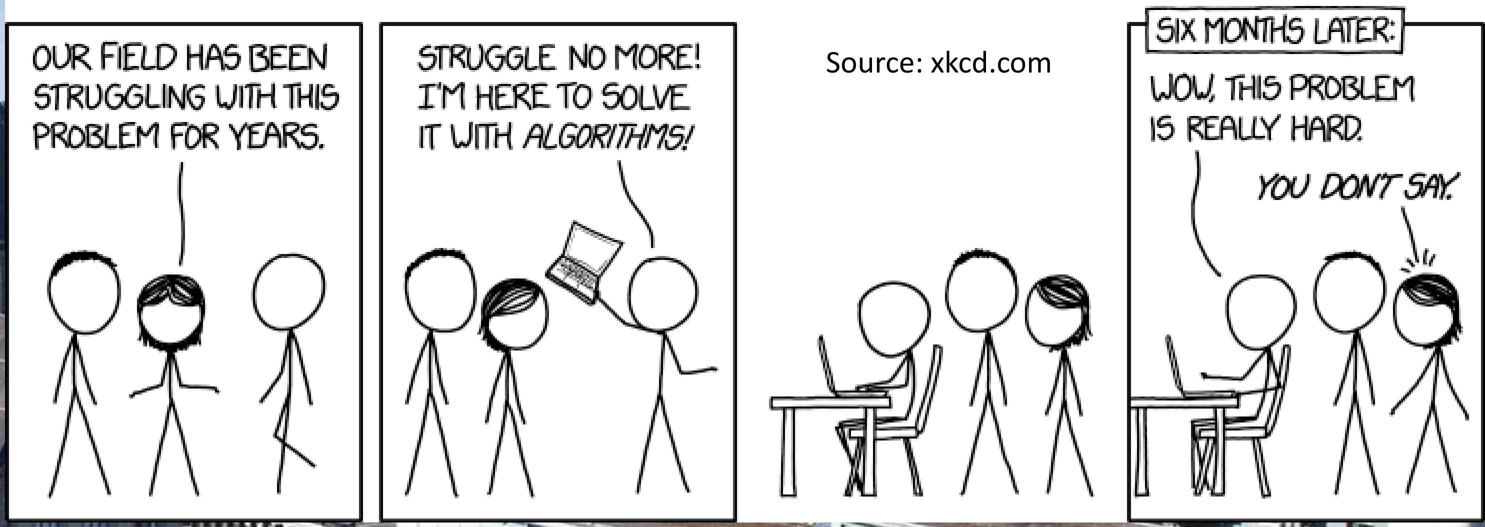
TORSTEN HOEFLER

# Parallel Programming

# Barriers, Readers/Writers Lock, Lock Granularity: Coarse Grained, Fine Grained, Optimal, and Lazy Synchronization

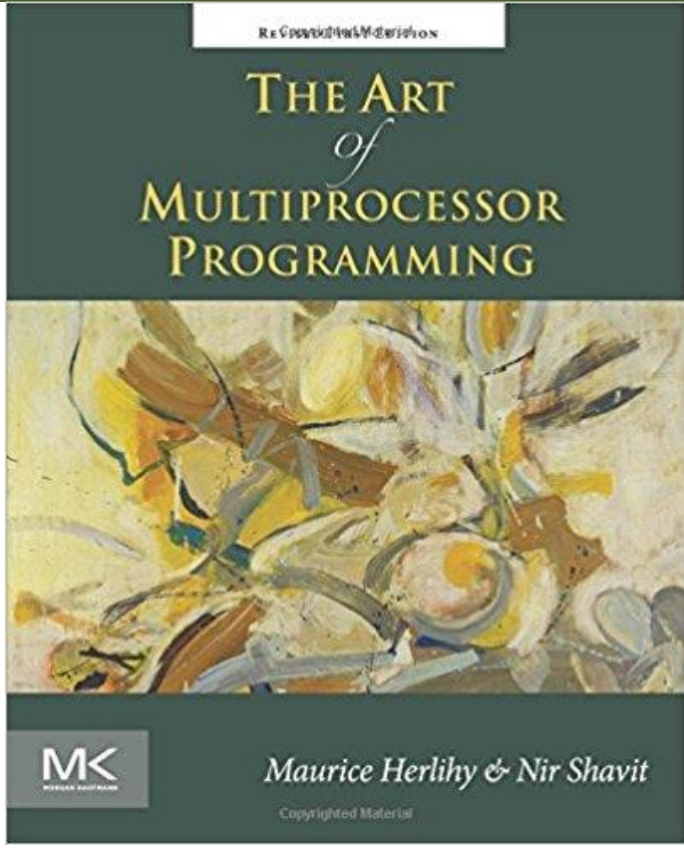



Source: xkcd.com

# Administrivia

- **Book for (the second part of) the lecture series:**
  - Herlihy & Shavit: The Art of Multiprocessor Programming
    *Contains more concepts (e.g., linearizability) than we use here!*

- **Locking objects – terminology clarification**
  - When we say “we lock an object” then I mean “we acquire the lock associated with the object” – data cannot be locked directly in Java (cf. advisory filesystem locks)




- **Starvation-freedom:**
  - *every* thread that tries to make progress, makes progress *eventually*

# Last week

- **Atomics**
  - Why do they exist (“extend” a limiting abstraction)
  - How do they work (example TAS, CAS)
  - Simple locks

- **Deadlock**
  - Cause (cyclic dependencies)
  - Avoidance (acquire resources in global order – many possibilites)

- **Semaphores**
  - Generalization of locks, can count (enables producer/consumer)

# Learning goals today

- **Barriers**
  - Multi-process synchronization, important in parallel programming
  - More examples for complexity of parallel programming (trial and error impossible)
- **Producer/Consumer in detail**
  - Queues, implementation
- **Monitors**
  - Condition variables, wait, signal, etc. (continued today)
- **More on locks (essentially a bag of tricks)**
  - Reader/writer
  - Coarse-grained vs. fine-grained
  - Optimistic synchronization
  - Lazy synchronization

# Example: scaled dot product

When is x ready?

- **Execute in parallel: $x = (\underline{a}^T * \underline{d}) * z$**
  - a and d are column vectors
  - x, z are scalar
- **Assume each vector has 4 elements**
  - $x = (a_1*d_1 + a_2*d_2 + a_3*d_3 + a_4*d_4) * z$
- **Parallelize on two processors (using two threads A and B)**
  - $x_A = a_1*d_1 + a_2*d_2$
  - $x_B = a_3*d_3 + a_4*d_4$
  - $x = (x_A + x_B) * z$
- **Which synchronization is needed where?**
  - Using locks?
  - Using semaphores?

```
Thread A
xA=…;
lock();
x=x+xA;
unlock()
```

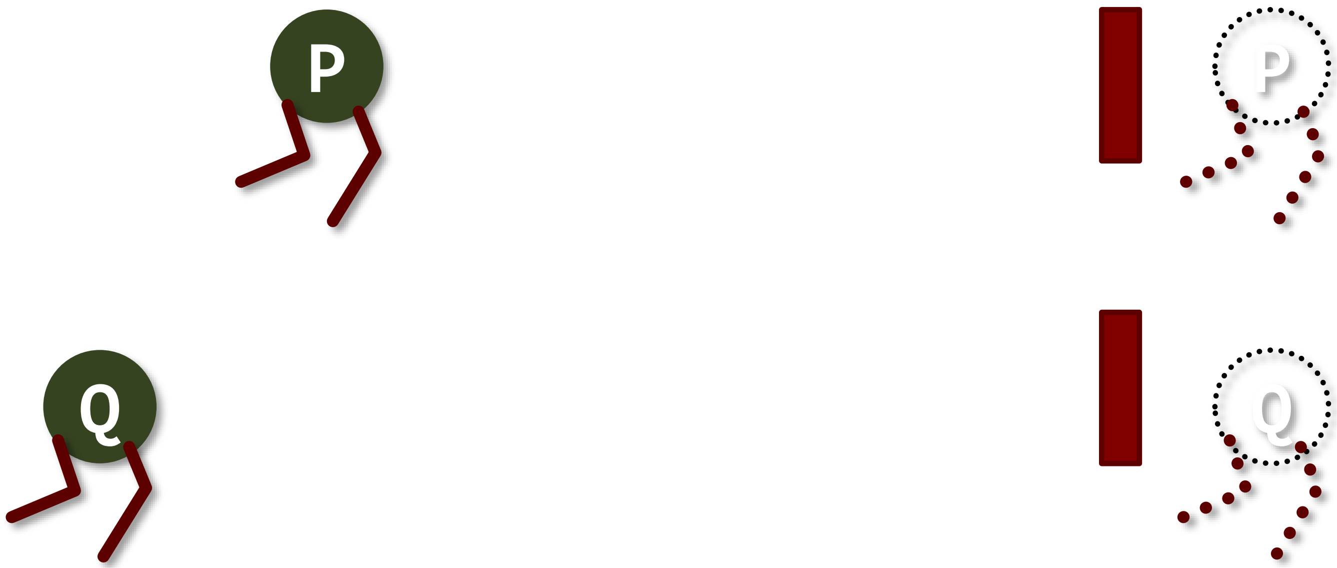
```
Thread B
xB=…;
lock();
x=x+xB;
unlock()
```

```
Thread A
xA=…;
x=x+xA;
release(S);
```

```
Thread A
xB=…;


acquire(S);
x=x+xA;
```

# Rendezvous with Semaphores

- Two processes P and Q executing code.
- Rendezvouz point: location in code where P and Q wait for the other to arrive. Synchronize P and Q.




How would you implement this using semaphores?

# Rendezvous with Semaphores

## Synchronize processes P and Q at one location (rendezvous)

## Semaphores `P_Arrived` and `Q_Arrived`

| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | ... | ... |
| *rendezvous* | ? | ? |
| *post* | ... | .. |

# Rendezvous with Semaphores

**Synchronize processes P and Q at one location (rendezvous)**

**Semaphores `P_Arrived` and `Q_Arrived`**

| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | `...` | `...` |
| *rendezvous* | `release(P_Arrived)`<br>`?` | `acquire(P_Arrived)`<br>`?` |
| *post* | `...` | `...` |

# Rendezvous with Semaphores

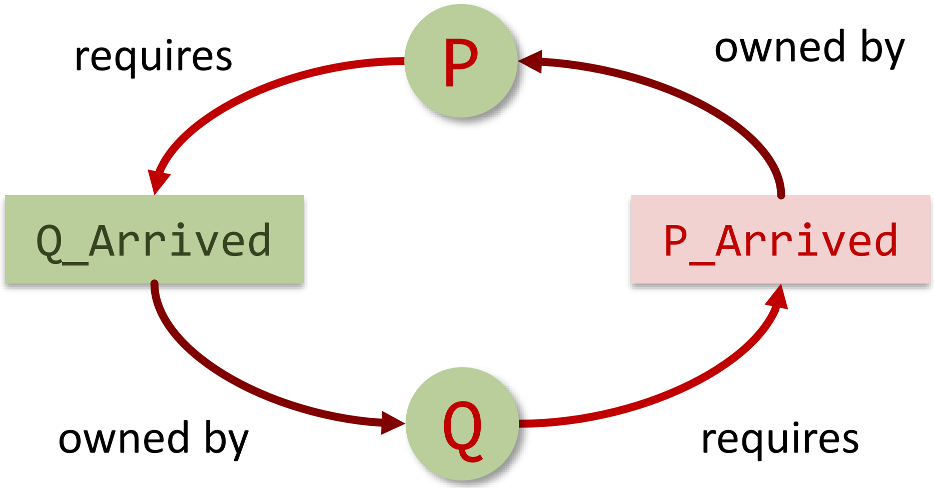
**Synchronize processes P and Q at one location (rendezvous)**

**Semaphores `P_Arrived` and `Q_Arrived`**

Dou you find the problem?

| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | `...` | `...` |
| *rendezvous* | `acquire(Q_Arrived)`<br>`release(P_Arrived)` | `acquire(P_Arrived)`<br>`release(Q_Arrived)` |
| *post* | `...` | `...` |

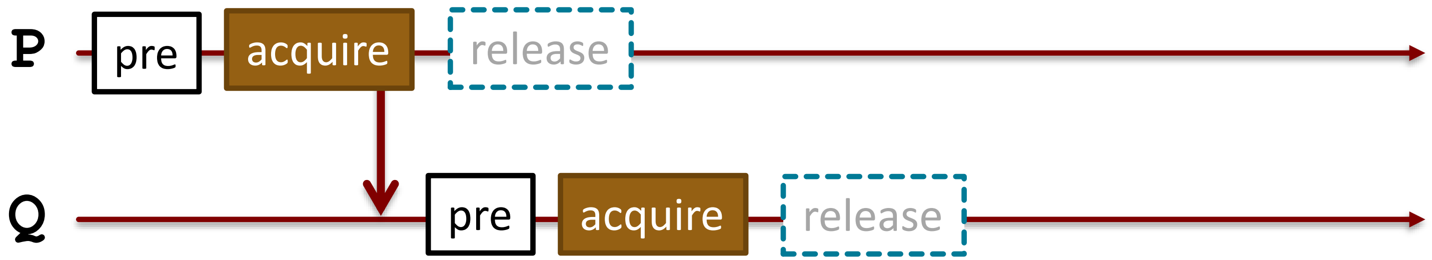
# Deadlock




| | **P** | **Q** |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | `...` | `...` |
| *rendezvous* | `acquire(Q_Arrived)`<br>`release(P_Arrived)` | `acquire(P_Arrived)`<br>`release(Q_Arrived)` |
| *post* | `...` | `...` |

# Rendezvous with semaphores

## Wrong solution with deadlock



# Rendezvous with Semaphores

**Synchronize processes P and Q at one location (rendezvous)**

**Assume semaphores `P_Arrived` and `Q_Arrived`**

| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | ... | ... |
| *rendezvous* | `release(P_Arrived)`<br>`acquire(Q_Arrived)` | `acquire(P_Arrived)`<br>`release(Q_Arrived)` |
| *post* | ... | .. |

# Detour: Implementing Semaphores without Spinning (blocking queues)

## Consider a process list $Q_S$ associated with semaphore S

**acquire(S)**

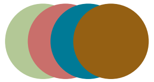
atomic

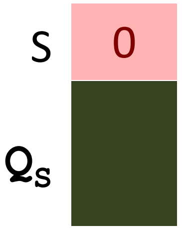
```
{if S > 0 then
     dec(S)
 else
     put(Q_S, self)
     block(self)
 end }
```

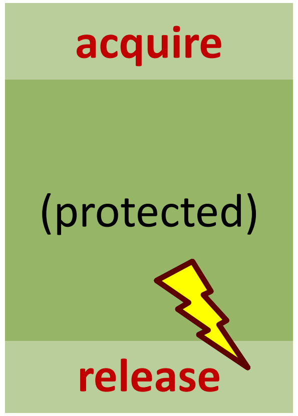
**release(S)**

atomic

```
{if Q_S == ∅ then
     inc(S)
 else
     get(Q_S, p)
     unblock(p)
 end }
```



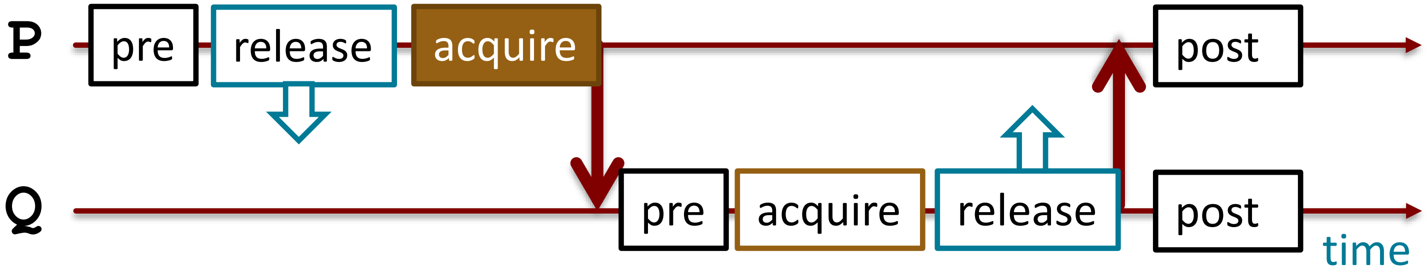


# Scheduling Scenarios

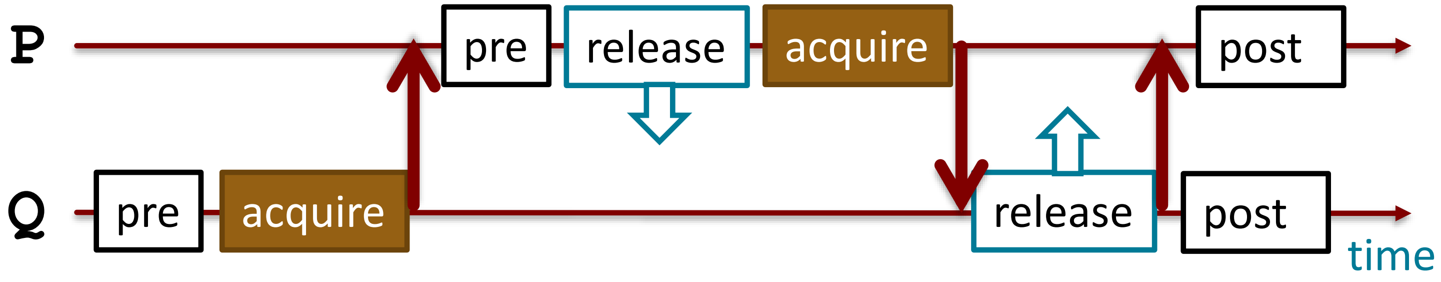
| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | `...` | `...` |
| *rendezvous* | `release(P_Arrived)`<br>`acquire(Q_Arrived)` | `acquire(P_Arrived)`<br>`release(Q_Arrived)` |
| *post* | `...` | `..` |

## P first




release signals (arrow)
acquire may wait (filled box)

## Q first



# Rendezvous with Semaphores

## Synchronize processes P and Q at one location (rendezvous)

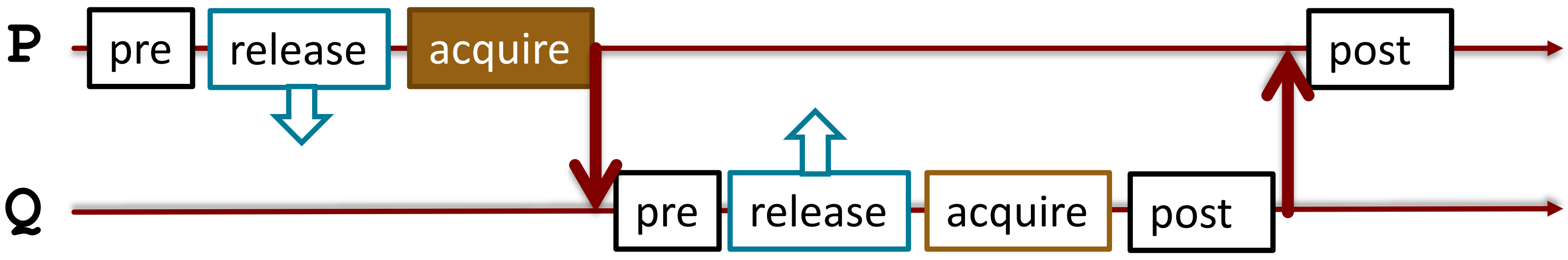
## Assume Semaphores P_Arrived and Q_Arrived

| | P | Q |
|---|---|---|
| *init* | P_Arrived=0 | Q_Arrived=0 |
| *pre* | ... | ... |
| *rendezvous* | release(P_Arrived)<br>acquire(Q_Arrived) | release(Q_Arrived)<br>acquire(P_Arrived) |
| *post* | ... | .. |

# That's even better.

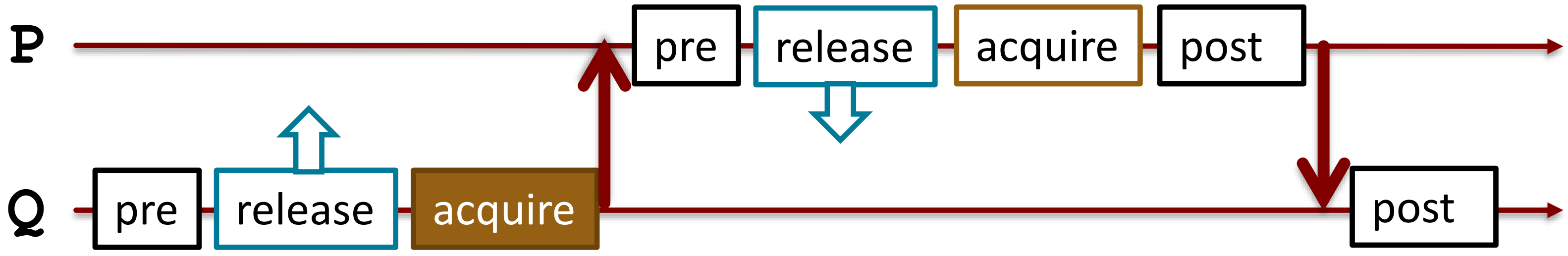
| | P | Q |
|---|---|---|
| *init* | `P_Arrived=0` | `Q_Arrived=0` |
| *pre* | `...` | `...` |
| *rendezvous* | `release(P_Arrived)`<br>`acquire(Q_Arrived)` | `release(Q_Arrived)`<br>`acquire(P_Arrived)` |
| *post* | `...` | `..` |

## P first




## Q first

release signals (arrow)
acquire may wait (filled box)



# Back to our dot-product

- **Assume now vectors with 1 million entries on 10,000 threads**
  - Very common! (we regularly run >1M threads on 20k+ GPUs)
  - How would you implement that?
  - Semaphores, locks?

- **Time for a higher-level abstraction!**
  - Supporting threads in bulk-mode

    *Move in lock-step*
  - And enabling a "bulk-synchronous parallel" (BSP) model

    *The full BSP is more complex (supports distributed memory)*

A Bridging Model for Parallel Computation

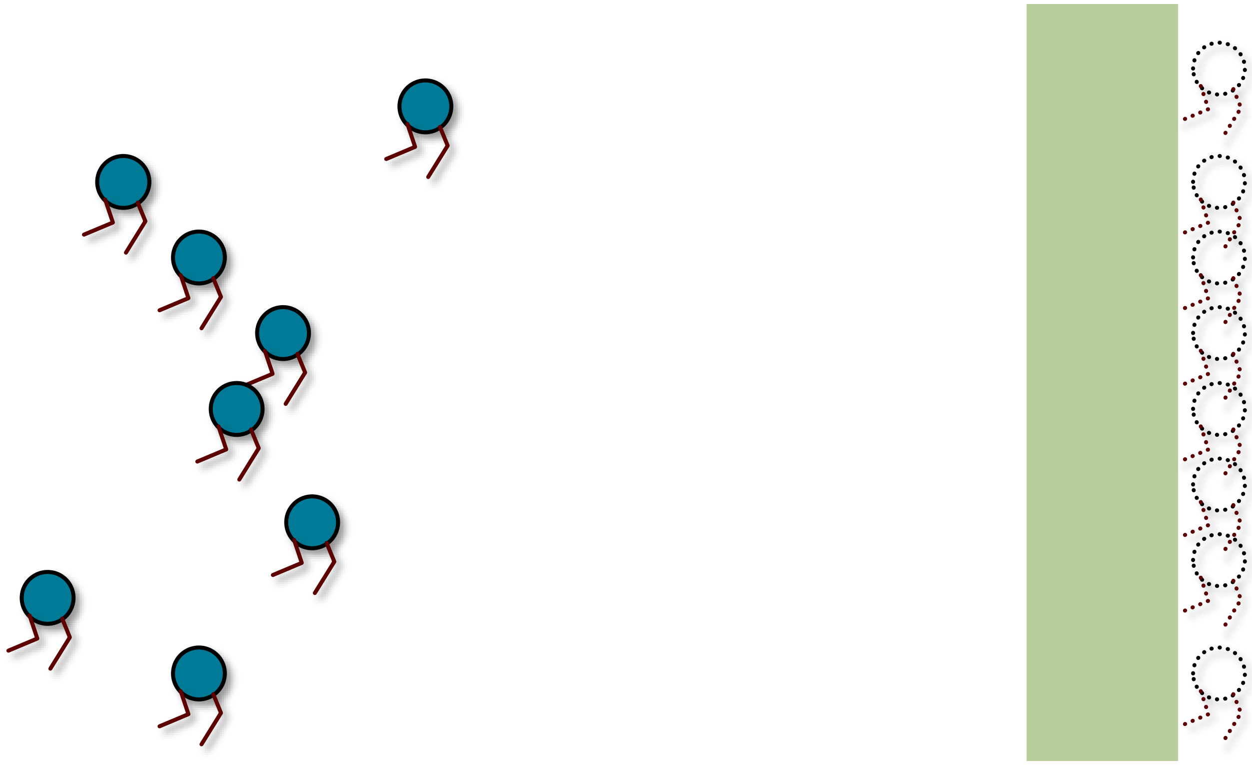
The success of the von Neumann model of sequential computation is attributable to the fact that it is an efficient bridge between software and hardware: high-level languages can be efficiently compiled on to this model; yet it can be efficiently implemented in hardware. The author argues that an analogous bridge between software and hardware is required for parallel computation if that is to become as widely used. This article introduces the bulk-synchronous parallel (BSP) model as a candidate for this role, and gives results quantifying its efficiency both in implementing high-level language features and algorithms, as well as in being implemented in hardware.

Leslie G. Valiant


# Barriers

# Barrier

**Synchronize a number of processes.**



How would you implement this using semaphores?

# Barrier – 1st try

**Synchronize a number (n) of processes.**

**Semaphore barrier. Integer count.**

| | P1 | P2 | ... | Pn |
|---|---|---|---|---|
| *init* | barrier = 0; volatile count = 0 | | | |
| *pre* | ... | | | |
| *barrier* | count++ | | | |
| | if (count==n) release(barrier) | ← | ← | ← |
| | acquire(barrier) | | | |
| *post* | ... | | | |

Race Condition !

May wait forever!

WRONG

# Barrier – 1st try

**Synchronize a number (n) of processes.**

**Semaphore barrier. Integer count.**

| | P1 |
|---|---|
| *init* | barrier = 0; volatile c |
| *pre* | ... |
| *barrier* | count++<br>if (count==n) release(barrier)<br>acquire(barrier) |
| *post* | ... |

WRONG

Deadlock !

Invariants

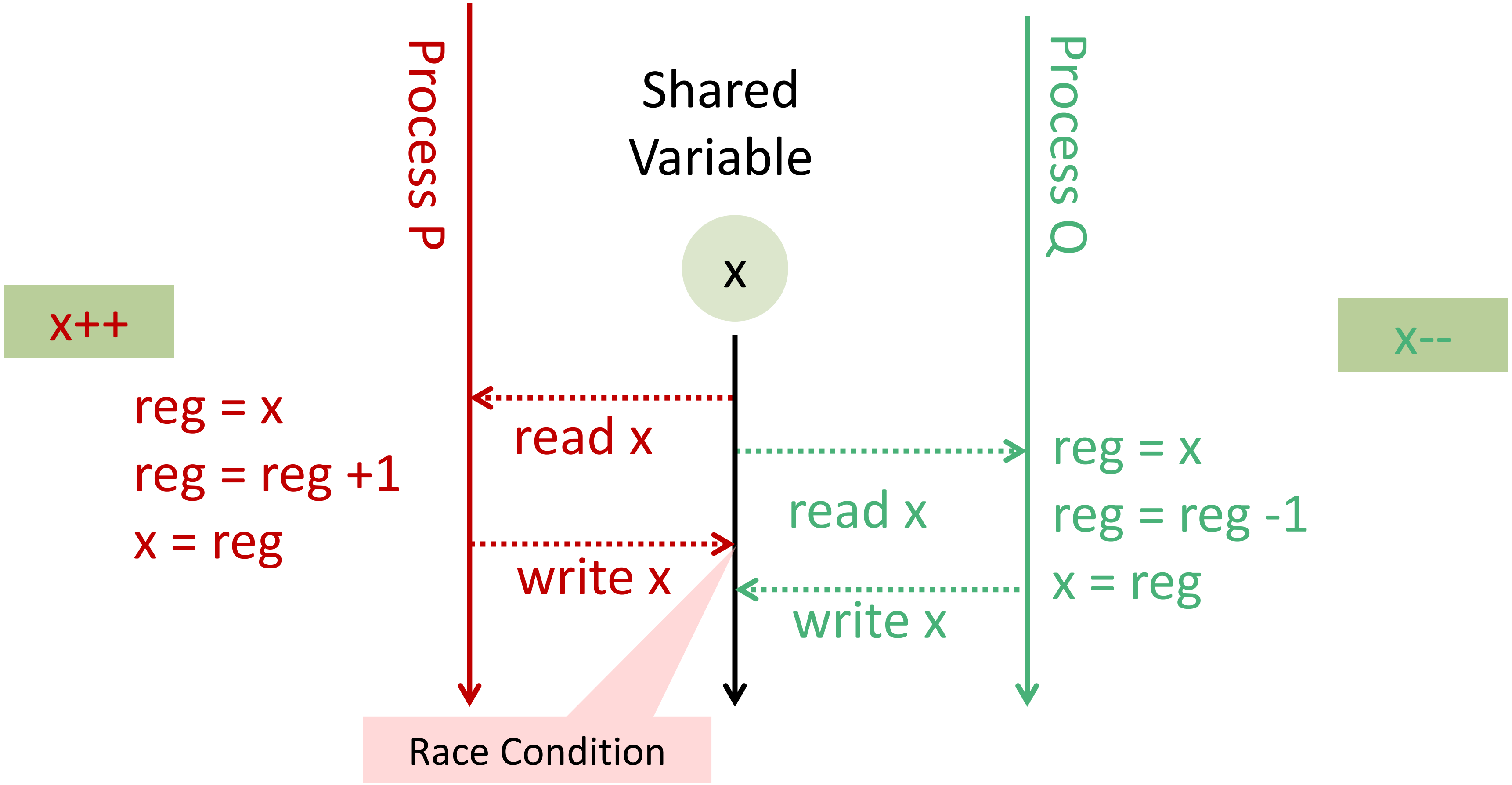
«Each of the processes eventually reaches the acquire statement"

«The barrier will be opened if and only if all processes have reached the barrier"

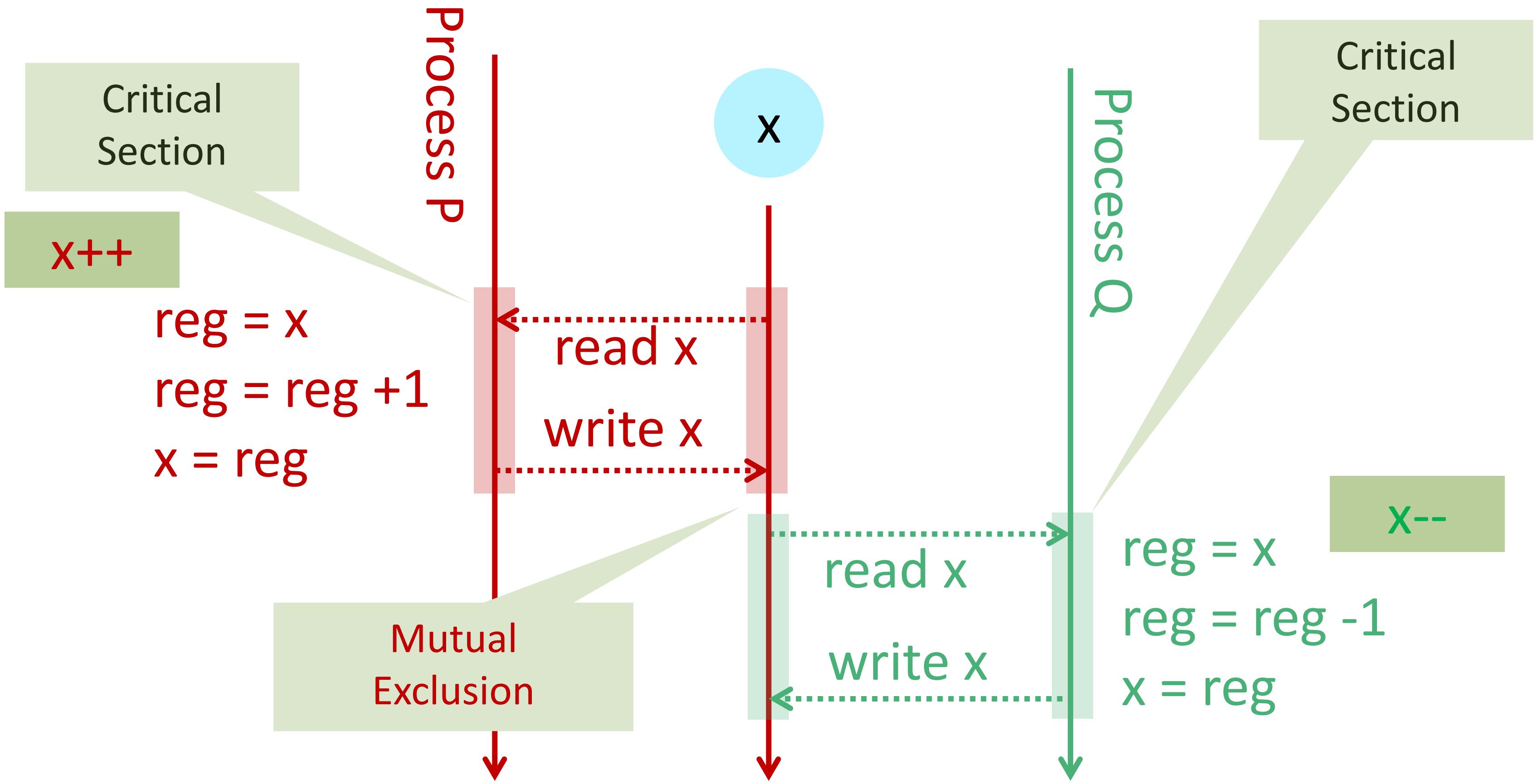
«count provides the number of processes that have passed the barrier" (violated)

«when all processes have reached the barrier then all waiting processes can continue" (violated)

# Recap: Race Condition on «count++»



# With Mutual Exclusion



# Barrier

**Synchronize a number (n) of processes.**

**Semaphores `barrier`, `mutex`. Integer `count`.**

Does this work for one iteration?

What is the value of count and barrier after the call?

| | P1 | P2 | ... | Pn |
|---|---|---|---|---|
| *init* | mutex = 1; barrier = 0; count = 0 | | | |
| *pre* | ... | | | |
| *barrier* | acquire(mutex)<br>  count++<br>release(mutex)<br>if (count==n) release(barrier)<br>acquire(barrier)<br>release(barrier) | ← | ← | ← |
| *post* | ... | | | |

$\}$ turnstile

# Reusable Barrier. 1st trial.

| | P1 | ... | Pn |
|---|---|---|---|
| *init* | `mutex = 1; barrier = 0; count = 0` | | |
| *pre* | ... | | |
| *barrier* | `acquire(mutex)`<br>`  count++`<br>`release(mutex)`<br>`if (count==n) release(barrier)`<br><br>`acquire(barrier)`<br>`release(barrier)`<br><br>`acquire(mutex)`<br>`  count--`<br>`release(mutex)`<br>`if (count==0) acquire(barrier)` | ← | ← |
| *post* | ... | | |

Dou you see the problem?

Race Condition !

Race Condition !

# Reusable Barrier. 1st trial.

| | P1 | ... | Pn |
|---|---|---|---|
| *init* | mutex = 1; barrier = 0; count = 0 | | |
| *pre* | ... | | |
| *barrier* | acquire(mutex)<br>  count++<br>release(mutex)<br>if (count==n) release(barrier)<br><br>acquire(barrier)<br>release(barrier)<br><br>acquire(mutex)<br>  count--<br>release(mutex)<br>if (count==0) acquire(barrier) | | ← |
| *post* | ... | | |

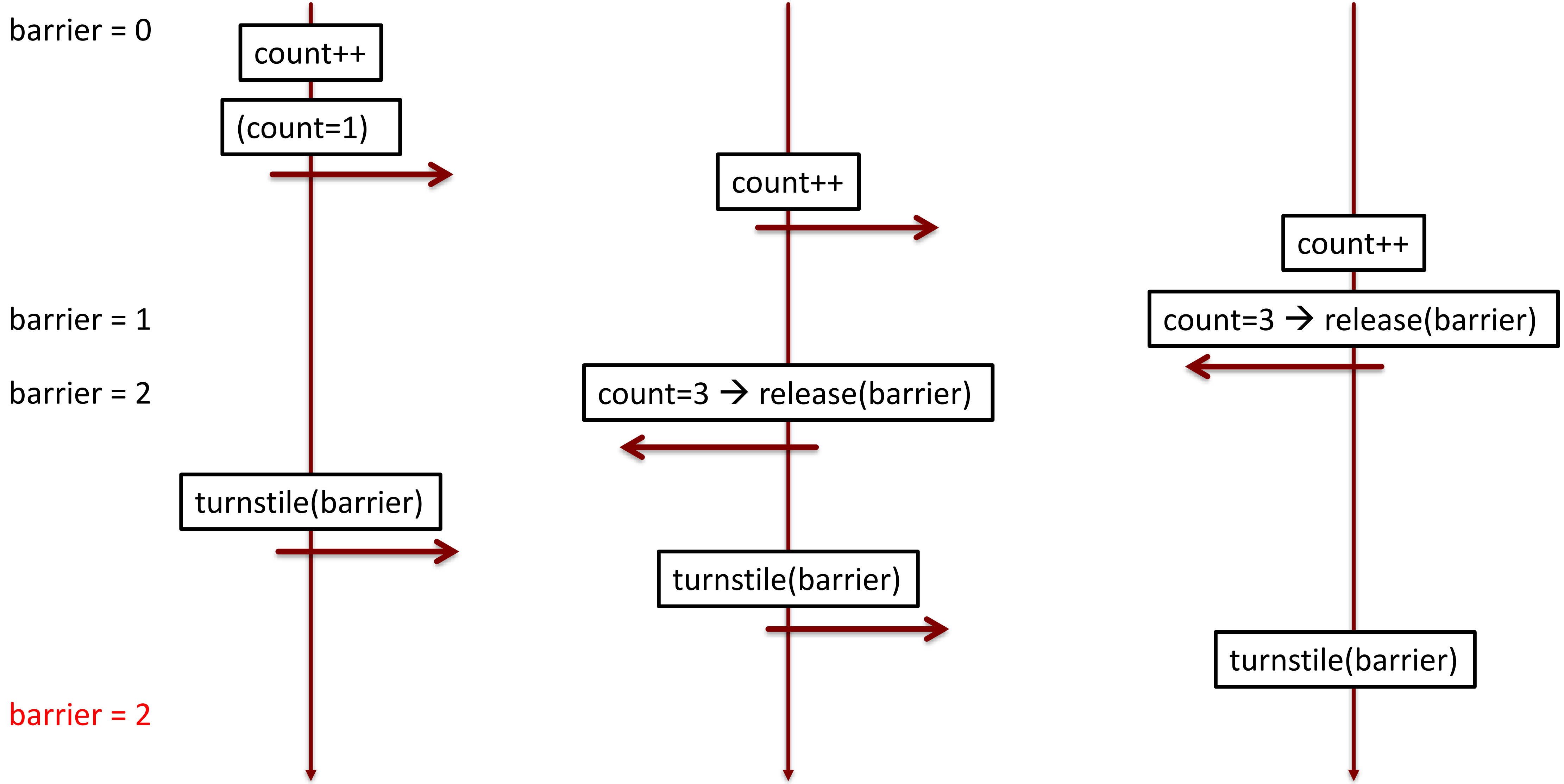
**Invariants**

«Only when all processes have reached the turnstyle it will be opened the first time"

«When all processes have run through the barrier then count = 0"

«When all processes have run through the barrier then barrier = 0" (violated)

# Illustration of the problem: scheduling scenario



# Reusable Barrier. 2nd trial.

| | P1 | ... | Pn |
|---|---|---|---|
| *init* | mutex = 1; barrier = 0; count = 0 | | |
| *pre* | ... | | |
| *barrier* | acquire(mutex)<br>  count++<br>  if (count==n) release(barrier)<br>release(mutex)<br><br>acquire(barrier)<br>release(barrier)<br><br>acquire(mutex)<br>  count--<br>  if (count==0) acquire(barrier)<br>release(mutex) | | |
| *post* | ... | | |

Dou you see the problem?

Process can pass other processes!

# Reusable Barrier. 2nd trial.

| | P1 | ... | Pn |
|---|---|---|---|
| *init* | mutex = 1; barrier = 0; count = 0 | | |
| *pre* | ... | | |
| *barrier* | acquire(mutex)<br>  count++<br>  if (count==n) release(barrier)<br>release(mutex)<br><br>acquire(barrier)<br>release(barrier)<br><br>acquire(mutex)<br>  count--<br>  if (count==0) acquire(barrier)<br>release(mutex) | | |
| *post* | ... | | |

n-1 processes here, one process cycles

**Invariants**

«When all processes have passed the barrier, it holds that barrier = 0"

« Even when a single process has passed the barrier, it holds that barrier = 0» (violated)

# Solution: Two-Phase Barrier

*init*

```
mutex=1; barrier1=0; barrier2=1; count=0
```

*barrier*

```
acquire(mutex)
  count++;
  if (count==n)
     acquire(barrier2); release(barrier1)
release(mutex)

acquire(barrier1); release(barrier1);
// barrier1 = 1 for all processes, barrier2 = 0 for all processes
acquire(mutex)
  count--;
  if (count==0)
      acquire(barrier1); release(barrier2)
release(mutex)

acquire(barrier2); release(barrier2)
// barrier1 = 0 for all processes, barrier2 = 1 for all processes
```

Of course, this is very slow in practice, see http://www.spiral.net/software/barrier.html for a specialized fast barrier for x86!

## Lesson Learned ?

- **Semaphore, Rendezvouz and Barrier:**
- **Concurrent programming is prone to errors in reasoning.**
- **A naive approach with trial and error is close-to impossible.**
- **Ways out:**
  - Identify invariants in the problem domain, ensure they hold for your implementation
  - Identify and apply established patterns
  - Use known good libraries (like in the Java API)

# Summary

**Locks are not enough: we need methods to wait for events / notifications**

**Semaphores**

**Rendezvous and Barriers**

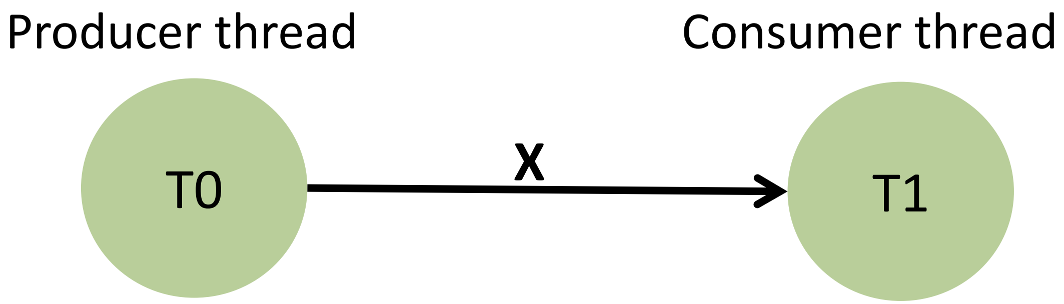
**Next:**

**Producer-Consumer Problem**

**Monitors and condition variables**

# Producer Consumer Pattern

# Producer / Consumer Pattern




**T0 computes X and passes it to T1**

**T1 uses X**

**Is synchronization for X needed?**

No because, at any point in time only one thread accesses X

we, however, need a synchronized mechanism to pass X from T0 to T1

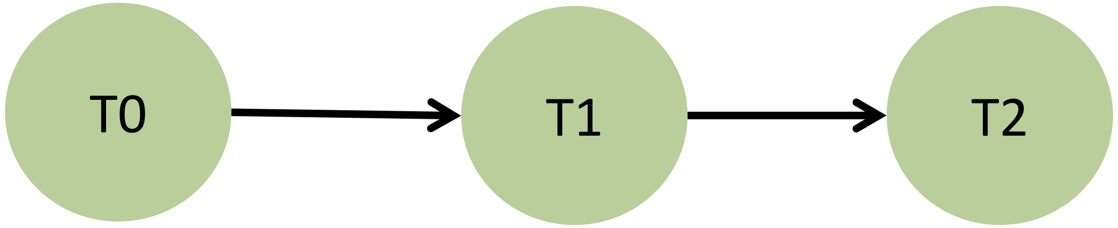
# Producer / Consumer Pattern


30 billion (30 * $10^9$) transistors, programmable at fine-grain!

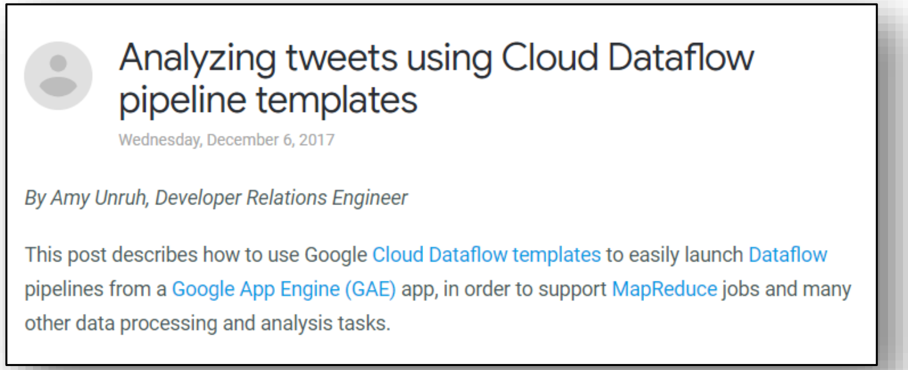
**Fundamental parallel programming pattern**

**Can be used to build data-flow parallel programs**

**E.g., pipelines:**




both producer
and consumer



Analyzing tweets using Cloud Dataflow pipeline templates

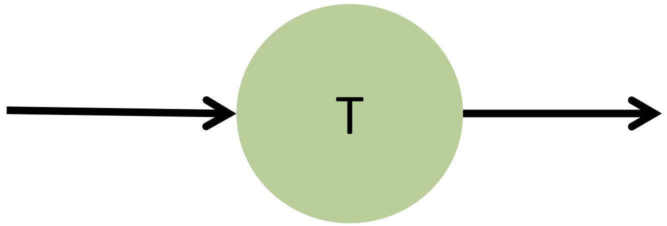
Wednesday, December 6, 2017

*By Amy Unruh, Developer Relations Engineer*

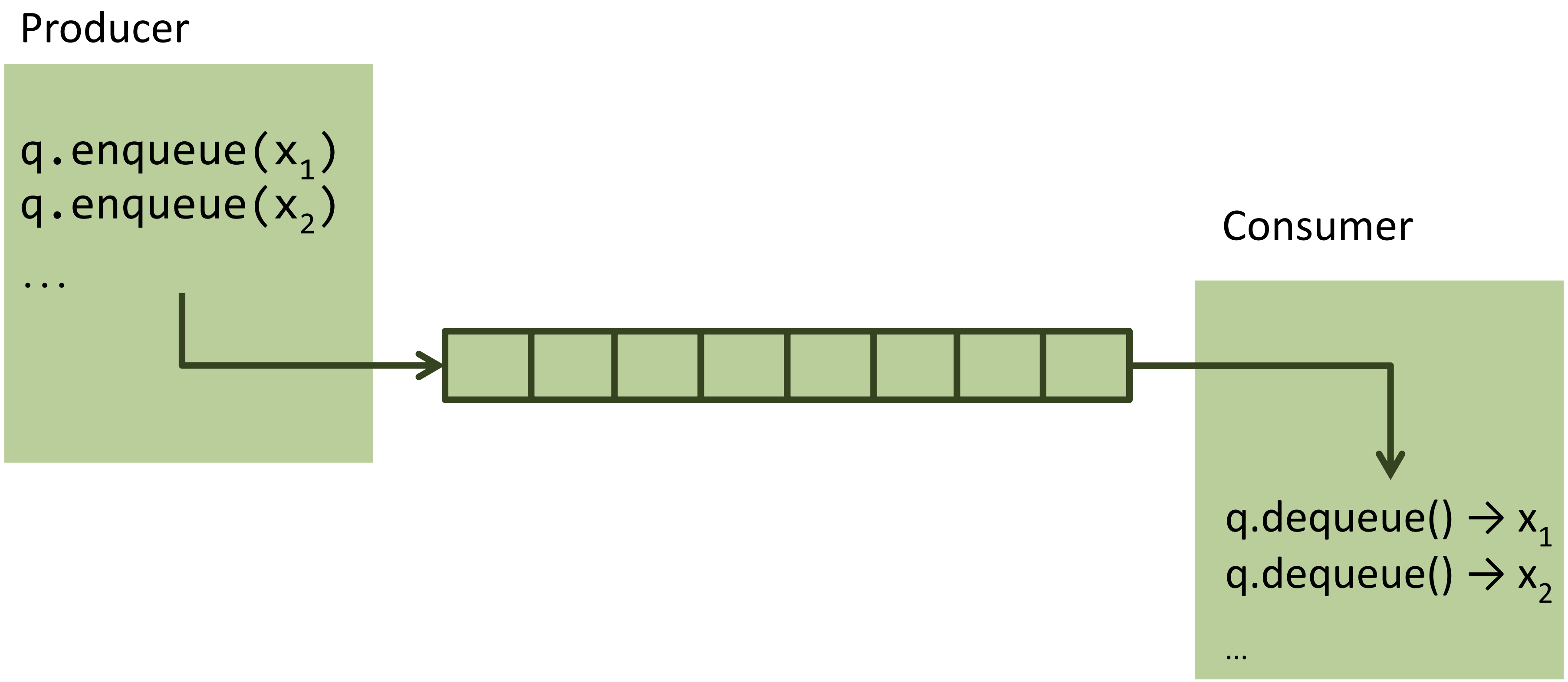
This post describes how to use Google Cloud Dataflow templates to easily launch Dataflow pipelines from a Google App Engine (GAE) app, in order to support MapReduce jobs and many other data processing and analysis tasks.

# Pipeline Node

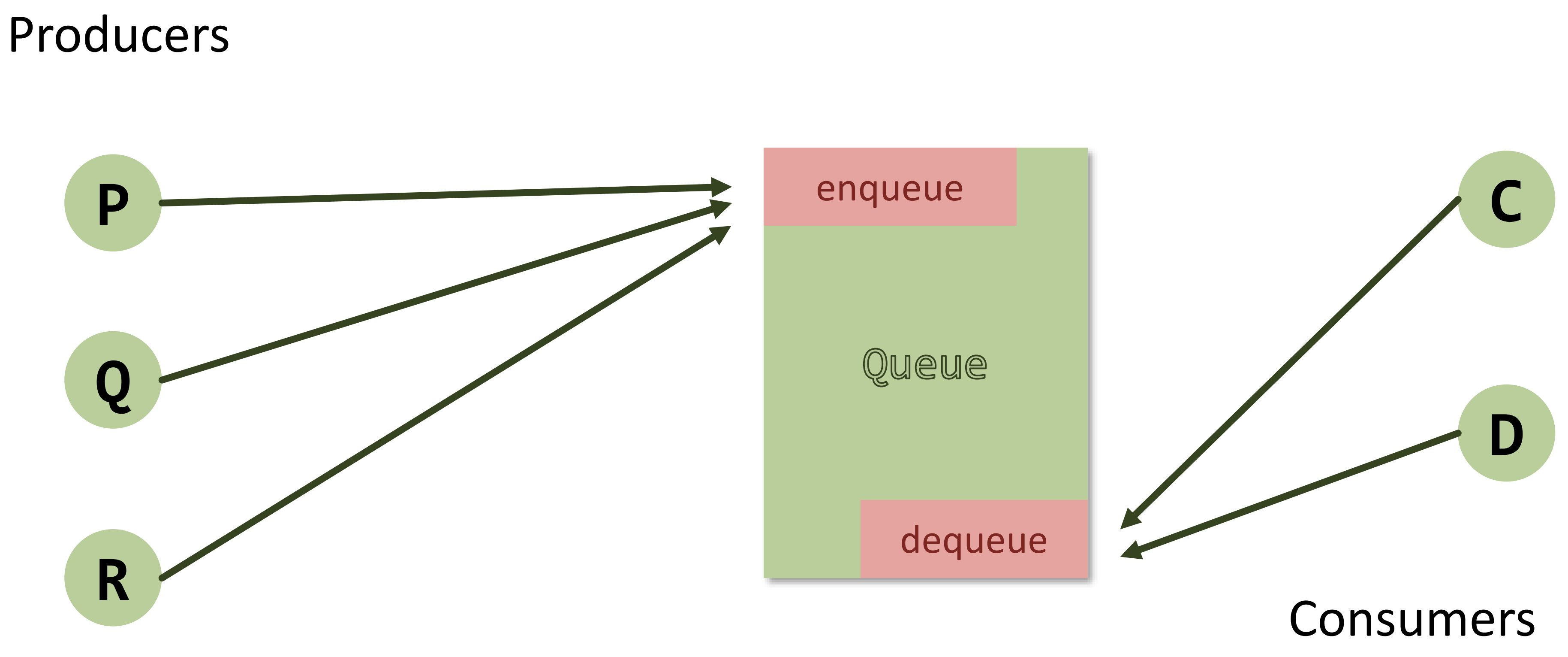
```
while (true) {
    input = q_in.dequeue();
    output = do_something(input);
    q_out.enqueue(output)
}
```



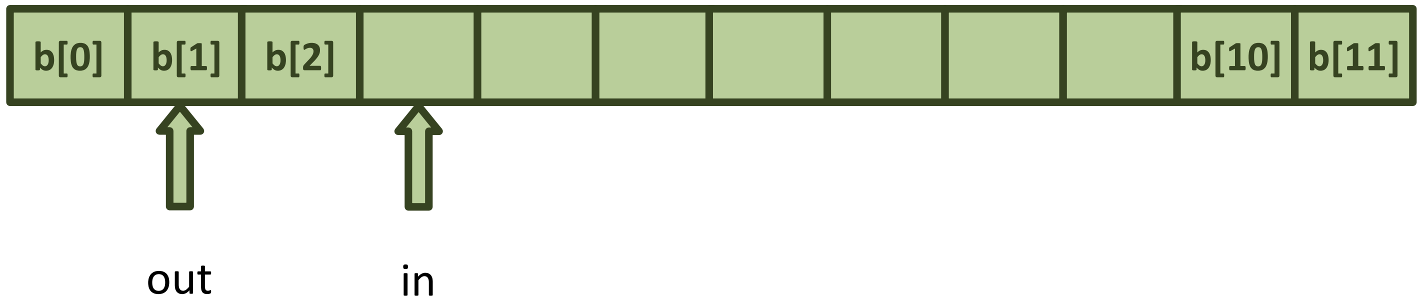
# Producer / Consumer queues



# Multiple Producers and Consumers

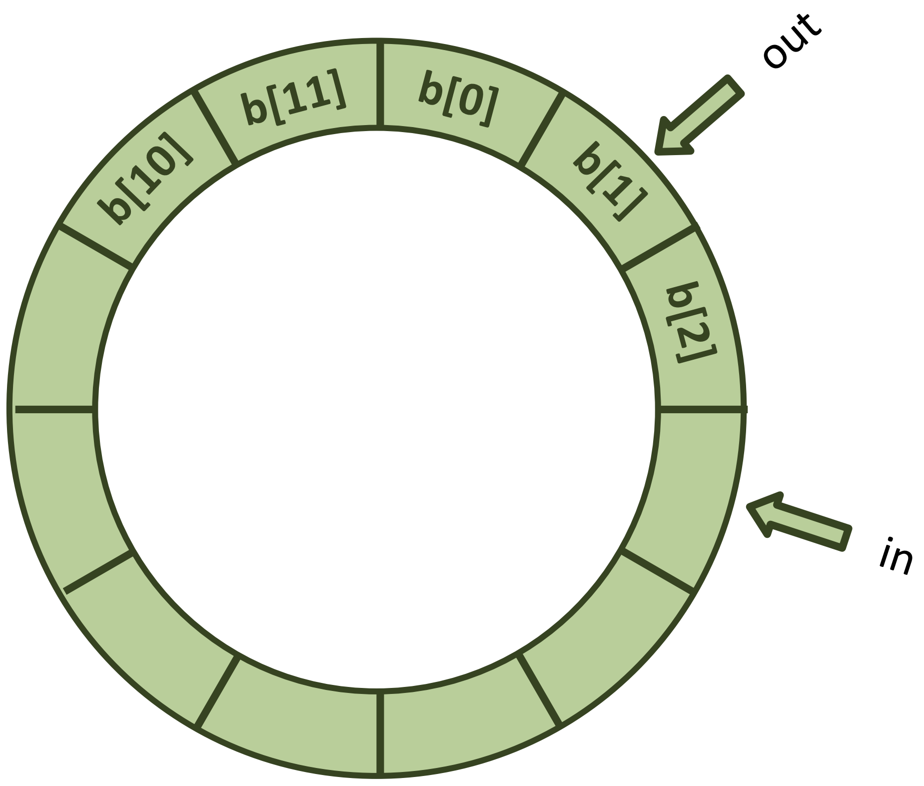


# Bounded FIFO as Circular Buffer

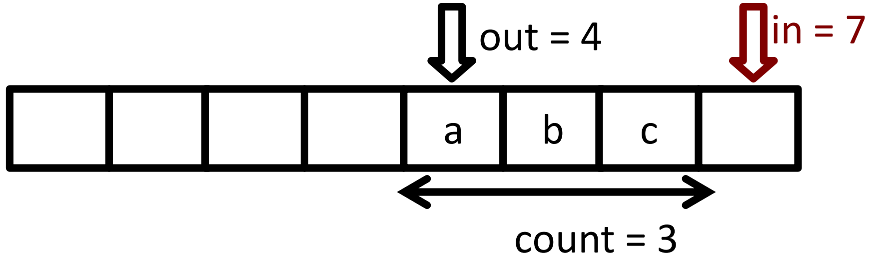



+ wrap around semantics

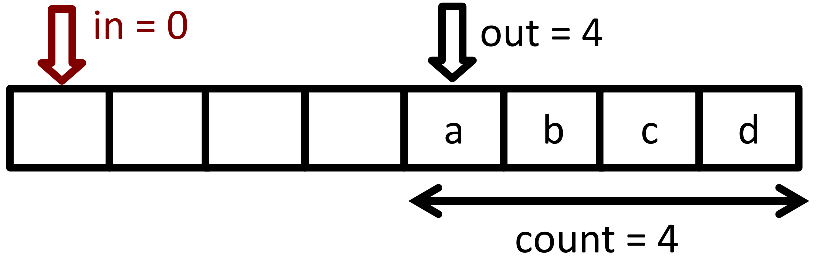
=



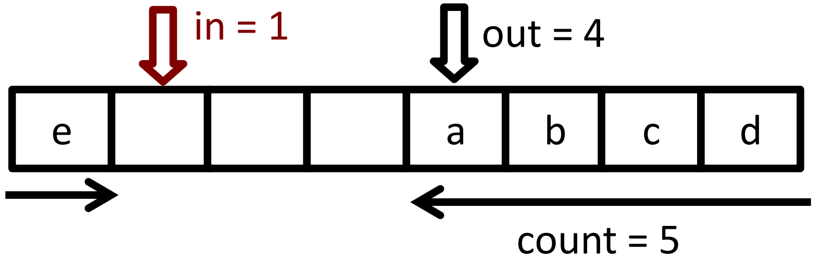
# Producer / Consumer queue implementation

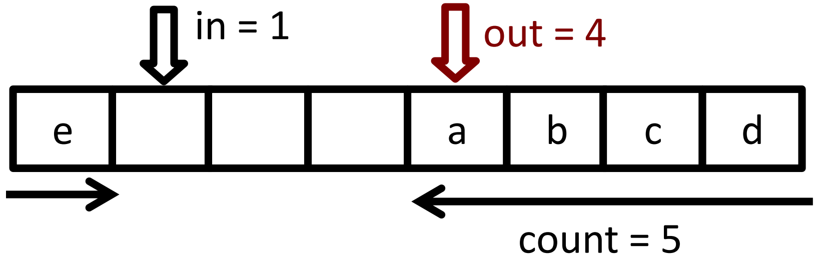



**Enqueue d**

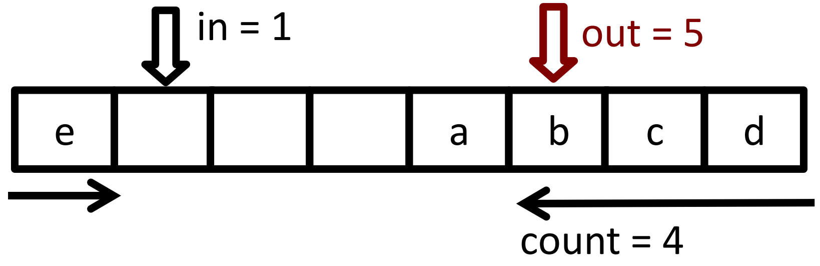



**Enqueue e**




**Dequeue → a**



# Producer / Consumer queue implementation

```
class Queue {
  private int in; // next new element
  private int out; // next element
  private int size; // queue capacity
  private long[] buffer;

  Queue(int size) {
    this.size = size;
    in = out = 0;
    buffer = new long[size];
  }

  private int next(int i) {
    return (i + 1) % size;
  }
```

```
public synchronized void enqueue(long item) {
   buffer[in] = item;
   in = next(in);
}
public synchronized long dequeue() {
   item = buffer[out];
   out = next(out);
   return item;
}
```

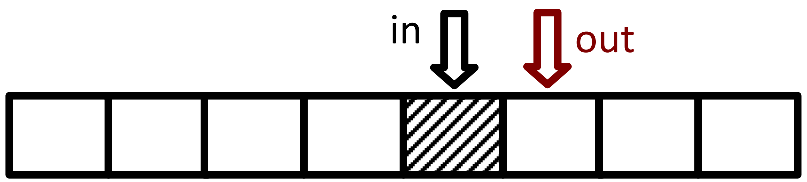
**What if we try to**

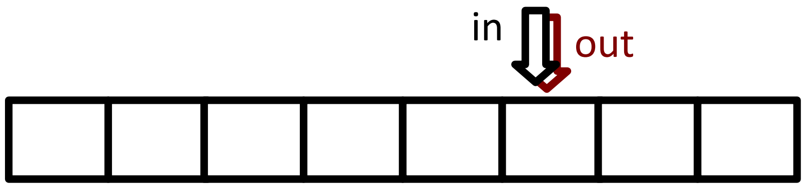
1. dequeue from an empty queue?
2. enqueue to a full queue?

# Producer / Consumer queues: helper functions

```
public void doEnqueue(long item) {
  buffer[in] = item;
  in = next(in);
}

public boolean isFull() {
  return (in+1) % size == out;
}
```

```
public long doDequeue() {
  long item = buffer[out];
  out = next(out);
  return item;
}
public boolean isEmpty() {
  return in == out;
}
```




full: one element not usable.
Still it has a benefit to not use a counter variable. Any idea what this benefit could be?

# Producer / Consumer queues

```
public synchronized void enqueue(long item) {
      while (isFull())
         ; // wait
      doEnqueue(item);
}
```

```
public void doEnqueue(long item) {
  buffer[in] = item;
  in = next(in);
}
public boolean isFull() {
  return (in+1) % size == out;
}
```

```
public synchronized long dequeue() {
      while (isEmpty())
         ; // wait
      return doDequeue();
}
```

Do you see the problem?

```
public long doDequeue() {
  long item = buffer[out];
  out = next(out);
  return item;
}
public boolean isEmpty() {
  return in == out;
}
```

**→ Blocks forever**
**infinite loops with a lock held …**

# Producer / Consumer queues using sleep()

```
public void enqueue(long item) throws InterruptedException {
  while (true) {
    synchronized(this) {
      if (!isFull()) {
        doEnqueue(item);
        return;
      }
    }
    Thread.sleep(timeout); // sleep without lock!
  }
}
```

What is the proper value for the timeout?
Ideally we would like to be notified when the change happens!
**When is that?**

# Producer / Consumer queues with semaphores

```
import java.util.concurrent.Semaphore;

class Queue {
    int in, out, size;
    long buf[];
    Semaphore nonEmpty, nonFull, manipulation;

    Queue(int s) {
        size = s;
        buf = new long[size];
        in = out = 0;
        nonEmpty = new Semaphore(0); // use the counting feature of semaphores!
        nonFull = new Semaphore(size); // use the counting feature of semaphores!
        manipulation = new Semaphore(1); // binary semaphore
    }
}
```

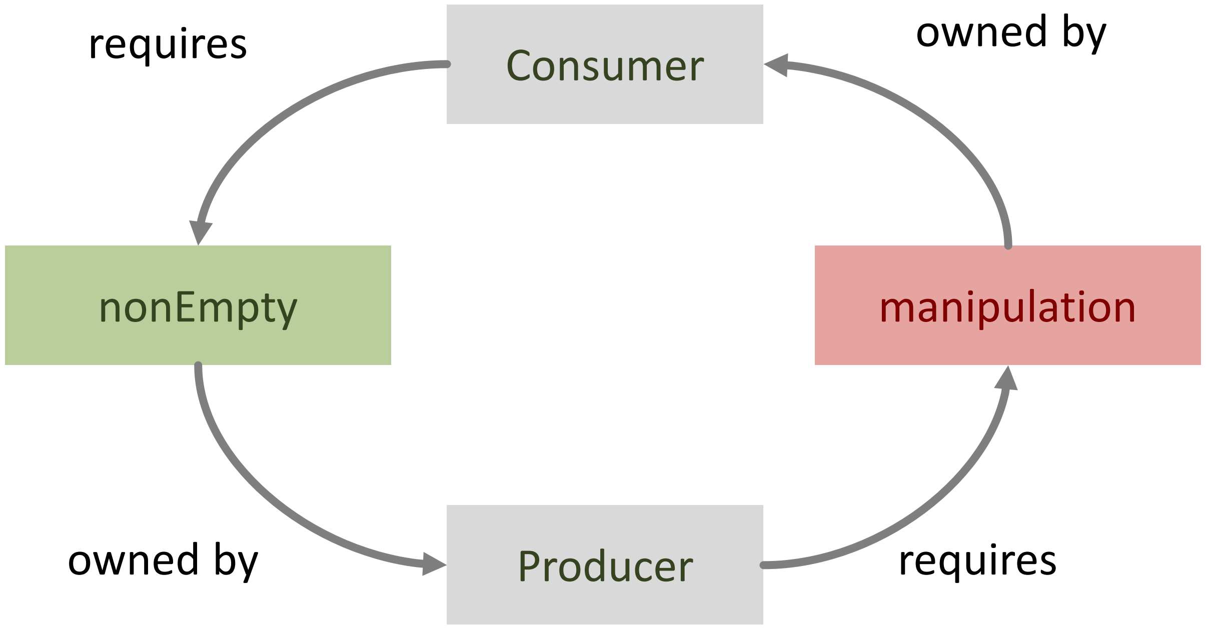
# Producer / Consumer queues with semaphores, correct?

Do you see the problem?

```
void enqueue(long x) {

    try {
        manipulation.acquire();
        nonFull.acquire();
        buf[in] = x;
        in = (in+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        manipulation.acquire();
        nonEmpty.acquire();
        x = buf[out];
        out = (out+1) % size;
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

# Deadlock (nearly the same as before, actually)!



# Producer / Consumer queues with semaphores

```
void enqueue(long x) {

    try {
        nonFull.acquire();
        manipulation.acquire();
        buf[in] = x;
        in = next(in);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonEmpty.release();
    }
}
```

```
long dequeue() {
    long x=0;
    try {
        nonEmpty.acquire();
        manipulation.acquire();
        x = buf[out];
        out = next(out);
    }
    catch (InterruptedException ex) {}
    finally {
        manipulation.release();
        nonFull.release();
    }
    return x;
}
```

# Why are semaphores (and locks) problematic?

Semaphores are unstructured. Correct use requires high level of discipline.

Easy to introduce deadlocks with semaphores.

We need: a lock that we can temporarily escape from when waiting on a condition.

# Monitors

# Monitors

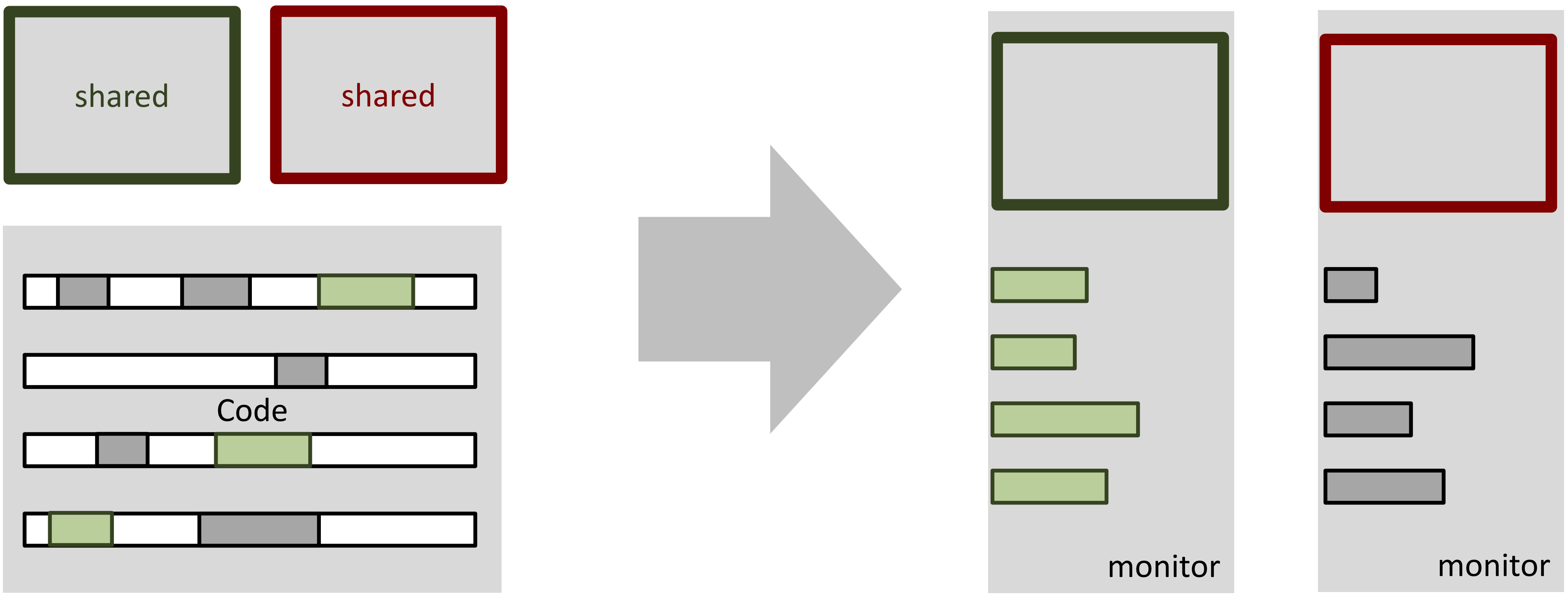
**Monitor:**
**abstract data structure equipped with a set of operations that run in mutual exclusion.**

**Invented by Tony Hoare and Per Brinch Hansen (cf. Monitors: An Operating System Structuring Concept, Tony Hoare, 1974)**

Tony Hoare
(1934-today)

Per Brinch Hansen
(1938-2007)

# Monitors vs. Semaphores/Unbound Locks



# Producer / Consumer queues

```
public void synchronized enqueue(long item) {
   "while (isFull()) wait"
   doEnqueue(item);
}
```

```
public long synchronized dequeue() {
   "while (isEmpty()) wait"
   return doDequeue();
}
```

The mutual exclusion part is nicely available already.
**But: while the buffer is full we need to give up the lock, how?**

# Monitors

Monitors provide, in addition to mutual exclusion, a mechanism to check conditions with the following semantics:

If a condition does not hold

- Release the monitor lock
- Wait for the condition to become true
- Signaling mechanism to avoid busy-loops (spinning)

# Monitors in Java

**Uses the intrinsic lock (`synchronized`) of an object**

**+ `wait / notify / notifyAll`:**

- `wait()` – the current thread waits until it is signaled (via notify)
- `notify()` – wakes up *one* waiting thread (an arbitrary one)
- `notifyAll()` – wakes up *all* waiting threads

# Producer / Consumer with monitor in Java

```
class Queue {
      int in, out, size;
      long buf[];

      Queue(int s) {
            size = s;
            buf = new long[size];
            in = out = 0;
      }
      ...
}
```

# Producer / Consumer with monitor in Java

```
synchronized void enqueue(long x) {

   while (isFull())
      try {
         wait();
      } catch (InterruptedException e) { }
   doEnqueue(x);
   notifyAll();
}
```

```
synchronized long dequeue() {
    long x;
    while (isEmpty())
        try {
            wait();
        } catch (InterruptedException e) { }
    x = doDequeue();
    notifyAll();
    return x;
}
```

Wouldn't an if be sufficient?

(Why) can't we use notify()?

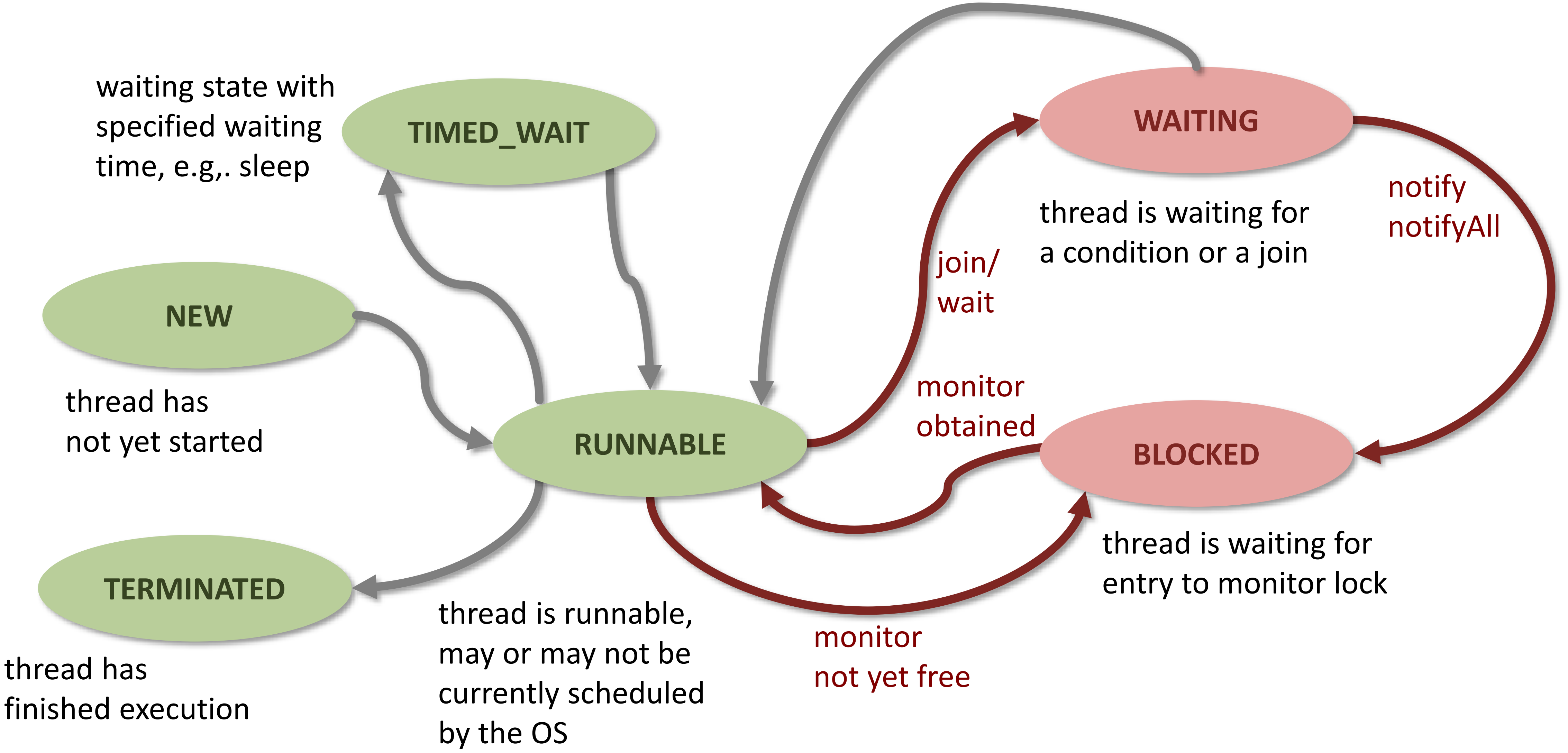
# IMPORTANT TO KNOW JAVA MONITOR IMPLEMENTATION DETAILS

# Thread States in Java

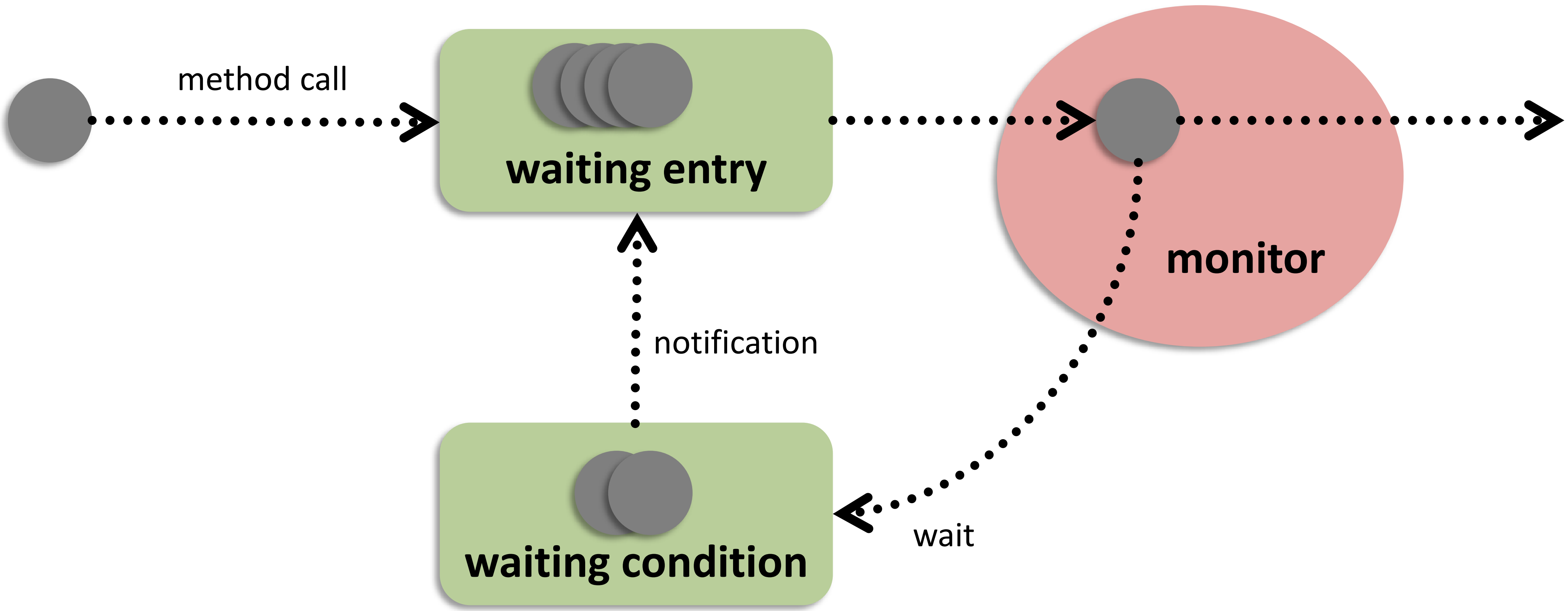
Spaced repetition

From Wikipedia, the free encyclopedia

**Spaced repetition** is a learning technique that incorporates increasing intervals of time between subsequent review of previously learned material in order to exploit the psychological spacing effect. Alternative names include *spaced rehearsal*, *expanding rehearsal*, *graduated intervals*, *repetition spacing*, *repetition scheduling*, *spaced retrieval* and *expanded retrieval*.[1]



# Monitor Queues



# Various (exact) semantics possible

Important to know for the programmer (you): what happens upon notification? Priorities?

signal and wait

- signaling process exits the monitor (goes to waiting entry queue)
- signaling process passes monitor lock to signaled process

signal and continue

- signaling process continues running
- signaling process moves signaled process to waiting entry queue

other semantics: signal and exit, signal and urgent wait …

# Why is this important? Let's try this implementing a semaphore:

```
class Semaphore {
   int number = 1; // number of threads allowed in critical section

   synchronized void acquire() {
      if (number <= 0)
         try { wait();   } catch (InterruptedException e) { };
      number--;
   }

   synchronized void release() {
      number++;
      if (number > 0)
         notify();
   }
}
```

Looks good, doesn't it?
But there is a problem.
Do you know which?

# Java Monitors = signal + continue

```
R synchronized void acquire() {
    if (number <= 0)
       try { wait(); } Q
       catch (InterruptedException e) { };
    number--;
  }

  synchronized void release() {
P   number++;
    if (number > 0)
       notify();
  }
```

**Scenario:**

1. Process P has previously acquired the semaphore and decreased number to 0.
2. Process Q sees number = 0 and goes to waiting list.
3. P is executing release. In this moment process R wants to enter the monitor via method acquire.
4. P signals Q and thus moves it into wait entry list (signal and continue!). P exits the function/lock.
5. R gets entry to monitor before Q and sees the number = 1
6. Q continues execution with number = 0!

**Inconsistency!**

# The cure – a while loop.

```
synchronized void acquire() {
 while (number <= 0)
  try { wait(); }
   catch (InterruptedException e) { };
 number--;
}
```

```
synchronized void release() {
   number++;
   if (number > 0)
      notify();
}
```

**If, additionally, different threads evaluate different conditions, the notification has to be a `notifyAll`. In this example it is not required.**

# Something different: Java Interface Lock

**Intrinsic locks ("synchronized") with objects provide a good abstraction and should be first choice**

**Limitations**

- **one implicit lock per object**
- **are forced to be used in blocks**
- **limited flexibility**

**Java offers the Lock interface for more flexibility (e.g., lock can be polled).**

```
final Lock lock = new ReentrantLock();
```

# Condition interface

**Java Locks provide *conditions that can be instantiated***

```
Condition notFull  = lock.newCondition();
```

**Java conditions offer**

**.await() – the current thread waits until condition is signaled**

**.signal() – wakes up one thread *waiting on this condition***

**.signalAll() – wakes up all threads *waiting on this condition***

# Condition interface

→ **Conditions are always associated with a lock**
lock.newCondition()

**.await()**
- called with the lock held
- **atomically** releases the lock and waits until thread is signaled
- When it returns, it is **guaranteed** to hold the lock
- thread **always** needs to check condition

**.signal{,All}() – wakes up one (all) waiting thread(s)**
- called with the lock held

# Producer / Consumer with explicit Lock

```
class Queue {
     int in=0, out=0, size;
     long buf[];
     final Lock lock = new ReentrantLock();
     final Condition notFull  = lock.newCondition();
     final Condition notEmpty = lock.newCondition();

     Queue(int s) {
          size = s;
          buf = new long[size];
     }

...

}
```

# Producer / Consumer with Lock

```
void enqueue(long x) {

   lock.lock();
   while (isFull())
      try {
         notFull.await();
      } catch (InterruptedException e){}
   doEnqueue(x);
   notEmpty.signal();
   lock.unlock();
}
```

```
long dequeue() {
   long x;
   lock.lock();
   while (isEmpty())
      try {
         notEmpty.await();
      } catch (InterruptedException e){}
   x = doDequeue();
   notFull.signal();
   lock.unlock();
   return x;
}
```

# The Sleeping Barber Variant (E. Dijkstra)

**Disadvantage of the solution: notFull and notEmpty signal will be sent in any case, even when no threads are waiting.**

**Seemingly simple solution (in barber analogy)**

1. ***Barber cuts hair, when done, check waiting room, if nobody left, sleep***
2. ***Client arrives, either enqueues or wakes sleeping barber***

**What can go wrong (really only in a threaded world)?**

***Sleeping barber*** **requires additional counters for checking if processes are waiting:**

$m \leq 0 \Leftrightarrow$ **buffer full &** $-m$ **producers (clients) are waiting**

$n \leq 0 \Leftrightarrow$ **buffer empty &** $-n$ **consumers (barbers) are waiting**

# Producer Consumer, Sleeping Barber Variant

```
class Queue {
    int in=0, out=0, size;
    long buf[];
    final Lock lock = new ReentrantLock();
    int n = 0; final Condition notFull  = lock.newCondition();
    int m; final Condition notEmpty = lock.newCondition();

    Queue(int s) {
        size = s; m=size-1;
        buf = new long[size];
    }
...
}
```

Two variables ☹ sic!
(cf. last lecture)

# Producer Consumer, Sleeping Barber Variant

```
void enqueue(long x) {

   lock.lock();
   m--; if (m<0)
      while (isFull())
         try { notFull.await(); }
         catch(InterruptedException e){}
   doEnqueue(x);
   n++;
   if (n<=0) notEmpty.signal();
   lock.unlock();

}
```

```
long dequeue() {
   long x;
   lock.lock();
   n--; if (n<0)
      while (isEmpty())
         try { notEmpty.await(); }
         catch(InterruptedException e){}
   x = doDequeue();
   m++;
   if (m<=0) notFull.signal();
   lock.unlock();
   return x;
}
```

# Guidelines for using condition waits

- **Always have a condition predicate**
- **Always test the condition predicate:**
  - before calling wait
  - after returning from wait
- **Always call wait in a loop**
- **Ensure state is protected by lock associated with condition**

# Check out java.util.concurrent

**Java (luckily for us) provides many common synchronization objects:**

- Semaphores
- Barriers (CyclicBarrier)
- Producer / Consumer queues
- and many more... (Latches, Futures, ...)

# Reader / Writer Locks

## Literature: Herlihy – Chapter 8.3

# Reading vs. writing

**Recall:**

- **Multiple concurrent reads of same memory: *Not* a problem**
- **Multiple concurrent writes of same memory: Problem**
- **Multiple concurrent read & write of same memory: Problem**

**So far:**

- **If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time**

**But this is unnecessarily conservative:**

- **Could still allow multiple simultaneous readers!**

# Example

**Consider a hashtable with one coarse-grained lock**

- So only one thread can perform operations at a time

**But suppose:**

- There are many simultaneous **`lookup`** operations
- **`insert`** operations are very rare

**Note: Important that `lookup` does not actually mutate shared memory, like a move-to-front list operation would**

WIKIPEDIA
The Free Encyclopedia

Number of edits (2007-11/27/2017): 921,644,695
Average views per day: ~200,000,000

→ 0.12% write rate

# Another Example

**Shared use of text, e.g., in an IDE**

**writers: editor(s), copy&paste agents, syntax highlighter**

**readers: compiler, editor(s), text viewers, copy&paste agents, search tools**

# Reader/writer locks

**A new abstract data type for synchronization : The reader/writer lock**

**This lock's states fall into three categories:**

- **"not held"**
- **"held for writing" by one thread**
- **"held for reading" by *one or more* threads**

$0 \leq \text{writers} \leq 1$

$0 \leq \text{readers}$

$\text{writers} * \text{readers} == 0$

# Reader/writer locks

| | |
|---|---|
| `new:` | make a new lock, initially "not held" |
| `acquire_write:` | block if currently "held for reading" or "held for writing", else make "held for writing" |
| `release_write:` | make "not held" |
| `acquire_read:` | block if currently "held for writing", else make/keep "held for reading" and increment *readers count* |
| `release_read:` | decrement readers count, if 0, make "not held" |

# Pseudocode example

```
class Hashtable<K,V> {
 …
 // coarse-grained, one lock for table
 RWLock lk = new RWLock();
 …
```

```
 V lookup(K key) {
   int bucket = hashval(key);
   lk.acquire_read();
   … read array[bucket] …
   lk.release_read();
 }
```

```
 void insert(K key, V val) {
   int bucket = hashval(key);
   lk.acquire_write();
   … write V to array[bucket] …
   lk.release_write();
 }
 …
}
```

# A Simple Monitor-based Implementation

```
class RWLock {
  int writers = 0;
  int readers = 0;

synchronized void acquire_read() {
  while (writers > 0)
    try { wait(); }
    catch (InterruptedException e) {}
  readers++;
}

synchronized void release_read() {
  readers--;
  notifyAll();
}
```

**Is this lock fair?**

The simple implementation gives priority to readers:

- when a reader reads, other readers can enter
- no writer can enter during readers reading

```
  synchronized void acquire_write() {
    while (writers > 0 || readers > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    writers++;
  }

  synchronized void release_write() {
    writers--;
    notifyAll();
  }
}
```

**Exercise: come up with a better performing version using condition variables!**

# Strong priority to the writers

```
class RWLock {
  int writers = 0;
  int readers = 0;
  int writersWaiting = 0;

  synchronized void acquire_read() {
    while (writers > 0 || writersWaiting > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    readers++;
  }

  synchronized void release_read() {
    readers--;
    notifyAll();
  }
```

```
  synchronized void acquire_write() {
    writersWaiting++;
    while (writers > 0 || readers > 0)
       try { wait(); }
      catch (InterruptedException e) {}
    writersWaiting--;
    writers++;
  }

  synchronized void release_write() {
   writers--;
   notifyAll();
  }
}
```


Is this lock now fair?
(this was just to see of you're awake)

# A fair(er) model

**What is fair in this context?**

**For example**

- When a writer finishes, a number k of currently waiting readers may pass.
- When the k readers have passed, the next writer may enter (if any), otherwise further readers may enter until the next writer enters (who has to wait until current readers finish).

# A fair(er) model

writers: # writers in CS
readers: # readers in CS
writersWaiting: # writers trying to enter CS
readersWaiting: # readers trying to enter CS
writersWait: # readers the writers have to wait

```
class RWLock{
  int writers = 0; int readers = 0;
  int writersWaiting = 0; int readersWaiting = 0;
  int writersWait = 0;

  synchronized void acquire_read() {
   readersWaiting++;
   while (writers > 0 ||
         (writersWaiting > 0 && writersWait <= 0))
      try { wait(); }
      catch (InterruptedException e) {}
   readersWaiting--;
   writersWait--;
   readers++;
  }

  synchronized void release_read() {
   readers--;
   notifyAll();
  }
```

Writers are waiting and the readers don't have priority any more.

When a writer finishes, the number of currently waiting readers may pass.

Writers have to wait until the waiting readers have finished.

```
  synchronized void acquire_write() {
   writersWaiting++;
   while (writers > 0 || readers > 0 || writersWait > 0)
      try { wait(); }
     catch (InterruptedException e) {}
   writersWaiting--;
   writers++;
  }

  synchronized void release_write() {
   writers--;
   writersWait = readersWaiting;
   notifyAll();
  }
}
```

**Exercise: come up with a better performing version using condition variables! Introduce an upper bound of k readers!**

# Reader/writer lock details

## A reader/writer lock implementation ("not our problem") usually gives *priority* to writers:

- Once a writer blocks, no readers *arriving later* will get the lock before the writer
- Otherwise an `insert` could *starve*

## Re-entrant?

- Mostly an orthogonal issue
- But some libraries support *upgrading* from reader to writer

# In Java

**Java's `synchronized` statement does not support readers/writer**

**Instead, library**

`java.util.concurrent.locks.ReentrantReadWriteLock`

**Different interface: methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods**

**Does *not* have writer priority or reader-to-writer upgrading**

- Always read the documentation

# LOCK GRANULARITY

Literature: Herlihy – Chapter 9

# The Five-Fold Path

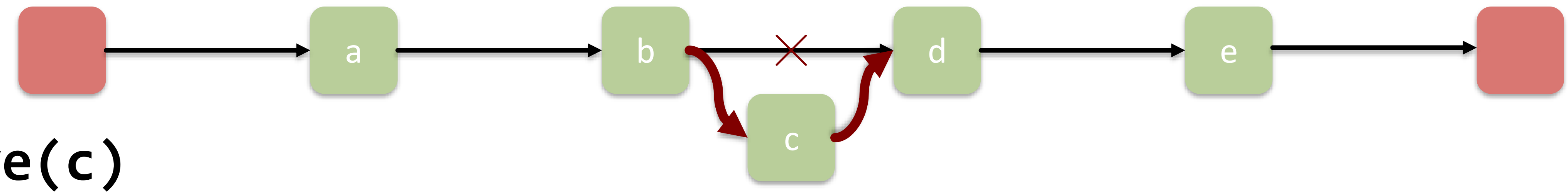
- Coarse-grained locking
- Fine-grained locking
- Optimistic synchronization (locking)
- Lazy synchronization (locking)

- Next lecture: Lock-free synchronization

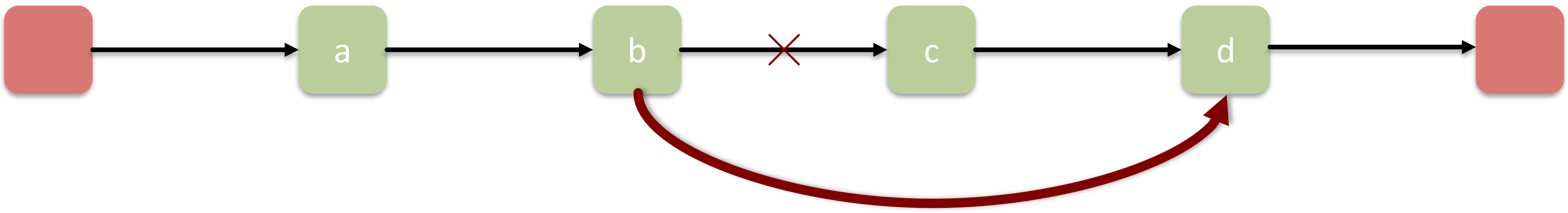
# Running Example: Sequential List Based Set

## Add, Remove, and Find unique elements in a sorted linked list.

- `add(c)`

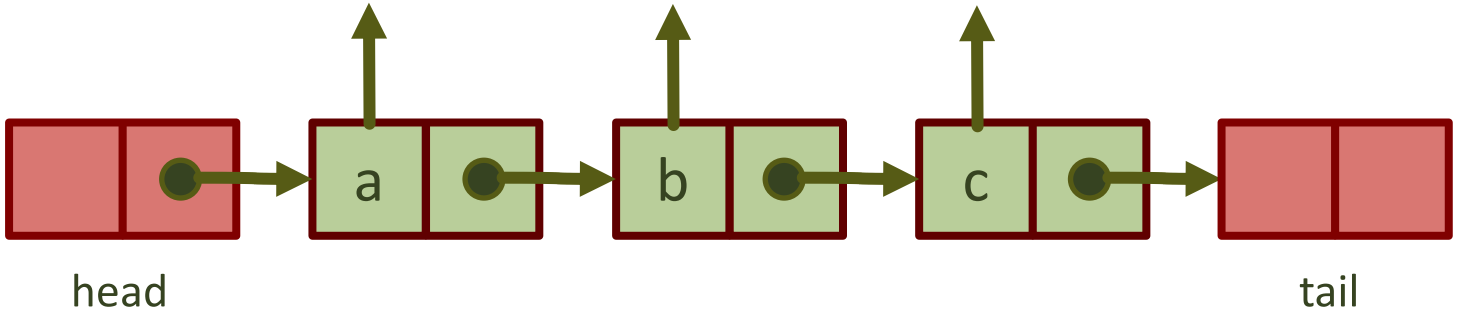



- `remove(c)`



# Set and Node

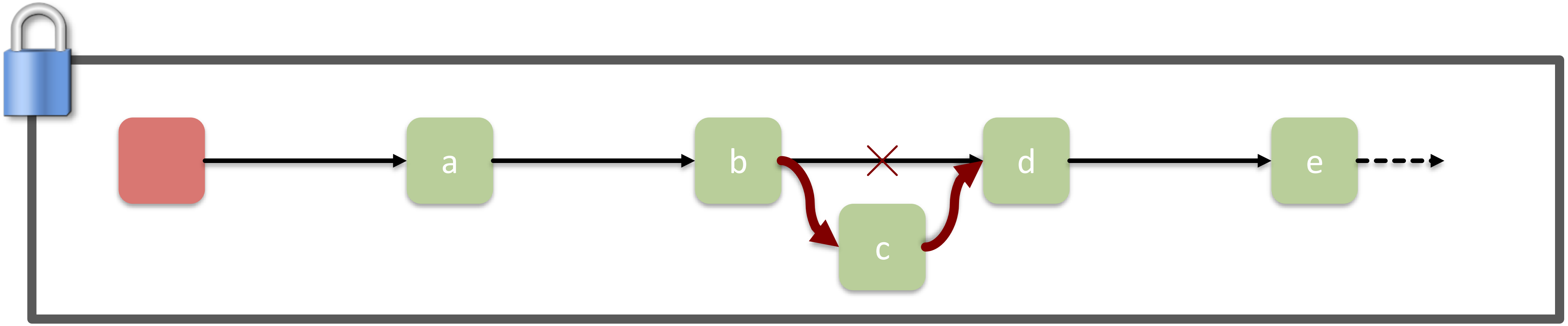
```
public class Set<T> {

   private class Node {
      T item;
      int key;
      Node next;
   }
   private Node head;
   private Node tail;

   public boolean add(T x) {...};
   public boolean remove(T x) {...};
   public boolean contains(T x) {...};
}
```




Note that the list is not "in place" but provides references to its items

# Coarse Grained Locking

```
public synchronized boolean add(T x) {...};
public synchronized boolean remove(T x) {...};
public synchronized boolean contains(T x) {...};
```




**Simple, but a bottleneck for all threads.**

# Fine grained Locking

**Often more intricate than visible at a first sight**

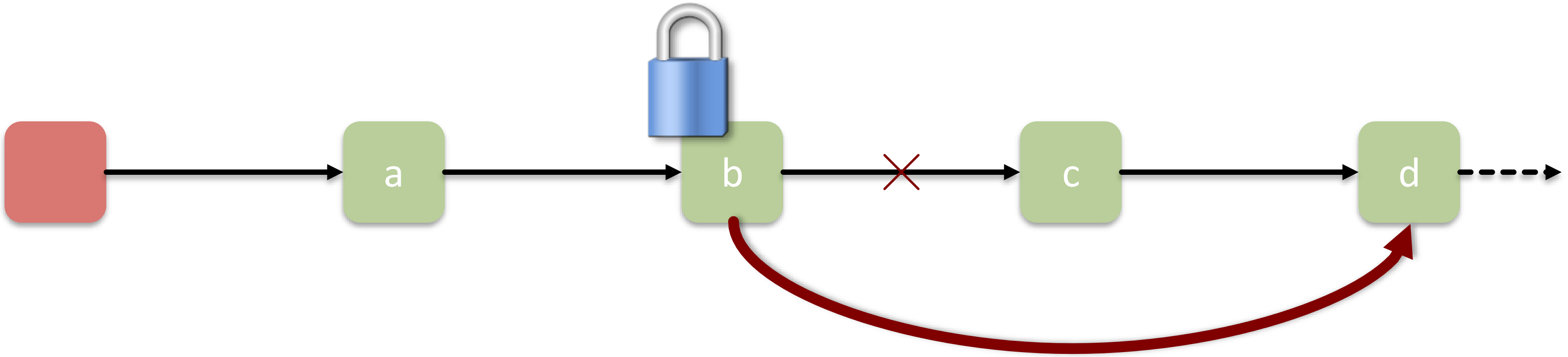
- requires careful consideration of special cases

**Idea: split object into pieces with separate locks**

- no mutual exclusion for algorithms on disjoint pieces

# Let's try this

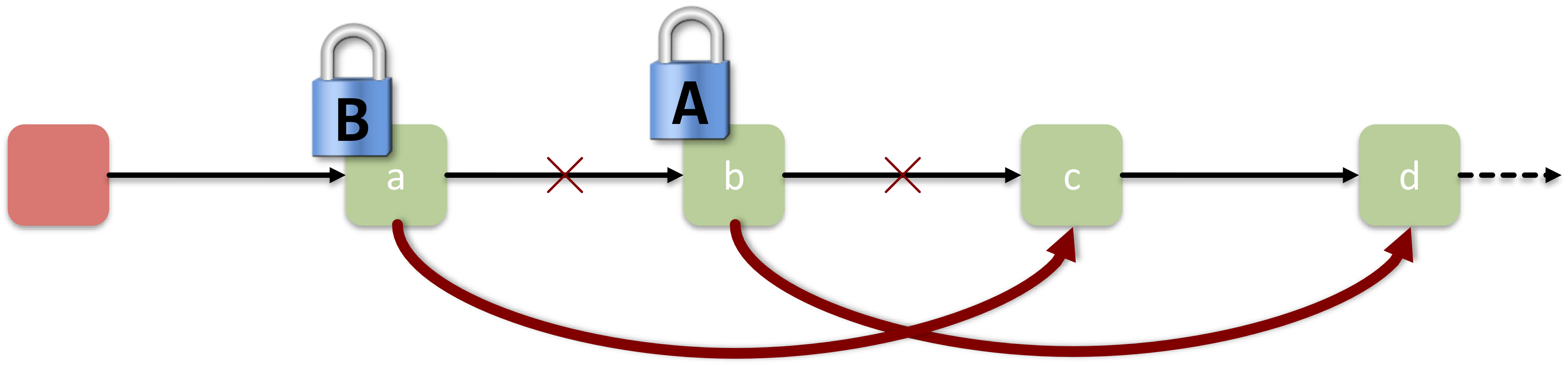
`remove(c)`




`Is this ok?`

# Let's try this

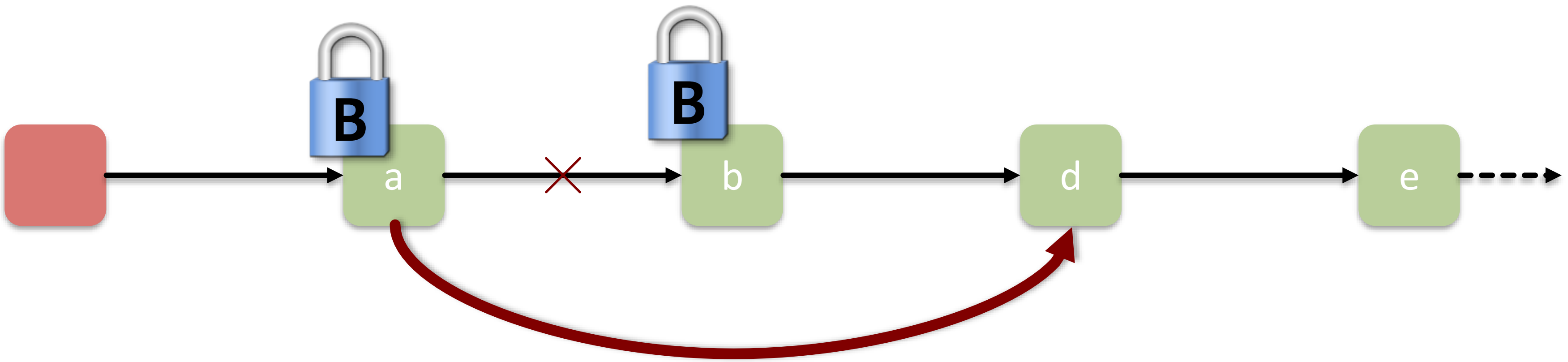
```
Thread A: remove(c)
Thread B: remove(b)
```




**c not deleted! ☹**

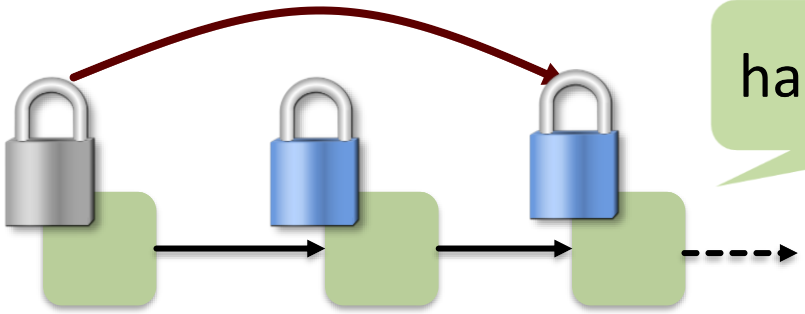
# What's the problem?

- **When deleting, the next field of next is read, i.e., next also has to be protected.**
- **A thread needs to lock both, predecessor and the node to be deleted (hand-over-hand locking).**



# Remove method

```
public boolean remove(T item) {
   Node pred = null, curr = null;
   int key = item.hashCode();
   head.lock();
   try {
    pred = head;
    curr = pred.next;
    curr.lock();
    try {
       // find and remove
    } finally { curr.unlock(); }
   } finally { pred.unlock(); }
}
```



hand over hand

```
while (curr.key < key) {
    pred.unlock();
    pred = curr; // pred still locked
    curr = curr.next;
    curr.lock(); // lock hand over hand
}
if (curr.key == key) {
    pred.next = curr.next; // delete
    return true;
}
return false;
```

remark: sentinel at front and end of list prevents an exception here

# Disadvantages?

- Potentially long sequence of acquire / release before the intended action can take place
- One (slow) thread locking "early nodes" can block another thread wanting to acquire "late nodes"

# OPTIMISTIC SYNCHRONIZATION

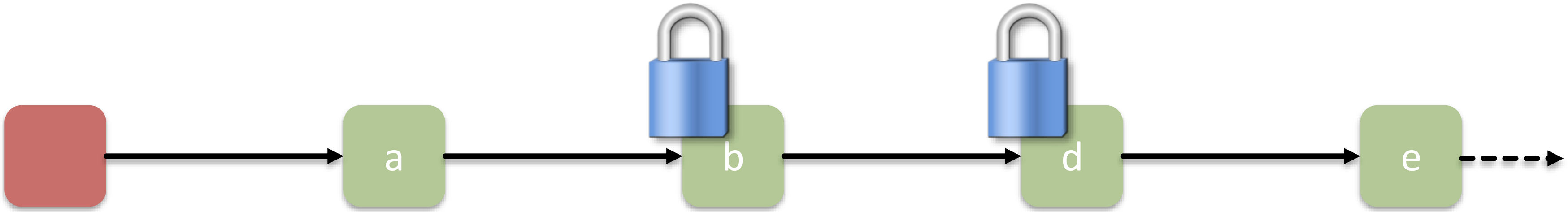
# Idea

## Find nodes without locking,

- then lock nodes and
- check that everything is ok (validation)

What do we need to "validate"?

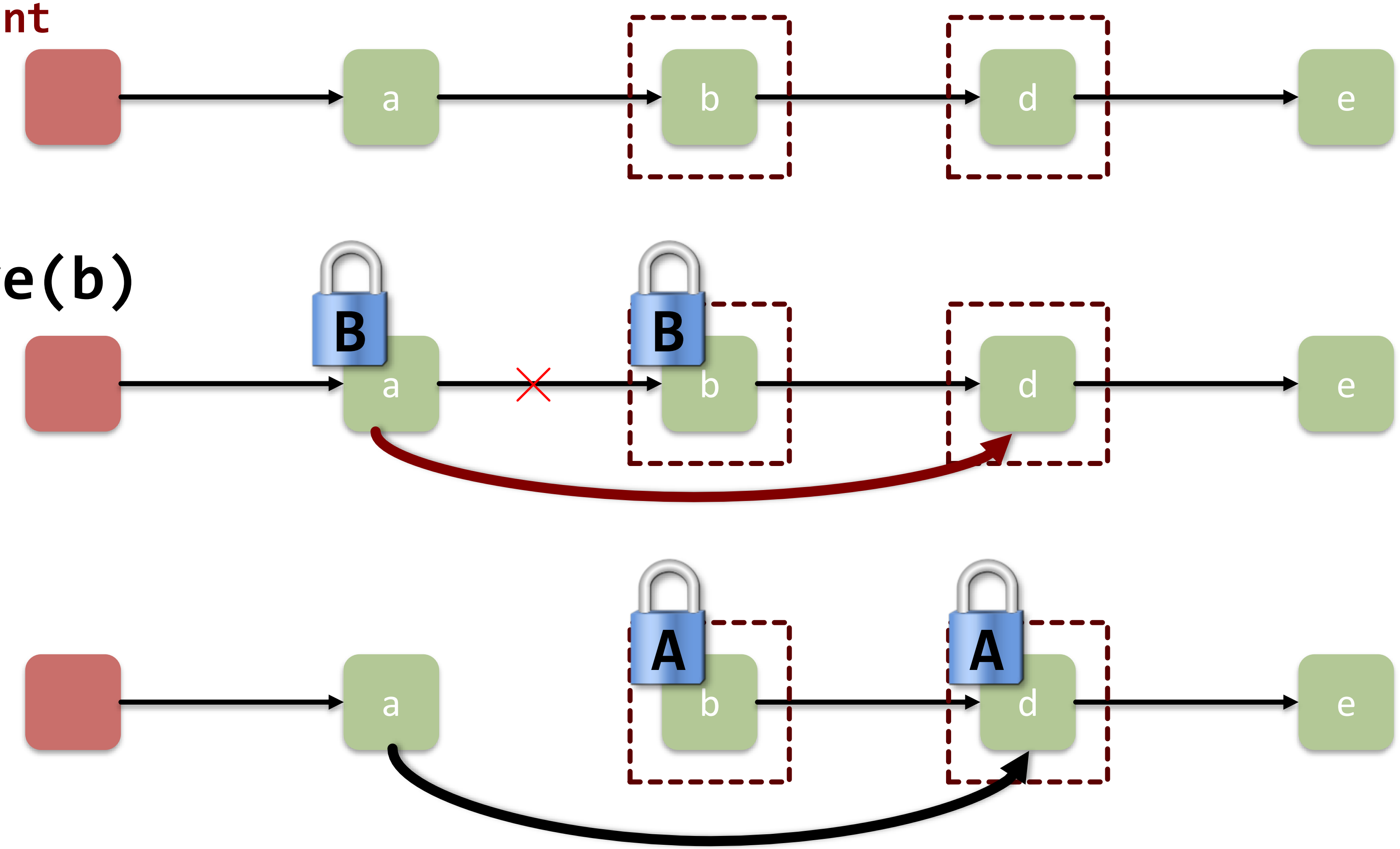
## e.g., add(c)



# Validation: what could go wrong?

## Thread A: add(c)

A: find insertion point




## Thread B: remove(b)

A: lock

A: validate: rescan

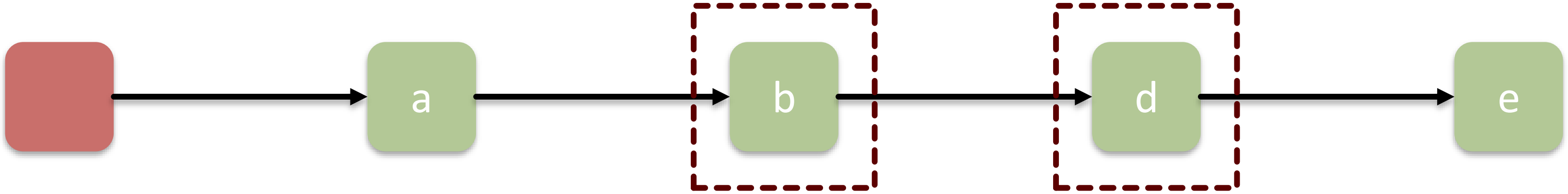
A: b not reachable

→return false

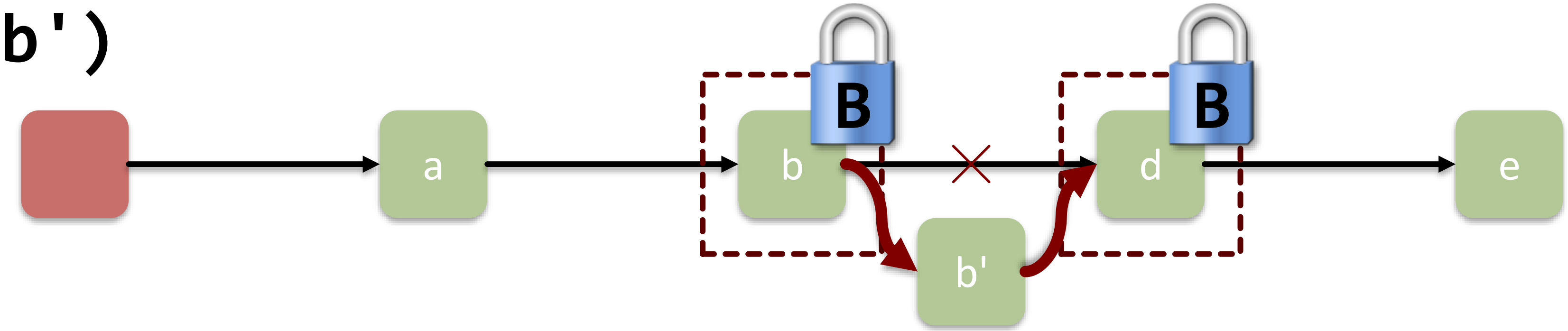
# Validation: what could go wrong?

## Thread A: add(c)

A: find insertion point




## Thread B: insert(b')

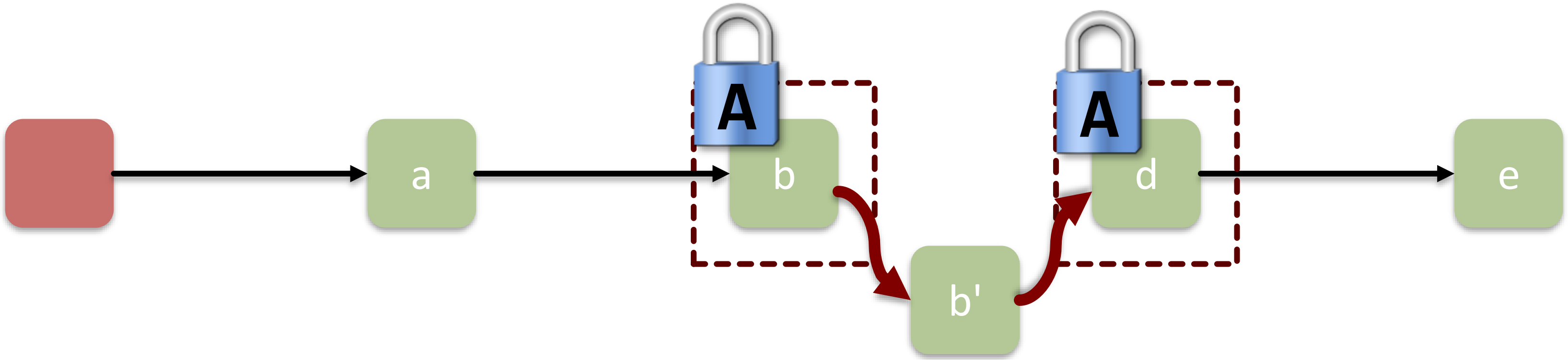



A: lock

A: validate: rescan

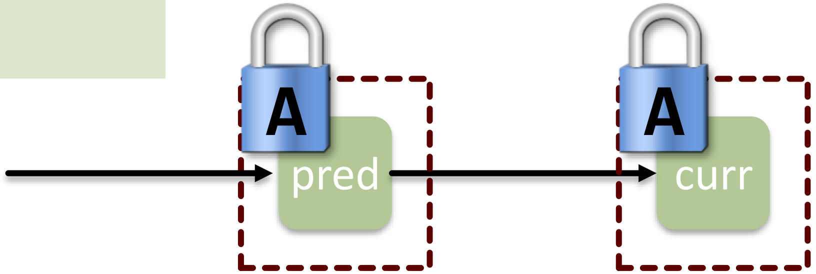
A: d != succ(b)

→return false



# Validate

```
private Boolean validate(Node pred, Node curr) {
  Node node = head;
  while (node.key <= pred.key) { // reachable?
      if (node == pred)
           return pred.next == curr; // correct?
      node = node.next;
  }
  return false;
}
```



# Correctness (remove c)

**If**

- **nodes b and c both locked**
- **node b still reachable from head**
- **node c still successor to b**

**then**

- **neither is in the process of being deleted**

**➔ ok to delete and return true**

# Correctness (remove c)

**If**

- **nodes b and d both locked**
- **node b still reachable from head**
- **node d still successor to b**


**then**

- **neither is in the process of being deleted,**
  **therefore a new element c must appear between b and d**
- **no thread can add between b and d:**
  **c cannot have appeared after our locking**

**➔ ok to return false**

# Optimistic List

## Good:

- No contention on traversals.
- Traversals are wait-free.
- Less lock acquisitions.

**Wait-Free:**
*Every* call finishes in a finite number of steps (NEVER waits for other threads).

## Bad:

- Need to traverse list twice
- The contains() method needs to acquire locks
- Not starvation-free

**Is the optimistic list starvation-free? Why/why not?**

# LAZY SYNCHRONISATION

## Laziness

The quality that makes you go to great effort to reduce overall energy expenditure […] **the first great virtue of a programmer.**

Larry Wall, Programming Perl
(emphasis mine)

# Lazy List

**Like optimistic list but**

- Scan only once
- Contains() never locks

**How?**

- Removing nodes causes trouble
- Use deleted-markers → invariant: every unmarked node is reachable!
- Remove nodes «lazily» after marking

# Lazy List: Remove

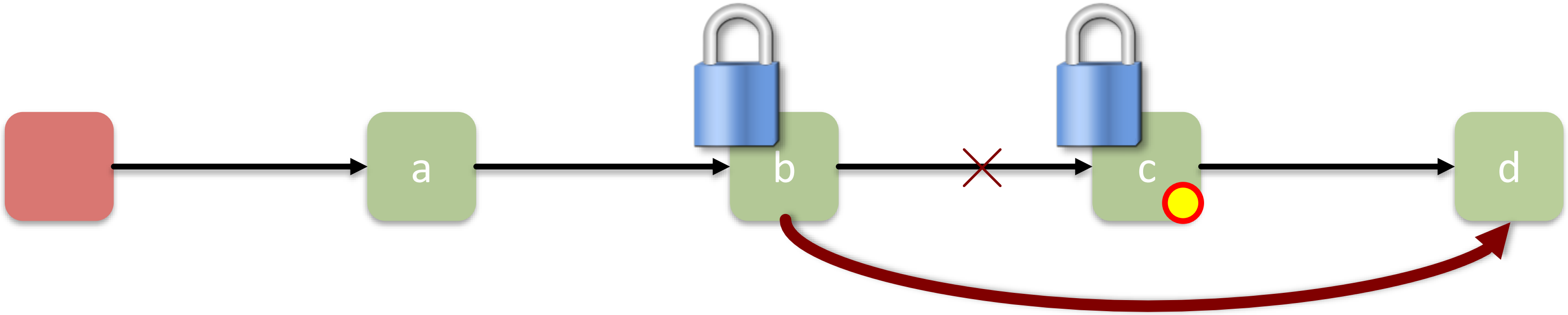
Scan list (as before)

Lock predecessor and current (as before)

Logical delete: mark current node as removed

Physical delete: redirect predecessor's next

e.g., remove(c)



# Invariant

## If a node is not marked then

- It is reachable from head
- And reachable from its predecessor

## A: remove(c)

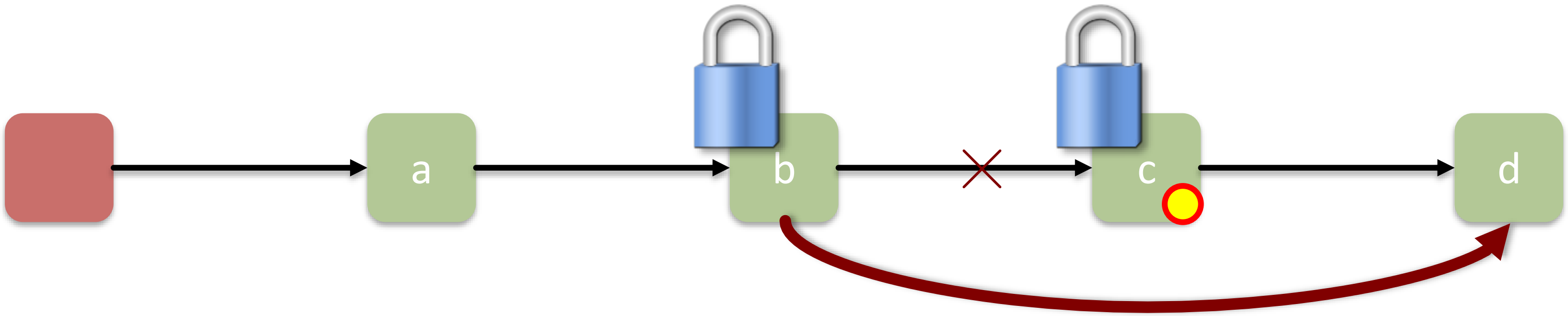
lock

check if b or c are marked

not marked? ok to delete:

mark c

delete c



# Remove method

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) { // optmistic, retry
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr;
      curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        // remove or not
      } finally { curr.unlock(); }
    } finally { pred.unlock(); }
  }
}
```

```
if (!pred.marked && !curr.marked &&
      pred.next == curr) {
   if (curr.key != key)
      return false;
   else {
      curr.marked = true;     // logically remove
      pred.next = curr.next; // physically remove
      return true;
   }
}
```

# Wait-Free Contains

```
public boolean contains(T item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

**This set data structure is again for demonstration only. Do not use this to implement a list! Now on to something more practical.**

Bill Pugh received a Ph.D. in Computer Science (with a minor in Acting) from Cornell University. He was a professor at the University of Maryland for 23.5 years, and in January 2012 became professor emeritus to start new adventure somewhere at the crossroads of software development and entrepreneurship.

Bill Pugh is a Packard Fellow, and invented Skip Lists, a randomized data structure that is widely taught in undergraduate data structure courses. He has also made research contributions in in techniques for analyzing and transforming scientific codes for execution on supercomputers, and in a number of issues related to the Java programming language, including the development of JSR 133 - Java Memory Model and Thread Specification Revision. Prof. Pugh's current research focus is on developing tools to improve software productivity, reliability and education. Current research projects include FindBugs, a static analysis tool for Java, and Marmoset, an innovative framework for improving the learning and feedback cycle for student programming projects.

Prof. Pugh has spoken at numerous developer conferences, including JavaOne, Goto/Jaoo in Aarhus, the Devoxx conference in Antwerp, and CodeMash. At JavaOne, he received six JavaOne RockStar awards, given to the speakers that receive the highest evaluations from attendees.

Professor Pugh spent the 2008-2009 school year on sabbatical at Google, where, among other activities, he learned how to eat fire.

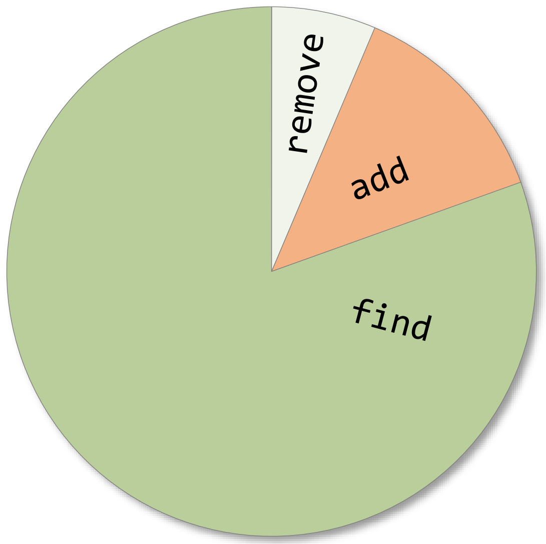
# More practical: Lazy Skiplists

Bill Pugh

# Skiplist

- **Collection of elements (without duplicates)**
- **Interface:**
  - `add` `// add an element`
  - `remove` `// remove an element`
  - `find` `// search an element`

- **Assumption:**
  - Many calls to find()
  - Fewer calls to add() and much fewer calls to remove()



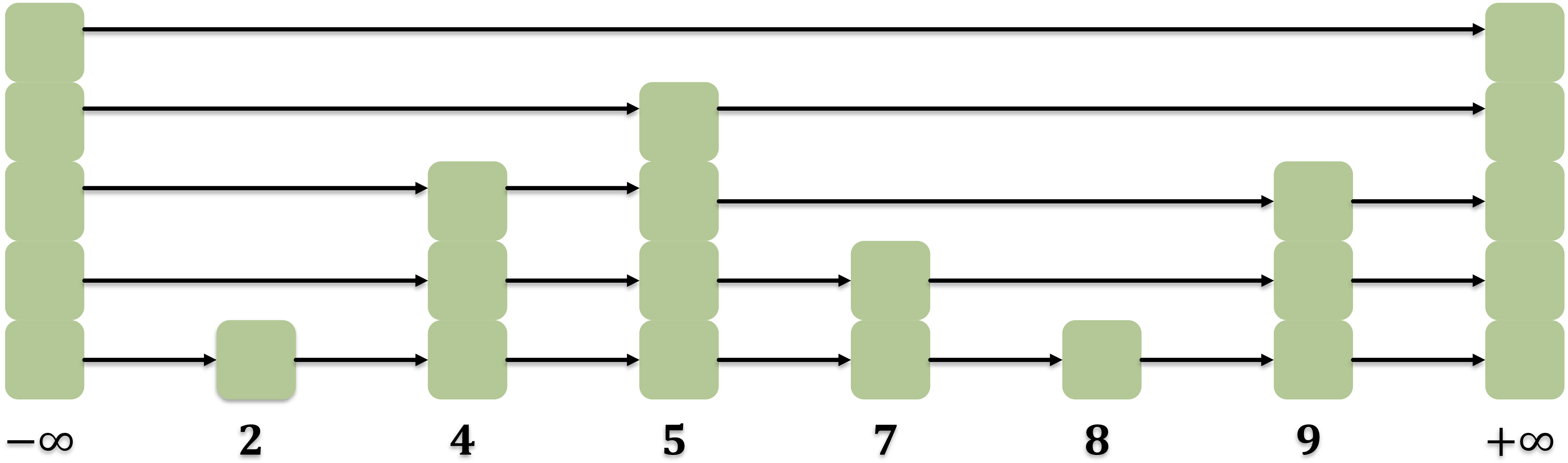
# How about balanced trees?

- **AVL trees, red-black trees, treaps, ...**
  - rebalancing after `add` and `remove` expensive
  - rebalancing is a *global* operation (potentially changing the whole tree)
  - particularly hard to implement in a lock-free way.

- **→ SkipList**

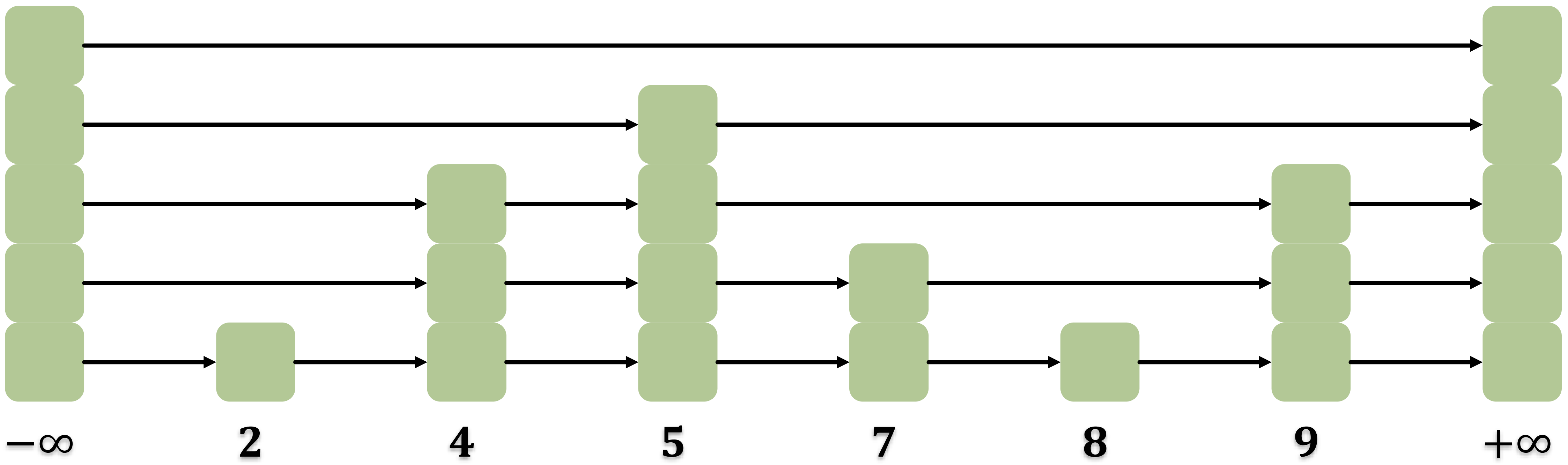
# Skip Lists

- **Sorted multi-level list**
- **Node height probabilistic, e.g.,** $\mathbb{P}(height = n) = 0.5^n$**, no rebalancing**



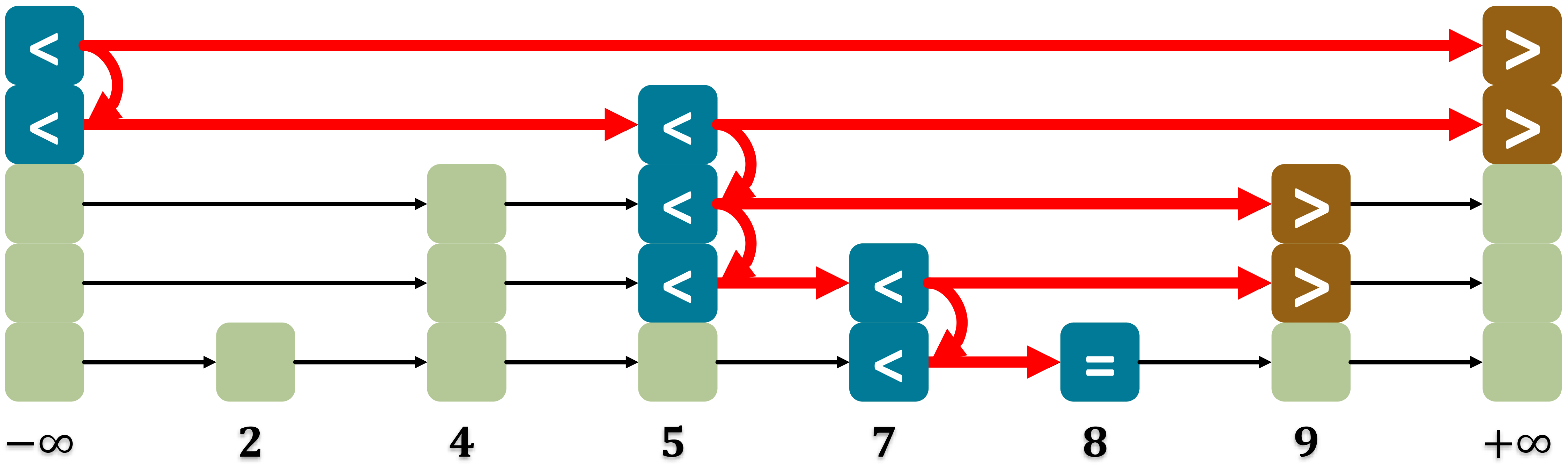
## Skip List Property

- **Sublist relationship between levels: higher level lists are always contained in lower-level lists. Lowest level is entire list.**



# Searching

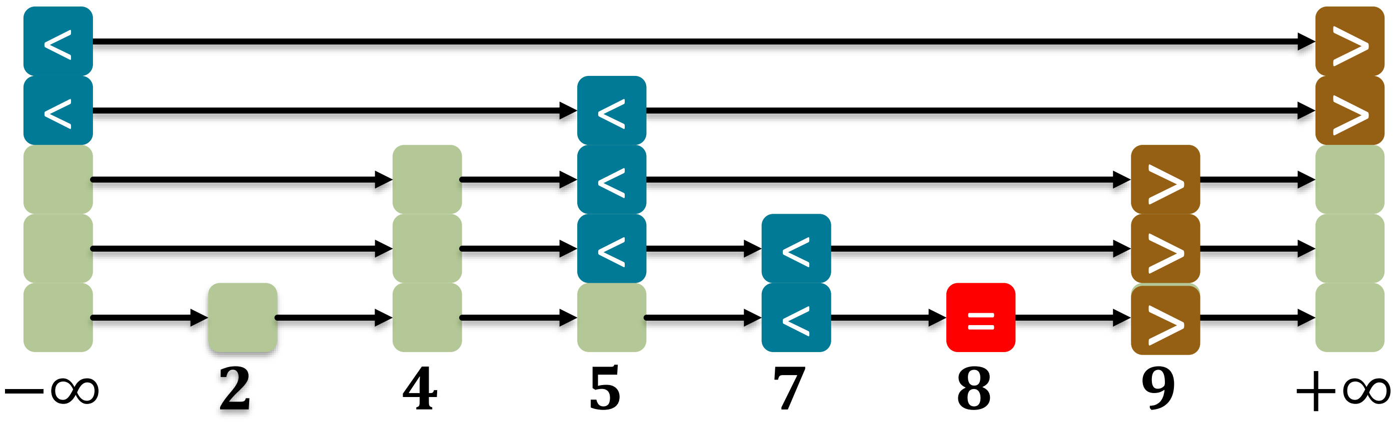
- Logarithmic Search (with high probability)
- Example: Search for 8



# Sequential Find

- `// find node with value x`
- `// return -1 if not found, node level otherwise`
- `// pre = array of predecessor node for all levels`
- `// succ = array of successor node for all levels`
- `int find(T x, Node<T>[] pre, Node<T>[] succ)`

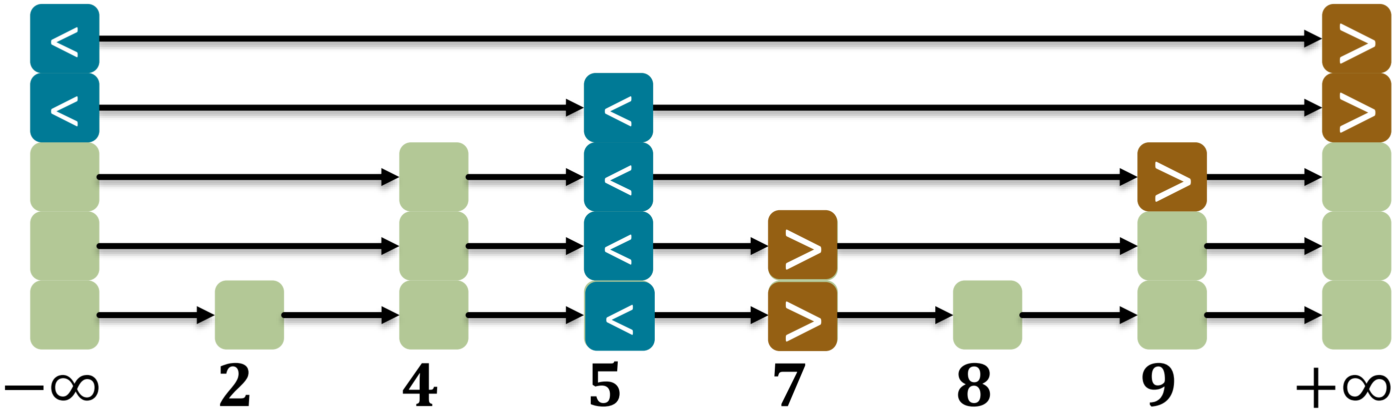
- `e.g., x = 8`
- `returns 0`



# Sequential Find

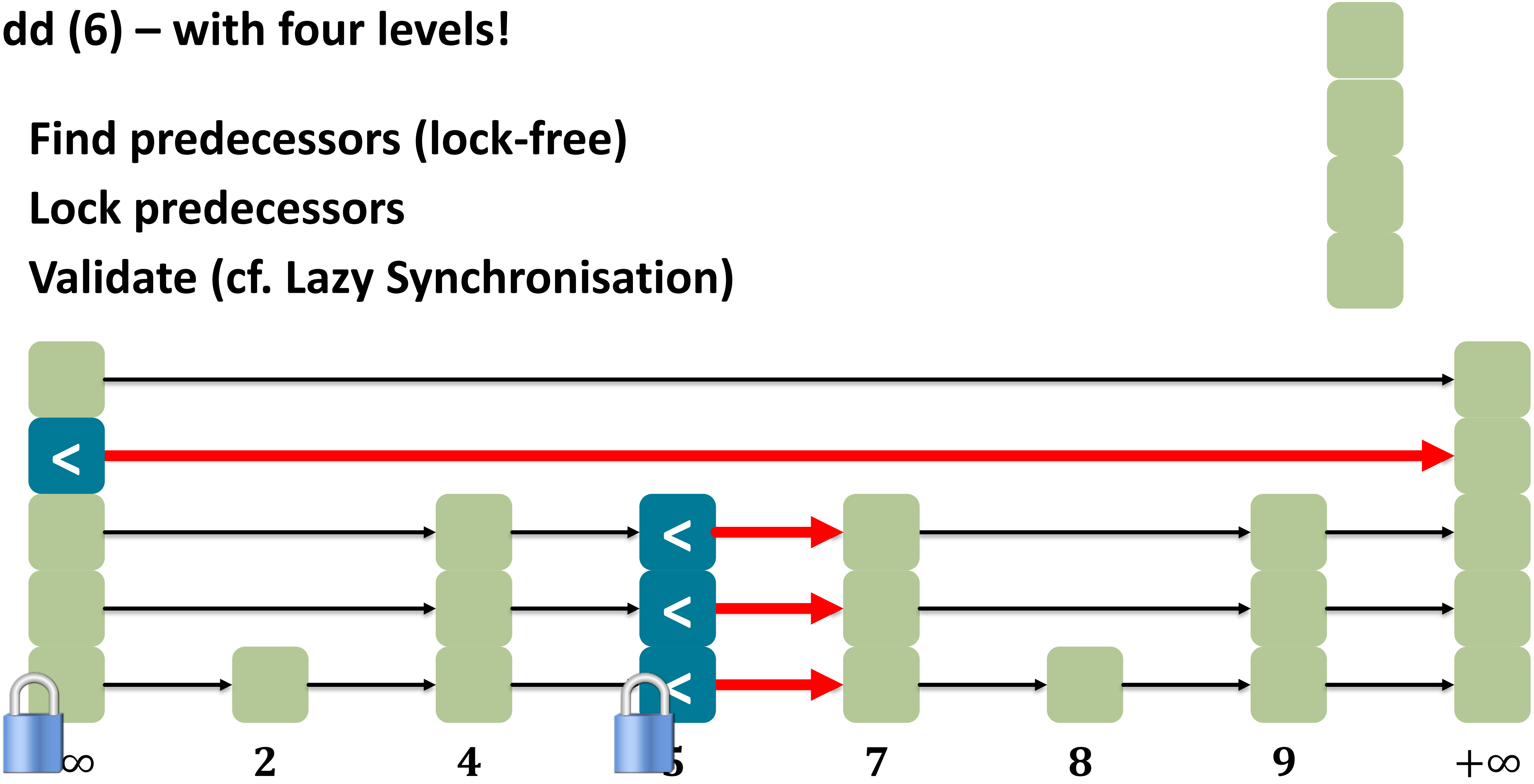
- // find node with value x
- // return -1 if not found, node level otherwise
- // pre = array of predecessor node for all levels
- // succ = array of successor node for all levels
- int find(T x, Node<T>[] pre, Node<T>[] succ)

- e.g., x = 6
- returns -1



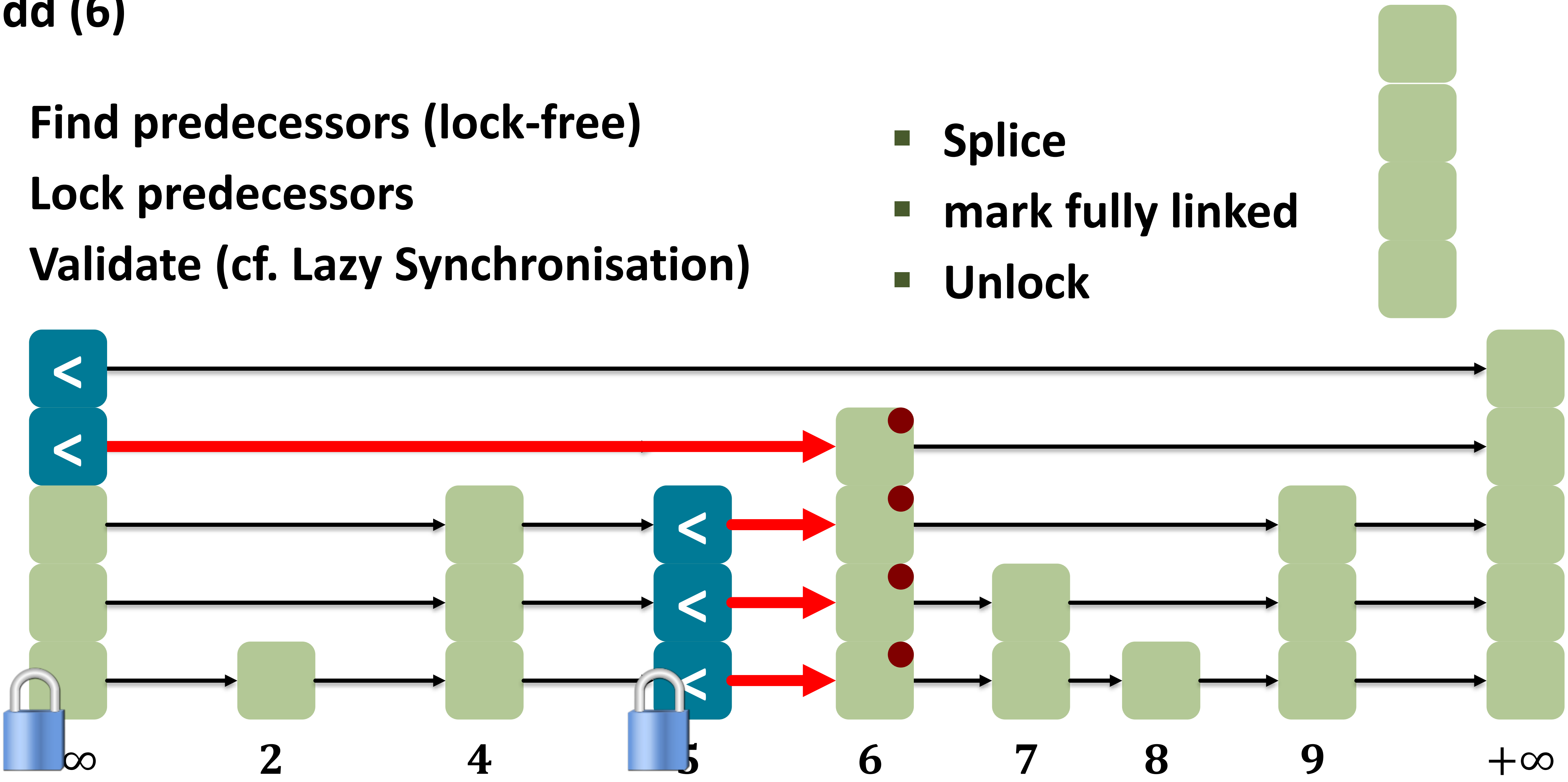
# add (6) – with four levels!

- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)



# add (6)

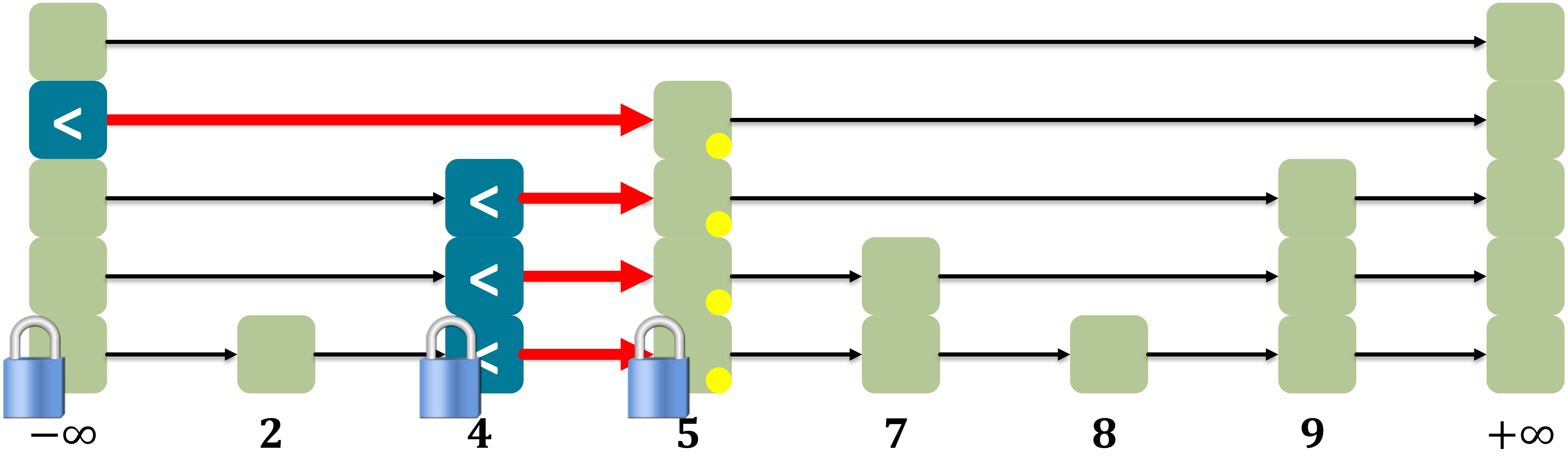
- Find predecessors (lock-free)
- Lock predecessors
- Validate (cf. Lazy Synchronisation)
- Splice
- mark fully linked
- Unlock



# remove(5)

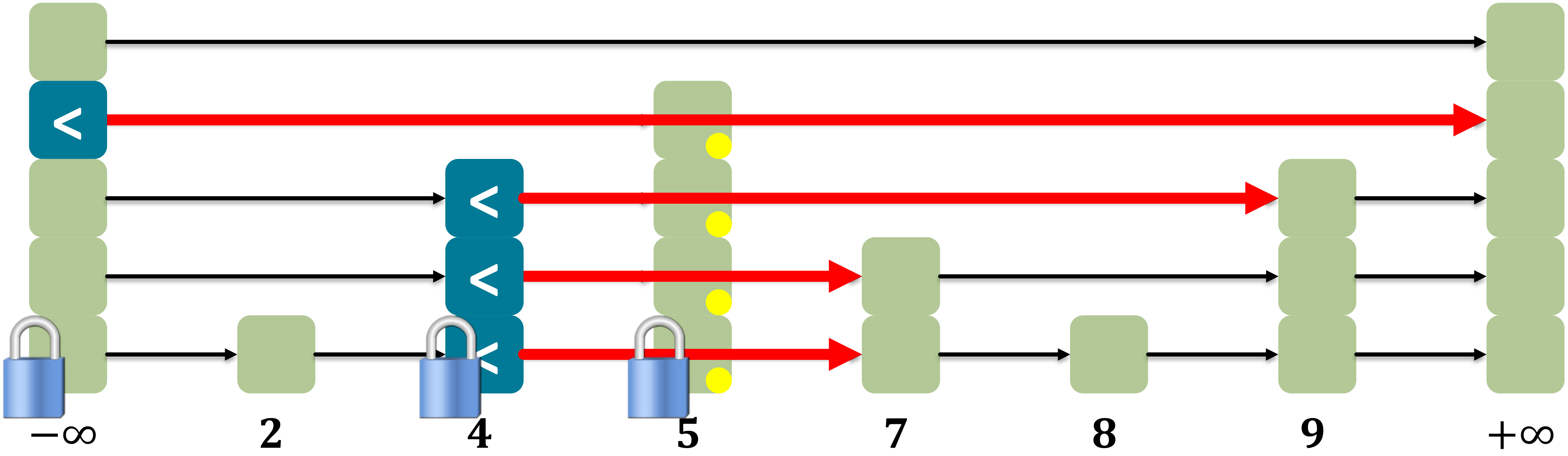
- find predecessors
- lock victim
- logically remove victim (mark)

- Lock predecessors and validate



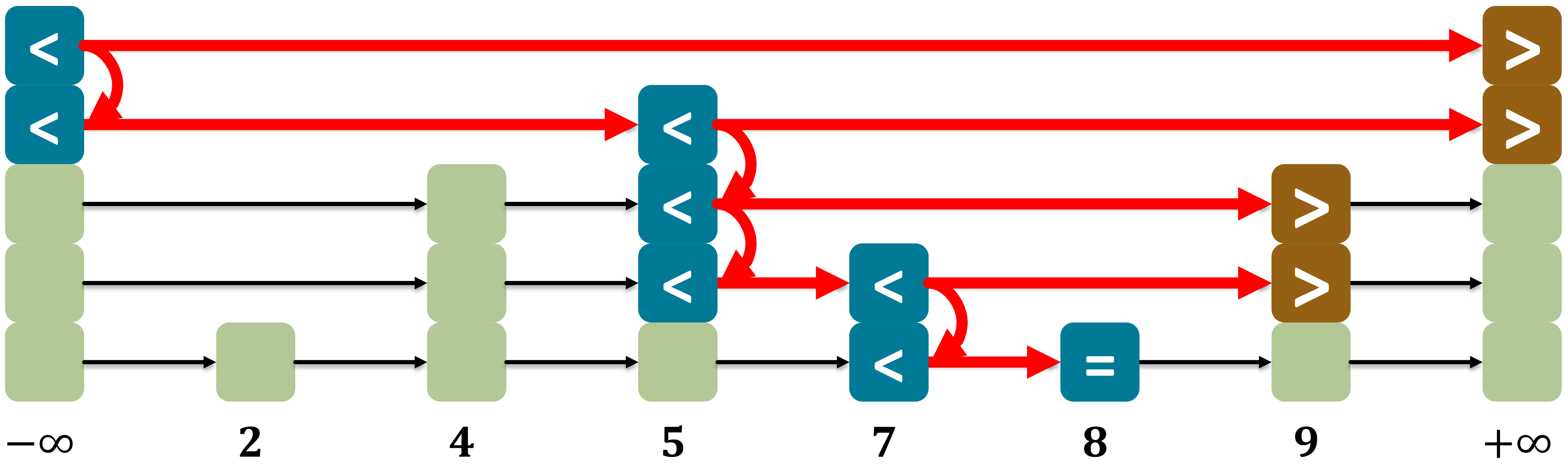
# remove(5)

- find predecessors
- lock victim
- logically remove victim (mark)
- Lock predecessors and validate
- physically remove
- unlock



# contains(8)

- sequential find() & not logically removed & fully linked
- even if other nodes are removed, it stays reachable
- contains is wait-free (while add and remove are not)



# Skiplist

- **Practical parallel datastructure**
- **Code in book (latest revision!) – 139 lines**
  - Too much to discuss in detail here
- **Review and implement as exercise**