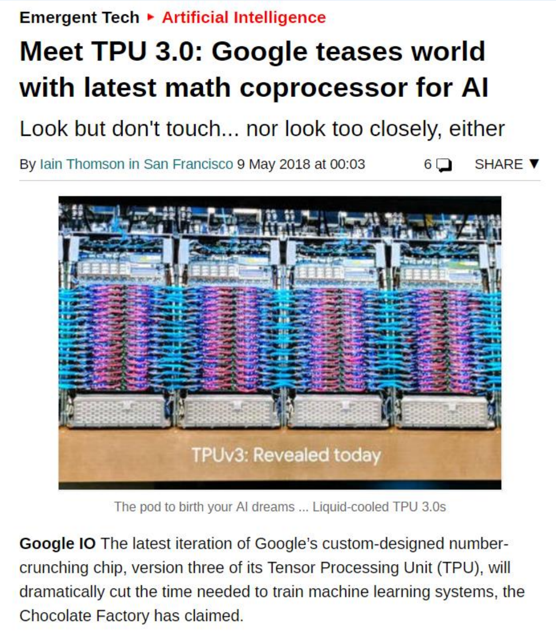
TORSTEN HOEFLER

Parallel Programming

Without locks II



Emergent Tech ▸ Artificial Intelligence

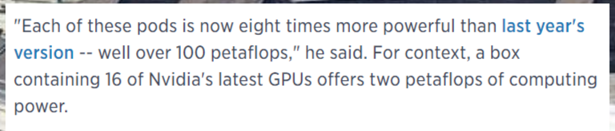
Meet TPU 3.0: Google teases world with latest math coprocessor for AI

Look but don't touch... nor look too closely, either

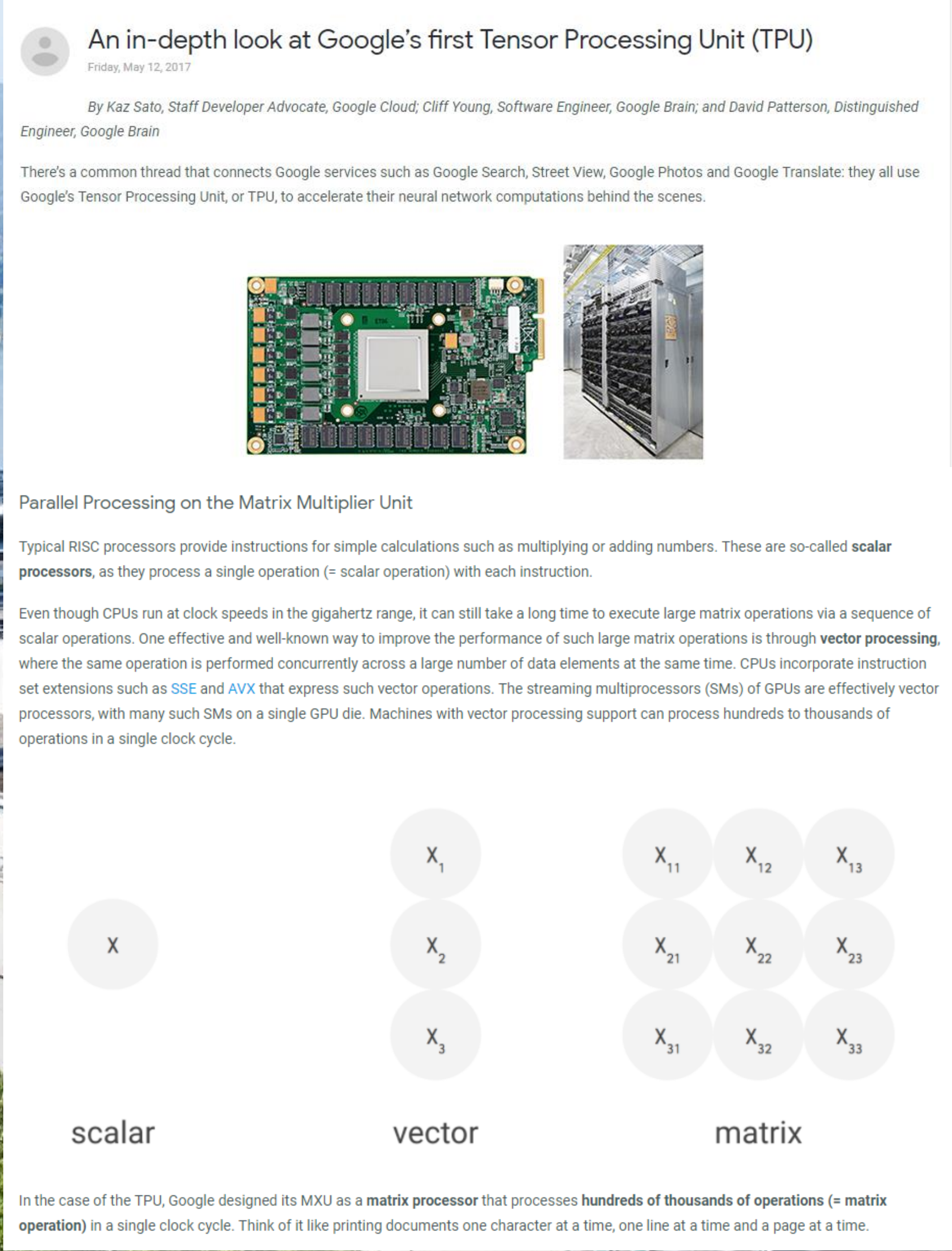
By Iain Thomson in San Francisco 9 May 2018 at 00:03 6 SHARE ▼

The pod to birth your AI dreams ... Liquid-cooled TPU 3.0s

Google IO The latest iteration of Google's custom-designed number-crunching chip, version three of its Tensor Processing Unit (TPU), will dramatically cut the time needed to train machine learning systems, the Chocolate Factory has claimed.



"Each of these pods is now eight times more powerful than last year's version -- well over 100 petaflops," he said. For context, a box containing 16 of Nvidia's latest GPUs offers two petaflops of computing power.



An in-depth look at Google's first Tensor Processing Unit (TPU)

Friday, May 12, 2017

By Kaz Sato, Staff Developer Advocate, Google Cloud; Cliff Young, Software Engineer, Google Brain; and David Patterson, Distinguished Engineer, Google Brain

There's a common thread that connects Google services such as Google Search, Street View, Google Photos and Google Translate: they all use Google's Tensor Processing Unit, or TPU, to accelerate their neural network computations behind the scenes.

Parallel Processing on the Matrix Multiplier Unit

Typical RISC processors provide instructions for simple calculations such as multiplying or adding numbers. These are so-called **scalar processors**, as they process a single operation (= scalar operation) with each instruction.

Even though CPUs run at clock speeds in the gigahertz range, it can still take a long time to execute large matrix operations via a sequence of scalar operations. One effective and well-known way to improve the performance of such large matrix operations is through **vector processing**, where the same operation is performed concurrently across a large number of data elements at the same time. CPUs incorporate instruction set extensions such as SSE and AVX that express such vector operations. The streaming multiprocessors (SMs) of GPUs are effectively vector processors, with many such SMs on a single GPU die. Machines with vector processing support can process hundreds to thousands of operations in a single clock cycle.

In the case of the TPU, Google designed its MXU as a **matrix processor** that processes **hundreds of thousands of operations (= matrix operation)** in a single clock cycle. Think of it like printing documents one character at a time, one line at a time and a page at a time.

Last week

- **Lock tricks on the list-based set example**
    - Fine-grained locking
    - Optimistic locking
    - Lazy (removal) locking

- **Skip lists**
    - Example for probabilistic parallel performance – conflict reduction

- **Lock-free programming**
    - Reminder of atomics (CAS)
    - Non-blocking counter

Learning goals today

- **Lock-free**
  - Stack
  - List

- **Unbounded Queues**
  - More complex example for lock-free, how to design a datastructure

- **Memory Reuse and the ABA Problem**
  - Understand one of the most complex pitfalls in shared memory parallel programming

Literature:
Herlihy: Chapter 10

Non-blocking counter

Deadlock/Starvation?

```
public class CasCounter {
    private AtomicInteger value;

    public int getVal() {
       return value.get();
    }

    // increment and return new value
    public int inc() {
       int v;
       do {
             v = value.get();
       } while (!value.compareAndSet(v, v+1));
       return v+1;
    }
}
```

What happens if some processes see the same value?

Assume one thread dies.
Does this affect other threads?

Mechanism

(a) read current value v

(b) modify value v'

(c) try to set with CAS

(d) return if success
restart at (a) otherwise

Positive result of CAS of (c) *suggests* that no other thread has written between (a) and (c)

Why not “guarantees”?

Handle CAS with care

Positive result of CAS *suggests* that no other thread has written

It is not always true, as we will find out (→ ABA problem).

However, it is still THE mechanism to check for exclusive access in lock-free programming.

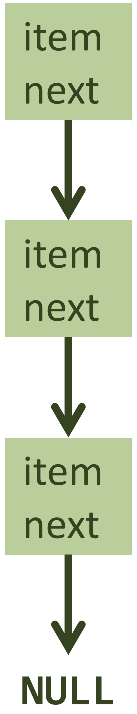
Sidenotes:

- **maybe transactional memory will become competitive at some point**
- **LL/SC or variants thereof may give stronger semantics avoiding ABA**

Lock-Free Stack

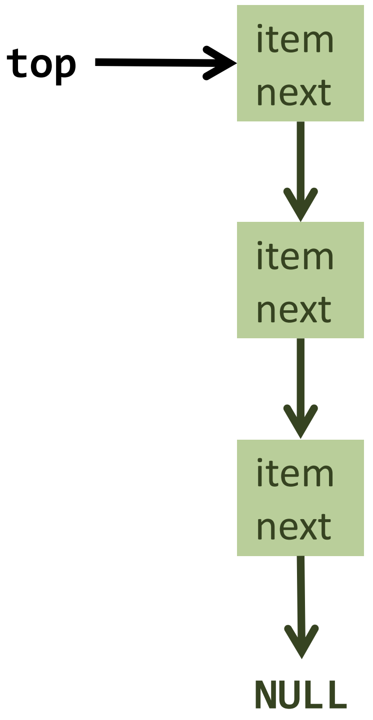
Stack Node

```
public static class Node {
   public final Long item;
   public Node next;

   public Node(Long item) {
       this.item = item;
   }

   public Node(Long item, Node n) {
       this.item = item;
       next = n;
   }

}
```



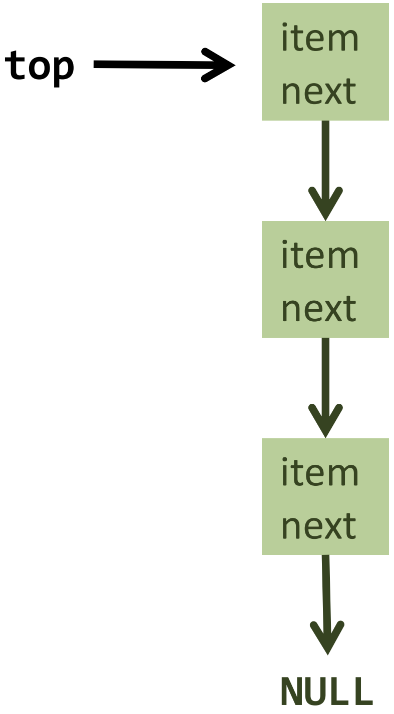
Blocking Stack

```
public class BlockingStack {
   Node top = null;

   synchronized public void push(Long item) {
      top = new Node(item, top);
   }

   synchronized public Long pop() {
      if (top == null)
         return null;
      Long item = top.item;
      top = top.next;
      return item;
   }
}
```



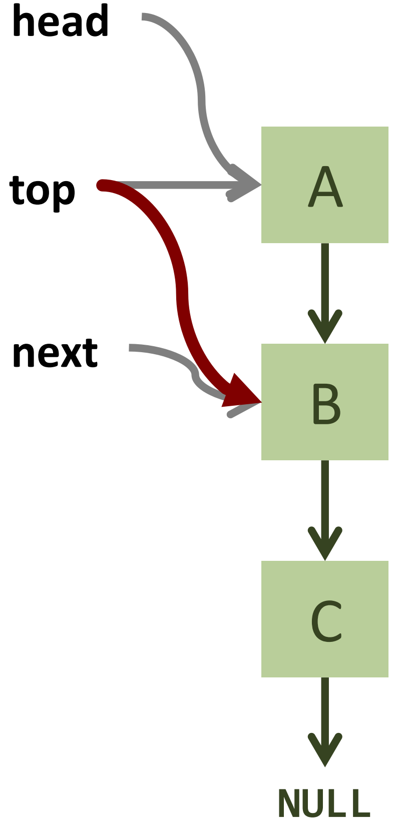
Non-blocking Stack

```
public class ConcurrentStack {
   AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```



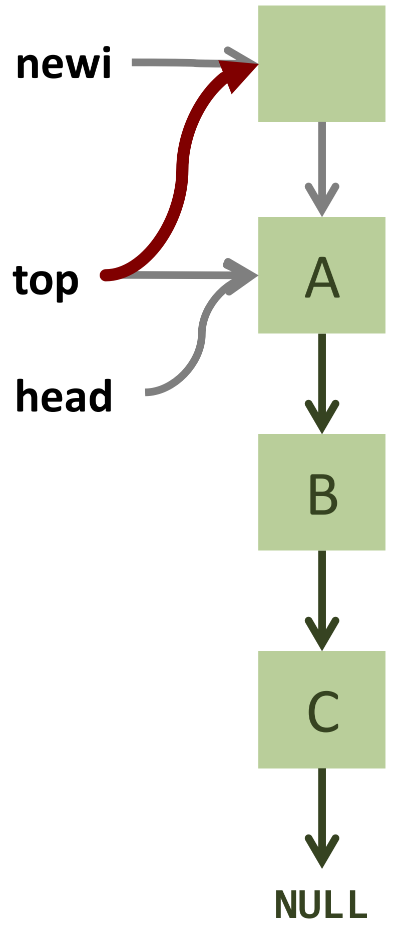
Pop

```
public Long pop() {
   Node head, next;

   do {
      head = top.get();
      if (head == null) return null;
      next = head.next;
   } while (!top.compareAndSet(head, next));

   return head.item;
}
```



Push

```
public void push(Long item) {
        Node newi = new Node(item);
        Node head;

        do {
                head = top.get();
                newi.next = head;
        } while (!top.compareAndSet(head, newi));
}
```



What's the benefit?

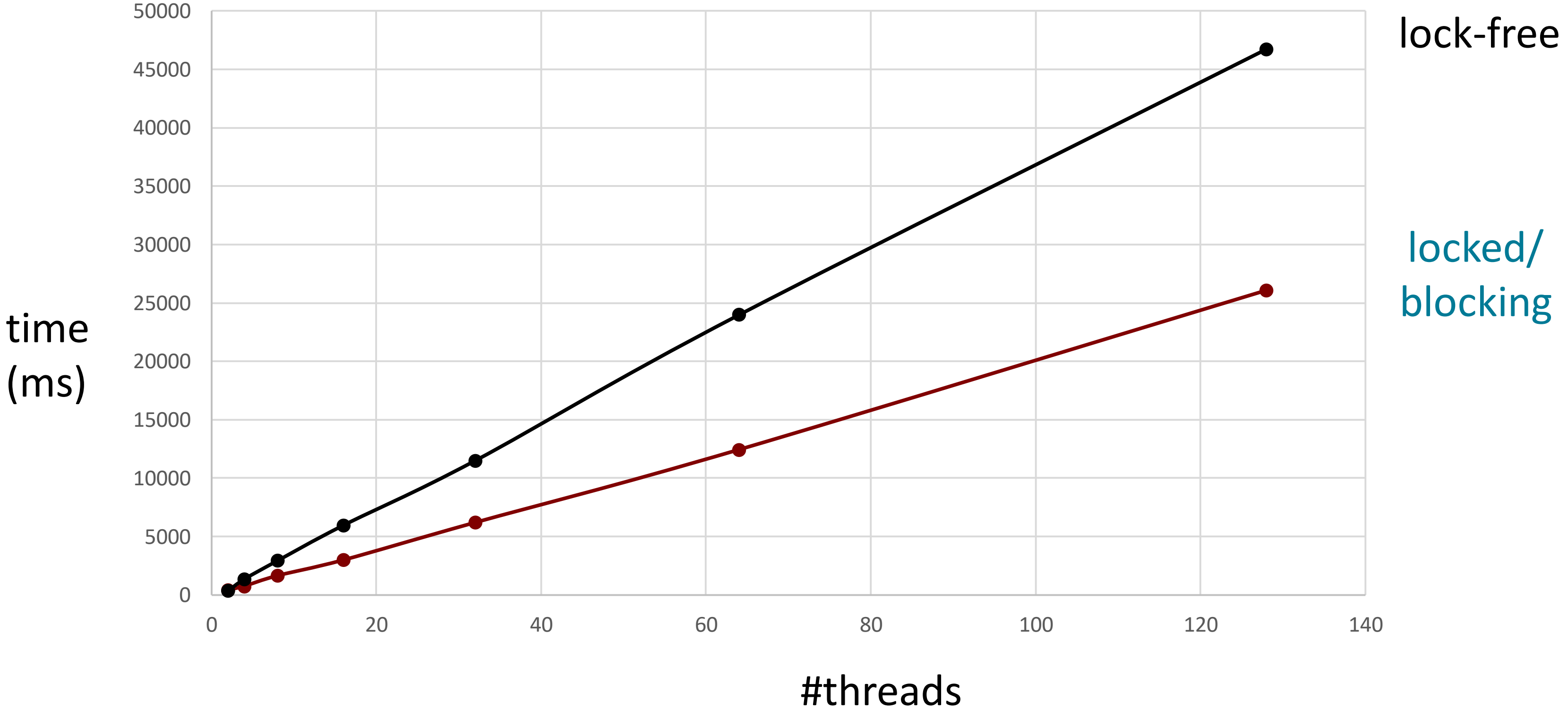
Lock-free programs are deadlock-free by design.

How about performance?

n threads

100,000 push/pop operations

10 times

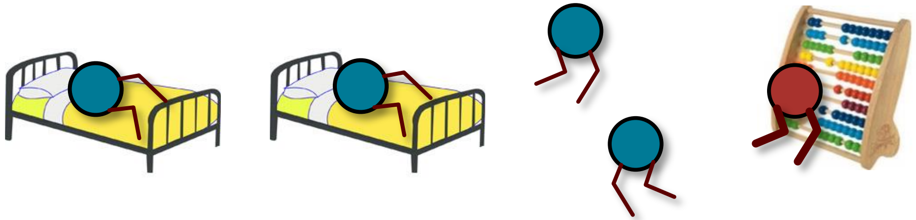


Performance

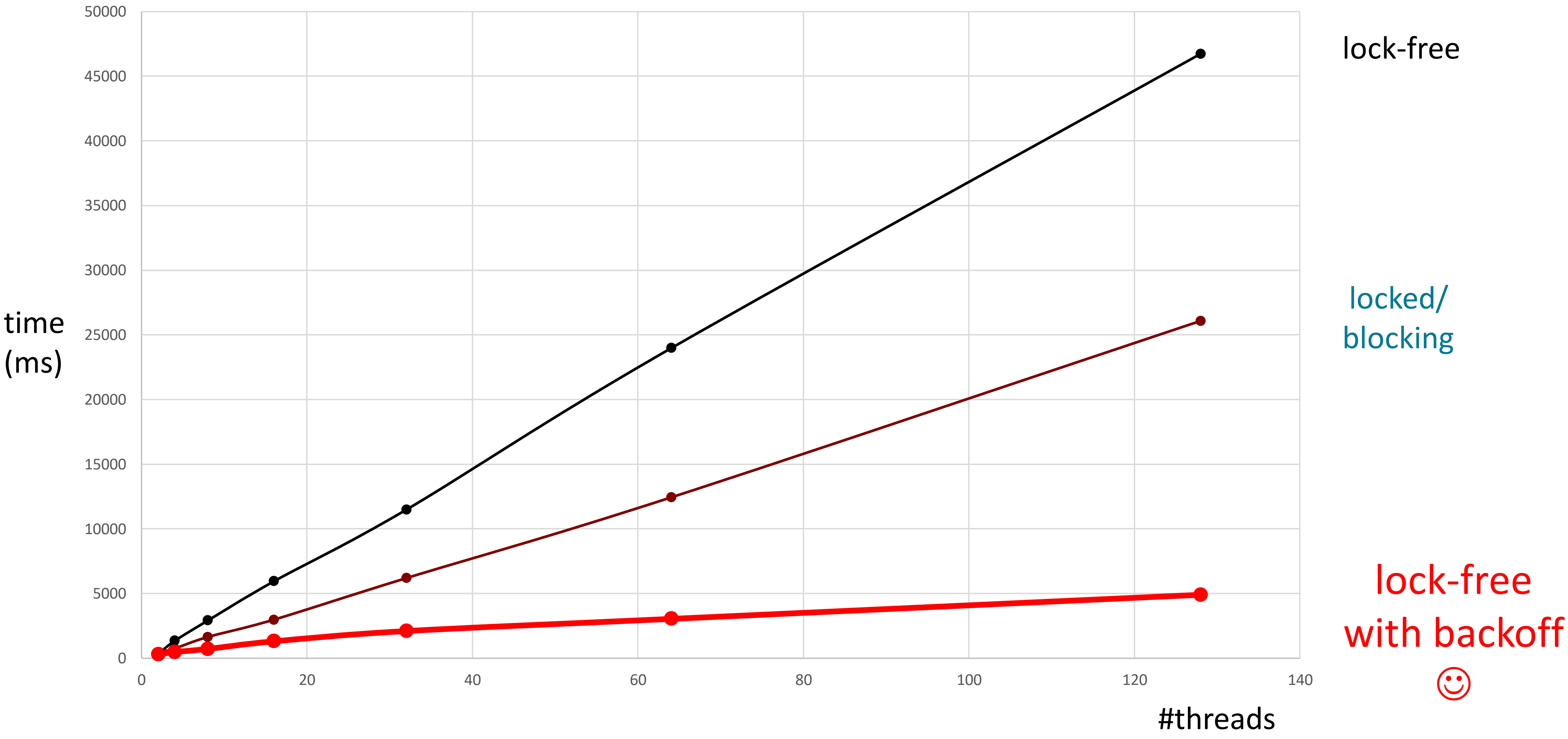
A lock-free algorithm does not automatically provide better performance than its blocking equivalent!

Atomic operations are expensive and contention can still be a problem.

→ Backoff, again.



With backoff



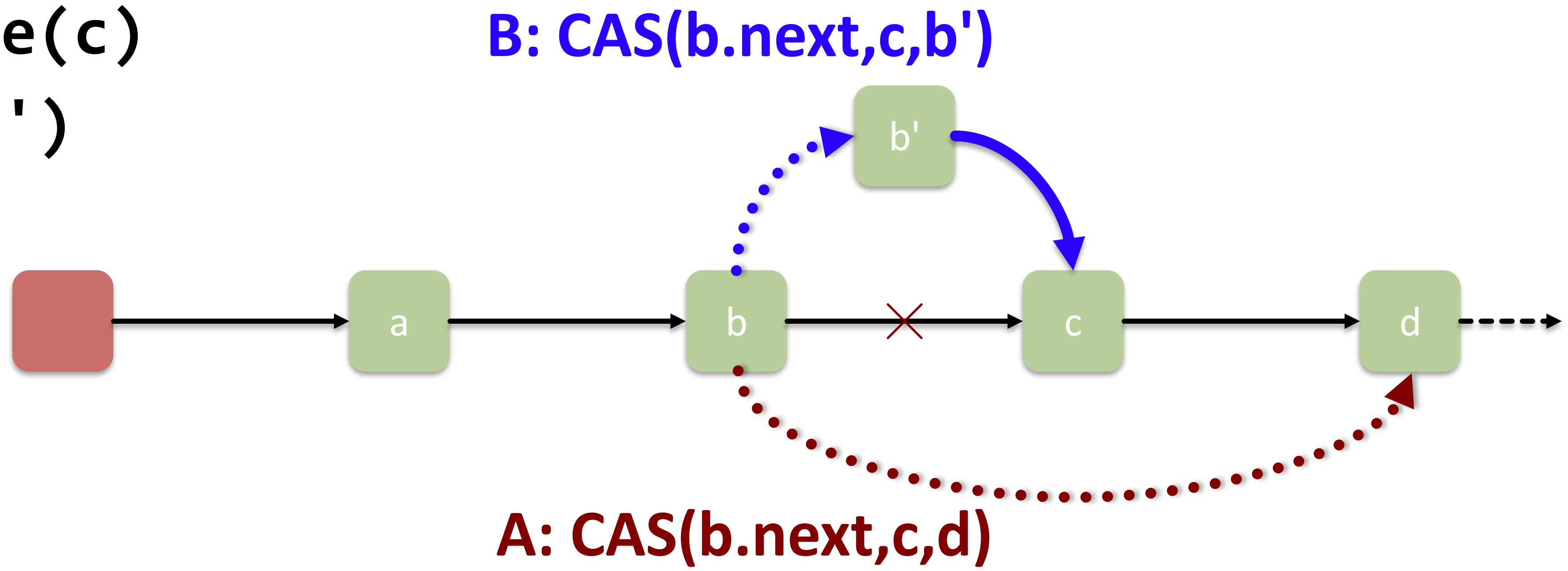
LOCK FREE LIST SET

(NOT SKIP LIST!)

Some of the material from "Herlihy: Art of Multiprocessor Programming"

Does this work?

```
A:  remove(c)
B:  add(b')
```



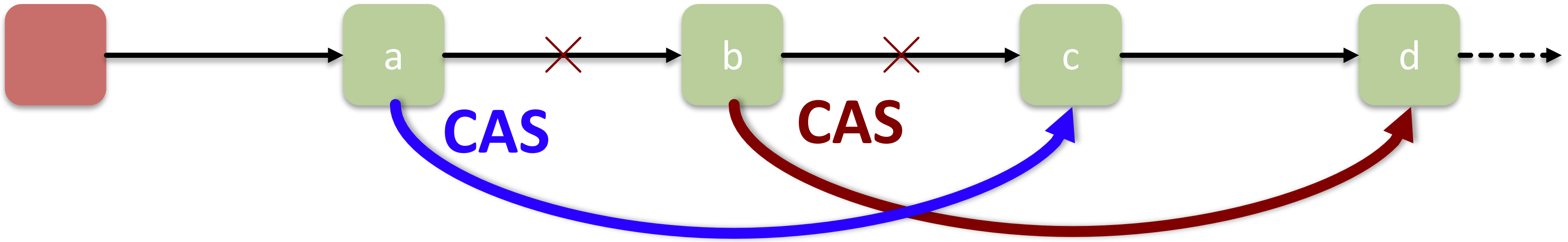
`ok?`

So does this CAS approach work generally??

CAS decides who wins → this seems to work

Another scenario

```
A:  remove(c)
B:  remove(b)
```

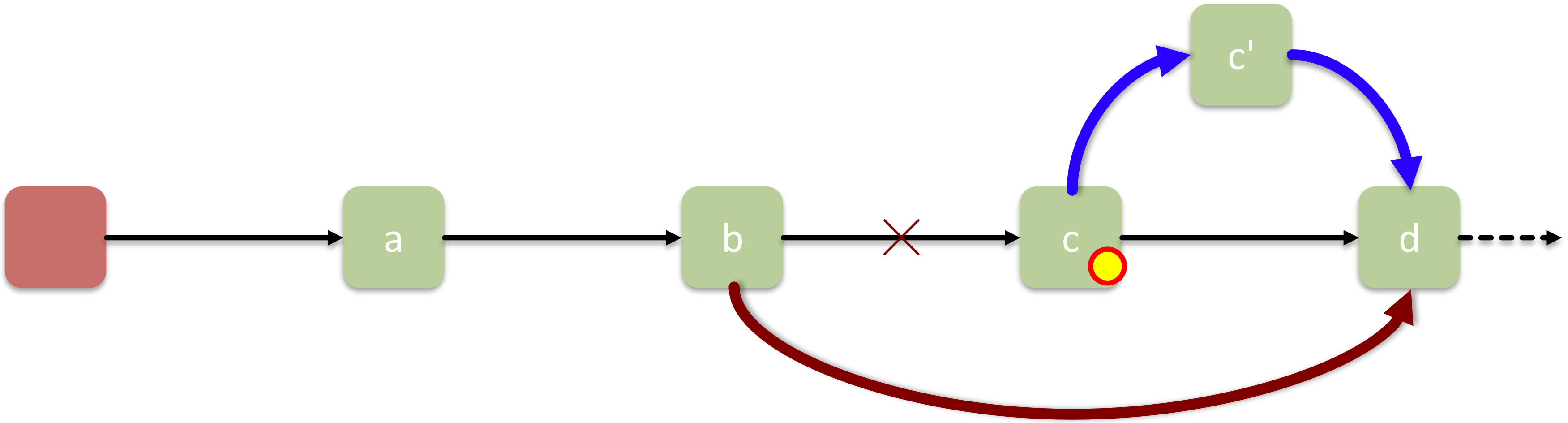


c not deleted! ☹

Mark bit approach?

`A: remove(c)`

`B: add(c')`



B: c.mark ?

B: CAS(c.next,d,c')

A: CAS(c.mark,false,true)

A: CAS(b.next,c,d)

c' not added! ☹

The problem

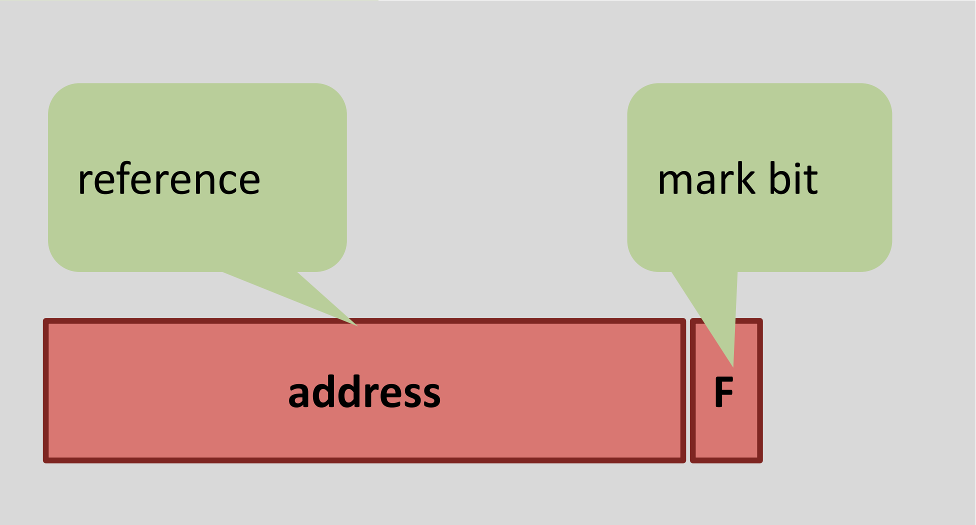
The difficulty that arises in this and many other problems is:

- We cannot (or don't want to) use synchronization via locks
- We still want to atomically establish consistency **of two things**
  Here: mark bit & next-pointer

The Java solution

```
Java.util.concurrent.atomic
AtomicMarkableReference<V> {
    boolean attemptMark(V expectedReference, boolean newMark)
    boolean compareAndSet(V expectedReference, V newReference,
                          boolean expectedMark, boolean newMark)
    V get(boolean[] markHolder)
    V getReference()
    boolean isMarked()
    set(V newReference, boolean newMark)
}
```

DCAS on V and mark



2^{64}Bytes=562,949,953,421,312 Petabytes

The algorithm using AtomicMarkableReference

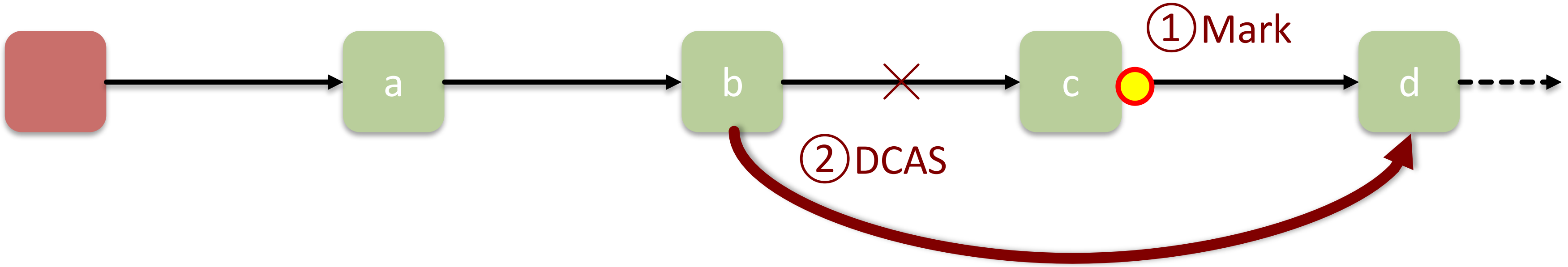
- **Atomically**
  - Swing reference *and*
  - Update flag
- **Remove in two steps**
  - Set mark bit in next field
  - Redirect predecessor’s pointer

Algorithm idea

`A: remove(c)`

Why "try to"? How can it fail? What then?

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,unmarked], [d,unmarked]);

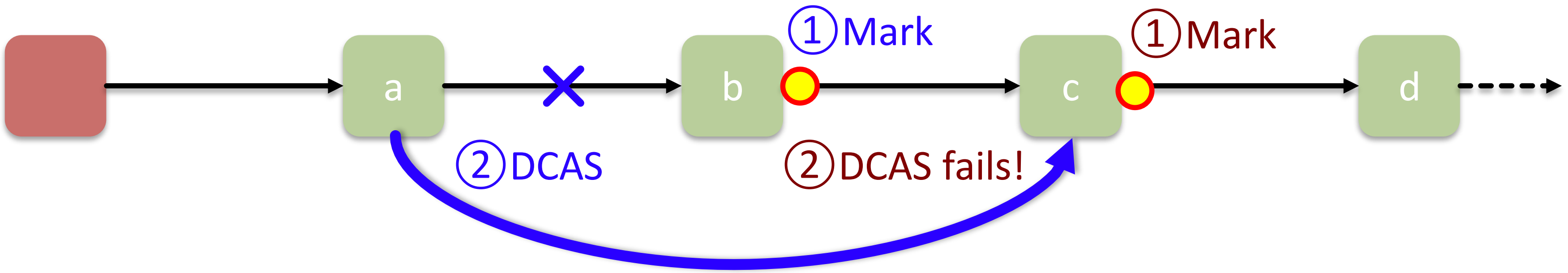


It helps!

`A: remove(c)`

`B: remove(b)`

1. try to set mark (c.next)
2. try CAS(
   [b.next.reference, b.next.marked],
   [c,**unmarked**], [d,unmarked]);



c remains marked ☹ (logically deleted)

1. try to set mark (b.next)
2. try CAS(
   [a.next.reference, a.next.marked],
   [b,unmarked], [c,unmarked]);

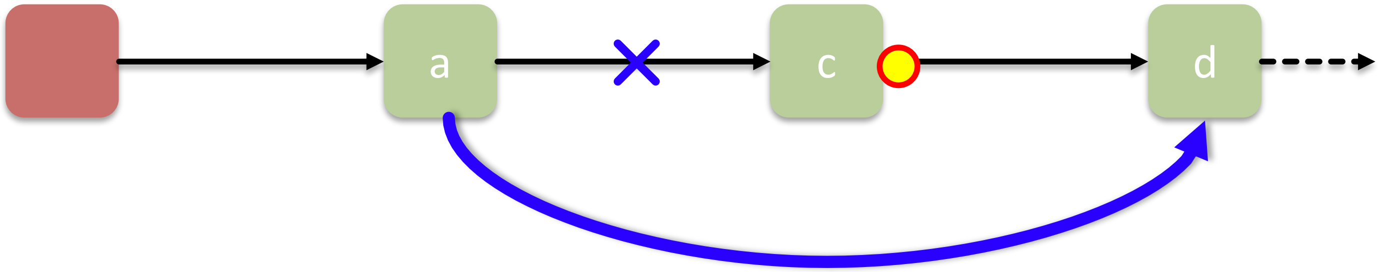
Traversing the list

Q: what do you do when you find a “logically” deleted node in your path?

A: finish the job.

CAS the predecessor’s next field

Proceed (repeat as needed)



Find node

```
public Window find(Node head, int key) {
    Node pred = null, curr = null, succ = null;
    boolean[] marked = {false}; boolean snip;
    while (true) {
        pred = head;
        curr = pred.next.getReference();
        boolean done = false;
        while (!done) {
            marked = curr.next.get(marked);
            succ = marked[1:n]; // pseudo-code to get next ptr
            while (marked[0] && !done) { // marked[0] is marked bit
                if pred.next.compareAndSet(curr, succ, false, false) {
                    curr = succ;
                    succ = curr.next.get(marked);
                }
                else done = true;
            }
            if (!done && curr.key >= key)
                return new Window(pred, curr);
            pred = curr;
            curr = succ;
} } }
```

loop over nodes until position found

if marked nodes are found, delete them, if deletion fails restart from the beginning

```
class Window {
    public Node pred;
    public Node curr;
    Window(Node pred, Node curr) {
        this.pred = pred;
        this.curr = curr;
    }
}
```

Remove

```
public boolean remove(T item) {
   Boolean snip;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key != key) {
         return false;
      } else {
         Node succ = curr.next.getReference();
         snip = curr.next.attemptMark(succ, true);
         if (!snip) continue;
         pred.next.compareAndSet(curr, succ, false, false);
         return true;
      }
   }
}
```

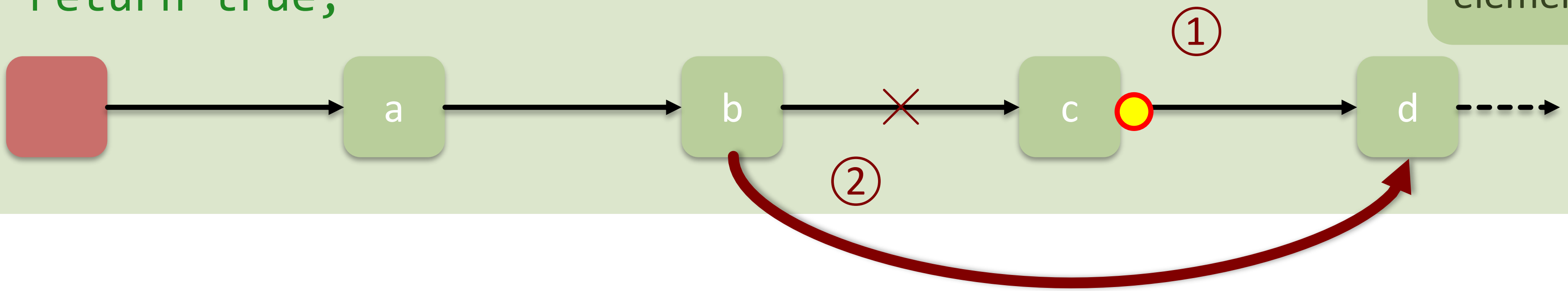
Find element and prev element from key

If no such element -> return false

Otherwise try to logically delete (set mark bit). ①

If no success, restart from the very beginning

Try to physically delete the element, ignore result ②



Add

```
public boolean add(T item) {
   boolean splice;
   while (true) {
      Window window = find(head, key);
      Node pred = window.pred, curr = window.curr;
      if (curr.key == key) {
         return false;
      } else {
         Node node = new Node(item);
         node.next = new AtomicMarkableRef(curr, false);
         if (pred.next.compareAndSet(curr, node, false, false))
            return true;
      }
   }
}
```

Find element and prev element from key

If element already exists, return false

Otherwise create new node, set next / mark bit of the element to be inserted

and try to insert. If insertion fails (next set by other thread or mark bit set), retry

Observations

- We used a special variant of DCAS (double compare and swap) in order to be able check two conditions at once.
  This DCAS was possible because one bit was free in the reference.
- We used a lazy operation in order to deal with a consistency problem. Any thread is able to repair the inconsistency.
  If other threads would have had to wait for one thread to cleanup the inconsistency, the approach would not have been lock-free!
- This «helping» is a recurring theme, especially in wait-free algorithms where, in order to make progress, threads must help others (that may be off in the mountains ☺)

LOCK FREE UNBOUNDED QUEUE

Motivation: a Lock-Free Operating System Kernel

At the heart of an Operating System is a scheduler.

A scheduler basically moves tasks between queues (or similar data structures) and selects threads to run on a processor core.

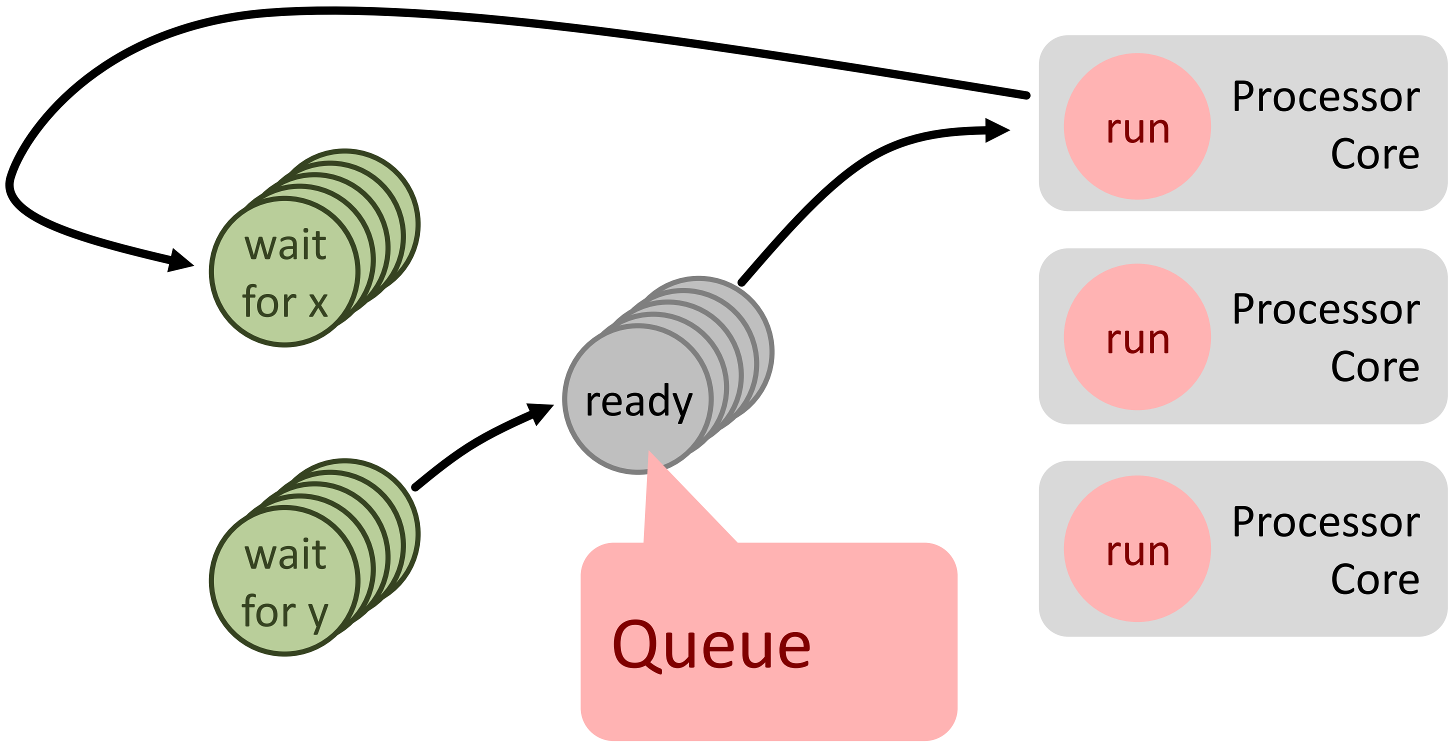
Scheduling decisions usually happen when

threads are created

threads end

threads block / wait

threads unblock



Motivation: a Lock-Free Operating System Kernel

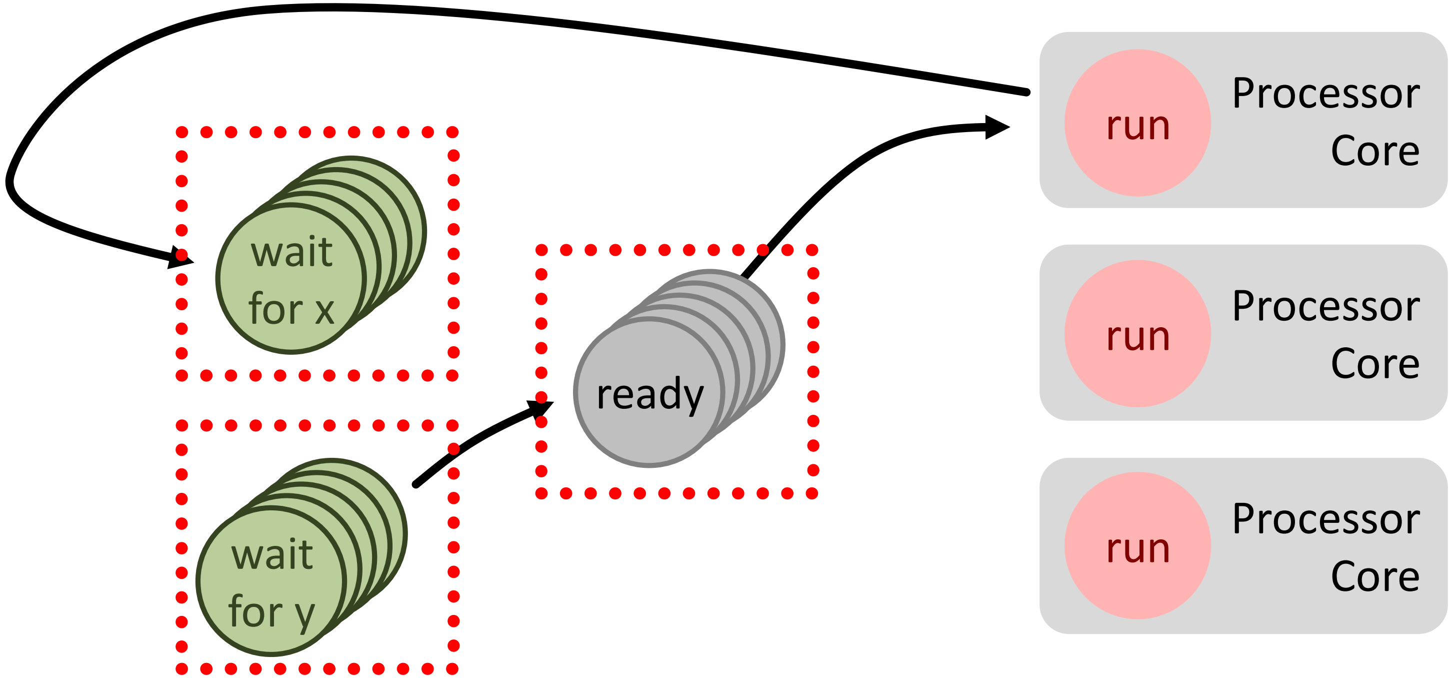
Data structures of a runtime or kernel need to be protected against concurrent access by

- threads and
- interrupt service routines

on different cores.

Conventionally, (spin-)locks are employed for protection

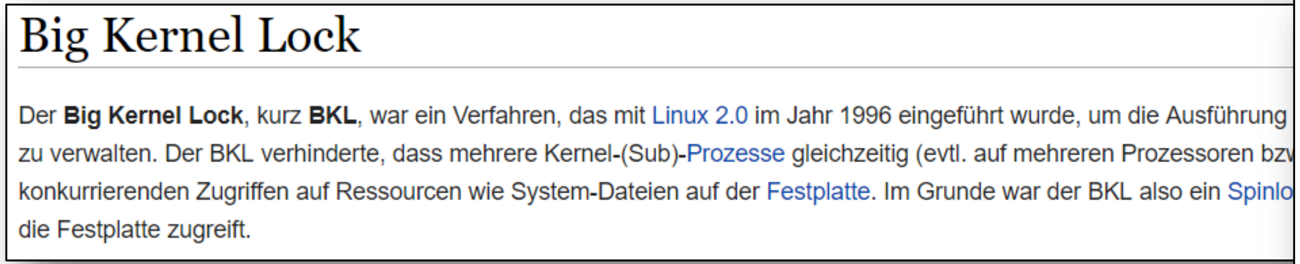
The granularity varies.



Motivation: a Lock-Free Operating System Kernel

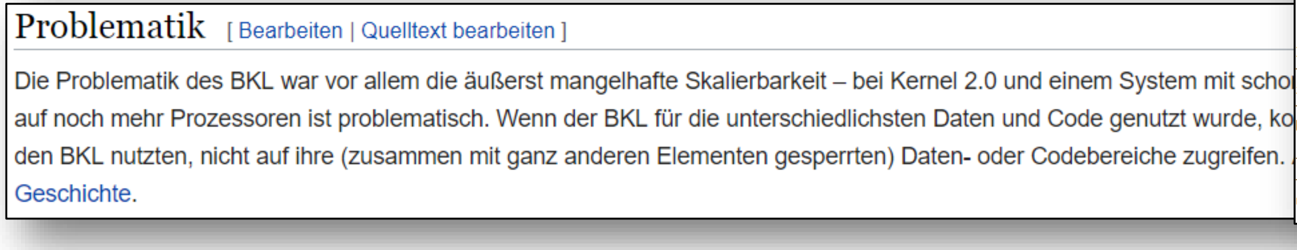
If we want to protect scheduling queues in a lock-free way, we obviously need

- an implementation of a lock-free unbounded queue
  We will again meet the problem of transient inconsistencies
- If we want to use the queues in a scheduler, usually we cannot rely on Garbage Collection, we need to reuse elements of the queue
  This will lead to a difficult problem, the ABA problem



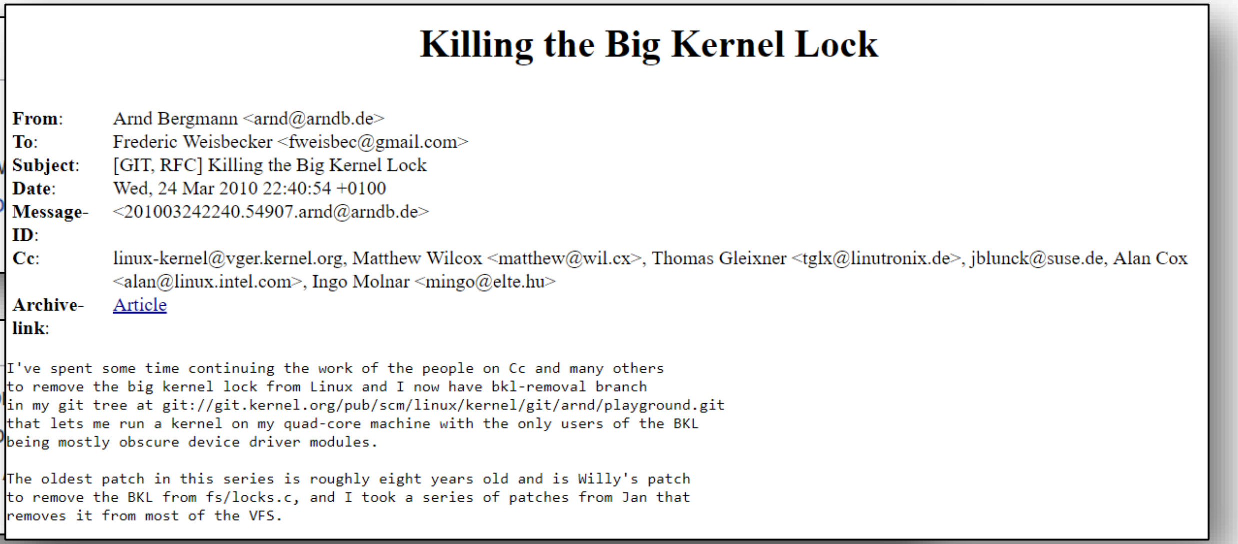
Big Kernel Lock

Der **Big Kernel Lock**, kurz **BKL**, war ein Verfahren, das mit Linux 2.0 im Jahr 1996 eingeführt wurde, um die Ausführung
zu verwalten. Der BKL verhinderte, dass mehrere Kernel-(Sub)-Prozesse gleichzeitig (evtl. auf mehreren Prozessoren bzw
konkurrierenden Zugriffen auf Ressourcen wie System-Dateien auf der Festplatte. Im Grunde war der BKL also ein Spinlo
die Festplatte zugreift.



Problematik [Bearbeiten | Quelltext bearbeiten]

Die Problematik des BKL war vor allem die äußerst mangelhafte Skalierbarkeit – bei Kernel 2.0 und einem System mit scho
auf noch mehr Prozessoren ist problematisch. Wenn der BKL für die unterschiedlichsten Daten und Code genutzt wurde, ko
den BKL nutzten, nicht auf ihre (zusammen mit ganz anderen Elementen gesperrten) Daten- oder Codebereiche zugreifen.
Geschichte.



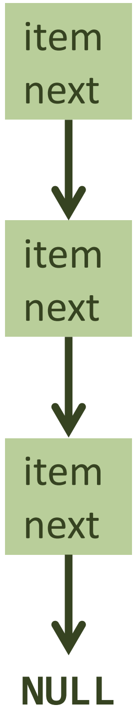
Killing the Big Kernel Lock

From: Arnd Bergmann <arnd@arndb.de>
To: Frederic Weisbecker <fweisbec@gmail.com>
Subject: [GIT, RFC] Killing the Big Kernel Lock
Date: Wed, 24 Mar 2010 22:40:54 +0100
Message-ID: <201003242240.54907.arnd@arndb.de>
Cc: linux-kernel@vger.kernel.org, Matthew Wilcox <matthew@wil.cx>, Thomas Gleixner <tglx@linutronix.de>, jblunck@suse.de, Alan Cox <alan@linux.intel.com>, Ingo Molnar <mingo@elte.hu>
Archive-link: Article

```
I've spent some time continuing the work of the people on Cc and many others
to remove the big kernel lock from Linux and I now have bkl-removal branch
in my git tree at git://git.kernel.org/pub/scm/linux/kernel/git/arnd/playground.git
that lets me run a kernel on my quad-core machine with the only users of the BKL
being mostly obscure device driver modules.

The oldest patch in this series is roughly eight years old and is Willy's patch
to remove the BKL from fs/locks.c, and I took a series of patches from Jan that
removes it from most of the VFS.
```

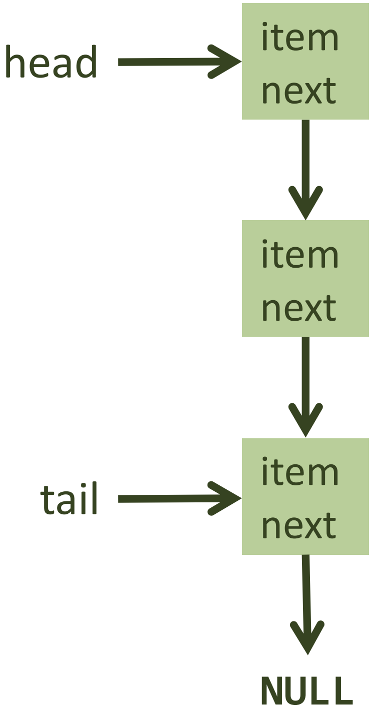
Queue Node

```
public class Node<T> {
   public T value;
   public Node<T> next;

   public Node(T item) {
       this.item = item;
       next = null
   }
}
```



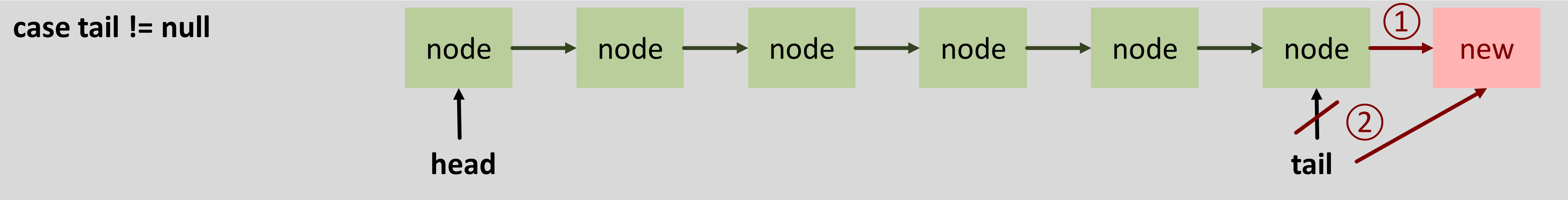
Blocking Queue

```
public class BlockingQueue<T> {
   Node<T> head, tail;

   public synchronized void Enqueue(T item) {
   }

   public synchronized T Dequeue() {
   }
}
```

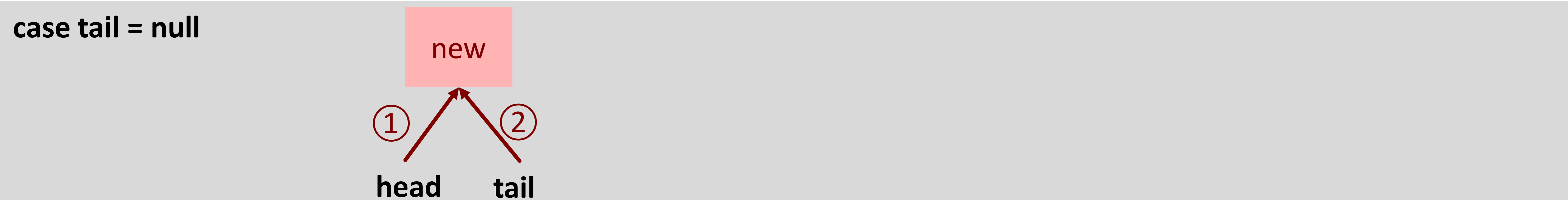


Enqueue

```
public synchronized void Enqueue(T item) {
   Node<T> node = new Node<T>(item);
   if (tail != null)
      tail.next = node;
   else
      head = node;
   tail = node;
}
```



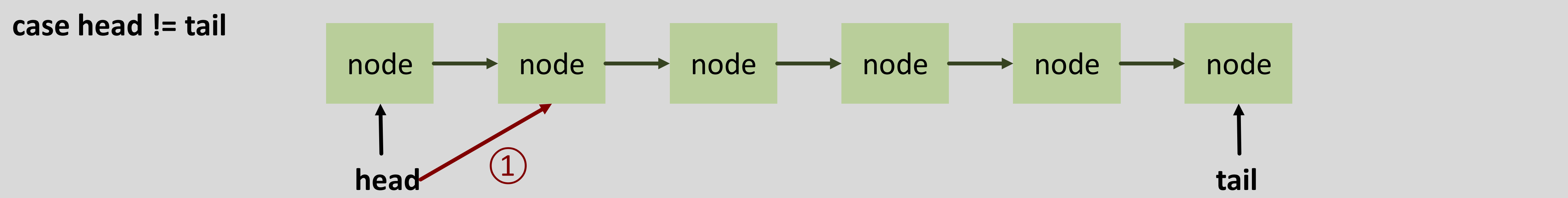
case tail != null



case tail = null

Dequeue

```
public synchronized T Dequeue() {
   T item = null;
   Node<T> node = head;
   if (node != null) {
      item = node.item;
      head = node.next;
      if (head == null) tail = null;
   }
   return item;
}
```



case head != tail

case head == tail

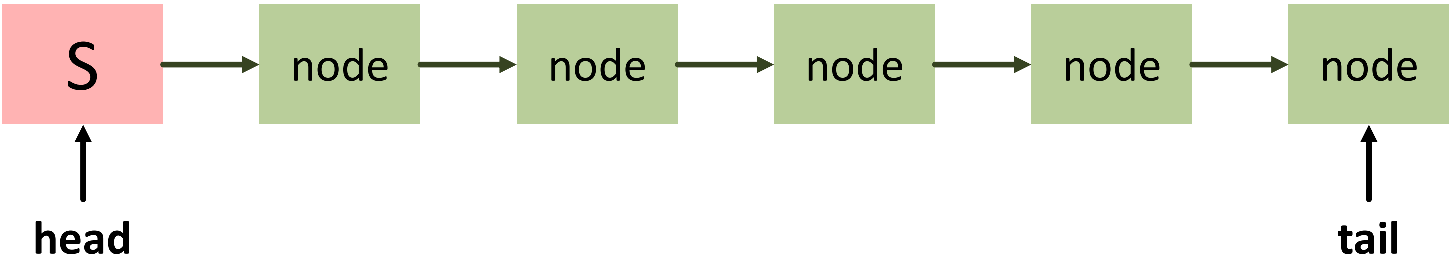
Observation

It turns out that when we want to implement a lock-free queue like this, we run into problems because of potentially simultaneous updates of

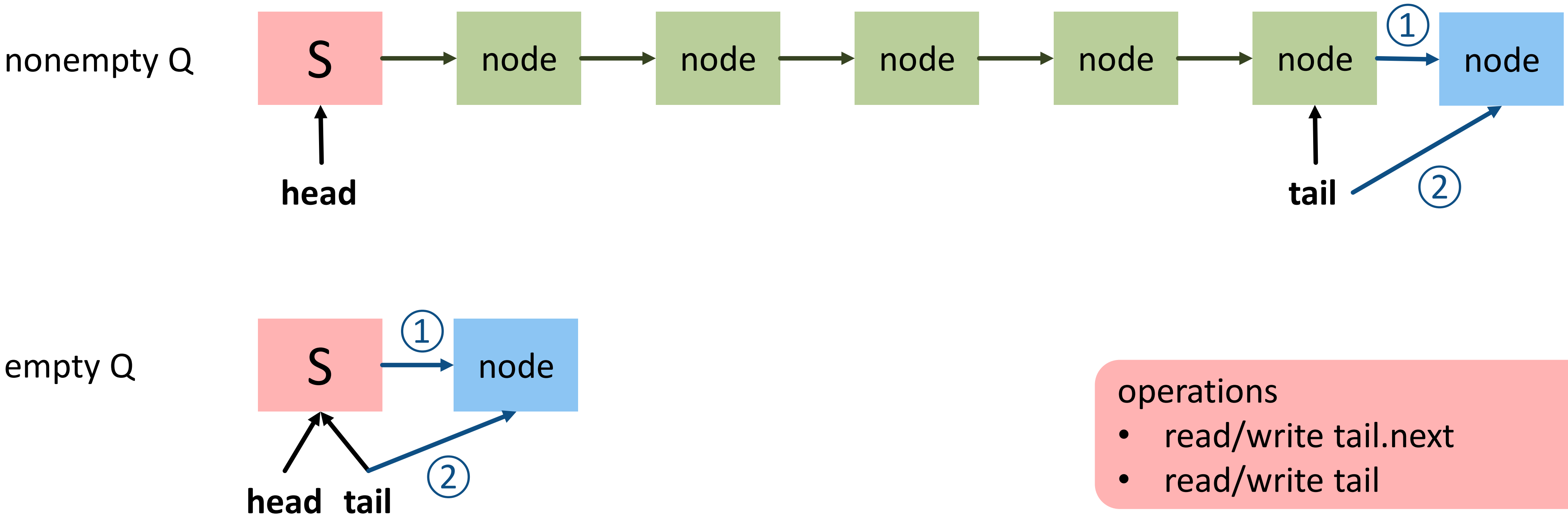
- **head**
- **tail**
- **tail.next**

How to solve this?

Idea: Sentinel at the front

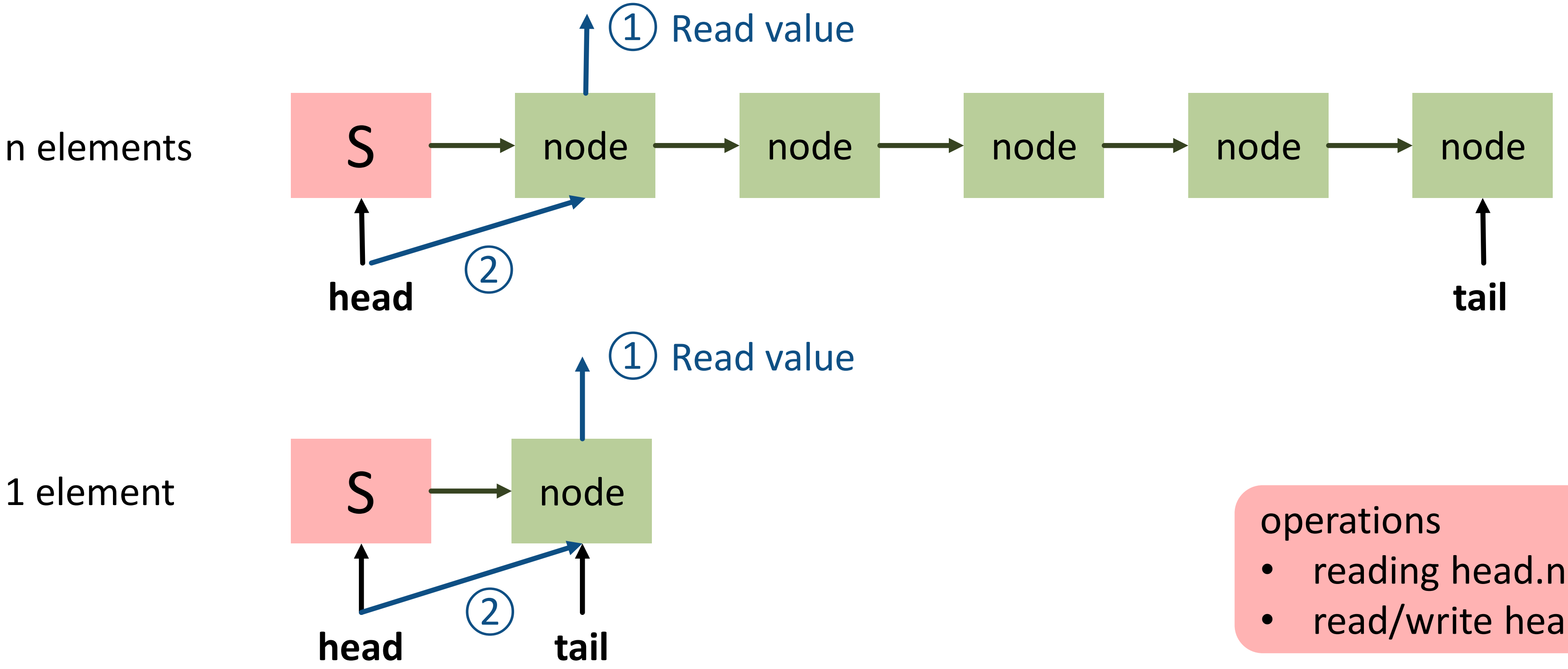


Sentinel at the front: Enqueue



operations
- read/write tail.next
- read/write tail

Sentinel at the front: Dequeue



operations
- reading head.next
- read/write head

Does this help?

Still have to update two pointers at a time!

- But enqueuers work on tail and dequeuers on head

Possible inconsistency?

- `tail` might (transiently) not point to the last element

What's the problem with this?

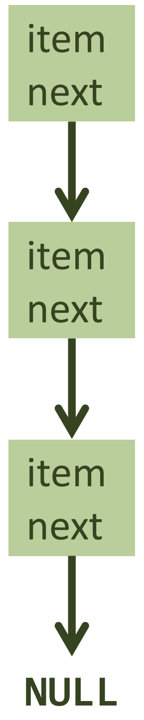
- Unacceptable that any thread has to wait for the consistency to be established -- this would be locking camouflaged

Solution

- Threads help making progress

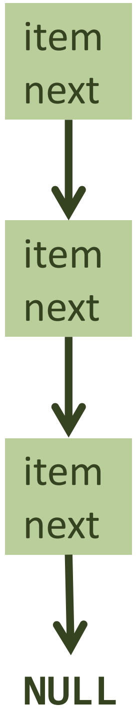
Queue Node needs Atomic next pointer

```
public class Node<T> {
    public T item;
    public AtomicReference<Node> next;

    public Node(T item) {
        next = new AtomicReference<Node>(null);
        this.item = item;
    }

    public void SetItem(T item) {
        this.item = item;
    }
}
```



Queue

```
public class NonBlockingQueue extends Queue {
   AtomicReference<Node> head = new AtomicReference<Node>();
   AtomicReference<Node> tail = new AtomicReference<Node>();

   public NonBlockingQueue() {
      Node node = new Node(null);
      head.set(node); tail.set(node);
   }

   public void Enqueue(T item);

   public T Dequeue();
}
```

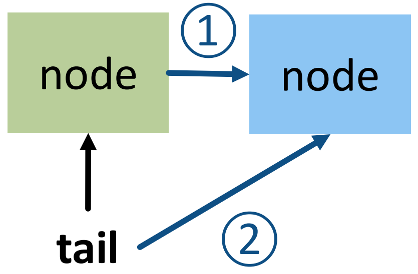


Protocol: Initial Version

Enqueuer

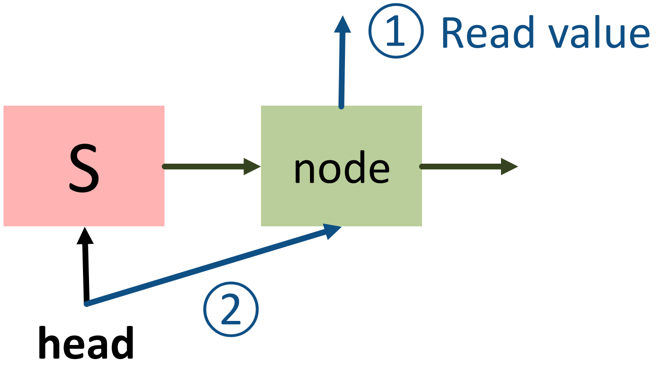
- read `tail` into `last`
- then tries to set `last.next`:
  CAS(last.next, null, new)
- If unsuccessful retry!
- If successful, try to set tail without retry

CAS(tail, last, new)



Dequeuer

- read head into `first`
- read `first.next` into next
- if `next` is available, read the item value of `next`
- try to set head from `first` to next
  CAS(head, first, next)
- If unsuccessful, retry!



Protocol

Enqueuer

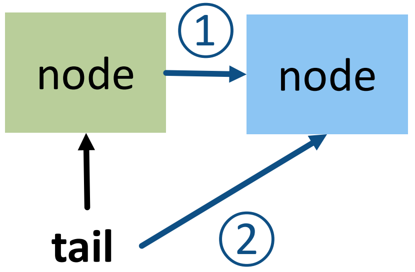
- read `tail` into `last`
- then tries to set `last.next`: **CAS(last.next, null, new)**
- If unsuccessful retry!
- If successful, try to set tail without retry

CAS(tail, last, new)

How can this be unsuccessful?

1. Some other thread has written last.next just before me
2. I have read a stale version of tail either
    a) because I just missed the update of other thread
    b) because the other thread failed in updating tail, for example because it has died

If the thread dies before calling this, tail is never updated.



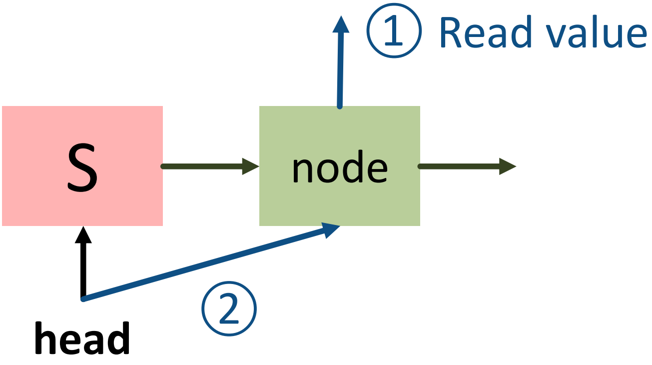
Protocol

Dequeuer

- read head into `first`
- read `first.next` into `next`
- if `next` is available, read the item value of `next`
- try to set head from `first` to `next`
  CAS(head, first, next)
- If unsuccessful, retry!

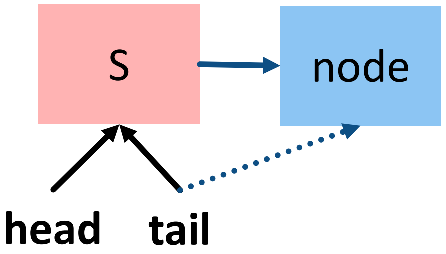
How can this be unsuccessful?

1. another thread has already removed next

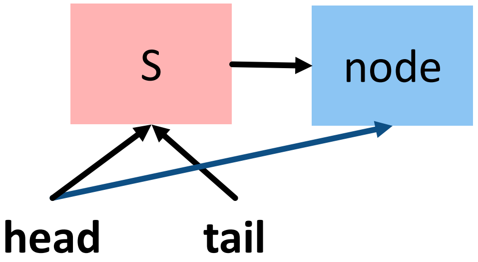


One more possible inconsistency

- Thread A enqueues an element to an empty list, but has not yet adapted tail



- Thread B dequeues (the sentinel)



- Now tail points to a dequeued element.

Final solution: enqueue

```
public void enqueue(T item) {
   Node node = new Node(item);
   while(true) {
      Node last = tail.get();
      Node next = last.next.get();
      if (next == null) {
         if (last.next.compareAndSet(null, node)) {
            tail.compareAndSet(last, node);
            return;
         }
      }
      else
         tail.compareAndSet(last, next);
   }
}
```

Help other threads to make progress !

Create the new node

Read current tail as last and last.next as next

Try to set last.next from null to node, if success then try to set tail

Ensure progress by advancing tail pointer if required and retry

Final solution: dequeue

```
public T dequeue() {
   while (true) {
      Node first = head.get();
      Node last = tail.get();
      Node next = first.next.get();
      if (first == last) {
         if (next == null) return null;
         else tail.compareAndSet(last, next);
      }
      else {
         T value = next.item;
         if (head.compareAndSet(first, next))
            return value;
      }
   }
}
```

Help other threads to make progress !

Read head as first, tail as last and first.next as next

Check if queue looks empty
(1) really empty: return
(2) next available: advance last pointer

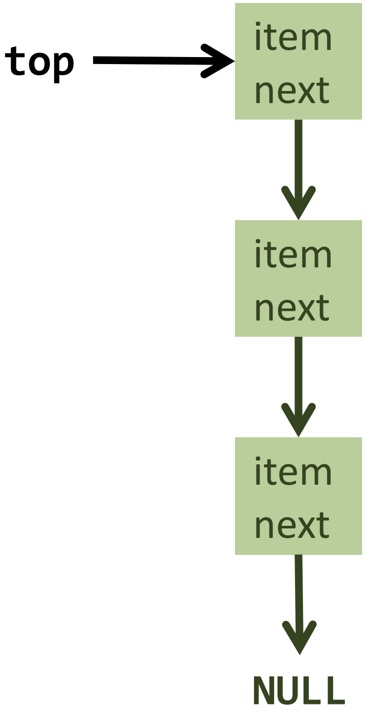
If queue is not empty, memorize value on next element and try to remove current sentinel

Retry if removal was unsuccessful

REUSE AND THE ABA PROBLEM

For the sake of simplicity: back to the Stack ☺

```
public class ConcurrentStack {
AtomicReference<Node> top = new AtomicReference<Node>();

   public void push(Long item) { … }
   public Long pop() { … }
}
```

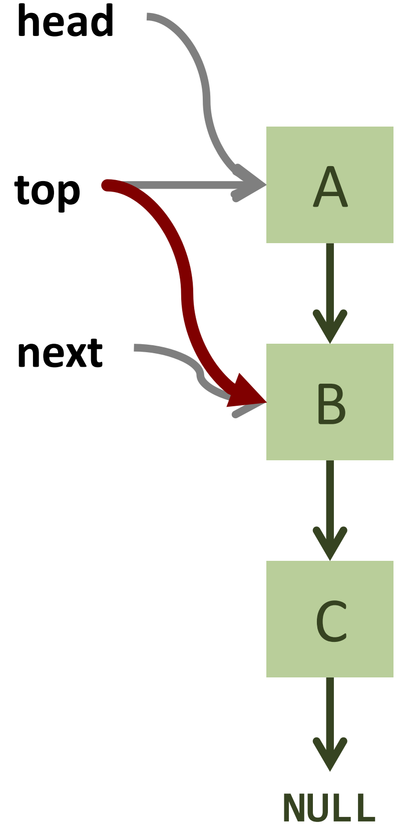


pop

```
public Long pop() {
  Node head, next;

  do {
    head = top.get();
    if (head == null) return null;
    next = head.next;
  } while (!top.compareAndSet(head, next));

  return head.item;
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change

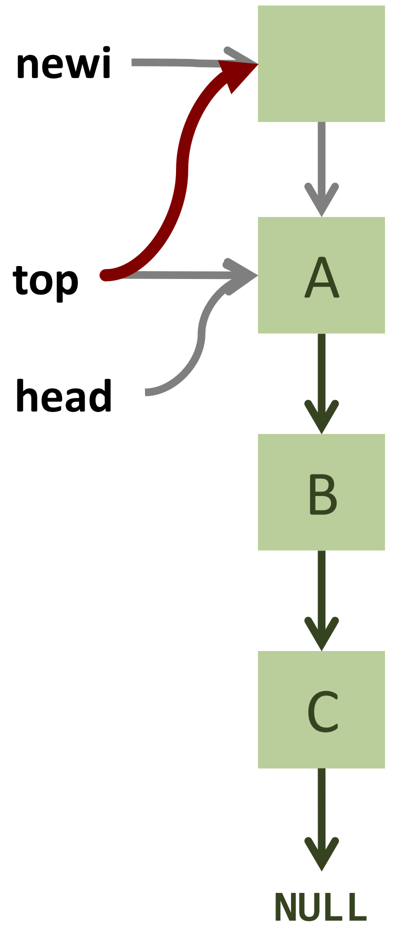


push

```
public void push(Long item) {
        Node newi = new Node(item);
        Node head;

        do {
                head = top.get();
                newi.next = head;
        } while (!top.compareAndSet(head, newi));
}
```

Memorize "current stack state" in local variable head

Action is taken only if "the stack state" did not change



Node Reuse

Assume we do not want to allocate for each push and maintain a Node-pool instead. Does this work?

```
public class NodePool {
   AtomicReference<Node> top new AtomicReference<Node>();

   public void put(Node n) { … }
   public Node get() { … }
}

public class ConcurrentStackP {
   AtomicReference<Node> top = newAtomicReference<Node>();
   NodePool pool = new NodePool();
   ...
}
```

NodePool put and get

```
public Node get(Long item) {
   Node head, next;
   do {
      head = top.get();
      if (head == null) return new Node(item);
      next = head.next;
   } while (!top.compareAndSet(head, next));
   head.item = item;
   return head;
}

public void put(Node n) {
   Node head;
   do {
      head = top.get();
      n.next = head;
   } while (!top.compareAndSet(head, n));
}
```

Only difference to Stack above: NodePool is in-place.

A node can be placed in one and only one in-place data structure. This is ok for a global pool.

***So far* this works.**

Using the node pool

```
public void push(Long item) {
   Node head;
   Node new = pool.get(item);
   do {
      head = top.get();
      new.next = head;
   } while (!top.compareAndSet(head, new));
}

public Long pop() {
   Node head, next;
   do {
      head = top.get();
      if (head == null) return null;
      next = head.next;
   } while (!top.compareAndSet(head, next));
   Long item = head.item;
   pool.put(head);
   return item;
}
```

Experiment

- run n consumer and producer threads
- each consumer / producer pushes / pops 10,000 elements and records sum of values
- if a pop returns an "empty" value, retry
- do this 10 times with / without node pool
- measure wall clock time (ms)
- check that sum of pushed values == sum of popped values

Result (of one particular run)

nonblocking stack **without** reuse

n = 1, elapsed= 15, normalized= 15
n = 2, elapsed= 110, normalized= 55
n = 4, elapsed= 249, normalized= 62
n = 8, elapsed= 843, normalized= 105
n = 16, elapsed= 1653, normalized= 103
n = 32, elapsed= 3978, normalized= 124
n = 64, elapsed= 9953, normalized= 155
n = 128, elapsed= 24991, normalized= 195

nonblocking stack **with** reuse

n = 1, elapsed= 47, normalized= 47
n = 2, elapsed= 109, normalized= 54
n = 4, elapsed= 312, normalized= 78
n = 8, elapsed= 577, normalized= 72
n = 16, elapsed= 1747, normalized= 109
n = 32, elapsed= 2917, normalized= 91
n = 64, elapsed= 6599, normalized= 103
n = 128, elapsed= 12090, normalized= 94

yieppieh...

But other runs …

nonblocking stack **with** reuse

n = 1, elapsed= 62, normalized= 62

n = 2, elapsed= 78, normalized= 39

n = 4, elapsed= 250, normalized= 62

n = 8, elapsed= 515, normalized= 64

n = 16, elapsed= 1280, normalized= 80

n = 32, elapsed= 2629, normalized= 82

Exception in thread "main"
java.lang.RuntimeException:
sums of pushes and pops don't match

at stack.Measurement.main(Measurement.java:107)

nonblocking stack **with** reuse

n = 1, elapsed= 48, normalized= 48

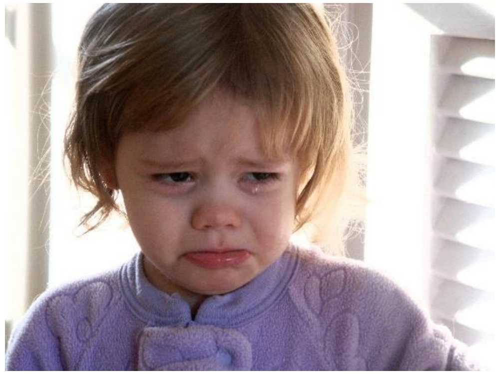
n = 2, elapsed= 94, normalized= 47

n = 4, elapsed= 265, normalized= 66

n = 8, elapsed= 530, normalized= 66

n = 16, elapsed= 1248, normalized= 78

[and does not return]



why?

ABA Problem

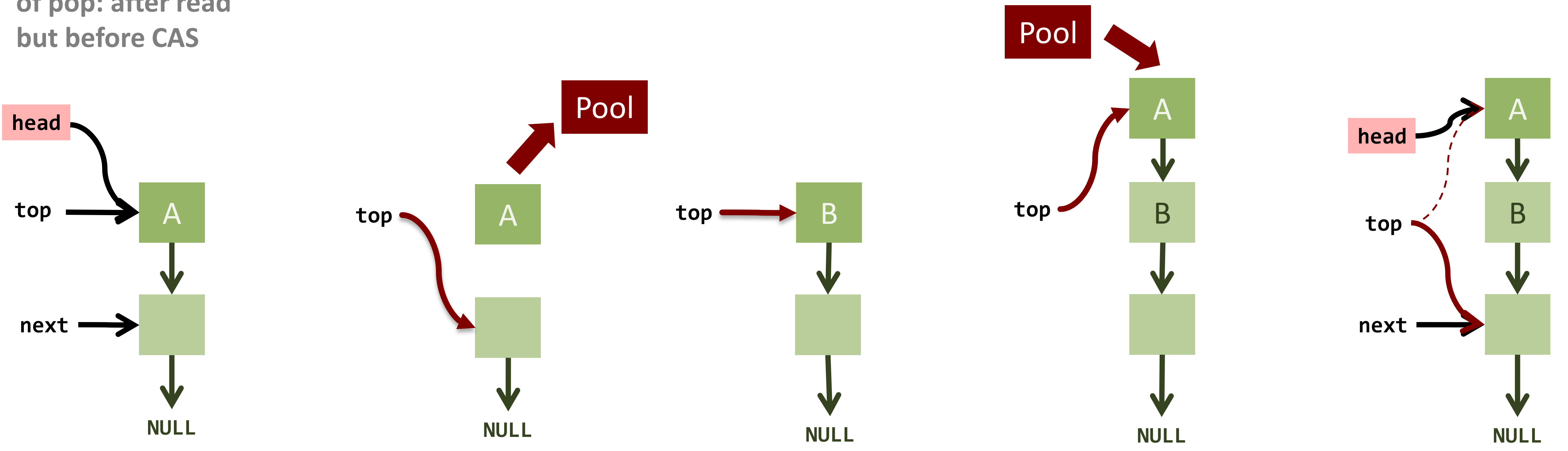
Thread X
in the middle
of pop: after read
but before CAS

Thread Y
pops A

Thread Z
pushes B

Thread Z'
pushes A

Thread X
completes pop

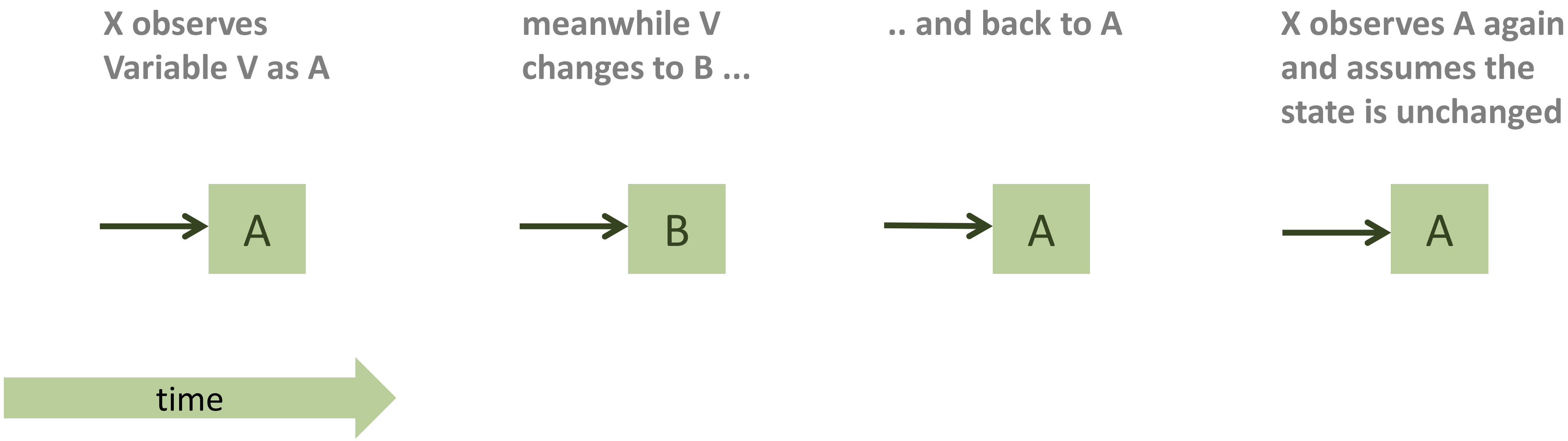


```
public Long pop() {
      Node head, next;
      do {
            head = top.get();
            if (head == null) return null;
            next = head.next;
      } while (!top.compareAndSet(head, next));
      Long item = head.item; pool.put(head); return item;
}
```

```
public void push(Long item) {
      Node head;
      Node new = pool.get(item);
      do {
            head = top.get();
            new.next = head;
      } while (!top.compareAndSet(head, new));
}
```

The ABA-Problem

"The ABA problem ... occurs when one activity fails to recognize that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overall state has not been changed."



X observes Variable V as A

meanwhile V changes to B ...

.. and back to A

X observes A again and assumes the state is unchanged

How to solve the ABA problem?

DCAS (double compare and swap)

- not available on most platforms (we have used a variant for the lock-free list set)

Garbage Collection

- relies on the existence of a GC
- much too slow to use in the inner loop of a runtime kernel
- can you implement a lock-free garbage collector relying on garbage collection?

Pointer Tagging

- does not cure the problem, rather delay it
- can be practical

Hazard Pointers

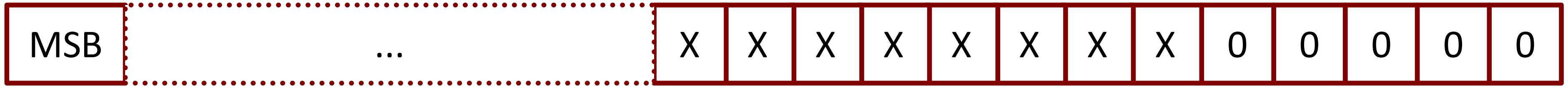
Transactional memory (later)

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging



Each time a pointer is stored in a data structure, the tag is increased by one.

Access to a data structure via address x – (x mod 32)

This makes the ABA problem very much less probable because now 32 versions of each pointer exist.

Hazard Pointers

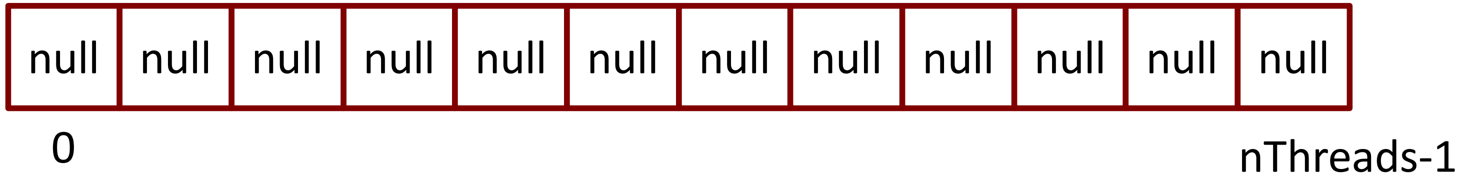
The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

- before X reads P, it marks it hazarduous by entering it in one of the n (n= number threads) slots of an array associated with the data structure (e.g., the stack)
- When finished (after the CAS), process X removes P from the array
- Before a process Y tries to reuse P, it checks all entries of the hazard array

Hazard Pointers

```
public class NonBlockingStackPooledHazardGlobal extends Stack {
    AtomicReference<Node> top = new AtomicReference<Node>();
    NodePoolHazard pool;
    AtomicReferenceArray<Node> hazarduous;

    public NonBlockingStackPooledHazardGlobal(int nThreads) {
        hazarduous =  new AtomicReferenceArray<Node>(nThreads);
        pool = new NodePoolHazard(nThreads);
    }
}
```

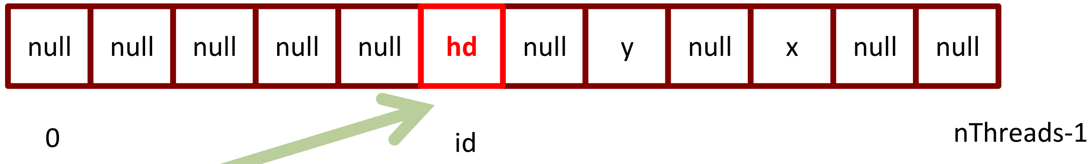


Hazard Pointers

null	null	null	null	null	**hd**	null	y	null	x	null	null
0					id						nThreads-1

```
boolean isHazarduous(Node node) {
    for (int i = 0; i < hazarduous.length(); ++i)
        if (hazarduous.get(i) == node)
            return true;
    return false;
}

void setHazardous(Node node) {
    hazarduous.set(id, node); // id is current thread id
}
```

Hazard Pointers



null	null	null	null	null	**hd**	null	y	null	x	null	null
0					id						nThreads-1

```
public int pop(int id) {
    Node head, next = null;
    do {
        do {
            head = top.get();
            setHazarduous(head);
        } while (head == null || top.get() != head);
        next = head.next;
    } while (!top.compareAndSet(head, next));
    setHazarduous(null);
    int item = head.item;
    if (!isHazardous(head))
        pool.put(id, head);
    return item;
}
```

This ensures that no other thread is already past the CAS and has not seen our hazard pointer

```
public void push(int id, Long item) {
    Node head;
    Node newi = pool.get(id, item);
    do{
        head = top.get();
        newi.next = head;
    } while (!top.compareAndSet(head, newi));
}
```

How to protect the Node Pool?

The ABA problem also occurs on the node pool.

Two solutions:

Thread-local node pools

- No protection necessary
- Does not help when push/pop operations are not well balanced

Hazard pointers on the global node pool

- Expensive operation for node reuse
- Equivalent to code above: node pool returns a node only when it is not hazarduous

Remarks

The Java code above does not really improve performance in comparison to memory allocation plus garbage collection.

But it demonstrates how to solve the ABA problem principally.

The hazard pointers are placed **in thread-local storage.**

When thread-local storage can be replaced by processor-local storage, it scales better*.

e.g., in *Florian Negele, *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System*, PhD Thesis, ETH Zürich 2014

Lessons Learned

Lock-free programming: new kind of problems in comparison to lock-based programming:

- Atomic update of several pointers / values impossible, leading to new kind of problems and solutions, such as threads that help each other in order to guarantee global progress
- ABA problem (which disappears with a garbage collector)