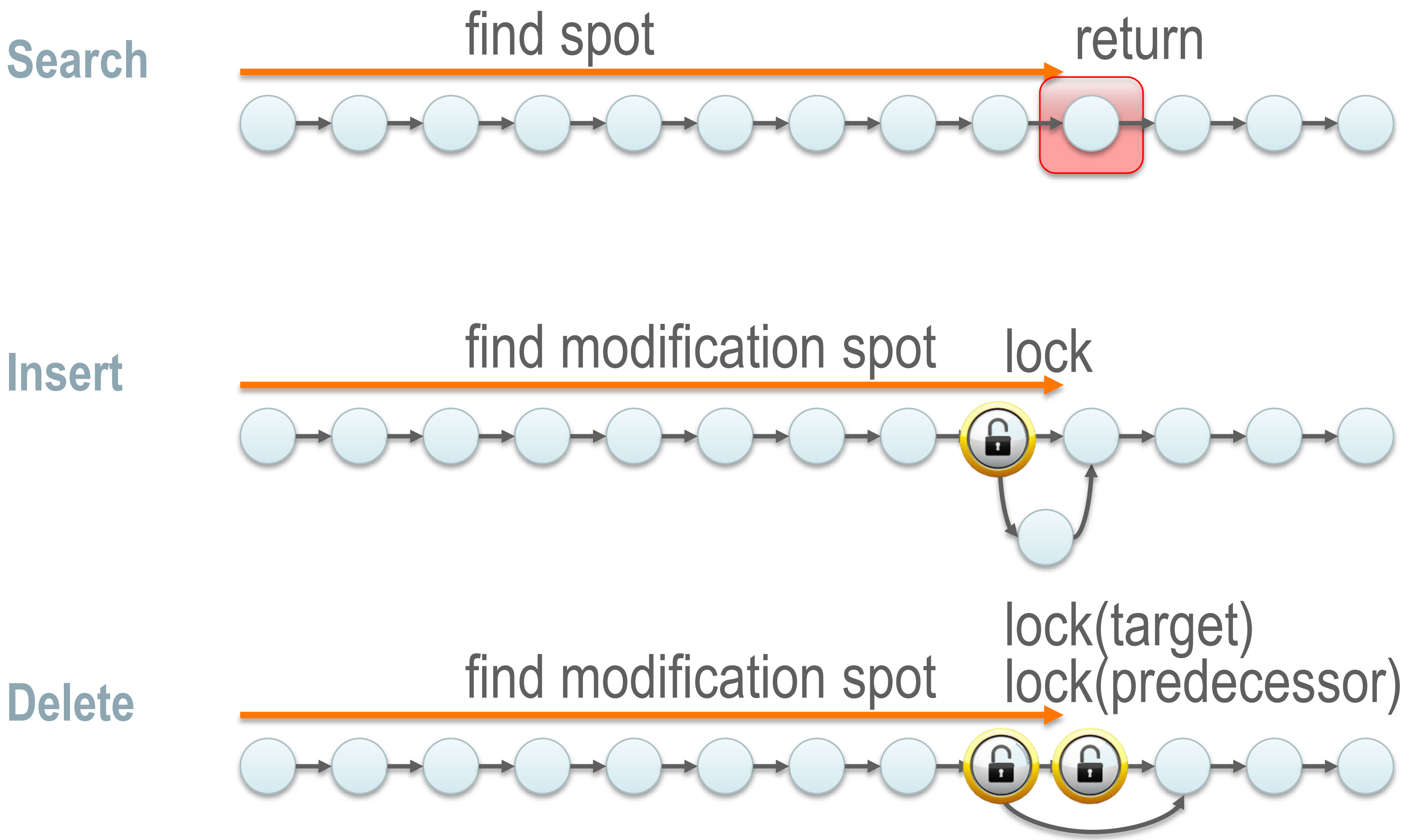
```
if (is_marked_node(n)) ...
```

2. **Unmarking every single pointer**
   - Increase complexity

```
curr = unmark_node(curr->next)
```

Can we simplify the design with locks?

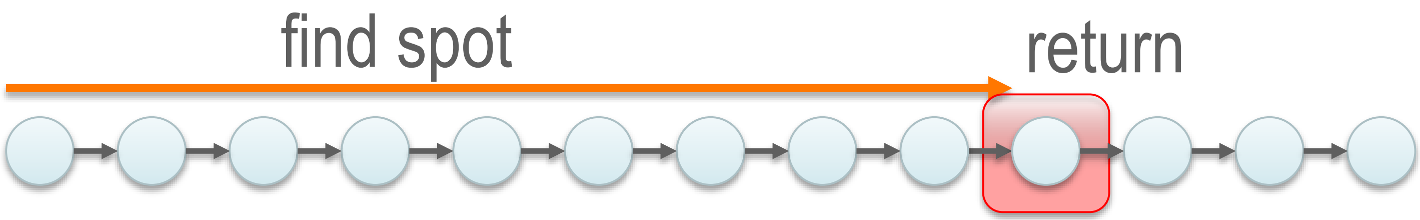
Lock-based Sorted Linked List: Naïve



Is this a correct (linearizable) linked list?

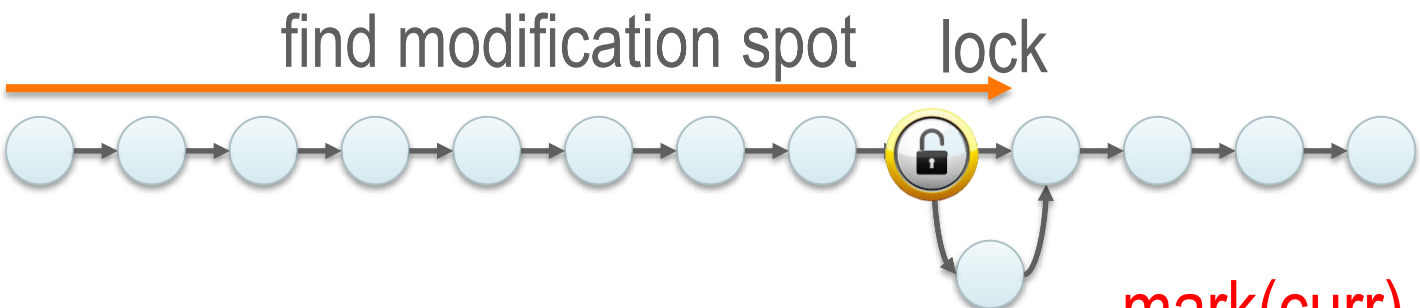
Lock-based List: Validate After Locking

Search

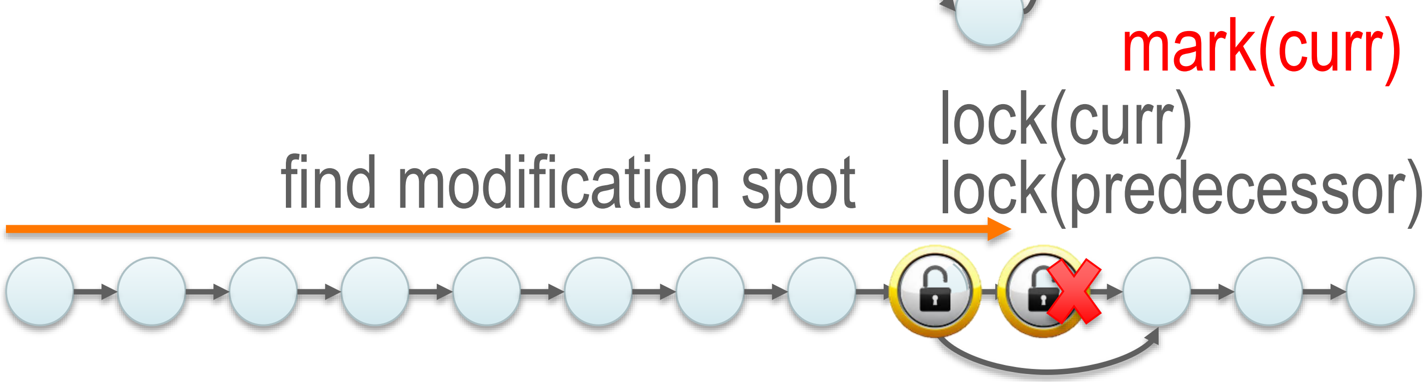


validate !**pred->marked** && **pred->next did not change**

Insert



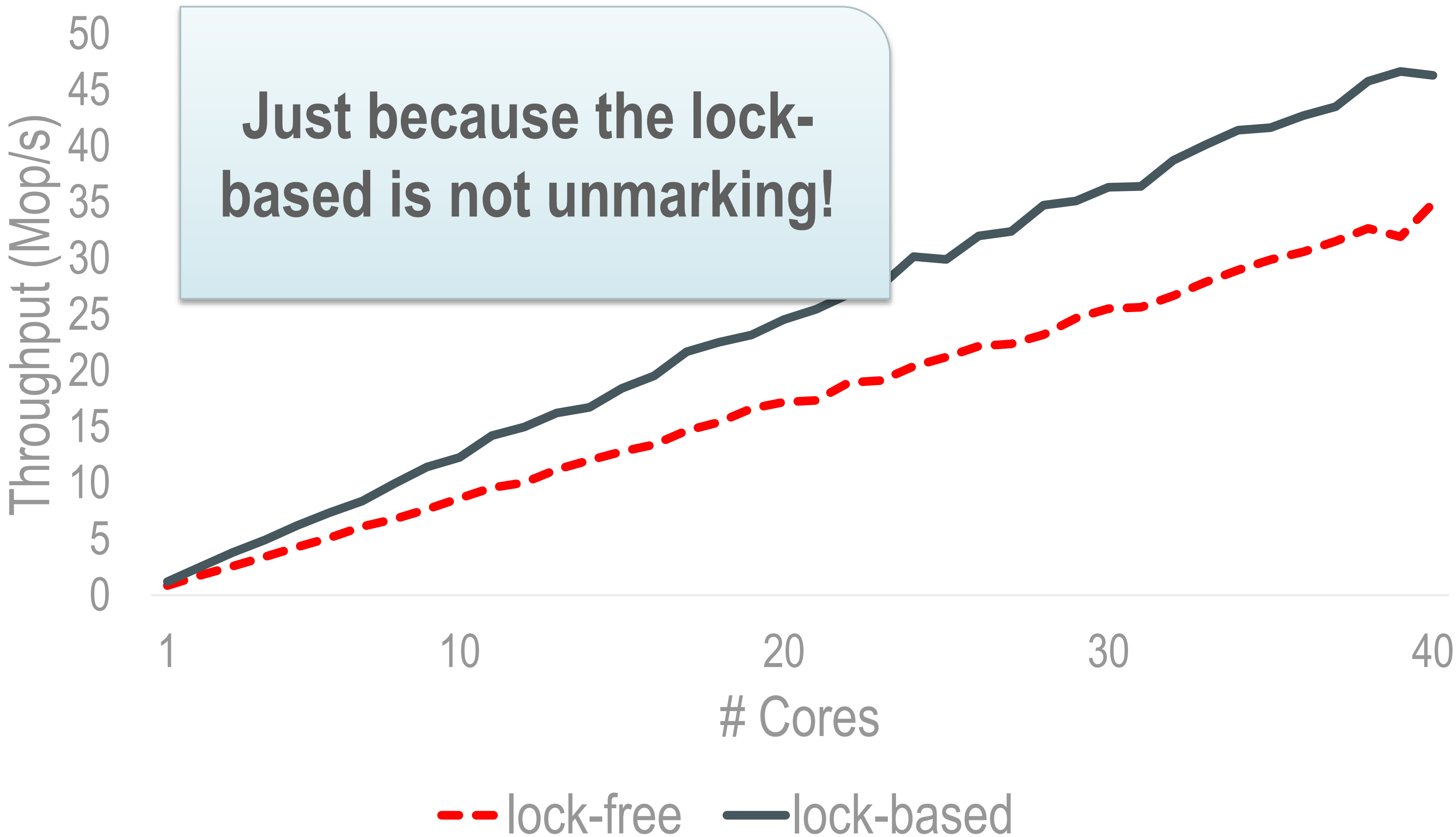
Delete



!pred->marked && **!curr->marked** && **pred->next did not change**

Concurrent Linked Lists – 0% updates

1024 elements
0% updates



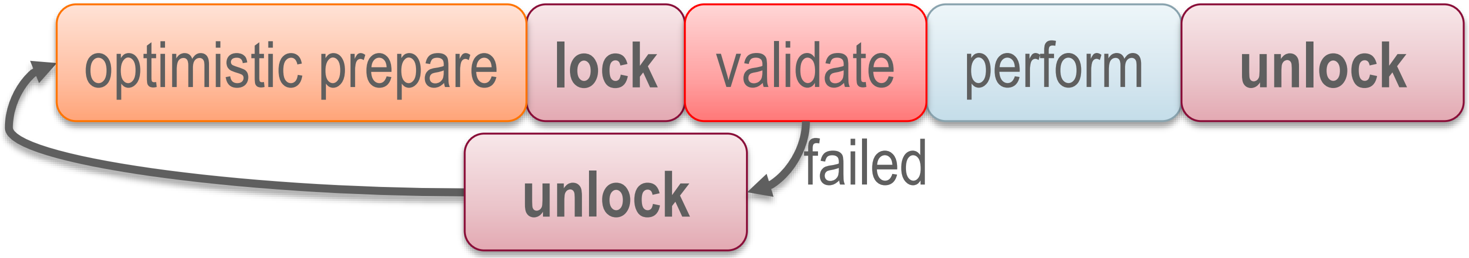
(**Lesson$_2$**) Sequential complexity matters → Simplicity ☺

Optimistic Concurrency Control: Summary

- Lock-free: atomic operations

  – marking pointers, flags, helping, …

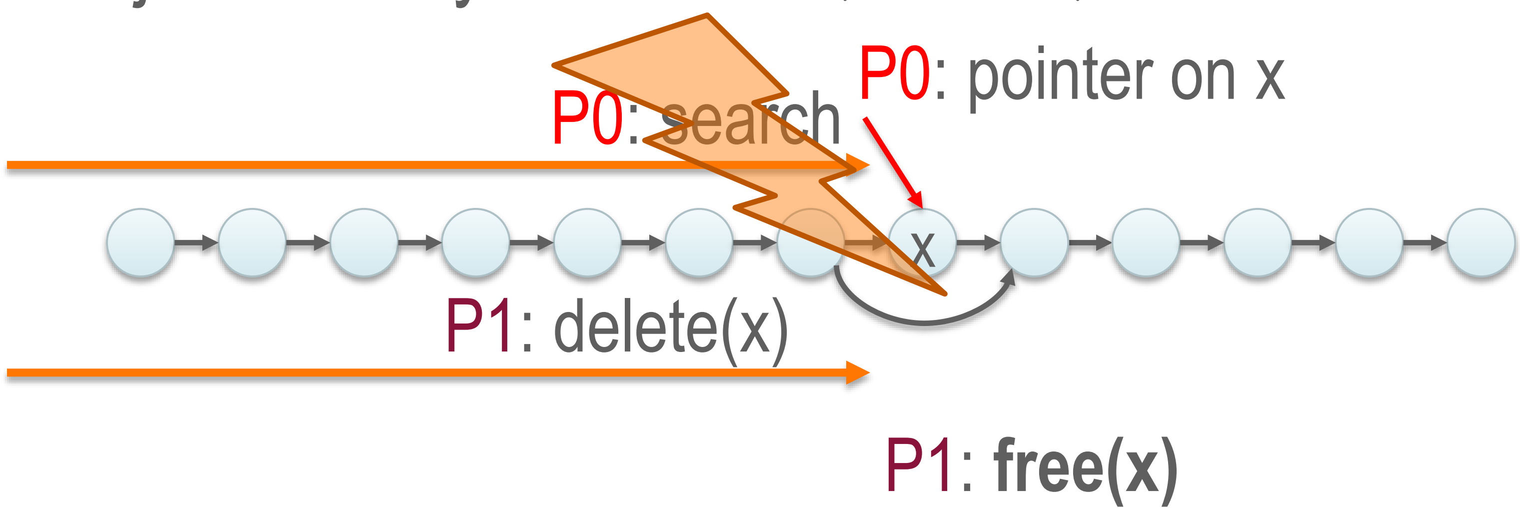
- Lock-based: lock → validate



  – flags, pointer reversal, parsing twice, …

Memory Reclamation: OCC's Side Effect

- Delete a node → free and reuse this memory
- Subset of the garbage collection problem
- Who is accessing that memory?
- Can we just directly do `free(node)`?



We cannot directly free the memory! Need memory reclamation

Memory Reclamation Schemes

1. **Reference counting**
   – Count how many references exist on a node
2. **Hazard pointers**
   – Tell to others what exactly you are reading
3. **Quiescent states**
   – Wait until it is certain than no one holds references
4. **Read-Copy Update (RCU)**
   – Quiescent states – The extreme approach

1. Reference Counting

- Pointer + Counter

rc_pointer

counter	pointer

- Dereference:

```
rc_dereference(rc_pointer* rcp)
    atomic_increment(&rcp->counter);
    return *pointer;
```

- “Release”:

```
rc_release(rc_pointer* rcp)
    atomic_decrement(&rcp->counter);
```

- Free: iff counter = 0

(**Lesson$_3$**) Readers cannot write on the shared nodes. Why?

Bad bad bad idea: Readers write on shared nodes!

2. Hazard pointers (1/2)

- Reference counter → property of the node
- Hazard pointer → property of the thread
  - A Multi-Reader Single-Writer (MRSW) register

hazard_pointer

address

- Protect:

```
hp_protect(node* n)
    hazard_pointer* hp = hp_get(n);
    hp->address = n;
```

Depends on the data structure type

- Release:

```
hp_release(hazard_pointer* hp)
    hp->address = NULL;
```

2. Hazard pointers (2/2)

- Free memory **x**
  1. Collect all hazard pointers
  2. Check if **x** is accessed by any thread
     1. If yes, buffer the free for later
     2. If not, free the memory
- Buffering the free is implementation specific

hazard_pointer

address

- + lock-free
- - not scalable

O(data structure size) hazard pointers `hp_protect`

3. Quiescent States

- Keep the memory until it is certain it is not accessed
- Can be implemented in various ways
- Example implementation

```
search / insert / delete
 qs_unsafe();   → I'm accessing shared data
 …
 qs_safe();     → I'm not accessing shared data
 return …
```

The data written in `qs_[un]safe` must be local-mostly

3. Quiescent States: qs_[un]safe Implementation

- List of "thread-local" (mostly) counters

(id = 0) qs_state	→	(id = x) qs_state	→	(id = y) qs_state

- **qs_state** (initialized to 0)
  - even : in safe mode (not accessing shared data)
  - odd : in unsafe mode
- `qs_safe / qs_unsafe`

```
qs_state++;
```

How do we free memory?

3. Quiescent States: Freeing memory

- List of "thread-local" (mostly) counters

(id = 0) qs_state → (id = x) qs_state → (id = y) qs_state

- Upon `qs_free`: Timestamp memory (vector_ts)
  - Can do this for batches of frees
- Safe to reuse the memory

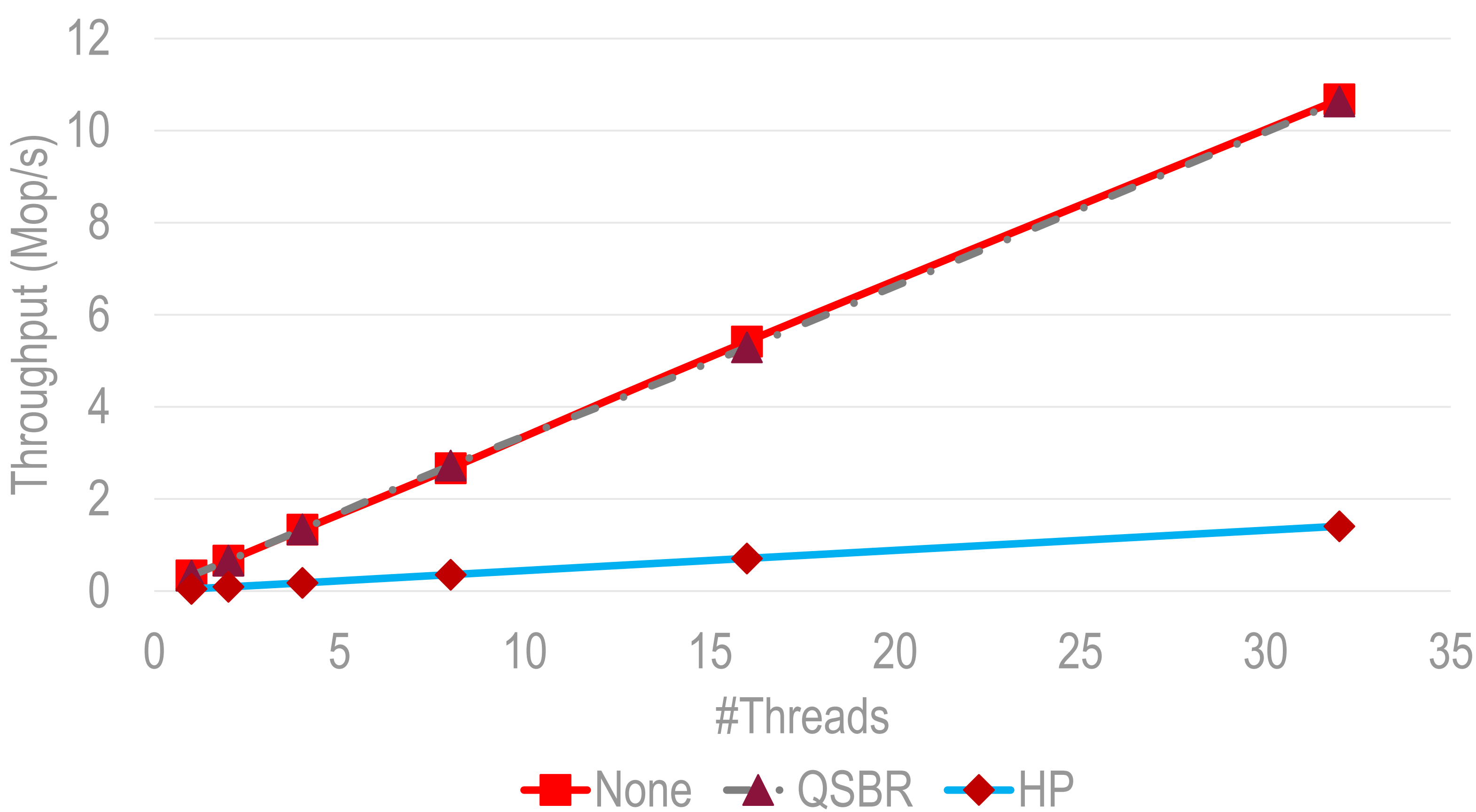
$vector\_ts_{now}$ >> $vector\_ts_{mem}$

```
for t in thread_ids
  if (vts_mem[t] is odd &&
    vts_now[t] = vts_mem[t])
    return false;
return true;
```

How do the schemes we have seen perform?

Hazard Pointers vs. Quiescent States

1024 elements
0% updates



Quiescent-state reclamation is as fast as it gets

4. Read-Copy Update (RCU)

- Quiescent states at their extreme
  - Deletions wait all readers to reach a safe state
- Introduced in the Linux kernel in ~2002
  - More than 10000 uses in the kernel!
- (Example) Interface
  - `rcu_read_lock` (= `qs_unsafe`)
  - `rcu_read_unlock` (= `qs_safe`)
  - `synchronize_rcu` → wait all readers

4. Using RCU

- **Search / Traverse**

```
rcu_read_lock()
…
rcu_read_unlock()
```

- **Delete**

… physical deletion of **x**

```
synchronize_rcu()
free(x)
```

- + simple
- + read-only workloads
- - bad for writes

Memory Reclamation: Summary

- **How and when to reuse freed memory**
- Many techniques, no silver bullet
  1. Reference counting
  2. Hazard pointers
  3. Quiescent states
  4. Read-Copy Update (RCU)

Summary

- **Concurrent data structures are very important**
- **Optimistic concurrency necessary for scalability**
  - Only recently a lot of active work for CDSs
- **Memory reclamation is**
  - Inherent to optimistic concurrency;
  - A difficult problem;
  - A potential performance/scalability bottleneck

(If time permits) Let's design one of the lists (array maps) of Java together!