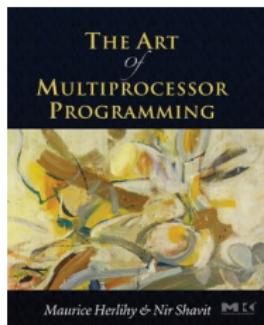


Concurrency & Multithreading



Maurice Herlihy & Nir Shavit
The Art of Multiprocessor Programming
Morgan Kaufmann, 2008
(or revised reprint of 1st edition, 2012)

Learning objectives of the course

Fundamental insight into multicore computing

Algorithms for multicore computing

Analyzing multicore algorithms

Concurrent datastructures

Multicore programming in Java

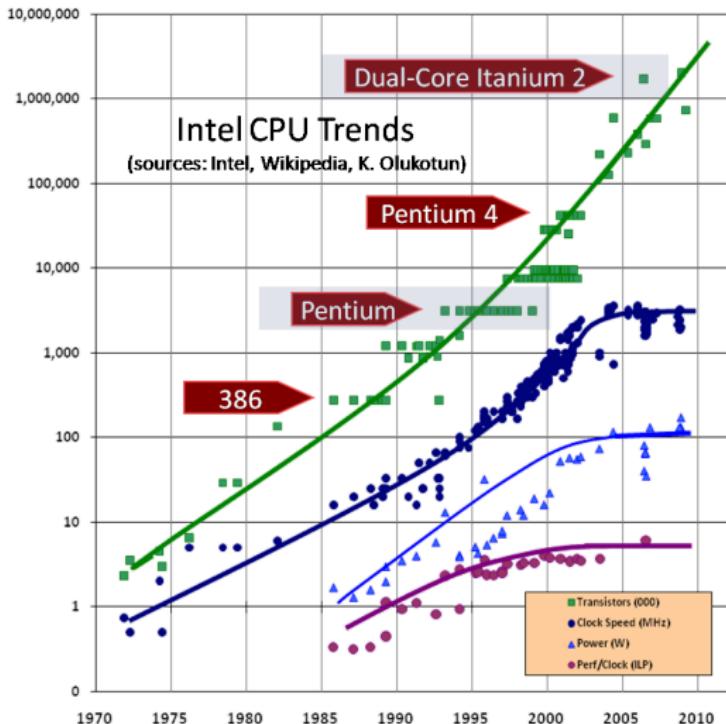
Moore's law versus clock speed

Moore's law from 1965 still holds: The number of transistors that can be placed on a chip doubles every two years.

However:

- ▶ As chip geometries shrink and clock frequencies rise, *leakage current* increases, leading to excessive power consumption and heat.
- ▶ Delays in signal transmission grow as feature sizes shrink, due to *parasitic capacitance* between parts in electrical circuits.
- ▶ *Memory wall*: memory access times haven't kept up with increasing clock frequencies.

Moore's law versus clock speed



Multiprocessors



A **multiprocessor** circumvents these problems by combining multiple CPUs in one computer system.



Moore's law continues to have effect

Parallelism doubles every two years !

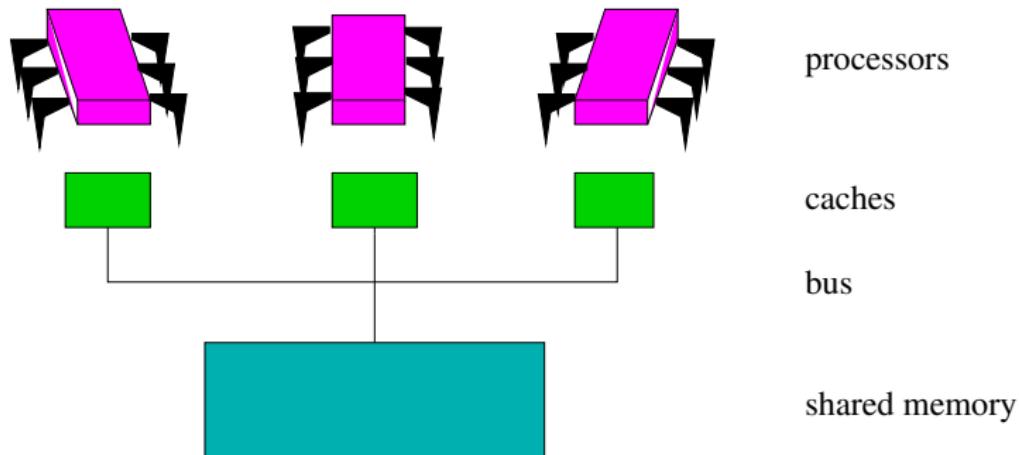
Laptops usually have between 4 and 8 cores.

This will increase when batteries improve.

Multicore programming is challenging:

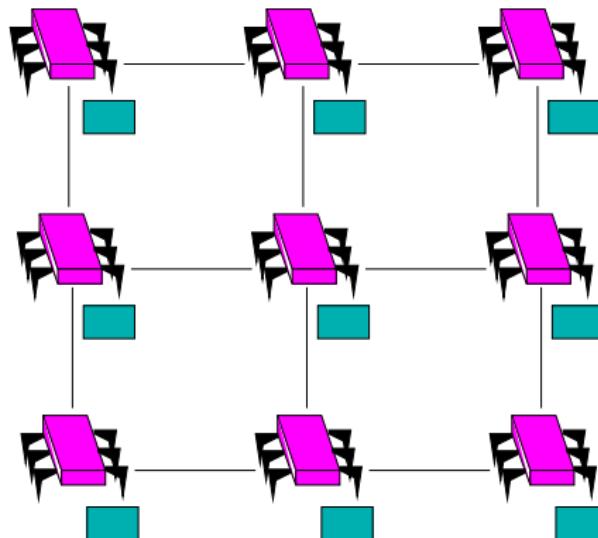
- ▶ At a *small* scale: e.g., processors within one chip must coordinate access to shared memory.
- ▶ At a *large* scale: e.g., processors in a supercomputer must coordinate routing of data.

Symmetric multiprocessing architecture



OpenMP is an API for programming on SMP architectures.

Non-uniform memory access architecture



MPI is a specification of an API for programming on **NUMA** architectures.

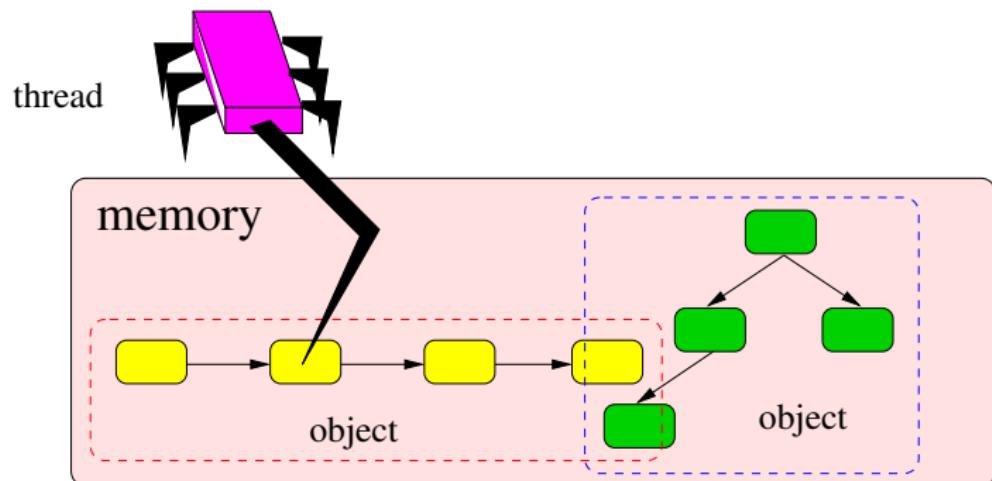
Threads are asynchronous

The software **threads** that run on processors are fully **asynchronous**.

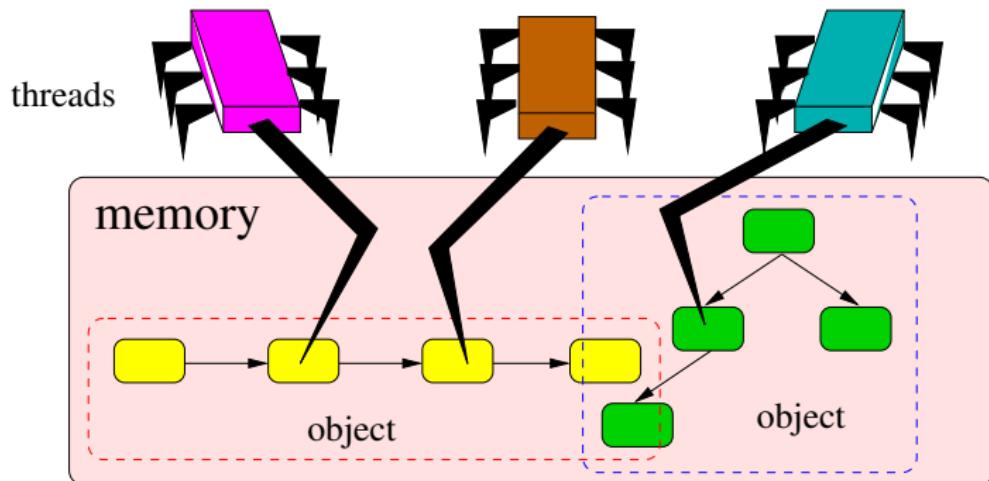
Threads may be **delayed** or even halted without warning.

Delays are unpredictable and vary greatly.

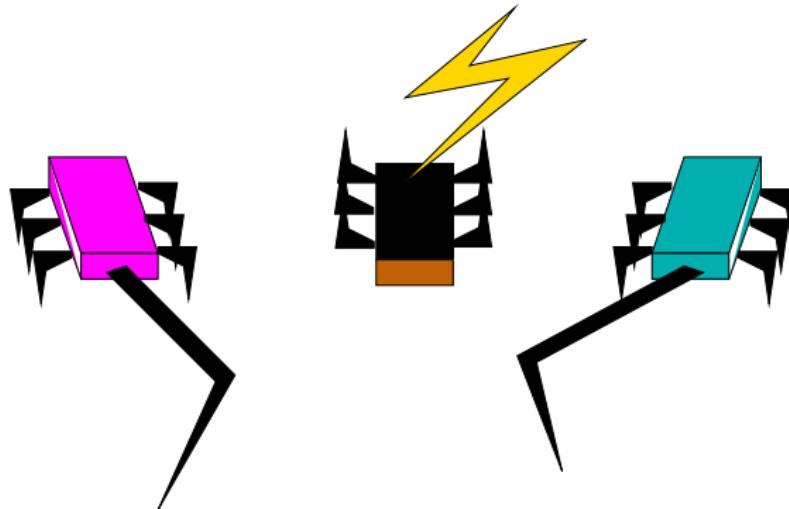
Sequential computation



Concurrent computation



Asynchrony



Sudden unpredictable delays of threads:

- ▶ cache miss (*short*)
- ▶ page fault (*long*)
- ▶ scheduling quantum used up (*really long*)
- ▶ crashed (*indefinite*)

Model summary

- ▶ multiple processors
- ▶ single shared memory
- ▶ objects live in memory
- ▶ multiple threads, which run on a processor
- ▶ threads are asynchronous and have unpredictable delays

Jargon:

- ▶ hardware: processors
- ▶ software: threads

Road map

We will first focus on **principles** of multiprocessor programming, then on “**practice**”.

- ▶ We start with *idealized* models.
- ▶ We look at *simplistic, fundamental* problems.
- ▶ *Correctness* is emphasized over *pragmatism*.

We need to understand the principles before we can discuss practice.

Example: Parallel primality testing

Challenge: Print all primes up to 10^{10} .

Given: Multiprocessor with ten processors, one thread per processor.

Goal: Get (close to) ten-fold speedup.

Naïve approach: Thread i takes care of interval $\langle (i-1) \cdot 10^9, i \cdot 10^9 \rangle$.

On the one hand, larger numbers are harder to test.

On the other hand, higher ranges contain fewer primes.

So thread workloads become *uneven*, and *hard to predict*.

Shared counter

Better idea: A *counter* is maintained, counting from 1 up to 10^{10} .

When a thread is ready to test a new number, it

- (1) **reads** the counter, and (2) **increases** the counter by 1.

(Actually, for the best known primality testing algorithm,
by Agrawal-Kayal-Saxena, this approach doesn't work.)

Question: What is the risk in using a shared counter ?

Shared counter

Problem: Two threads can *concurrently* read the value k of the counter in shared memory, and increase the value by 1.

Because reading and increasing the counter are distinct **atomic** steps.

Then both threads analyze k , while $k + 1$ is skipped.

Possible solutions:

- ▶ Use **mutual exclusion** (e.g., a *lock*) to guarantee that only one thread at a time reads and increases the counter.
- ▶ Use a **read-modify-write** hardware primitive to turn reading and increasing the counter into a *single* atomic step.

Non-atomic operations on integer variables in Java

Consider an integer variable `x` in Java

`x++` isn't an atomic operation.

(Not even if `x` is declared *volatile*.)

First `x` is read, and then written to, in two distinct atomic steps.

If `x` is 64-bits (`long`) and *non-volatile*, a write to `x` isn't atomic
(on 32-bit machines).

Mutual exclusion: A story

Neighbors Alice and Bob share a yard; Alice owns a *cat*, Bob a *dog*.

The pets don't get along;
they must never be in the yard together.



Idea: Look at the yard, to see whether it is empty.

Gotcha: Alice and Bob might look at (almost) the same time,
and both conclude that the yard is empty.

Interpretation: Looking at the yard and releasing a pet are
distinct atomic steps.

Explicit communication is required for coordination.

Cell phone protocol

Idea: Bob calls Alice (or vice versa).



Gotcha: Alice may be taking a shower, or shopping for pet food.
Or her cell phone may be off or dead.

Interpretation: *Transient* communication (like talking) isn't ideal,
because the recipient may be non-responsive.

Communication should be *persistent* (like writing).

Can protocol

- ▶ A can on Alice's window-sill, with a string to Bob's house. And vice versa.
- ▶ Alice pulls the string (knocking down Bob's can) when she wants to let her pet into the yard. And vice versa.
- ▶ Alice lets her cat in the yard if Bob's can is down and her can is up. And vice versa.



Gotcha: Alice needs Bob to reset his can after the cat has left the yard.
And vice versa.

Interpretation: **Interrupts** aren't ideal for solving mutual exclusion.
Alice and Bob better control their own signals.

Flag protocol

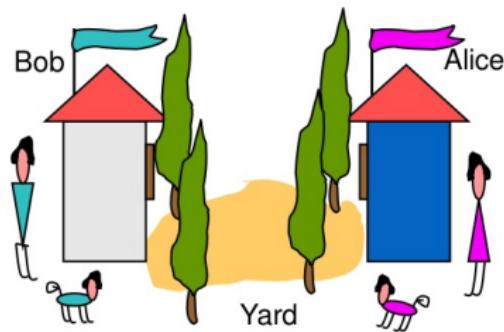
Alice's protocol:

- ▶ raise flag
- ▶ wait until Bob's flag is down
- ▶ release pet
- ▶ after pet returned, lower flag

Bob's protocol:

- ▶ raise flag
- ▶ while Alice's flag is up:
 - ▶ lower flag
 - ▶ wait until Alice's flag is down
 - ▶ raise flag
- ▶ release pet
- ▶ after pet returned, lower flag

Protocol 3 – flags



17

Question

Why isn't it a good idea to let both Alice and Bob be “polite” ?

Answer: A deadlock could occur if infinitely often Alice and Bob concurrently raise and lower their flag.



"AFTER YOU", SAID MISS MANNERS.
"OH, YOU FIRST" INSISTED MRS. ETIQUETTE.

Flag protocol: Proof of mutual exclusion

Mutual exclusion: The pets are never in the yard together.

Suppose Bob releases the dog.

When Bob looked, Bob's flag was up,
and Alice's flag down.

So Alice can only release the cat
after Bob has lowered his flag.



The same argumentation applies to Alice's cat.

Flag protocol: Proof of deadlock-freeness

Deadlock-free: If a pet wants to enter the yard, one of the pets eventually succeeds.

If only one pet wants to enter the yard, it succeeds.

If both pets want to enter the yard at the same time, Bob sees Alice's raised flag and gives her priority.



Flag protocol: Starvation-freeness

The flag protocol is **not starvation-free**:

The dog might never get in, while the cat keeps on entering and leaving the yard.



Question: How can the flag protocol be made starvation-free ?

The flag protocol is **not lock-free**:

If e.g. Bob dies while his flag is raised, the cat can't enter the yard anymore.

Moral of the story

Mutual exclusion can't be solved effectively by:

- ▶ transient communication
- ▶ interrupts

It can be solved by (multi-reader, single-writer) shared variables.

Safety and liveness properties

Safety property: Something “bad” will never happen.

For example, *mutual exclusion* (e.g. at no moment in time more than one thread has write access to some variable).

Liveness property: Something “good” will eventually happen.

For example:

- ▶ *deadlock-free* (e.g. if a pet wants to enter the yard, one of the pets eventually succeeds)
- ▶ *starvation-free* (e.g. if a pet wants to enter the yard, it eventually succeeds)

Progress properties

Blocking

Deadlock-free: *Some* thread trying to get the **lock** eventually succeeds.

Starvation-free: *Every* thread trying to get the **lock** eventually succeeds.

Non-blocking

Lock-free: *Some* thread calling the **method** eventually returns.

Wait-free: *Every* thread calling the **method** eventually returns.

Lock- and wait-free disallow blocking methods like locks.

They guarantee that the system can cope with crash-failures.

Picking a progress property for a given application depends on its needs and what is feasible. We will look at all four properties.

Producer-consumer: The story continues

Alice and Bob fall in love and marry; then they divorce.

She gets the pets (who now get along),
while he has to feed them.

The pets side with Alice and attack Bob.



Bob must put food in the yard, when the pets aren't there.

Alice only wants to release the pets when there is food.

Bob only wants to put food in the yard if there is none left.

Can protocol revisited

A can on Alice's window-sill; the string leads to Bob's house.

Alice's protocol:

- ▶ wait until the can is down
- ▶ release the pets
- ▶ every time the pets return, check whether there is food left
- ▶ if not, reset the can; keep the pets inside until the can is down

Bob's protocol:

- ▶ wait until the can (on Alice's window-sill) is up
- ▶ put food in the yard
- ▶ go inside, and pull the string to knock down the can

Can protocol: Correctness

Mutual exclusion: Bob and the pets are never in the yard together.

Suppose Bob enters the yard.

When Bob looked, the can was up.

When Alice reset the can, the pets weren't in the yard.

Alice can only release the pets after Bob went inside and knocked down the can.

Starvation-free: If Bob is always willing to feed and Alice always alert, and the pets are always famished, then they will eat infinitely often.

Producer-consumer: The pets only enter the yard when there is food.

Bob only enters the yard when there is no food left.

Exercise 3

Design a producer-consumer protocol using cans and strings that works even if Bob can't see the can on Alice's window-sill.

Answer: Let a string from Alice's house lead to a can on Bob's window-sill.

The two cans are multi-writer shared variables.

(This is how real-world interrupt bits work.)

Exercise 3: Solution

A string from Bob's house leads to a can on Alice's window-sill,
and a string from Alice's house leads to a can on Bob's window-sill.

- ▶ Alice waits until the can on her window-sill is down.
- ▶ She releases the pets.
- ▶ When the pets return and the food is gone, she resets her can.
- ▶ *She pulls her string to knock down the can on Bob's window-sill.*
- ▶ She keeps the pets inside until her can is down.

- ▶ Bob waits until *the can on his window-sill is down.*
- ▶ *He resets his can.*
- ▶ He puts food in the yard.
- ▶ He goes inside, and pulls his string to knock down the can on Alice's window-sill.

Questions

Why must Alice reset her can before pulling the string ?

Why must Bob reset his can before pulling the string ?

Amdahl's law

Given a job, that is executed on n processors.

Let $p \in [0, 1]$ be the fraction of the job that can be parallelized (over any number of processors).

Let sequential execution of the job take 1 time unit.

Parallel execution of the job takes (at least) $(1 - p) + \frac{p}{n}$ time units.

So the speedup is

$$\frac{1}{(1 - p) + \frac{p}{n}}$$

Amdahl's law: Examples

$$n = 10$$

$$p = 0.6 \text{ gives speedup of } \frac{1}{0.4 + \frac{0.6}{10}} = 2.2$$

$$p = 0.9 \text{ gives speedup of } \frac{1}{0.1 + \frac{0.9}{10}} = 5.3$$

$$p = 0.99 \text{ gives speedup of } \frac{1}{0.01 + \frac{0.99}{10}} = 9.2$$

Conclusion: To make efficient use of multiprocessors, it is important to minimize sequential parts, and reduce idle time in which threads wait.

Question: What is the maximum speedup if $p = 0.6$?

Exercise 8

You want to perform a job either on:

- ▶ a *multiprocessor* consisting of 10 processors; or
- ▶ a *uniprocessor*, 5 times faster than each of those 10 processors.

How large should the fraction of the job be that can be parallelized, in order to prefer the multiprocessor ?

Exercise 8: Solution

The question is when a parallelization of the job on the multiprocessor yields a speedup of more than 5.

According to Amdahl's law, the speedup is

$$\frac{1}{(1-p) + \frac{p}{10}} = \frac{10}{10 - 9p}$$

So if $p > \frac{8}{9}$, the speedup is greater than 5.

This lecture in a nutshell

Moore's law versus non-increasing clock speed

SMP architecture

mutual exclusion

software lock / read-modify-write hardware primitive

flag protocol for mutual exclusion

interrupts for producer-consumer

safety and liveness properties

deadlock-, starvation-, lock-, wait-freeness

Amdahl's law

minimize sequential part of multicore program

Events

A thread exhibits a sequence of events a_0, a_1, a_2, \dots

(E.g., read or write to a variable / invoke or return from a method.)

Events are *instantaneous* and *never simultaneous*.

(You're free to break ties between simultaneous events.)

Consider the events of an execution by a multicore system.

$a \rightarrow b$ denotes that a happens before b . This is a *total order*.

Let a and b be events by the same thread, with $a \rightarrow b$.

(a, b) denotes the time interval between these events.

We write $(a, b) \rightarrow (a', b')$ if $b \rightarrow a'$. This is a *partial order*.

Mutual exclusion

A **critical section** is a block of code that should be executed by at most one thread at a time.

Let CS and CS' be time intervals in which two different threads execute their critical sections.

Mutual exclusion: For each such pair, $CS \rightarrow CS'$ or $CS' \rightarrow CS$.

Question

Suppose one thread leaves its critical section exactly at the moment another thread enters its critical section.



Does this mean mutual exclusion is violated?

Locks in Java

```
public interface Lock {  
    public void lock();           acquire the lock  
    public void unlock();         release the lock  
}  
  
lock.lock();  
try {  
    critical section  
} finally {  
    lock.unlock();  
    unlocking instructions  
}
```

Even if an output is returned or exception is thrown in the critical section, the `finally` part will be executed, to release the lock properly.

Locks and memory management in Java

When a thread *acquires* a lock, it invalidates its working memory, to ensure that fields are reread from shared memory.

When a thread *releases* a lock, modified fields in its working memory are written back to shared memory.

Deadlock- and starvation-free locks

Assumption: No thread holds the lock forever.

Deadlock-free: Suppose some thread calls `lock()` but never acquires the lock.

Then other threads must be completing an infinite number of critical sections.

Starvation-free: If some thread calls `lock()`, then it will eventually acquire the lock.

LockOne

Given two threads, with identities 0 and 1.

`ThreadID.get()` returns the identity of the calling thread.

```
class LockOne implements Lock {  
    private boolean[] flag = new boolean[2];  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        while flag[j] {}  
    }  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false  
    }  
}
```

my id
other id
set my flag
wait until other flag is false

my id
reset my flag

LockTwo

LockOne provides *mutual exclusion*, but may *deadlock* (if threads concurrently set their flag to true).

```
class LockTwo implements Lock {  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();           my id  
        victim = i;                     let other go first  
        while victim == i {}           wait for permission  
    }  
    public void unlock() {}
```

LockTwo provides *mutual exclusion*, but may *deadlock* (if one thread never tries to get the lock).

Peterson lock

```
class Peterson implements Lock {  
    private boolean[] flag = new boolean[2];  
    private int victim;  
    public void lock() {  
        int i = ThreadID.get();  
        int j = 1 - i;  
        flag[i] = true;  
        victim = i;  
        while (flag[j] && victim == i) {};  
    }  
    public void unlock() {  
        int i = ThreadID.get();  
        flag[i] = false;  
    }  
}
```

Question

What could go wrong if `victim = i` were performed *before* `flag[i] = true` ?

Peterson lock: Mutual exclusion

The Peterson lock provides *mutual exclusion* (for two threads).

Let thread i enter its critical section (so `flag[i] == true`).

There are two possibilities:

1. Before entering, i read `flag[j] == false`.

To enter, j first sets `flag[j] = true` and `victim = j`.

2. Before entering, i read `victim == j`.

Then j performed `victim = j` after i performed `victim = i` (and so after i performed `flag[i] = true`).

In both cases, j can only enter after i sets `flag[i] = false` or `victim = i`.

Hence j can't enter while i is in its critical section.

Peterson lock: Starvation-free

The Peterson lock is *starvation-free*.

Let thread i try to enter its critical section.

Then it sets `flag[i] = true` and `victim = i`.

Since i can enter when `flag[j] = false`, j could only starve i by repeatedly re-entering its critical section.

However, before entering, j sets `victim = j`.

Then i can enter its critical section.

Volatile variables

In Java, a variable can be declared **volatile**.

- ▶ When a volatile variable is **read**, its value is fetched from memory (instead of from the cache).
- ▶ When a volatile variable is **written**, the new value is immediately written back to memory.
- ▶ **Out-of-order execution** by the hardware with regard to a volatile variable is *not* allowed.

In the Peterson lock, the elements of the **flag** array and **victim** must be declared **volatile**.

Else threads could read stale **flag** and **victim** values.

Volatile arrays

The pseudocode of Herlihy and Shavit simply declares the `flag` array volatile.

It is questionable whether this is enough, because a volatile array isn't an array of volatile elements.

But now we drift into yucky details of Java's semantics.

In any case, some *memory barrier* is needed for the `flag` array.

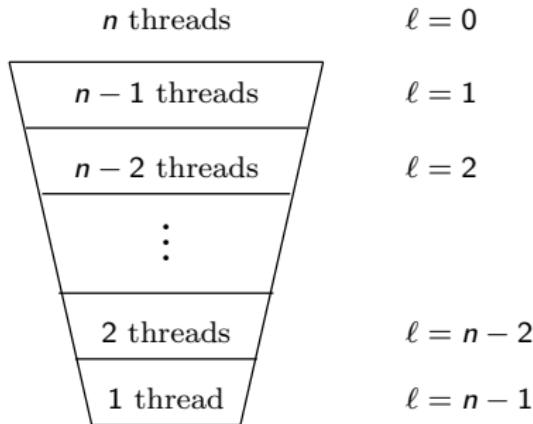
Filter lock

The filter lock generalizes the Peterson lock to $n \geq 2$ threads.

There are $n - 1$ “waiting rooms”, called **levels**, from 0 up to $n - 1$.

Threads start at level 0. The critical section is at level $n - 1$.

At most $n - \ell$ threads can concurrently proceed to a level $\geq \ell$.



Filter lock

For levels $\ell = 1, \dots, n - 1$, there is a variable `victim[ℓ]`.

A thread i at a level $\ell - 1$ that wants to go to level ℓ , sets `level[i] = ℓ` and `victim[ℓ] = i`.

Thread i must wait with going to level ℓ until either `level[j] < ℓ` for all $j \neq i$ or `victim[ℓ] ≠ i`.

That is, thread i *spins* on:

- ▶ `level[j]` for each $j \neq i$, to check whether they are all $< \ell$; and
- ▶ `victim[ℓ]` to check whether it is unequal to i .

Again, the `level[_]` and `victim[_]` fields must be *volatile*.

Question

What does a thread i do when it leaves its critical section ?

Answer: It sets `level[i]` to 0.

Filter lock: Example

Threads A,B,C are all at level 0.

Thread B sets $\text{level}[B] = 1$ and $\text{victim}[1] = B$.

Since no other thread has a level ≥ 1 , thread B proceeds to level 1.

Thread C sets $\text{level}[C] = 1$ and $\text{victim}[1] = C$.

Thread A sets $\text{level}[A] = 1$ and $\text{victim}[1] = A$.

Since $\text{victim}[1] \neq C$, thread C proceeds to level 1.

Thread C sets $\text{level}[C] = 2$ and $\text{victim}[2] = C$.

Thread B sets $\text{level}[B] = 2$ and $\text{victim}[2] = B$.

Since $\text{victim}[2] \neq C$, thread C proceeds to its critical section.

Filter lock: Mutual exclusion

Let $\ell \leq n - 1$.

At most $n - \ell$ threads can concurrently proceed to a level $\geq \ell$.

Namely, either:

1. at most one thread is at a level $\geq \ell$;
2. or a thread is waiting at each level $0, \dots, \ell - 1$.

In both cases the claim holds.

Taking $\ell = n - 1$, only one thread can be in its critical section.

So the filter lock provides *mutual exclusion*.

Filter lock: Starvation-free

The filter lock is *starvation-free*.

Namely, consider a thread i waiting to go to a level $\ell \geq 1$.

If other threads keep on entering and leaving their critical sections, eventually a thread $j \neq i$ wants to enter level ℓ and sets $\text{victim}[\ell] = j$.

Then thread i can proceed to level ℓ .

Question

Suppose a thread i that finds $\text{level}[j] < \ell$ for all $j \neq i$, is allowed to access its critical section immediately.

Give a scenario to show that then mutual exclusion isn't guaranteed.

Threads A,B,C are all at level 0.

$\text{level}[B] = 1$ and $\text{victim}[1] = B$.

Thread B finds no other thread has a level ≥ 1 .

$\text{level}[C] = 1$ and $\text{victim}[1] = C$.

$\text{level}[A] = 1$ and $\text{victim}[1] = A$.

Thread C finds $\text{victim}[1] == A$, and proceeds.

$\text{level}[C] = 2$ and $\text{victim}[2] = C$.

Thread C finds no other thread has a level ≥ 2 .

Threads B and C concurrently access their critical sections.

Peterson locks in a binary tree

Another way to generalize the Peterson lock to $n \geq 2$ threads is to use a **binary tree**, where each node holds a Peterson lock for two threads.

Threads start at a leaf in the tree, and move one level up when they acquire the lock at a node.

A thread that holds the lock of the root can enter its critical section.

When a thread exits its critical section, it releases the locks of nodes that it acquired.

Filter lock: Doorway

The `lock()` method can be split into two parts:

- ▶ a **doorway** part (which completes in a finite number of steps)
- ▶ a **waiting** part (which may include spinning, i.e., repeatedly reading variables until certain values are read)

Fairness: If a thread *i* completes its doorway before another thread *j* starts its doorway, then *i* enters its critical section before *j*.



Question: What is the doorway of the filter lock ?

Filter lock: Not fair

In the filter lock, the **doorway** of a thread i consists of setting `level[i] = 1` and `victim[1] = i`.

The filter lock isn't fair.



In the example, thread B completed its doorway before C, but C entered its critical section first.

Bakery algorithm

The bakery algorithm provides *mutual exclusion* and is *fair*.

To enter its critical section, a thread sets a **flag**, and takes a **number** greater than the numbers of all other threads.

When all lower numbers have been served, the thread can enter.

At leaving its critical section, the thread resets its **flag**.



"What would the lady at the back like?"

Complication: Threads may concurrently take the same number.

Solution: Lexicographical order:

$$(\ell, i) < (m, j) \quad \text{if } \ell < m, \text{ or } \ell = m \text{ and } i < j$$

Bakery algorithm

```
class Bakery implements Lock {  
    boolean[] flag;  
    Label[] label;  
    public Bakery (int n) {  
        flag = new boolean[n];  
        label = new Label[n];  
        for (int k = 0; k < n; k++) {  
            flag[k] = false; label[k] = 0; }  
    }  
    public void lock() { int i = ThreadID.get();  
        flag[i] = true;  
        label[i] = max(label[0],...,label[n-1]) + 1;  
        while ∃k (flag[k] && (label[k],k) < (label[i],i)) {};  
    }  
    public void unlock() { int i = ThreadID.get();  
        flag[i] = false; }  
}
```

Bakery algorithm: Example

flag[1] = true
flag[0] = true
 A_0 and A_1 read label[1] resp. label[0]
label[0] = 1
 A_0 reads flag[1] == true and (label[1],1) < (label[0],0)
label[1] = 1
 A_1 reads flag[0] == true and (label[0],0) < (label[1],1)
 A_0 reads (label[0],0) < (label[1],1)
 A_0 enters its critical section
 A_0 exits its critical section
flag[0] = false
flag[0] = true
 A_0 reads label[1]
label[0] = 2
 A_0 reads flag[1] == true and (label[1],1) < (label[0],0)
 A_1 reads (label[1],1) < (label[0],0)
 A_1 enters its critical section

(n is 2)

Bakery algorithm: Mutual exclusion

The bakery algorithm provides *mutual exclusion*.

Suppose, toward a contradiction, that two threads i and j are concurrently in their critical sections.

Let $(\text{label}[i], i) < (\text{label}[j], j)$.

When j successfully completed the test in its waiting section, it read either $\text{flag}[i] == \text{false}$ or $(\text{label}[j], j) < (\text{label}[i], i)$.

Since the `label` value of a thread only increases over time, j must have read $\text{flag}[i] == \text{false}$.

So before entering its critical section, i must have selected a label greater than $\text{label}[j]$.

This contradicts $(\text{label}[i], i) < (\text{label}[j], j)$.

Bakery algorithm: Fairness

The bakery algorithm is *fair*.

The **doorway** of a thread consists of **setting its flag** and **computing its new label**.

If thread j starts its doorway after thread i has completed it, then j will select a label greater than $\text{label}[i]$.

Since $\text{flag}[i] == \text{true}$, i will enter its critical section before j .

Question

The bakery algorithm is correct, elegant and fair.

But it doesn't scale to large, dynamic systems. Why ?

Answer 1: Labels may become arbitrarily large.

But this can be circumvented.

Answer 2: The number of threads is fixed beforehand.

Answer 3: For n threads, it requires reading n distinct variables.

With only *read/write variables*, this can't be avoided !

Registers

Shared memory locations are called **registers**.

The three most common types are:

- ▶ Single-reader single-writer (SRSW)

For example, `i` and `j` in the Peterson lock.

- ▶ Multi-reader single-writer (MRSW)

For example, `flag[]` and `label[]` in the bakery algorithm.

- ▶ Multi-reader multi-writer (MRMW)

For example, `victim` in the Peterson lock.

Question

Why is the mutual exclusion algorithm below for two threads flawed?

A MRMW register initially has the value -1 .

When thread A_0 (or A_1) wants to enter its critical section, it spins on the register until it is -1 .

Then A_0 (or A_1) writes the value 0 (or 1) into the register.

Next A_0 (or A_1) checks whether the value of the register is 0 (or 1).

If not, it returns to spinning on the register until it is -1 .

If so, it enters its critical section.

When a thread exits its critical section, it writes -1 into the register.

Lower bound on the number of registers

Theorem: At least n read/write registers are needed to solve deadlock-free mutual exclusion for n threads.

Proof (for $n = 2$): Given threads A, B , and one MRMW register R .

Before A or B can enter its critical section, it must write to R .

Bring A and B in a position where they are about to write to R , after which they perform reads, and may enter their critical sections.

Let A write to R first, perform reads, and enter its critical section.

The subsequent write by B obliterates the value A wrote to R , so B can no longer tell that A is in its critical section.

B also performs reads and enters its critical section.

Question

How does this proof idea carry over to general n ?

Answer: With only $n - 1$ registers, two threads must share a register to signal to other threads that they have entered their critical sections.

Then the scenario from the previous slide applies.

Fischer's algorithm

There are n threads A_0, \dots, A_{n-1} .

turn is a MRMW register with range $\{-1, 0, \dots, n - 1\}$.

Initially it has the value -1 .

An A_i wanting to enter its critical section, spins on *turn* until it is -1 .

Within one time unit of this read, A_i sets the value of *turn* to i .

A_i waits for more than one time unit, and then reads *turn*.

If it still has the value i , then A_i enters its critical section.

Else A_i returns to spinning on *turn* until it is -1 .

When a thread exits its critical section, it sets the value of *turn* to -1 .

Fischer's algorithm: Correctness

Fischer's algorithm guarantees **mutual exclusion**.

When $turn = -1$, no thread is in its critical section.

If a thread sets $turn$, other threads can only concurrently set $turn$ within one time unit of this first write.

Since threads re-check $turn$ one time unit after setting it, only the thread that set $turn$ last will enter its critical section.

Fischer's algorithm is **deadlock-free**.

When a thread exits the critical section, $turn$ becomes -1 .

The last thread to set $turn$ within one time unit of the first write becomes privileged.

Fischer's algorithm: Drawbacks

Not starvation-free.

Needless delay in case there is no contention.

Requires a global clock.

All threads spin on the same variable *turn*.

This lecture in a nutshell

Peterson lock

filter lock

volatile variables

fairness / doorway

bakery algorithm

n registers needed for n threads

Fischer's algorithm

Correctness of methods for sequential objects

An object has a *state* (values of variables) and a set of *methods*.

- ▶ if (precondition)
 - ▶ the object is in such-and-such a state
 - ▶ before the method is called
- ▶ then (postcondition)
 - ▶ the method call will return a particular value
 - ▶ or throw a particular exception
- ▶ and (postcondition, continued)
 - ▶ the object will be in some other state
 - ▶ when the method call returns

Pre- and postconditions: Example

Consider a **dequeue** method on a FIFO queue.

precondition: the queue is nonempty

postcondition: returns the head of the queue

postcondition: removes the head of the queue

precondition: the queue is empty

postcondition: throws `EmptyException`

postcondition: the queue remains unchanged

Why sequential objects totally rock

An object state is meaningful between method calls.

Intermediate states of the object while a method call is in progress can be ignored.

Interactions among methods depend only on side-effects on the object state.

Each method can be considered in isolation.

New methods can be added without changing the description of old methods.

Welcome to the jungle of concurrent objects

Method calls on concurrent threads can overlap (in time).

As a result, an object may *never* be between method calls.

All possible interactions between method calls must be taken into account.

What does it mean for a concurrent object to be correct ?

Linearizability

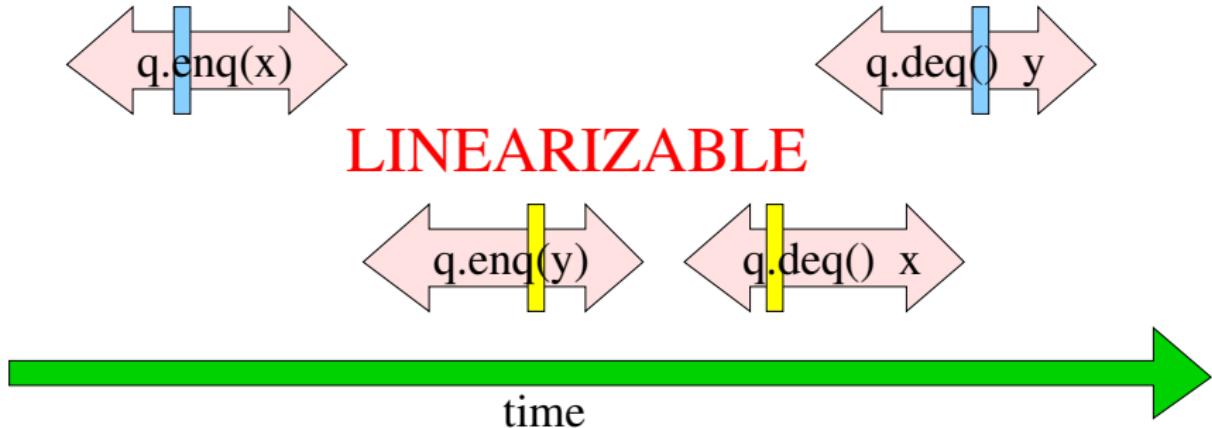
We order the method calls in an execution, by associating each of them to a single moment in time, when it is active.

That is, an **execution** on a concurrent object is **linearizable** if each method call in the execution:

- ▶ appears to take effect instantaneously,
- ▶ at a moment in time between its invocation and return events,
- ▶ in line with the system specification.

An **object** is **linearizable** if all its possible executions are linearizable.

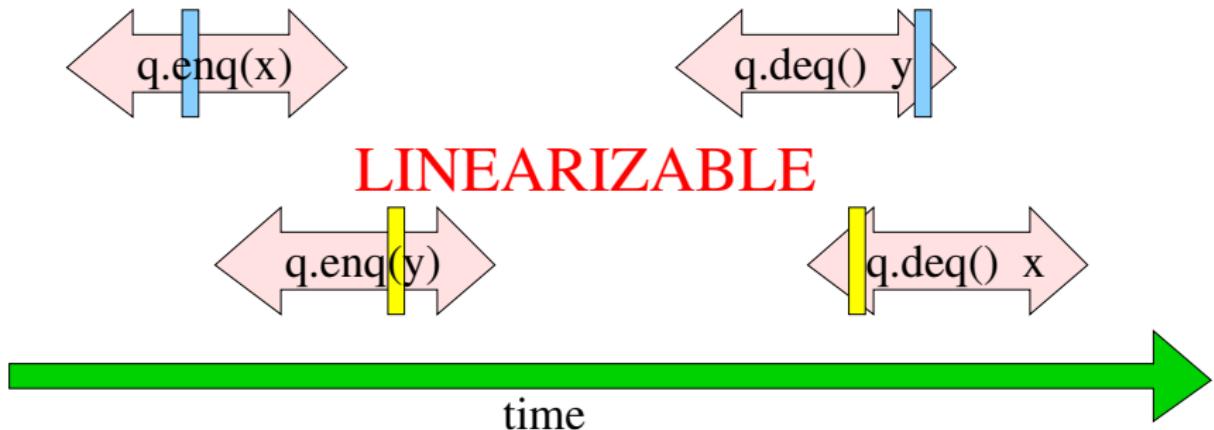
Linearizability: Example 1



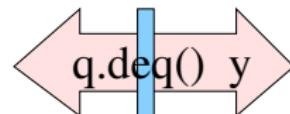
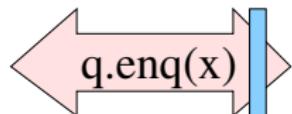
Consider a FIFO queue q .

Since x is enqueued before y (in the FIFO queue),
it should also be dequeued before y .

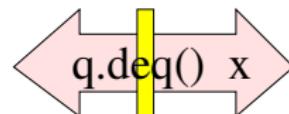
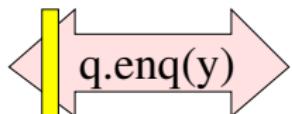
Linearizability: Example 2



Linearizability: Example 2



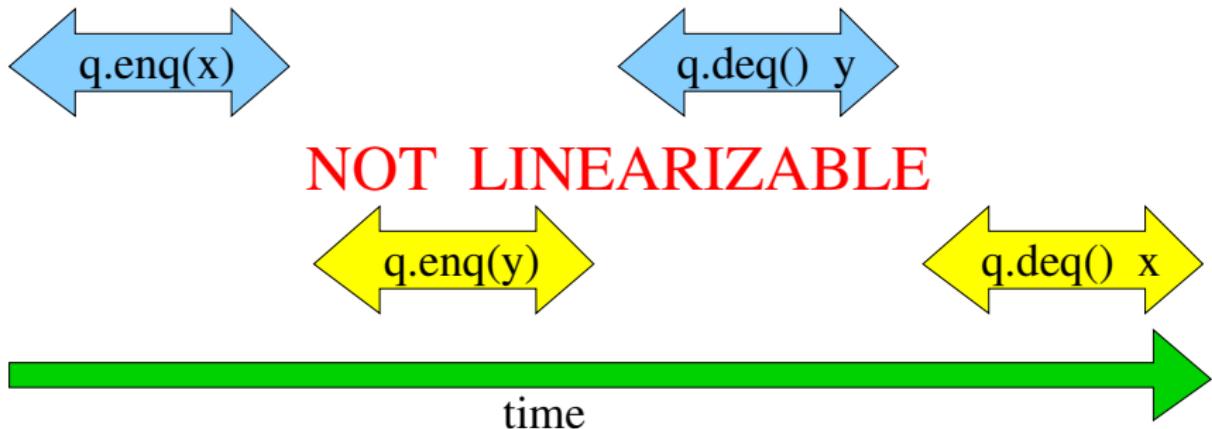
LINEARIZABLE



time

A thick green arrow pointing to the right, with the word "time" centered below it, indicating the progression of time from left to right.

Linearizability: Example 3



Linearization of unfinished method calls

For a method call that has an invocation but no return event, one can either:

- ▶ let it take effect before the end of the execution; or
- ▶ omit the method call from the execution altogether
(i.e., it didn't take effect before the end of the execution).

Wait-free bounded FIFO queue for two threads

Consider a *concurrent* bounded FIFO queue, with an enqueue (`q.enq(x)`) and a dequeue (`q.deq()`) method.

There are two threads: An enqueuer and a dequeuer.

Conflicts between the threads can be avoided by protecting the queue using a lock.

But this results in an inefficient implementation (recall *Amdahl's law*) that is vulnerable to crashes.

The next slide shows a *wait-free* implementation: The enqueuer or dequeuer can always progress by itself.

Wait-free bounded FIFO queue for two threads

```
class WaitFreeQueue<T> {
    volatile int head, tail;
    T[] items;
    public WaitFreeQueue(int capacity) {
        items = T[] new Object[capacity];
        head = 0;  tail = 0;
    }
    public void enq(T x) throws FullException {
        if tail - head == items.length throw new FullException();
        items[tail % items.length] = x;  tail++;
    }
    public T deq() throws EmptyException {
        if tail == head throw new EmptyException();
        T y = items[head % items.length];  head++;
        return y;
    }
}
```

Wait-free bounded FIFO queue: Correctness

Intuitively, this algorithm is correct for the following reasons:

- ▶ Only the *enqueuer* writes to `tail` and `items[...]`, and only the *dequeuer* writes to `head`.
- ▶ The condition `if tail - head == items.length` stops the *enqueuer* from overwriting an element in the queue before the *dequeuer* has read it.

Here it is used that `head` is **volatile**, and the dequeuer only increases `head` after it read `items[head % items.length]`.

- ▶ The condition `if tail == head` stops the *dequeuer* from reading an element in the queue before the *enqueuer* has placed it in the queue.
- Here it is used that `tail` is **volatile**, and the enqueueuer only increases `tail` after it wrote to `items[tail % items.length]`.

Question

Why don't we need to declare the slots in the `items` array volatile?

Answer: Reading or writing to a volatile variable (here `head` or `tail`) imposes a memory barrier in which the entire cache is flushed/invalidated.

Wait-free bounded FIFO queue is linearizable

The **linearization point** of a *successful* `enq()` is at `tail++`.

The **linearization point** of an *unsuccessful* `enq()` is at (reading the `head` value to perform) `if tail - head == items.length`.

The **linearization point** of a *successful* `deq()` is at `head++`.

The **linearization point** of an *unsuccessful* `deq()` is at (reading the `tail` value to perform) `if tail == head`.

So any execution of the wait-free bounded FIFO queue is linearizable.

Questions

Why is it **too late** to linearize an *unsuccessful* `enq()` at

```
throw new FullException() ?
```

Why is it **too early** to linearize a *successful* `enq()` at

```
if tail - head == items.length ?
```

Flawed bounded FIFO queue

In the dequeuer, let's swap the order of two program statements:

```
head++; T y = items[(head - 1) % items.length];
```

Let `items.length` be 1.

Suppose the enqueueer performs the methods `enq(a)` and `enq(b)`, and the dequeuer performs the method `deq()`.

Let `head` and `tail` be 0. The following execution isn't linearizable:

enq: tail - head == 0	enq: tail - head == 0
enq: items[0] = a	enq: items[0] = b
enq: tail = 1	deq: y = b
deq: tail - head == 1	deq: return b
deq: head = 1	

Sequential consistency

For multiprocessor memory architectures, linearizability is often considered too strict.

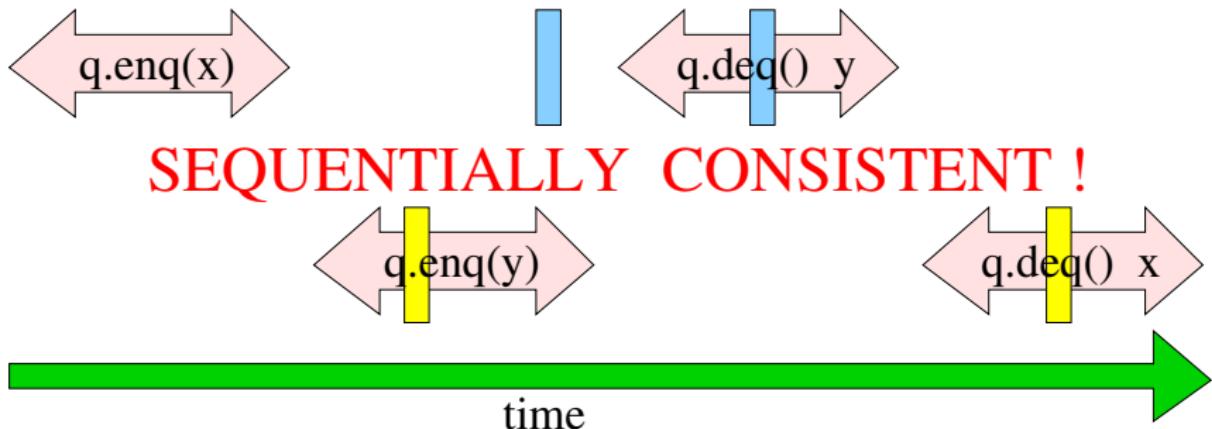
An **execution** on a concurrent object is **sequentially consistent** if each method call in the execution:

- ▶ appears to take effect instantaneously,
- ▶ *in program order on each thread,*
- ▶ in line with the system specification.

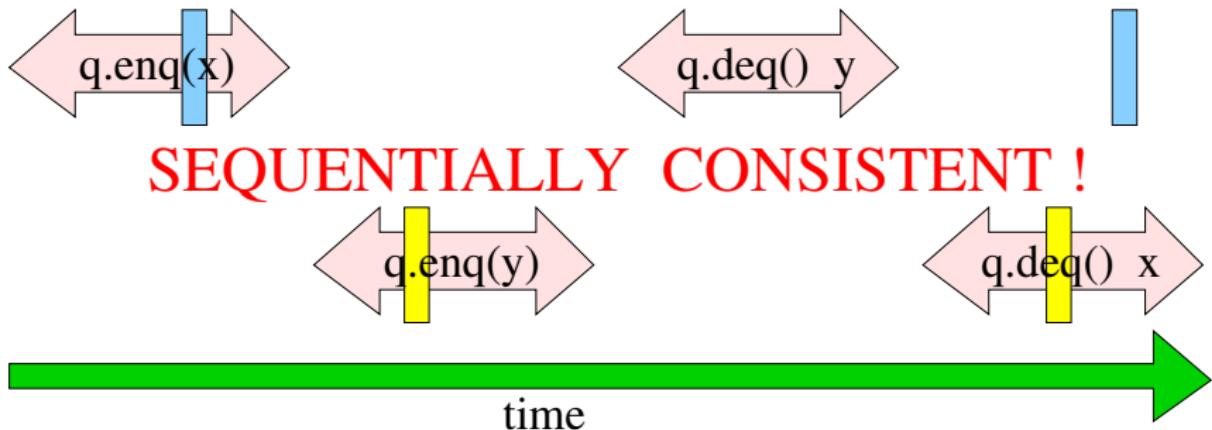
An **object** is **sequentially consistent** if all its possible executions are sequentially consistent.

Sequential consistency is less restrictive than linearizability, because it allows method calls to take effect *after* they returned.

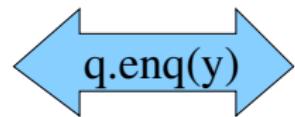
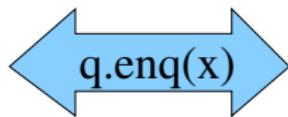
Sequential consistency: Example 1



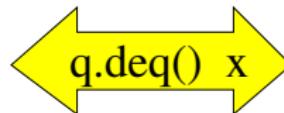
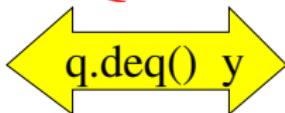
Sequential consistency: Example 1



Sequential consistency: Example 2



NOT SEQUENTIALLY CONSISTENT



time

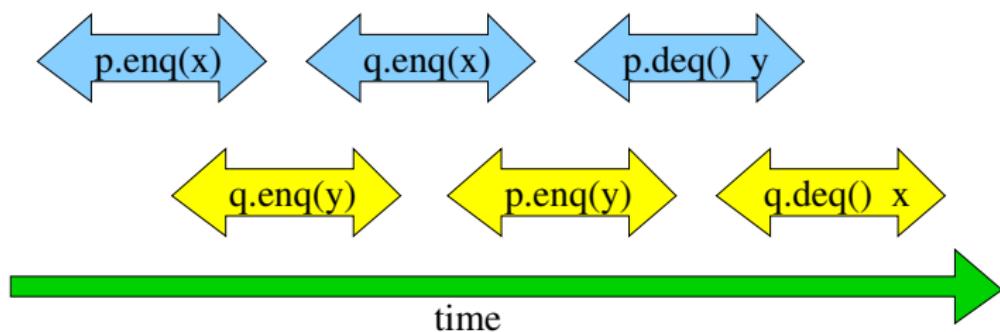
A thick green arrow pointing to the right, representing the progression of time. The word "time" is centered below the arrow.

Compositionality

Linearizability is **compositional**:

The composition of linearizable objects is again linearizable.

By contrast, **sequential consistency** isn't compositional.



The executions for objects p and q are by themselves sequentially consistent, but their composition isn't.

Out-of-order execution

Most hardware architectures *don't* support sequential consistency, as it would outlaw widely used compiler optimizations.

A processor executes instructions ordered by the availability of input data, rather than by their order in the program.

This makes sense because the vast majority of instructions isn't for synchronization.

Reads/writes for synchronization should be announced explicitly.

This comes with a performance penalty.

We stick to linearizability

Which correctness requirement is right for a given application ?

This depends on the needs of the application.

- ▶ A printer server can allow jobs to be reordered.
- ▶ A banking server better be sequentially consistent.
- ▶ A stock-trading server requires linearizability.

We will stick to linearizability, as it is well-suited for high-level objects.

Safe registers

A *single-writer Boolean* register is **safe** if every `read()` that doesn't overlap with a `write()` returns the last written value.

A `read()` that overlaps with a `write()` may return any value.

At the hardware level, only safe registers are provided:

- ▶ The writer sets a voltage level either high or low (i.e., 1 or 0).
- ▶ Setting a level high when it is already high may cause a temporary perturbation of the level.

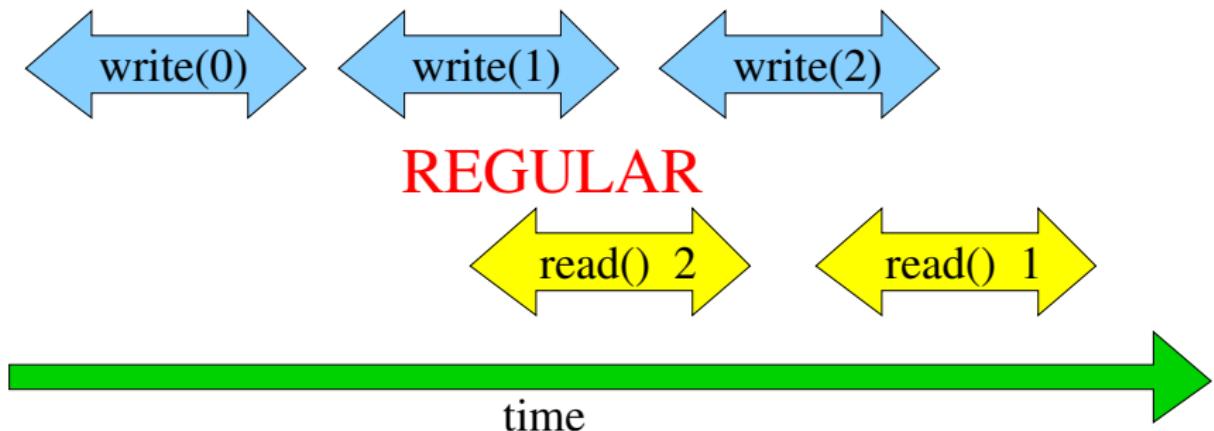
Regular and atomic registers

A *single-writer* register is **regular** if:

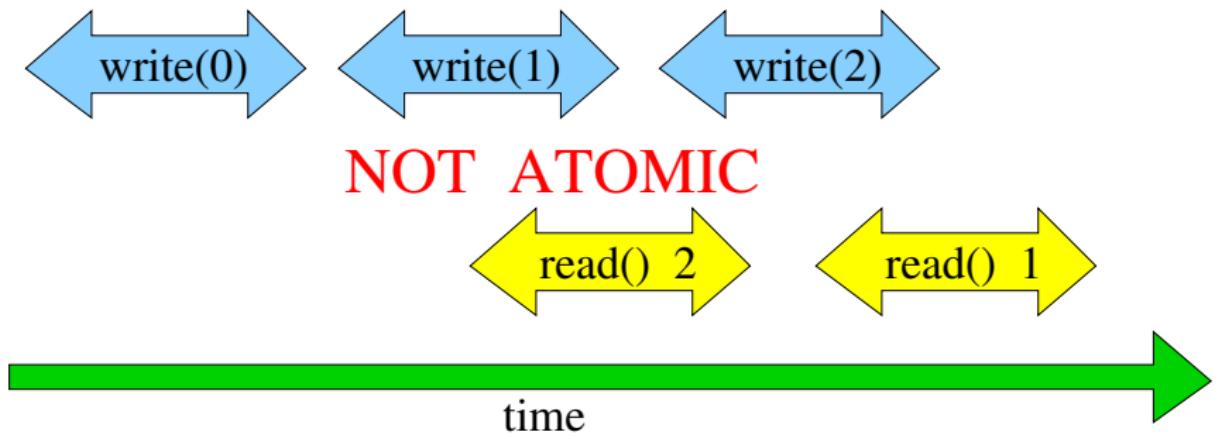
- ▶ it is safe; and
- ▶ every `read()` that overlaps with `write()`'s returns either one of the values written by these overlapping `write()`'s, or the last written value before the `read()`.

A (possibly multi-writer) register is **atomic** if it is linearizable to the sequential register (on a uniprocessor).

Regular and atomic registers: Example



Regular and atomic registers: Example



The first and second `read()` would have to be linearized after and before `write(2)`, respectively.

From safe MRSW to regular Boolean MRSW

From **safe SRMW** registers, we build **atomic MRMW** registers.

As a first step, a **safe** Boolean MRSW register is turned into a **regular Boolean** MRSW register.

Before a write, first the value in the (safe) register is checked.

If the written value is the same as the current value, then the write leaves the register untouched.

Only if the written value is different from the current value, the write is performed physically.

From safe MRSW to regular Boolean MRSW: Correctness

If a read doesn't overlap with a write, then clearly it returns the last written value.

Suppose a read overlaps with writes.

- ▶ If all writes leave the value unchanged, no write is physically performed, so the read returns the last written value.
- ▶ If a write changes the value, then the read is allowed to return both a 0 and a 1.

From regular Boolean MRSW to regular M -valued MRSW

An M -valued register can contain values $0, 1, \dots, M - 1$.

It is represented by M (regular) Boolean registers.

The value of the M -valued register is k if its Boolean registers $0, \dots, k - 1$ contain 0 while the k th Boolean register contains 1.

Write value k : Write 1 in the k th Boolean register.

Then write 0 in Boolean registers $k - 1, \dots, 0$.

Read: Read the Boolean registers from 0 up to $M - 1$, until the value 1 is encountered. The index of this slot is returned.

From regular Boolean MRSW to regular M -valued MRSW

Let k be the last written value in the M -valued register before a certain read.

If the read doesn't overlap with a write, then clearly it returns k .

If the read returns an $\ell < k$, then the 1 in the ℓ th Boolean register was written by an overlapping write.

Suppose the read encounters 0 at the k th Boolean register.

Then an overlapping write wrote 1 at an ℓ th Boolean register with $\ell > k$, and 0 at Boolean registers $\ell - 1, \dots, k$.

This implies the read will return the value of an overlapping write.

Questions

What could go wrong if a write of value k would first write 0 in Boolean registers $k - 1, \dots, 0$ and then 1 in Boolean register k ?

Answer: A read could then encounter M 0's.

What could go wrong if a write of value k would first write 1 in Boolean register k and then 0 in Boolean registers $0, \dots, k - 1$?

Answer: A read could then return an obsolete value between the last written value and k .

Question

How can a **regular** SRSW register (with the reader and the writer distinct threads) be made **atomic**?

(*Hint: Let the writer use timestamps.*)

From regular SRSW to atomic SRSW

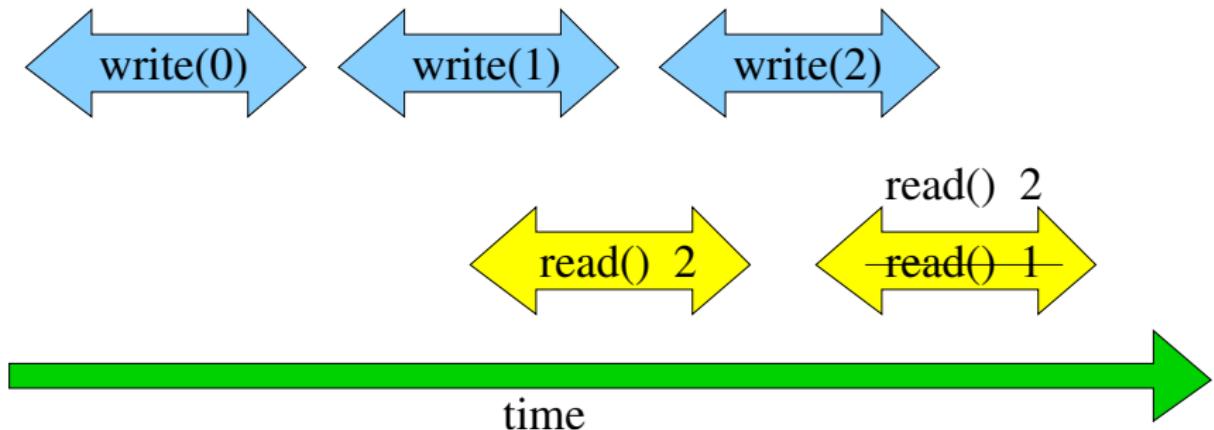
The **writer** to the (regular) SRSW register provides each write with a **timestamp** (*provided by hardware*), which increases at each write.

(The initial value of the register carries the timestamp 0.)

The **reader** remembers the *latest* value/timestamp pair it ever read (i.e., the pair with the greatest timestamp).

If it reads a value with a timestamp smaller than the previous read, the reader ignores that value, and uses the remembered last value.

From regular SRSW to atomic SRSW: Example



Linearize `write(2)` before the two reads.

Question

Where do we have to be careful ?

(*Hint: It concerns the value/timestamp pair.*)



Pointers for atomic updates of multiple registers

The value/timestamp pair must be a single unit for reads/writes.

In Java, one can let the register contain a reference to such a pair.

First a new pair is built, and next the reference is updated.

Properties of regular and atomic registers

Let every read value be written at some point.

Let W^i be the i th write on a (single-writer or linearizable) register, and R^i a read of the corresponding value.

The value of W^i is indexed by i , so that it is unique.

W^0 “writes” the initial value at the start; R^0 ’s read this value.

A register is **regular** if:

- ▶ never $R^i \rightarrow W^i$; and
- ▶ never $W^i \rightarrow W^j \rightarrow R^i$.

A register is **atomic** if moreover:

- ▶ $R^i \rightarrow R^j$ implies $i \leq j$.

From regular SRSW to atomic SRSW: Correctness

Since the original register is regular, clearly never $R^i \rightarrow W^i$ or $W^i \rightarrow W^j \rightarrow R^i$.

Owing to the timestamps, a read never returns an older value than earlier reads.

That is, if $R^i \rightarrow R^j$, then $i \leq j$.

From atomic SRSW to atomic MRSW

Given n readers and one writer. First we discuss an *incorrect* attempt.

The MRSW register consists of n atomic SRSW registers, one for each **reader**.

The **writer** writes a value to each SRSW register (one at a time).

The following scenario shows that this MRSW register isn't atomic:

- (1) The writer starts writing to the MRSW register, by writing a new value to the SRSW register of reader A .
- (2) A reads the *new* value in its SRSW register, and returns.
- (3) B reads the *old* value in its SRSW register, and returns.
- (4) The writer writes the new value to the SRSW register of B .

From atomic SRSW to atomic MRSW

Given n readers A_i (for $i = 0, \dots, n - 1$), and one writer.

The MRSW register consists of $n \times n$ atomic SRSW registers $a_table[0..n - 1][0..n - 1]$ with timestamped values.

- ▶ The writer can write to the registers $a_table[i][i]$.
- ▶ Each A_i can write to the registers $a_table[i][j]$ for all $j \neq i$.

The writer writes a value/timestamp to $a_table[i][i]$ for each i ; the timestamp is increased at each write call.

Each reader A_i at a read:

- ▶ reads $a_table[j][i]$ for all j , and picks the value with the highest timestamp; and
- ▶ writes this value/timestamp to $a_table[i][j]$ for all $j \neq i$.

From atomic SRSW to atomic MRSW: Example 1

	0	1
0	t	t
1	t	t

$n = 2$, and all slots in `a_table` carry value u and timestamp t .

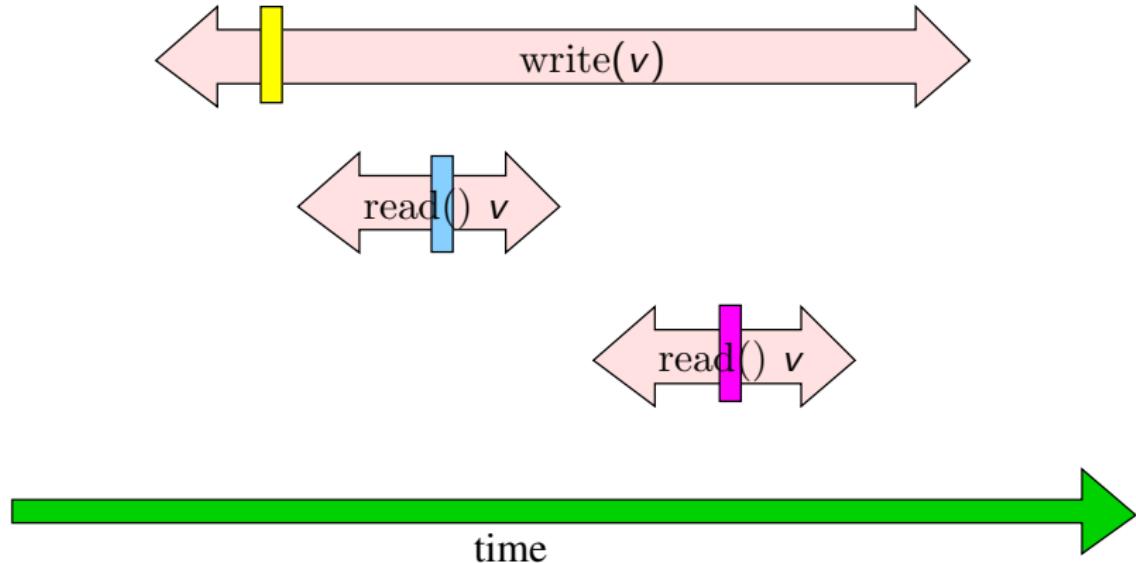
A **write** starts, and writes a new value v with timestamp $t+1$ to `a_table[0, 0]`.

A **read** by A_0 starts, reads `a_table[0][0]` and `a_table[1][0]`, and selects $v/t+1$.

Next it writes $v/t+1$ to `a_table[0][1]`, and returns v .

A **read** by A_1 starts, reads `a_table[0][1]` and `a_table[1][1]`, and selects $v/t+1$.

From atomic SRSW to atomic MRSW: Example 1



From atomic SRSW to atomic MRSW: Example 2

	0	1
0	t	t
1	t	t

$n = 2$, and all slots in `a_table` carry value u and timestamp t .

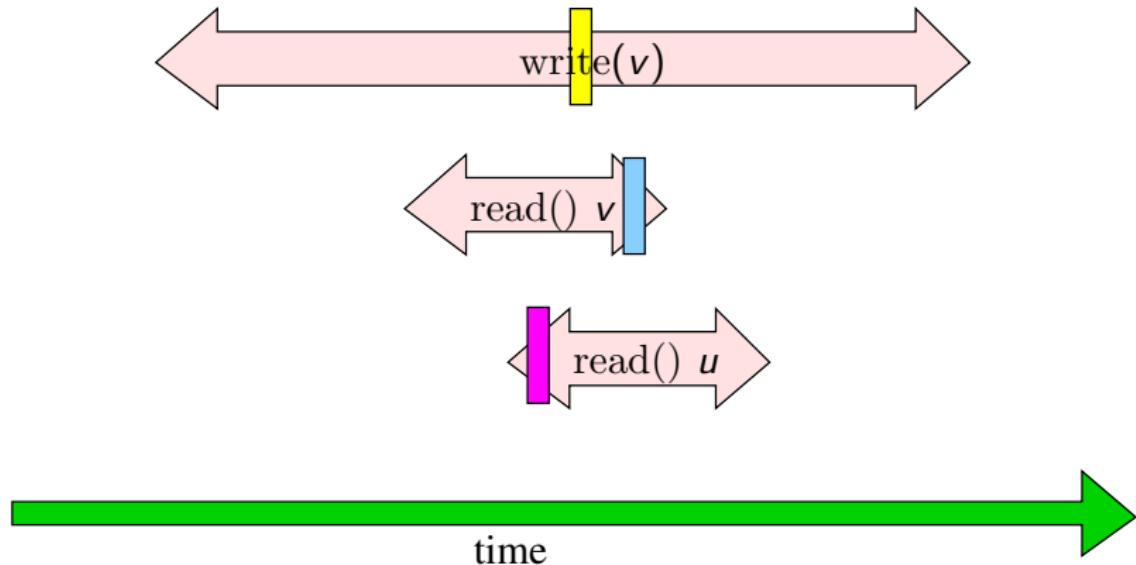
A **write** starts, and writes a new value v with timestamp $t+1$ to `a_table[0, 0]`.

A **read** by A_0 starts, reads `a_table[0][0]` and `a_table[1][0]`, and selects $v/t+1$.

A **read** by A_1 starts, reads `a_table[0][1]` and `a_table[1][1]`, and selects the *old* value u with timestamp t .

A_0 writes $v/t+1$ to `a_table[0][1]`, and returns v .

From atomic SRSW to atomic MRSW: Example 2



From atomic SRSW to atomic MRSW: Correctness

Clearly never $R^i \rightarrow W^i$.

Each write call overwrites the diagonal of `a_table` by values with a higher timestamp.

Read calls consider a pair on the diagonal, and preserve the diagonal.

This guarantees that never $W^i \rightarrow W^j \rightarrow R^i$.

Suppose a read by A_k completely precedes a read by A_ℓ .

Let the read by A_k return a value v with timestamp t .

A_k writes v/t to `a_table`[k][ℓ] before the read by A_ℓ starts.

A_ℓ reads `a_table`[m][ℓ] for all m , so `a_table`[k][ℓ] in particular.

Therefore the read by A_ℓ returns a value with a timestamp $\geq t$.

Hence, if $R^i \rightarrow R^j$, then $i \leq j$.

From atomic MRSW to atomic MRMW

Given n readers/writers A_i , for $i = 0, \dots, n - 1$.

The MRMW register consists of n atomic MRSW registers $a_table[0..n - 1]$ with *timestamped* values.

Each A_i , to **write** a value v :

- ▶ reads $a_table[j]$ for all j ;
- ▶ picks a timestamp t , higher than any it observed; and
- ▶ writes v/t to $a_table[i]$.

Each A_i , to **read**:

- ▶ reads $a_table[j]$ for all j ; and
- ▶ returns a value with the highest timestamp; if multiple registers carry this timestamp, it takes the one with the largest index.

From atomic MRSW to atomic MRMW: Example 1

0	1	2	3
$w/t-2$	$x/t+1$	y/t	$z/t+1$

A_0 and A_1 start a **write** of value v_0 and v_1 , respectively.

They concurrently read the registers, and both pick timestamp $t+2$.

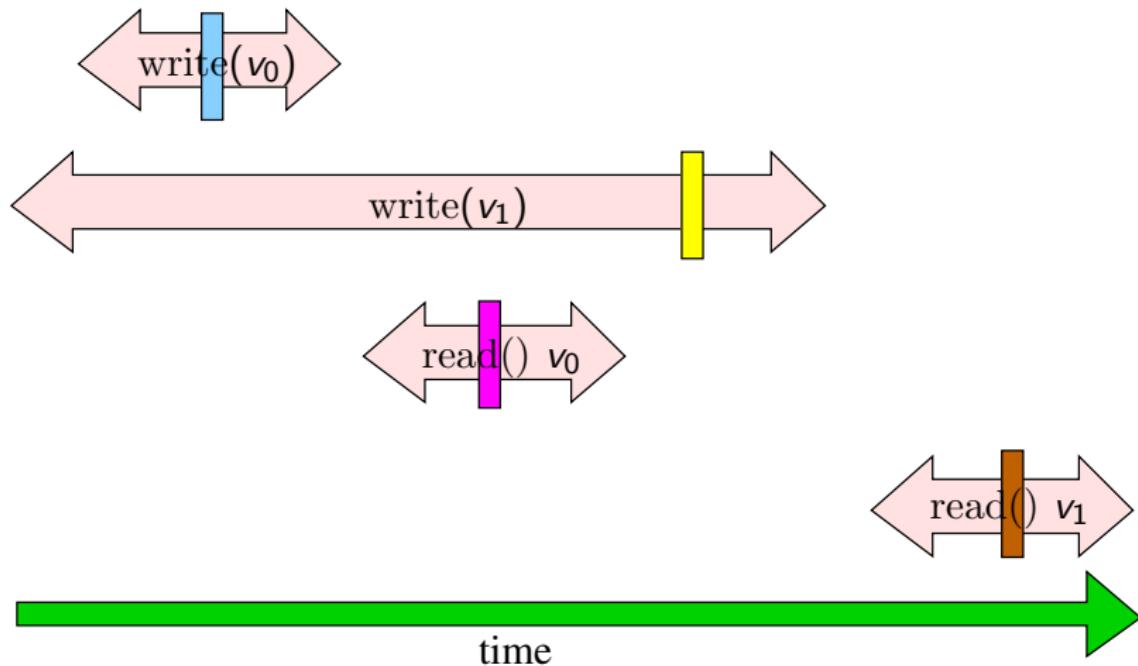
A_0 **writes** $v_0/t+2$ to `a_table[0]`.

A_2 starts a **read**. It reads the registers, and **returns** v_0 .

A_1 **writes** $v_1/t+2$ to `a_table[1]`.

A_3 starts a **read**. It reads the registers, and **returns** v_1 .

From atomic MRSW to atomic MRMW: Example 1



From atomic MRSW to atomic MRMW: Example 2

0	1	2	3
$w/t-2$	$x/t+1$	y/t	$z/t+1$

A_0 and A_1 start a **write** of value v_0 and v_1 , respectively.

They concurrently read the registers, and both pick timestamp $t+2$.

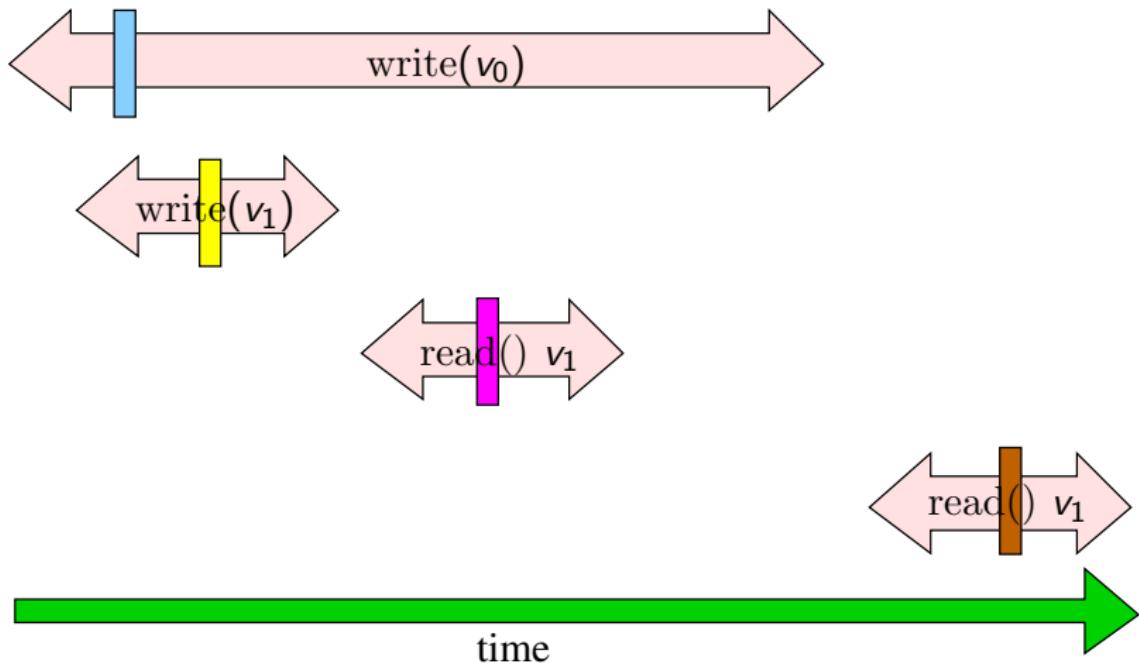
A_1 **writes** $v_1/t+2$ to `a_table[1]`.

A_2 starts a **read**. It reads the registers, and **returns** v_1 .

A_0 **writes** $v_0/t+2$ to `a_table[0]`.

A_3 starts a **read**. It reads the registers, and **returns** v_1 .

From atomic MRSW to atomic MRMW: Example 2



From atomic MRSW to atomic MRMW: Correctness

W^i denotes the i th write, with regard to a linearization order on writes.

Clearly never $R^i \rightarrow W^i$.

Suppose a *write* by A_k completely precedes a *write* by A_ℓ .

A_k writes its value/timestamp in `a_table[k]`.

So A_ℓ will pick a timestamp greater than the timestamp of A_k .

Hence subsequent reads will never return the write from A_k .

So never $W^i \rightarrow W^j \rightarrow R^i$.

From atomic MRSW to atomic MRMW: Correctness

Suppose a *read* by A_k completely precedes a *read* by A_ℓ .

Let the read by A_k return a value from `a_table[i]` with timestamp t .

The read by A_ℓ reads in `a_table[i]` a value with a timestamp $\geq t$.

So it will return a value with a timestamp $\geq t$.

And if A_ℓ returns a value with timestamp t ,

then this pair must originate from `a_table[j]` for some $j \geq i$.

We **linearize** writes *lexicographically on timestamp and index*.

(Note that writes with the same timestamp must overlap in time.)

So the write corresponding to the read by A_k is linearized no later than the write corresponding to the read by A_ℓ .

Hence, if $R^i \rightarrow R^j$, then $i \leq j$.

Examples 1 and 2 revisited

Both in example 1 and in example 2, the write by A_0 is linearized before the write by A_1 because:

- ▶ they have equal timestamps, and
- ▶ A_0 has the smaller index.

Question

Suppose reading is adapted as follows:

*if multiple registers carry the largest timestamp,
take the one with the **smallest** index.*

How should we then adapt the argumentation for

if $R^i \rightarrow R^j$, then $i \leq j$.

Answer: A write by A_k is linearized before a write by A_ℓ if

- ▶ either it has a smaller timestamp,
- ▶ or the same timestamp and $k > \ell$.

This lecture in a nutshell

linearizability

wait-free bounded FIFO queue with one reader and one writer

sequential consistency

safe, regular and atomic registers

from regular SRSW registers one can build atomic MRMW registers

Consensus

A fundamental problem in distributed computing is to guarantee reliability if processes crash.

Consensus: Threads need to agree on a decision (while some of them may crash).

Basically, each concurrent program needs to solve consensus problems.

Example: Agree on whether to commit a distributed transaction to a database.



Search ID: cwin350
"Everyone **finally** agrees: the only consensus is there's no consensus."

Consensus protocol

Each thread randomly chooses an *input* value 0 or 1, and (if it doesn't crash) eventually *decides* for 0 or 1.

In a **(binary) consensus protocol**, each execution satisfies two requirements:

- ▶ **Consensus**: All (non-crashed) threads eventually decide for the same value.
- ▶ **Validity**: This value is some thread's input.

We will aim for **wait-free** consensus protocols.

Wait-free consensus is tricky because the thread that enforces the decision may crash immediately after this event.

Lock- and wait-free consensus coincide

Since in a consensus protocol all executions are finite, lock- and wait-free consensus coincide.

Namely, each thread eventually either terminates or crashes.

Hence lock-freeness implies that *each* alive thread can eventually proceed and decide.

Question: How can consensus be achieved if we are allowed to use a lock?

Consensus: Bivalent and critical states

A state of a consensus protocol is **bivalent** if it exhibits executions to decisions 0 and 1. Else it is **univalent**.

Lemma: Each wait-free consensus protocol has a *bivalent initial* state.

If one thread gets input 0 and another 1, then by wait-freeness, either of them can *run solo*, and decide for its value.

A state is **critical** if:

- ▶ it is **bivalent**; and
- ▶ any move by a thread results in a **univalent** state.

Lemma: Every wait-free consensus protocol has a *critical* state.

Else there would be an infinite execution visiting only bivalent states.

No consensus with atomic registers

Theorem: Wait-free 2-thread consensus can't be solved by atomic registers.

Proof: Suppose toward a contradiction that a solution does exist.

Given (deterministic) threads A and B . Consider a *critical* state s .

Let a move from A lead to decision 0, and a move from B to decision 1.

- ▶ If A does a **read**, then B can still run solo to a decision 1.
Likewise, if B does a **read**, A can still run solo to a decision 0.
- ▶ Let A and B do **writes** to *different* registers.
Both orders of these two moves lead to the same state.
- ▶ Let A and B do **writes** to the *same* register.
If A does its write, then B can still run solo to a decision 1.
Likewise, if B does its write, A can still run solo to a decision 0.

All cases contradict the fact that s is critical.

2-thread consensus with a FIFO queue

Theorem: Wait-free 2-thread consensus can be solved by a wait-free FIFO queue with a dequeue method.

Proof: Given two threads.

The queue initially contains two items: *WIN* and *LOSE*.

Each thread *first writes its value in a MRSW register*, and then dequeues an item from the queue.

- ▶ If it dequeues *WIN*, it decides for its own value.
- ▶ If it dequeues *LOSE*, it gets and decides for the value of the other thread.

2-thread consensus with a FIFO queue

Corollary: It is *impossible* to implement a **wait-free FIFO queue** with two dequeuers using only atomic registers.

(Earlier we saw a wait-free implementation using atomic registers of a FIFO queue with one enqueuer and one dequeuer.)

Likewise one can show that it is impossible to implement a wait-free **stack**, **list** or **set** using only atomic registers.

No 3-thread consensus with FIFO queues

Theorem: Wait-free 3-thread consensus can't be solved by wait-free FIFO queues (with only the enqueue and dequeue methods).

Proof: Suppose toward a contradiction that a solution does exist.

Given threads A , B and C . Consider a *critical* state s .

Let a move from A lead to decision 0, and a move from B to decision 1.

The following cases all contradict the fact that s is critical.

- ▶ Let A and B perform moves on *different* queues.
Both orders of these two moves lead to the same state.
- ▶ Let A and B perform **dequeues** on the *same* queue (and crash).
 C can't (on its own) distinguish in which order they were done.

No 3-thread consensus with FIFO queues

- ▶ Let A enqueue and B dequeue on the same queue (and crash).
If the queue is nonempty, these two moves lead to the same state (because they operate on different ends of the queue).
If the queue is empty, C can't distinguish the dequeue of B followed by the enqueue of A , from only the enqueue of A .
- ▶ Likewise if A dequeues and B enqueues on the same queue.
- ▶ Let A enqueue a and B enqueue b on the same queue.
Let A run solo until it dequeues the a or b (and crash).
(This must happen before A can decide between 0 or 1, for else A can't determine whether B enqueued first.)
Next let B run solo until it dequeues the b or a (and crash).
Now C can't distinguish whether a or b was enqueued first.

Read-modify-write operations

On the *hardware* level, before a **read** or **write** operation is performed, first the bus (between processors and memory) must be *locked*.

A **read-modify-write** operation allows a read followed by a write, while in the meantime the lock on the bus is kept.

The written value is determined using the value returned by the read.

In Java these operations can be performed on an **AtomicInteger**.

Remark: Threads crash as a consequence of a hardware instruction.

This means they can't crash during a read-modify-write instruction and keep the lock on the bus.

(At a hardware crash, no correctness guarantees can be given.)

Read-modify-write operations

Some standard read-modify-write operations for AtomicInteger in Java:

- ▶ `getAndSet(v)`: Assign v, and return the prior value.
- ▶ `getAndIncrement()`: Add 1, and return the prior value.
- ▶ `get()` returns the value of the register.
- ▶ `compareAndSet(e,u)`: If the prior value is e, then replace it by u, else leave it unchanged;
return a Boolean to indicate whether the value was changed.

It is advisable to use read-modify-write operations sparingly:

- ▶ They take significantly more clock cycles to complete than an atomic register.
- ▶ They include a barrier, invalidate cache lines, and prevent out-of-order execution and various other compiler optimizations.

Question

`testAndSet()` writes *true* in a Boolean register and returns the prior value.

How can wait-free 2-thread consensus be solved with `testAndSet()` ?

Answer: Let a MRMW register contain *false*.

Each thread first writes its value in a MRSW register.

Then it performs `testAndSet()` on the MRMW register:

- ▶ If it returns *false*, it decides for its own value.
- ▶ If it returns *true*, it gets and decides for the value of the other thread.

Commuting/overwriting read-modify-write operations

Theorem: Let F be a set of functions such that for all $f_i, f_j \in F$ and values v , $f_i(f_j(v)) = f_j(f_i(v))$ or $f_i(f_j(v)) = f_i(v)$ or $f_j(f_i(v)) = f_j(v)$.

3-thread consensus can't be solved by read-modify-write operations using only functions in F .

Many early read-modify-write operations were in this category, e.g.

- ▶ testAndSet (IBM 360)
- ▶ fetchAndAdd (NYU Ultracomputer)
- ▶ swap (original SPARCcs)

Due to their limited synchronization capabilities, they fell from grace.

Commuting/overwriting read-modify-write operations

Proof: Suppose a *wait-free* 3-thread consensus protocol does exist.

Given threads A , B and C . Consider a *critical* state s .

Let a move from A lead to decision 0, and a move from B to decision 1.

If A and B perform read-modify-write moves on *different* registers, then both orders of these two moves lead to the same state.

Let A and B perform f_A and f_B on the *same* register (and crash).

- ▶ If $f_A(f_B(v)) = f_B(f_A(v))$, then C can't distinguish in which order these moves were performed.
- ▶ If $f_A(f_B(v)) = f_A(v)$, then C can't distinguish whether first B and then A , or only A moved.
- ▶ Likewise if $f_B(f_A(v)) = f_B(v)$.

All cases contradict the fact that s is critical.

compareAndSet isn't commuting or overwriting

Consider a register with the value 0.

- ▶ First compareAndSet(0, 1) and then compareAndSet(1, 2) yields the final value 2.
- ▶ First compareAndSet(1, 2) and then compareAndSet(0, 1) yields the final value 1.

Consider a register with the value 0.

- ▶ First compareAndSet(0, 1) and then compareAndSet(1, 2) yields the final value 2.
- ▶ Only compareAndSet(1, 2) yields the final value 0.

Consider a register with the value 1.

- ▶ First compareAndSet(1, 2) and then compareAndSet(0, 1) yields the final value 2.
- ▶ Only compareAndSet(0, 1) yields the final value 1.

n -thread consensus with compareAndSet

Theorem: Wait-free n -thread consensus can be solved by compareAndSet, for any n .

Proof: The register initially contains *FIRST*.

Each thread performs `compareAndSet(FIRST, v)`, with v its input value.

- ▶ if *true* is returned, it decides for v ;
- ▶ if *false* is returned, it gets and decides for the value in the register.

n-thread consensus with compareAndSet

Actually threads store their value in a MRSW array slot beforehand, and write their index in the MRMW register.

```
private final int FIRST = -1;
private AtomicInteger r = new AtomicInteger(FIRST);
public Object decide(Object value) {
    int i = ThreadID.get()
    proposed[i] = value
    if r.compareAndSet(FIRST, i)
        return proposed[i]
    else
        return proposed[r.get()]
}
```

Universality of consensus

The wait-free consensus protocol for any number of threads can be used to obtain a wait-free implementation of *each* object.

First we describe a **lock-free** implementation.

Next we will adapt it to a **wait-free** implementation.

A lock-free universal construction

The methods that have been applied to the object are placed in an (unbounded) *linked list*.

A thread that wants to apply a method to the object, creates a node ν holding this method call.

The thread repeatedly tries to let the head of the list point to ν , by participating in a consensus protocol.

When successful, the thread performs *all* method calls in the list (stored in local memory) to a private copy of the object, to compute the state that results from its own method call.

(Admittedly, this is not very efficient...)

A lock-free universal construction: head array

The head of the list cannot be tracked by a consensus object, as

- ▶ the head is updated repeatedly, and
- ▶ a consensus object can only be accessed once by each thread.

Traversing the entire list over and over again to find the head would be very clumsy.

Solution: Given n threads with id's $0, \dots, n-1$.



The **array head** contains n MRSW pointers to nodes in the linked list.

`head[i]` points to the last node in the list that thread i observed.

Initially they all point to a sentinel node *tail* with sequence number 1.

A lock-free universal construction: Nodes

Each node in the **linked list** contains:

- ▶ a method call
- ▶ a sequence number
- ▶ a consensus object
- ▶ a pointer to the next node in the list
(or *null* in case of the head of the list)

A lock-free universal construction

Suppose thread i wants to apply a method to the object.

It creates a node ν holding this method call, with sequence number 0.

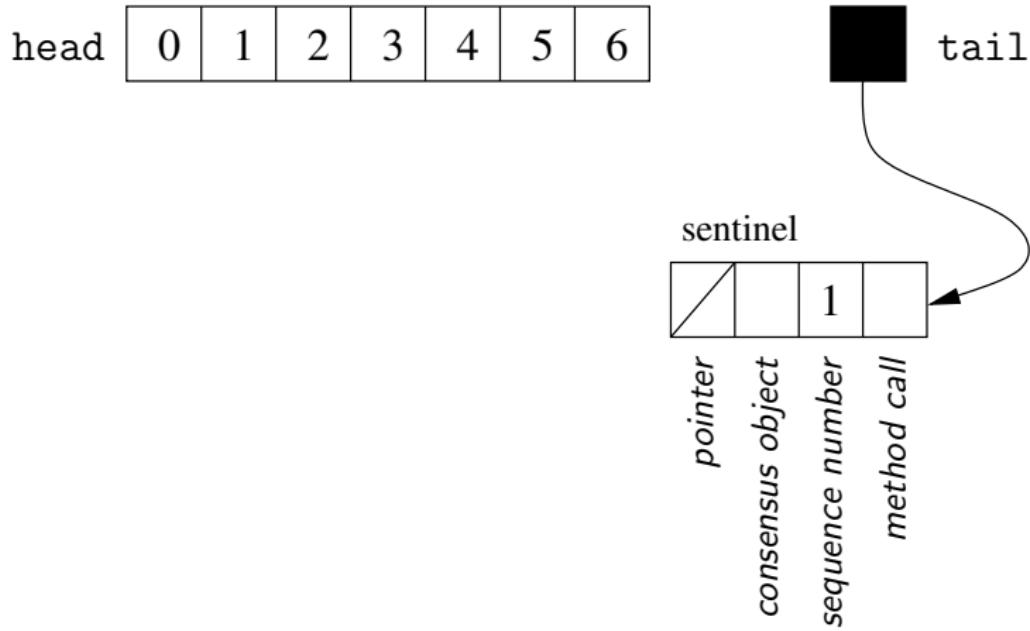
Thread i repeatedly (until it wins) does the following:

- ▶ To determine the head of the list, traverse the array `head`, and return the node ν' with the highest sequence number m .
- ▶ Take part in the consensus object of ν' .

If thread i wins, it:

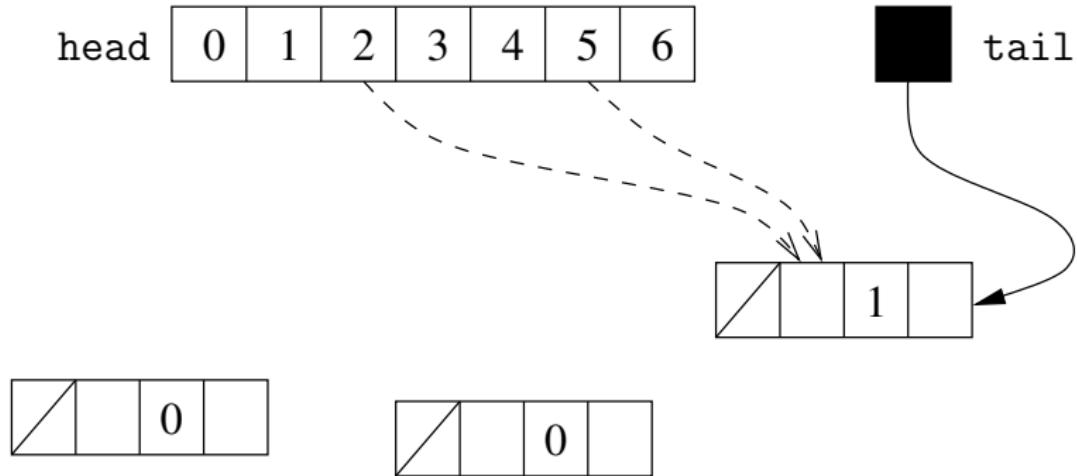
- ▶ lets ν' point to ν ,
- ▶ sets the sequence number of ν to $m + 1$, and
- ▶ lets `head[i]` point to ν .

A lock-free universal construction: Example



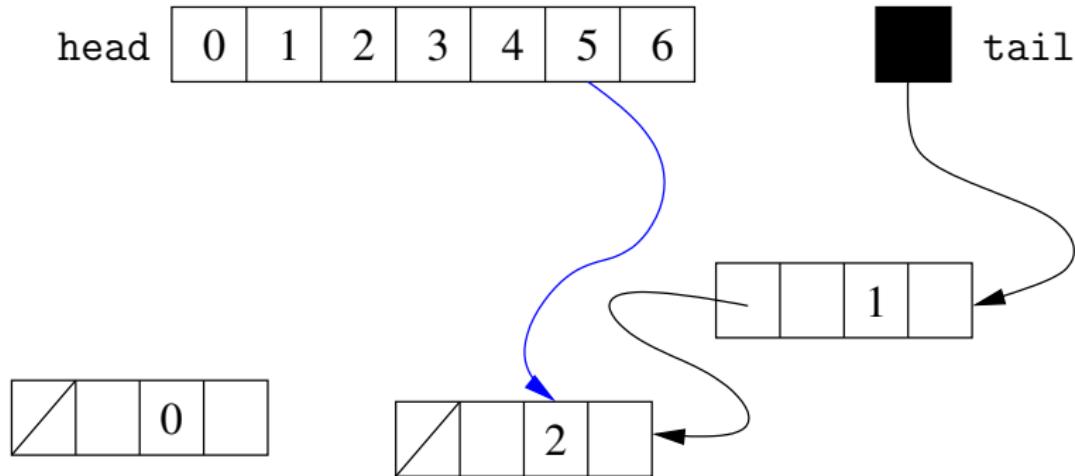
All entries of the array **head** initially point to the sentinel node.

A lock-free universal construction: Example



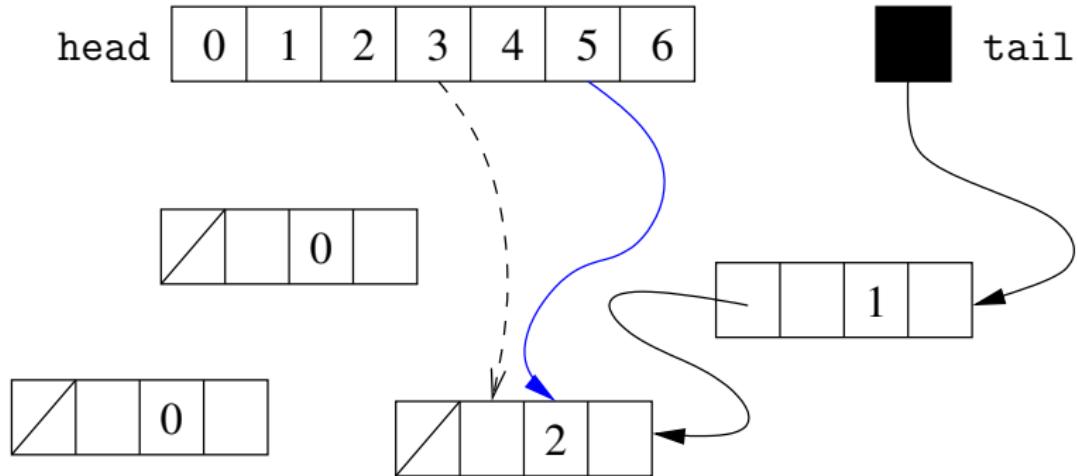
Threads 2 and 5 want to invoke a method call.

A lock-free universal construction: Example



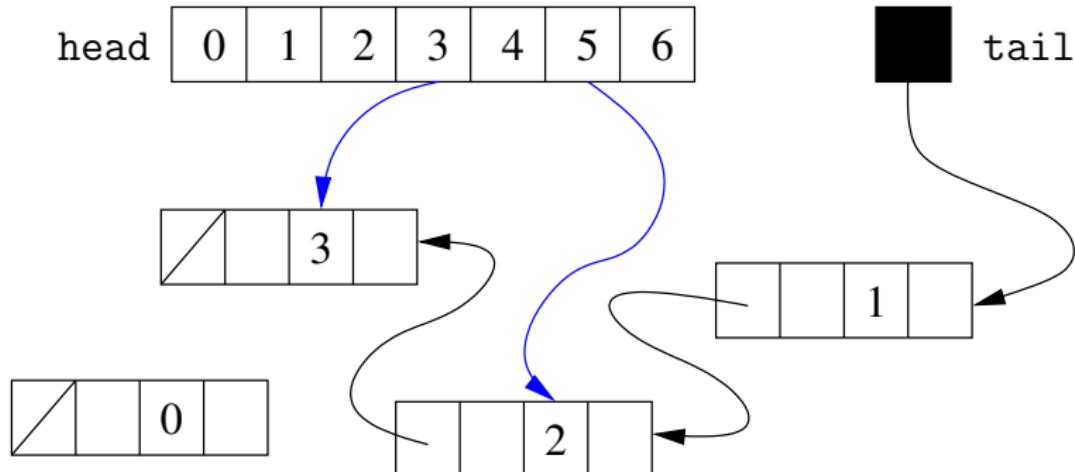
Thread 5 wins (beating thread 2).

A lock-free universal construction: Example



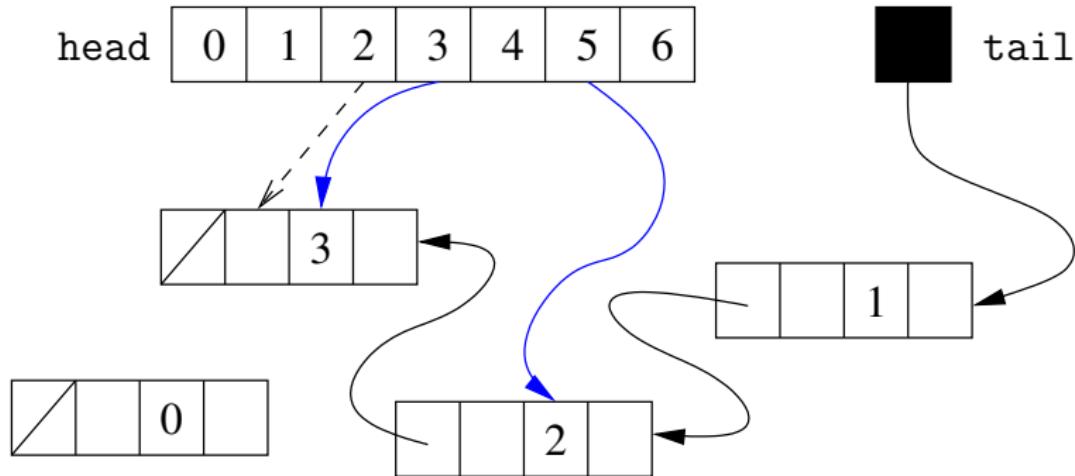
Thread 3 wants to invoke a method call.

A lock-free universal construction: Example



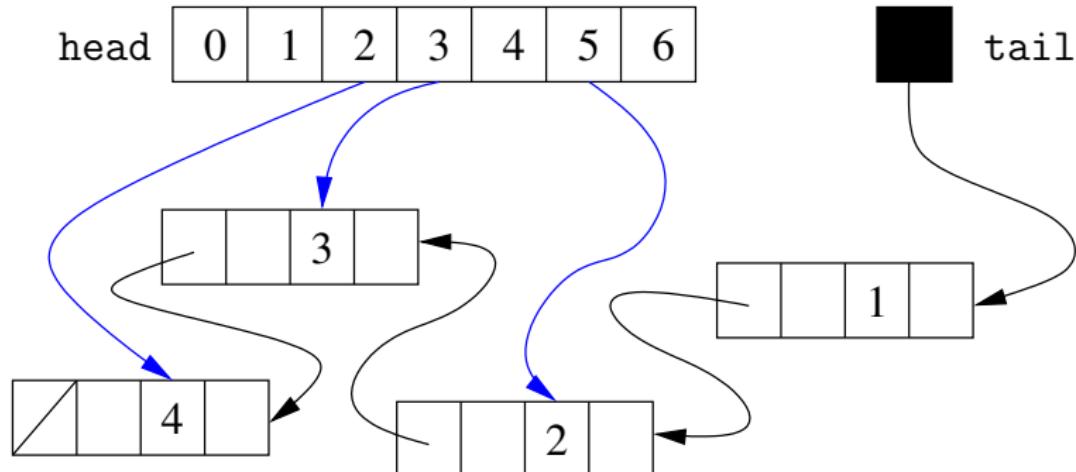
Thread 3 wins (without contention).

A lock-free universal construction: Example



Thread 2 retries on the consensus object of the new head of the list.

A lock-free universal construction: Example



Thread 2 wins (without contention).

A lock-free universal construction

If thread i finds that another thread j won on the consensus object at the head ν' of the list, then i :

- ▶ finds the node ν'' of j (via the consensus object);
- ▶ directs ν' to ν'' ;
- ▶ writes the correct sequence number in ν'' ; and
- ▶ directs $\text{head}[i]$ to ν'' .

Else the algorithm wouldn't be lock-free:

A node that wins on the consensus object and then crashes would halt all other threads.

A lock-free universal construction: Correctness

The construction is **lock-free**:

- ▶ As long as there are method calls, such calls keep on being added to the head of the list.
- ▶ Competing threads help to advance the pointer from the old to the new head, and update the sequence number of the new head.

A **linearization point** of a method call is the moment it wins on a consensus object.

The construction isn't wait-free:

A thread may infinitely often fail to add its method call at the head of the list.

A wait-free universal construction

When a thread i wants to add a method call to the list:

- ▶ It places its node holding this method call in `announce[i]`.
- ▶ It determines the head of the list, by traversing the array `head`.
- ▶ It checks if `announce[k]`, with k the next seq. nr. modulo n , contains a pending method call (i.e., has seq. nr. 0).
- ▶ It competes for the consensus object at the head.
- ▶ If i loses, it helps the winning thread advance the head of the list.

A wait-free universal construction

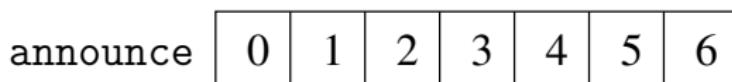
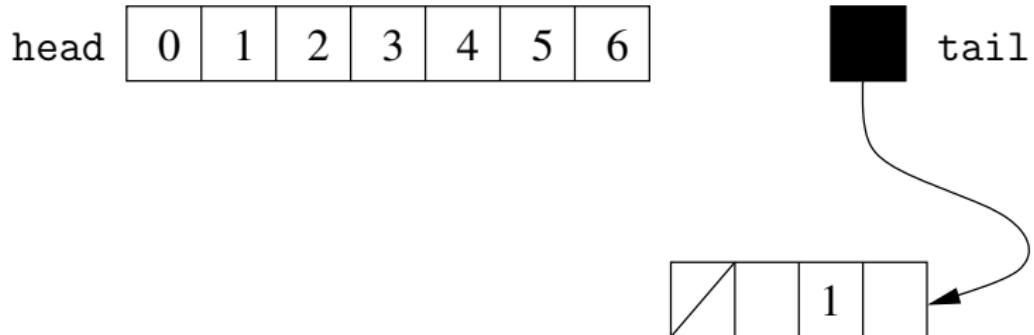
Suppose thread i wins on the consensus object.

Let ν denote either `announce[k]` if it contains a pending method call, or else i 's node.

- ▶ i makes the consensus object point to ν ,
- ▶ lets the old head point to ν ,
- ▶ sets ν 's sequence number to the sequence number of the old head plus one, and
- ▶ lets `head[i]` point to ν .

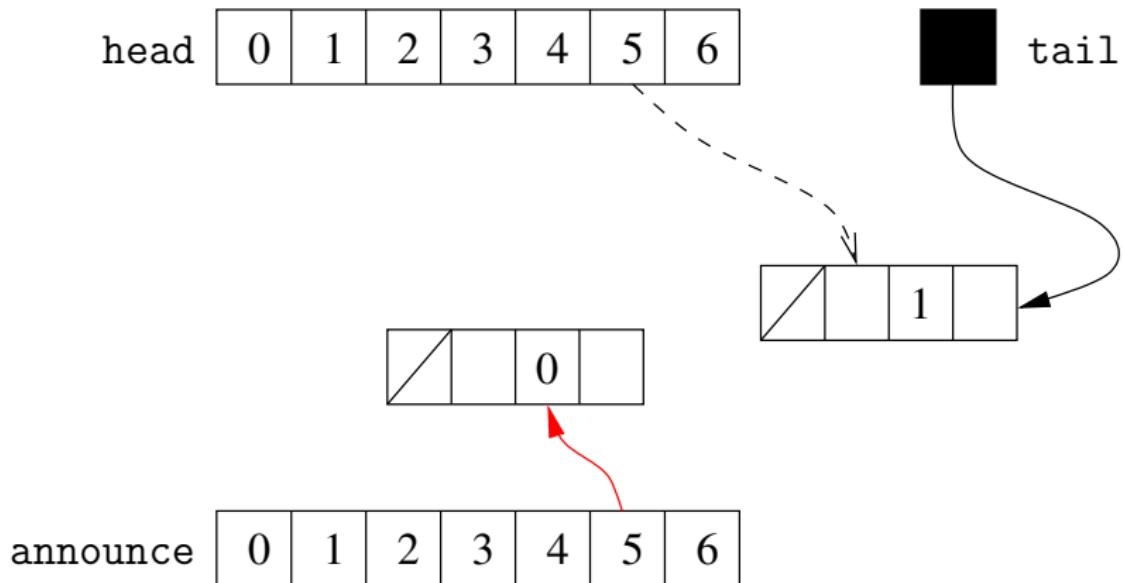
Thread i may continue to help (at most $n - 1$) other threads, until its own method call is appended to the list.

A wait-free universal construction: Example



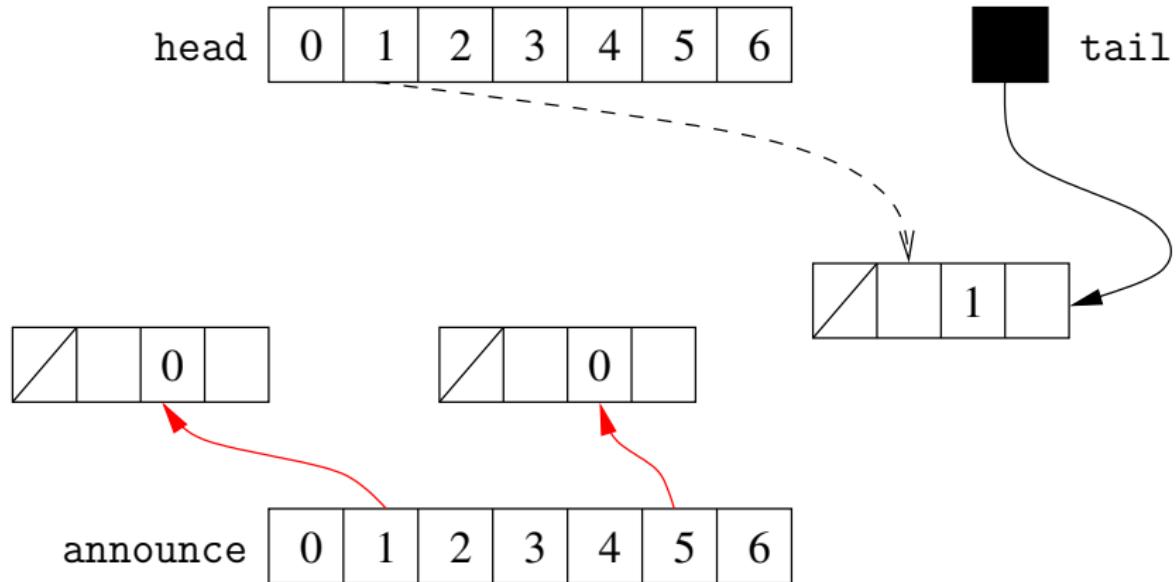
All entries of the array head initially point to the sentinel node.

A wait-free universal construction: Example



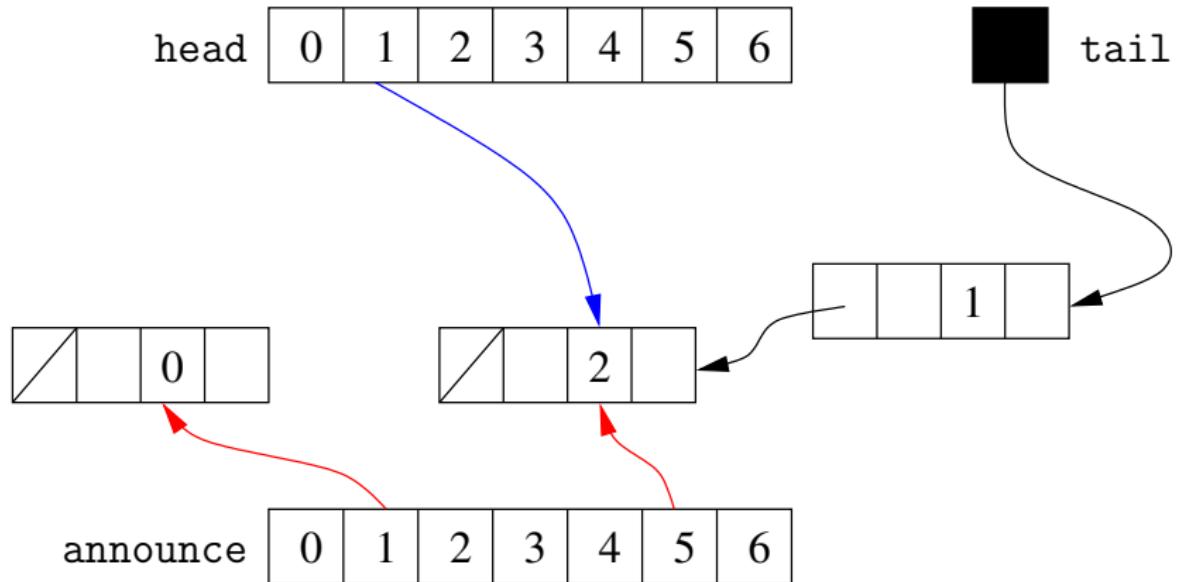
Thread 5 wants to invoke a method call, wins, and halts after that.

A wait-free universal construction: Example



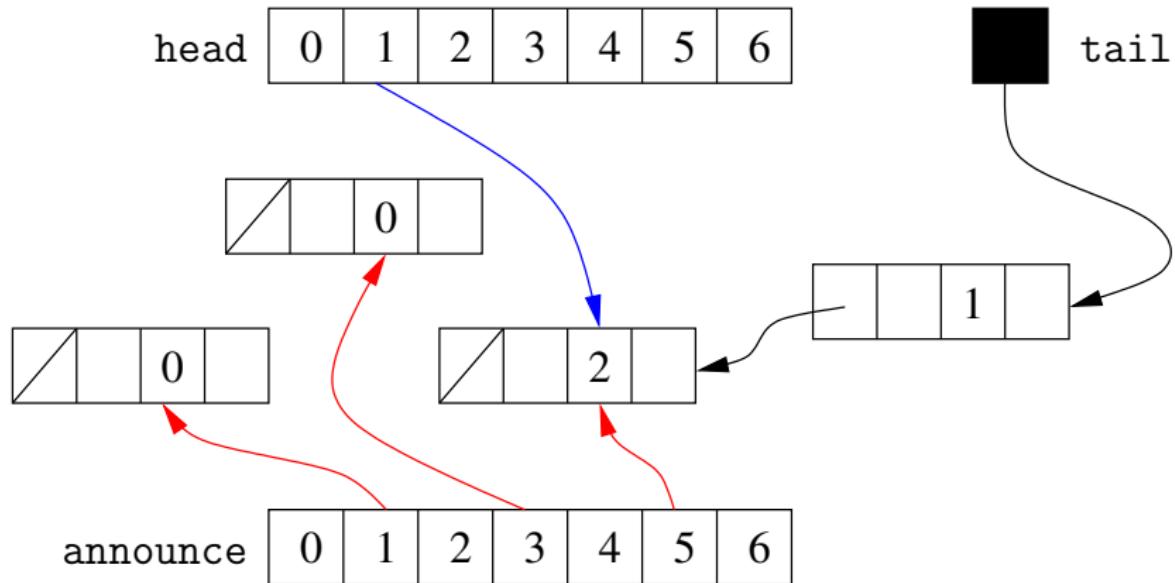
Thread 1 wants to invoke a method call, and loses.

A wait-free universal construction: Example



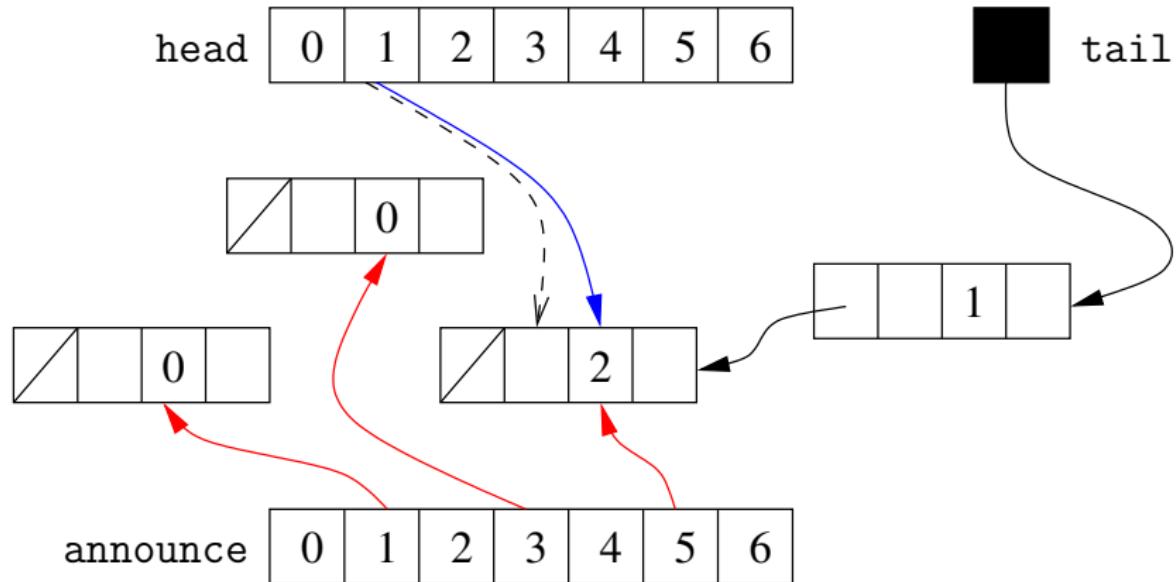
Thread 1 points the old head to the new head, updates the sequence number of the new head, and points `head[1]` to the new head.

A wait-free universal construction: Example



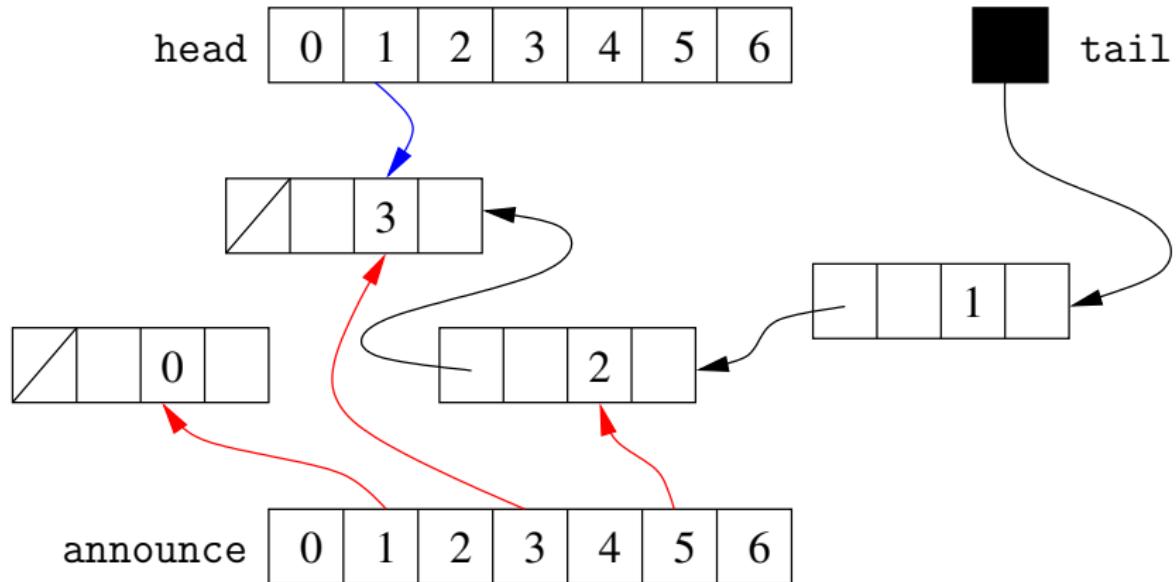
Thread 3 wants to invoke a method call, but is very slow.

A wait-free universal construction: Example



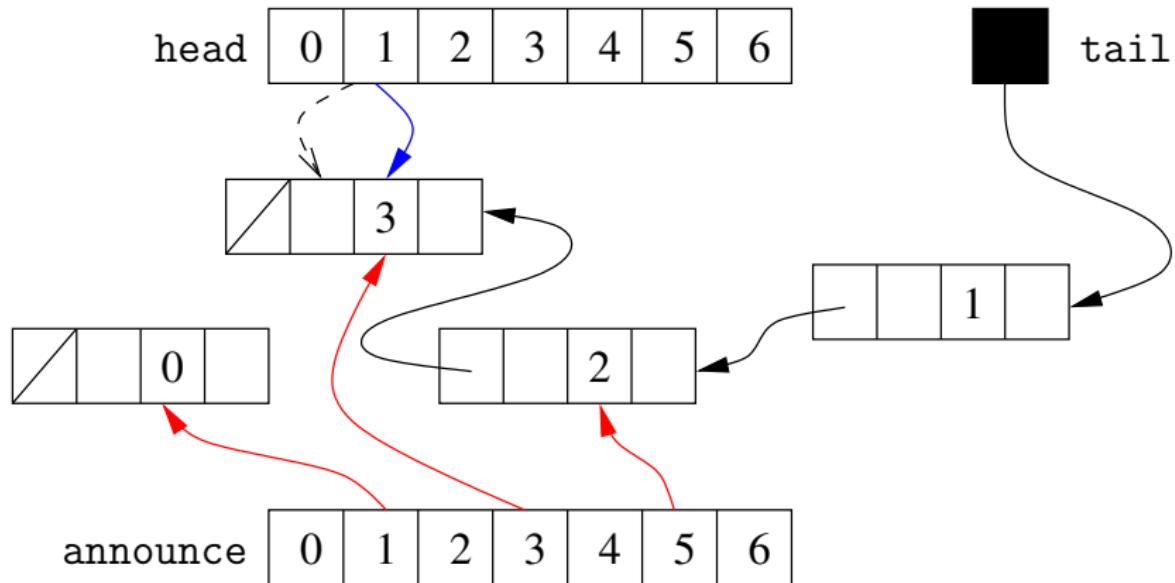
Thread 1 wins. Since 3 is the sequence number of the head plus one, thread 1 helps to make the node of thread 3 the new head.

A wait-free universal construction: Example



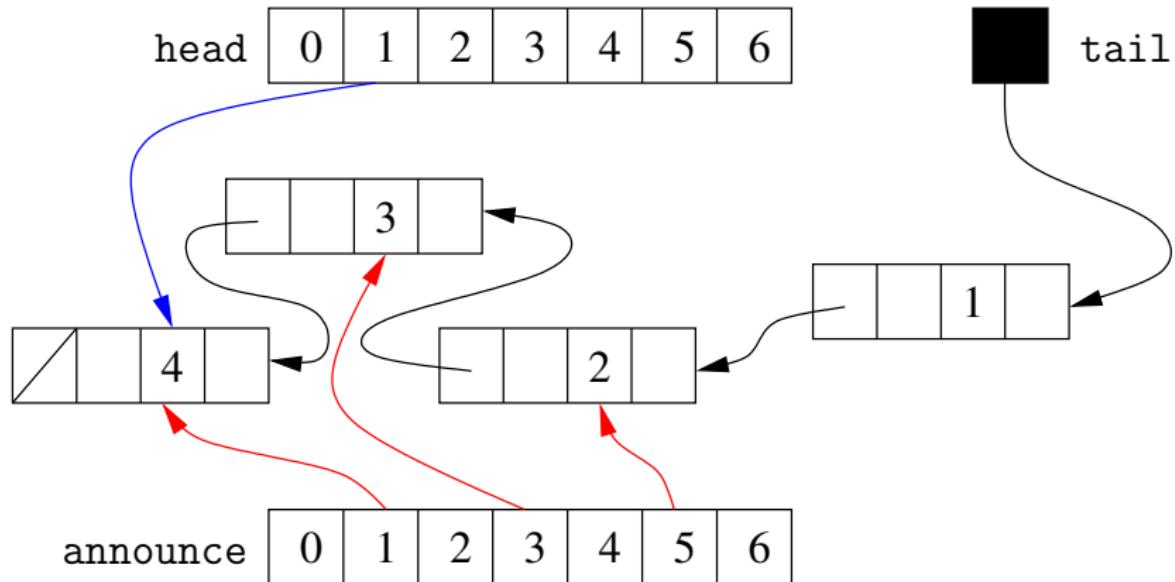
Thread 1 points the old head to the new head, updates the sequence number of the new head, and points `head[1]` to the new head.

A wait-free universal construction: Example



Thread 1 wins. Since thread 4 isn't trying to invoke a method, thread 1 makes its own node the new head.

A wait-free universal construction: Example



Thread 1 points the old head to the new head, updates the sequence number of the new head, and points `head[1]` to the new head.

A wait-free universal construction: Correctness

The construction is **wait-free**: If a thread i has a pending method call, and the next sequence number of the list equals i modulo n , then this method call is certain to be added to the head of the list.

A **linearization point** of a method call is the moment its node is appended to the list.

Remark: A **blockchain** (that underlies Bitcoin) works similarly, but uses *proof-of-work* and a *longest chain rule* to achieve consensus.

This lecture in a nutshell

consensus

no 2-thread consensus with atomic registers

2-thread but no 3-thread consensus with a FIFO queue

no 3-thread consensus with commuting/overwriting operations

n-thread consensus with compareAndSet

wait-free implementation of any object, using consensus

threads help each other to achieve lock- and wait-freeness

Focus so far: Correctness and progress

- ▶ Models

- ▶ accurate (I never lied to you)
- ▶ but idealized (I forgot to mention a few things)

- ▶ Algorithms

- ▶ elegant
- ▶ important for understanding
- ▶ but naive



New focus: Performance

- ▶ Models

- ▶ a bit more complicated
- ▶ still focus on principles

- ▶ Algorithms

- ▶ elegant in their fashion
- ▶ important for understanding and in practice
- ▶ realistic



Mutual exclusion revisited

The **filter lock** and the **bakery algorithm** don't scale.

Because they require at least n atomic registers for n threads.

Since hardware allows out-of-order execution, and due to caches, care is needed to place **barriers** or declare atomic registers **volatile**.

Read-modify-write operations are about as expensive as barriers and volatile variables.

Spinning versus backoff

What to do if you can't get the lock ?

- ▶ Keep trying
 - ▶ spin (also called **busy-waiting**)
 - ▶ good if delay is expected to be *short* and contention is *low*
- ▶ Give up the processor and retry later
 - ▶ **exponential backoff**
 - ▶ good if delay is expected to be *long*, contention is *high*, or a descheduled thread urgently needs processor time

A sensible approach can be to spin for a while, and retry later if the delay becomes too long.

We will now study **spin locks**.



Test-and-set lock

`testAndSet()` sets the value of a Boolean variable to *true*, and returns the previous value of this variable, in one atomic step.

Recall that read-modify-write operations include a barrier.

The **TAS lock**:

- ▶ The Boolean lock variable originally contains *false*.
- ▶ `lock()` repeatedly applies `testAndSet()` to the lock variable.
The lock is obtained if *false* is returned.
- ▶ `unlock()` writes *false* to the lock variable.

Test-and-test-and-set lock

The TTAS lock:

- ▶ The Boolean lock variable originally contains *false*.
- ▶ `lock()` spins on a *cached copy* of the lock variable.

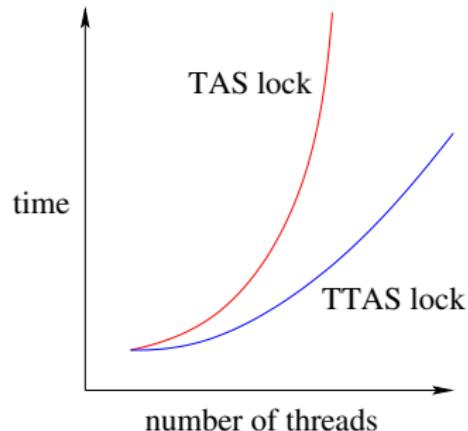
When *false* is returned, apply `testAndSet()` to the lock variable.

The lock is obtained if *false* is returned.

Else go back to spinning on the cached lock variable.

- ▶ `unlock()` writes *false* to the lock variable.

Performance of TAS and TTAS locks

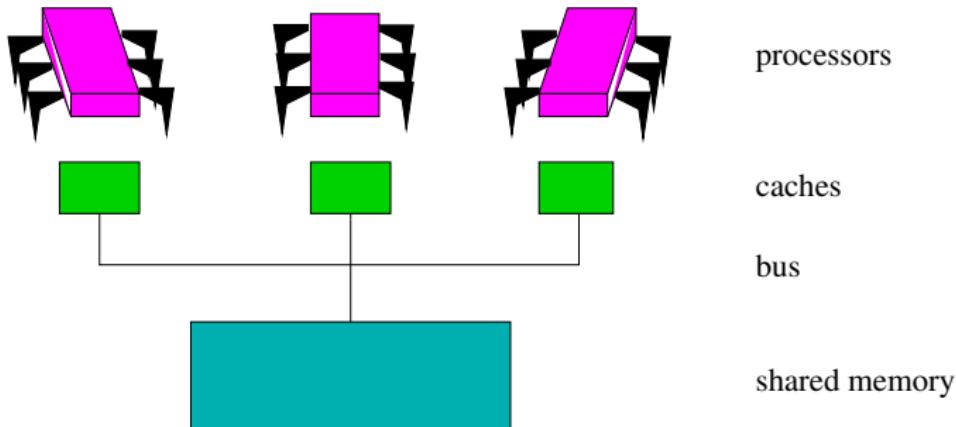


The performance of the TAS lock is *very bad*.

The performance of the TTAS lock is *pretty bad* in case of high contention.

To grasp this poor performance, we must dive into cache coherence.

Symmetric multiprocessing architecture



The shared bus is claimed by one broadcaster at a time.

Processors and memory “snoop” on the bus.

Random access memory is slow (tens of machine cycles).

Caches are fast (one or two machine cycles).

Caches

Changes in a cache are accumulated, and written back when needed (to make place in the cache, or when another processor wants it, or at a barrier).

Cache coherence: When a processor writes a value in its cache, all copies of this variable in other caches must be *invalidated*.

When a processor takes a **cache miss**, the required data is provided by memory, or by a snooping processor.

Why the TAS lock performs poorly

A `testAndSet()` call on the lock variable invalidates cache lines at all processors.

As a result, all spinners take a cache miss, and go to the bus to fetch the (mostly unchanged) value.

So the spinners produce a continuous storm of unnecessary messages over the bus.

To make matters worse, this delays the thread holding the lock.

The Achilles heel of the TTAS lock

When the lock is released, *false* is written to the lock variable, invalidating all cached copies.

All spinners take a cache miss, and go to the bus to fetch the value.

Then they concurrently call `testAndSet()` to acquire the lock.

These calls invalidate the cached copies at other threads, leading to another round of cache misses.

Then the storm lies down, and threads return to local spinning.

TTAS lock with exponential backoff

Improvement of the TTAS lock:

If the lock was free but I fail to get it, back off
(to avoid collisions, because there is contention).

Each subsequent failure to get the lock increases the waiting time,
for instance by doubling it.

Wait durations are **randomized**, to avoid that conflicting threads
fall into lock-step.

Two important parameters: **minDelay**, the initial minimum delay
maxDelay, the final maximum delay

TTAS lock with exponential backoff: Performance

The TTAS lock with exponential backoff is easy to implement, and gives excellent performance in case of low contention.

Drawbacks:

- ▶ All threads still spin on the same lock variable, causing cache coherence traffic when the lock is released.
- ▶ It isn't starvation-free.
- ▶ Exponential backoff may delay threads longer than necessary, causing underutilization of the critical section.
- ▶ Its performance is very sensitive to `minDelay` and `maxDelay`.
Optimal values are platform- and application-dependent.

Array-based queue lock

A Boolean `array` represents the threads that are waiting for the lock.

The array's size `n` is the (maximal) number of (concurrent) threads.

Initially slot 0 holds `true`, while slots $1, \dots, n - 1$ hold `false`.

Initially the `counter` is 0.

To acquire the lock, a thread applies `getAndIncrement()` to the counter. The returned value modulo `n` is its slot in the array.

A thread waiting for the lock keeps spinning on (a cached copy of) its slot in the array, until it is `true`.

To unlock, a thread first sets its slot in the array to `false`, and then the next slot (modulo `n`) to `true`.

Question

What could go wrong if the unlock method would first set the next slot to *true*, and then its own slot to *false*?

(Take $n = 3$.)

Array-based queue lock: Performance

Each waiting thread spins on (a cached copy of) a different slot in the array, so releasing a lock gives **no cache coherence overhead**.

Short hand-over time compared to exponential backoff.

Scales well to large numbers of threads.

Provides (first-come-first-served) **fairness**.

Drawbacks: Protecting L different objects takes $O(L \cdot n)$ space.

Vulnerable to *false sharing*.

False sharing

If different items of the array are on the same cache line, releasing a lock may give cache coherence overhead after all.

This can be avoided by *padding*: The array size is say quadrupled, and slots are separated by three (unused) places in the array.

Guidelines to avoid false sharing:

- ▶ Fields that are accessed independently should end up on different cache lines (e.g. by padding).
- ▶ Keep read-only data separate from data that is modified often.
- ▶ Where possible, split an object into thread-local pieces.
- ▶ If a lock protects data that is frequently modified, then keep lock and data on different cache lines.

Craig Landin Hagersten queue lock

Threads wait for the lock in a (virtual) **queue** of *nodes*.

A node's **locked** field is:

- ▶ *true* while its thread is waiting for or holds the lock;
- ▶ *false* when it has released the lock.

tail points to the most recently added node.

(Initially it points to a dummy node containing *false*.)

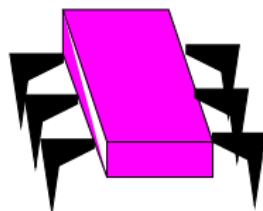
A thread that wants the lock creates a node ν , containing *true*.

- ▶ It applies **getAndSet(ν)** to tail, to make ν the tail of the queue and get the node of its predecessor.
- ▶ Next it spins on (a cached copy of) the locked field of its predecessor's node until it becomes *false*. Then it takes the lock.

A thread that releases the lock, sets the locked field of its node to *false*.

CLH queue lock: Example

idle

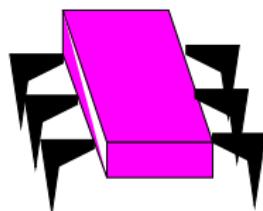


tail



CLH queue lock: Example

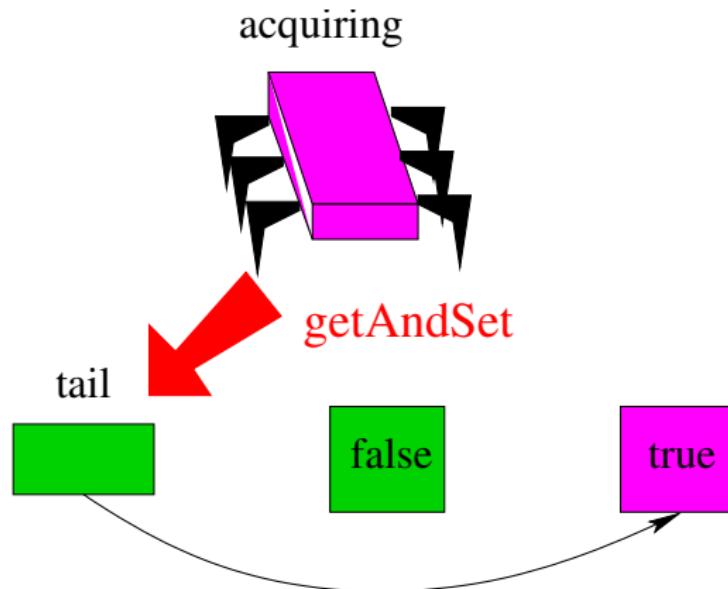
acquiring



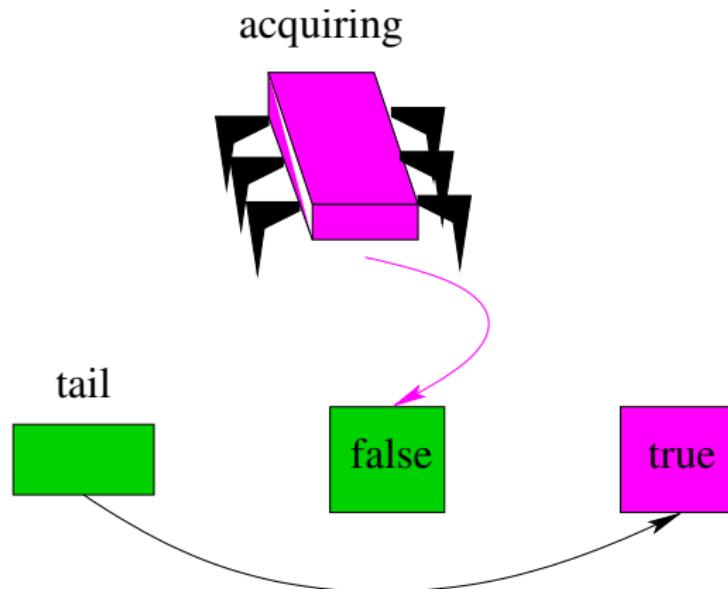
tail



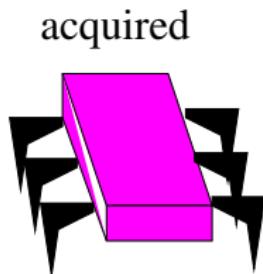
CLH queue lock: Example



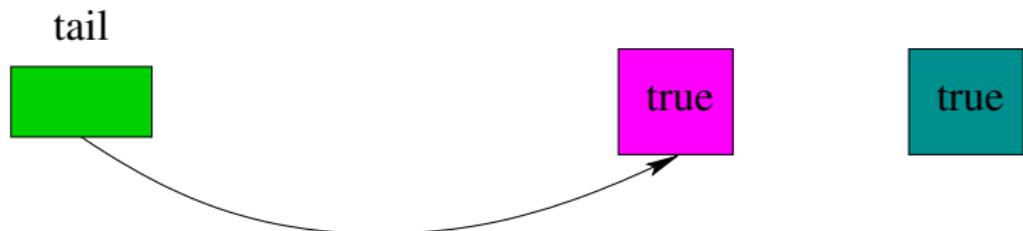
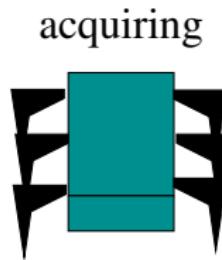
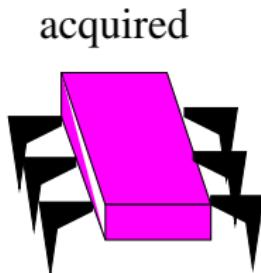
CLH queue lock: Example



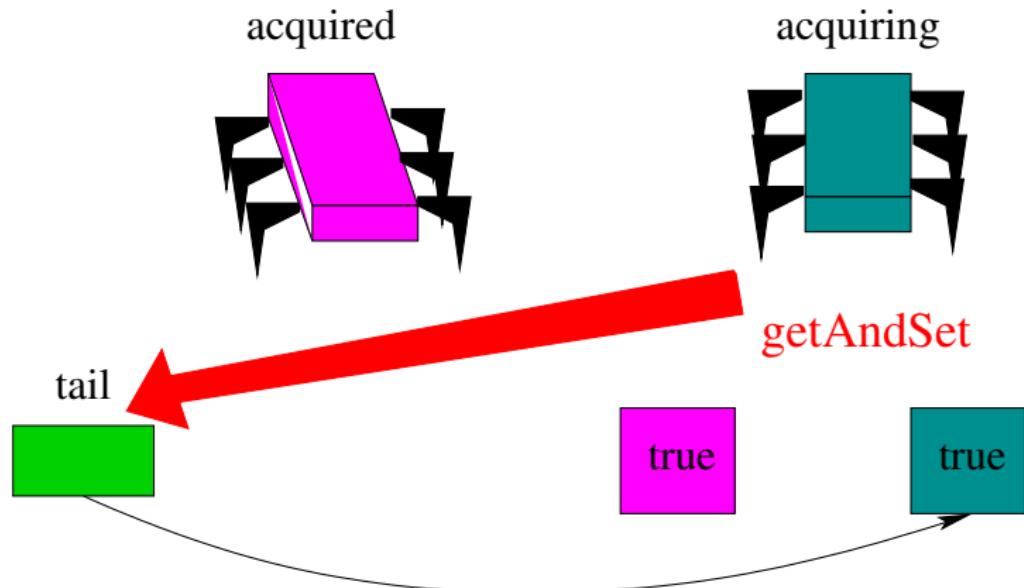
CLH queue lock: Example



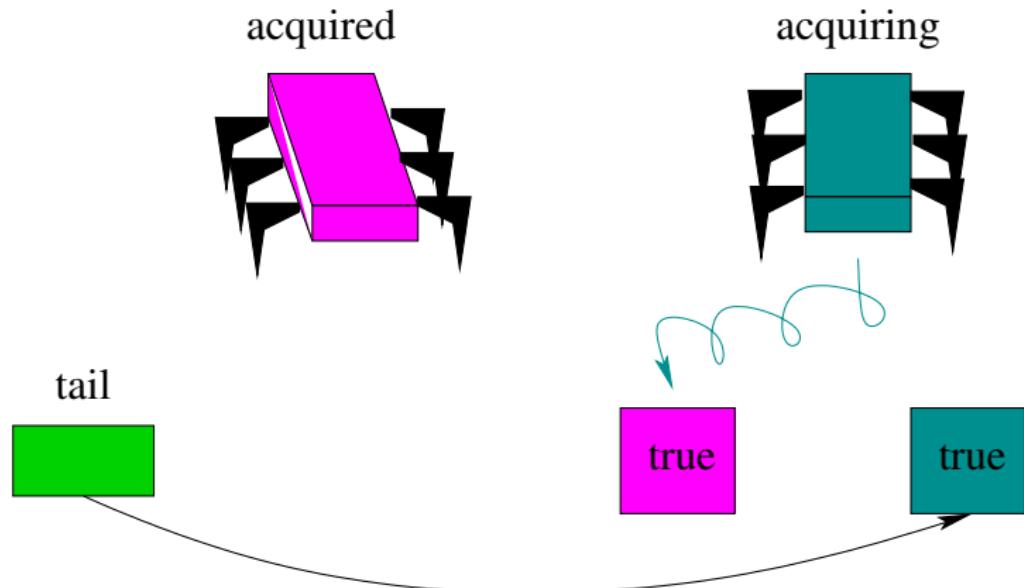
CLH queue lock: Example



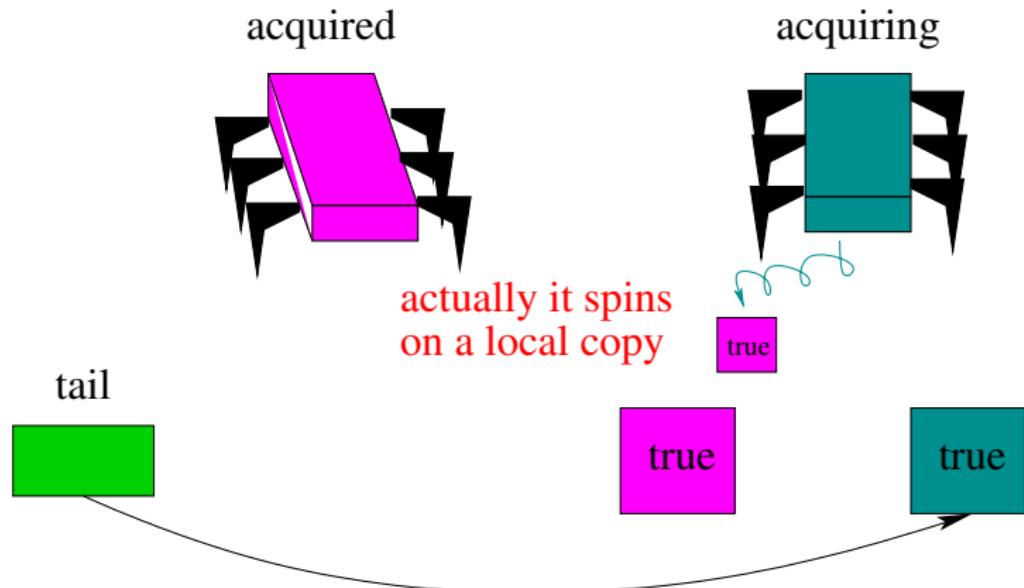
CLH queue lock: Example



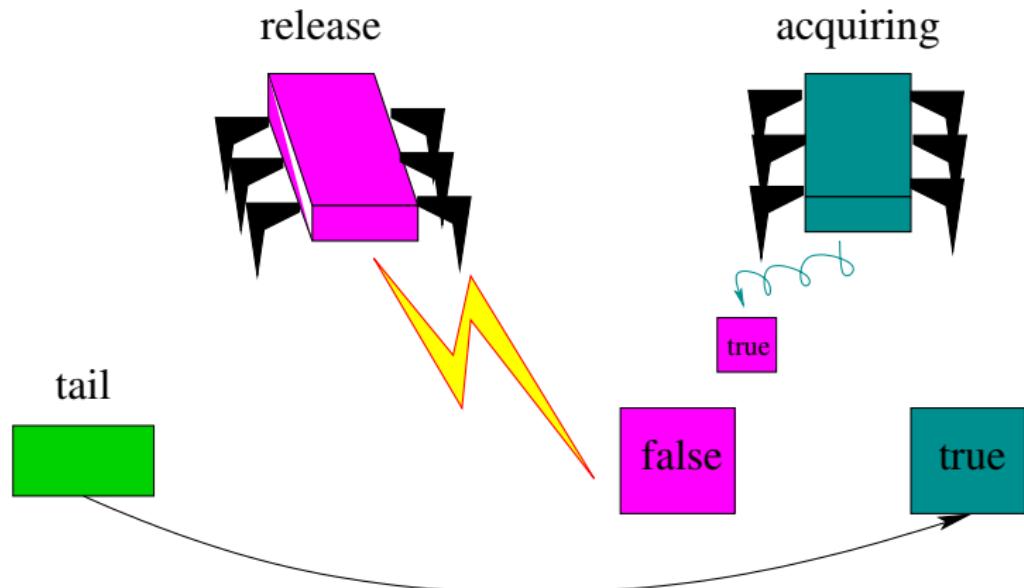
CLH queue lock: Example



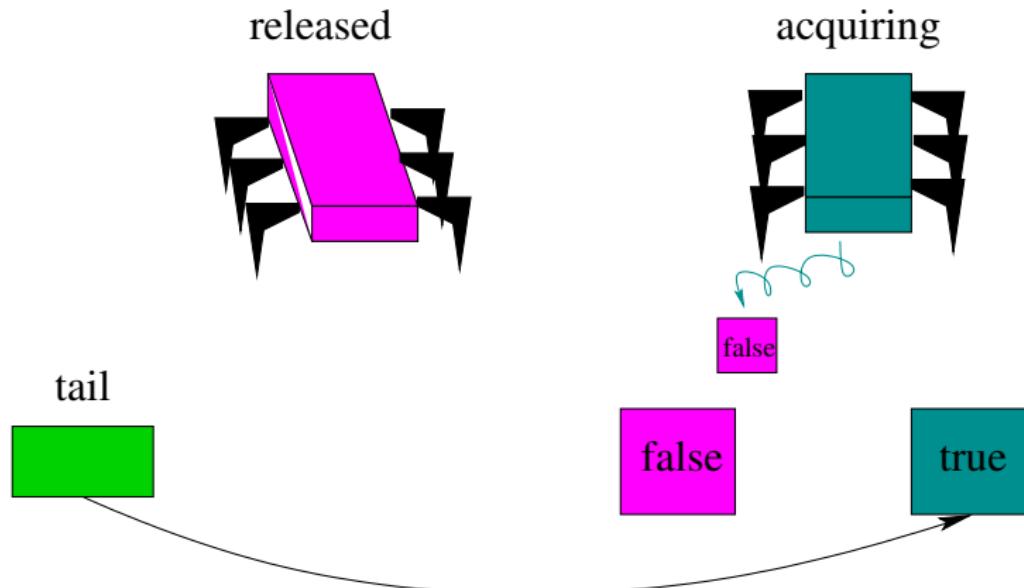
CLH queue lock: Example



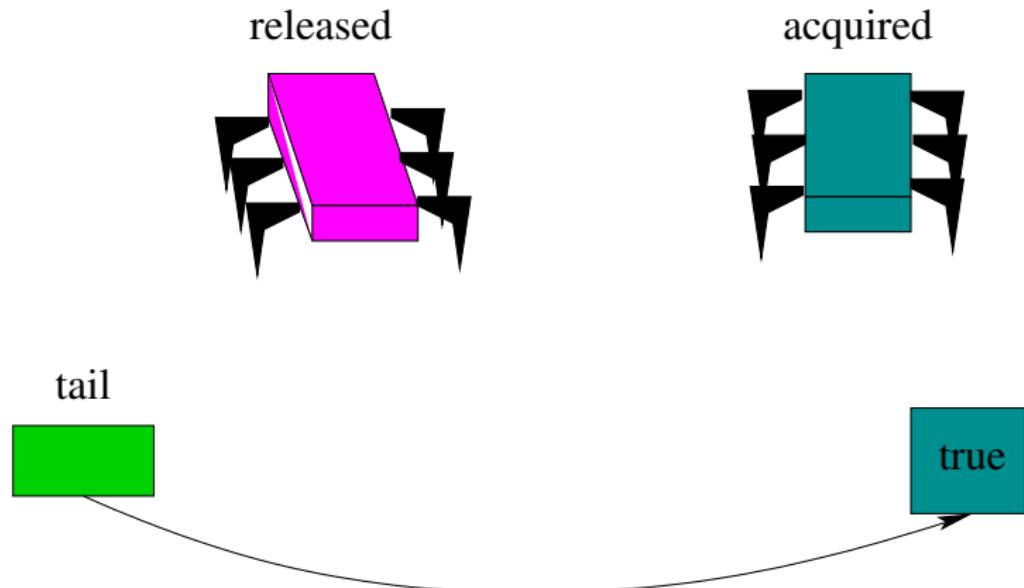
CLH queue lock: Example



CLH queue lock: Example



CLH queue lock: Example



CLH queue lock: Performance

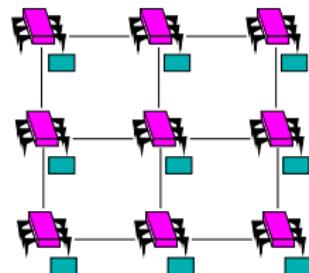
The CLH lock has the same good performance as the array-based lock, also provides fairness, and uses less space.

After releasing the lock, a thread can reuse the node of *its predecessor* for a future lock access.

Protecting L different objects takes $O(L + n)$ space.

But the CLH lock performs poorly in a **cacheless non-uniform memory access architecture**, where remote spinning is expensive.

Question: How can the CLH lock be adapted to let threads spin on a local variable?



Mellor-Crummey Scott queue lock

Threads again wait for the lock in an (explicit) queue of nodes.

`tail` points to the most recently added node (initially it is *null*).

A node's `locked` field is *true* while its thread is waiting for the lock (initially it is *false* (!)).

A node's `next` field points to the node of the successor of its thread in the queue (initially it is *null*).

A thread that wants the lock creates a node ν .

- ▶ It applies `getAndSet(ν)` to `tail`, to make ν the tail of the queue and get the node of its predecessor.
- ▶ If `tail` was *null*, the thread takes the lock immediately.
- ▶ Else it sets the `locked` field of ν to *true* and the `next` field of its predecessor's node to ν . Next it spins on the `locked` field of ν until it becomes *false*. Then the thread takes the lock.

MCS queue lock

A thread that releases the lock, checks its `next` field.

If it points to a node, the thread sets the `locked` field of that node to *false*.

Question: What to do if its `next` field is *null*?

Then the thread applies `compareAndSet(ν , null)` to `tail`, with ν the node of the thread.

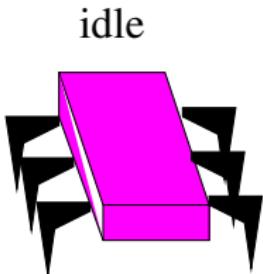
If this call fails, another thread is trying to acquire the lock.

Question: What to do if this call fails?

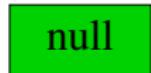
Then the thread spins on its `next` field until a node is returned.

Next it sets the `locked` field of that node to *false*.

MCS queue lock: Example 1

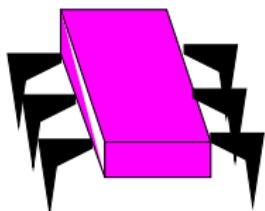


tail



MCS queue lock: Example 1

acquiring

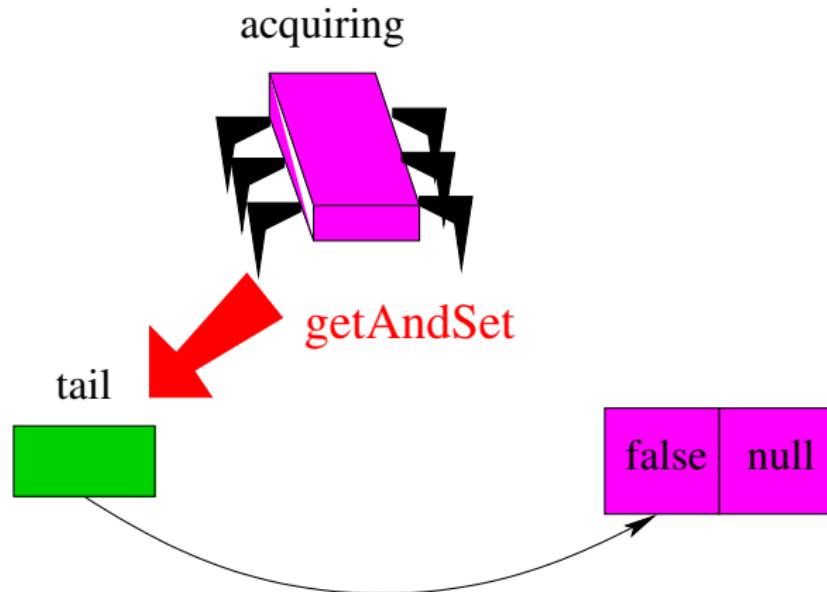


tail

null

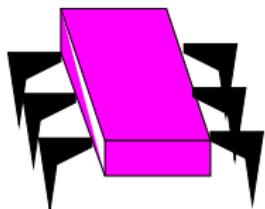
false	null
-------	------

MCS queue lock: Example 1

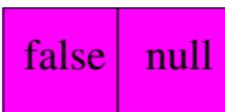


MCS queue lock: Example 1

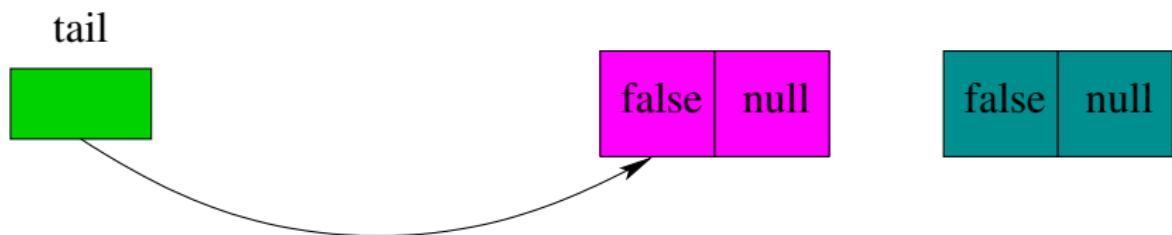
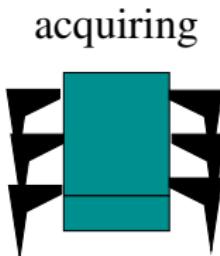
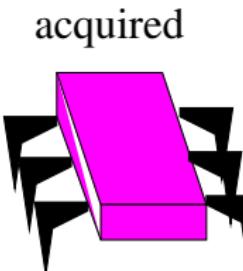
acquired



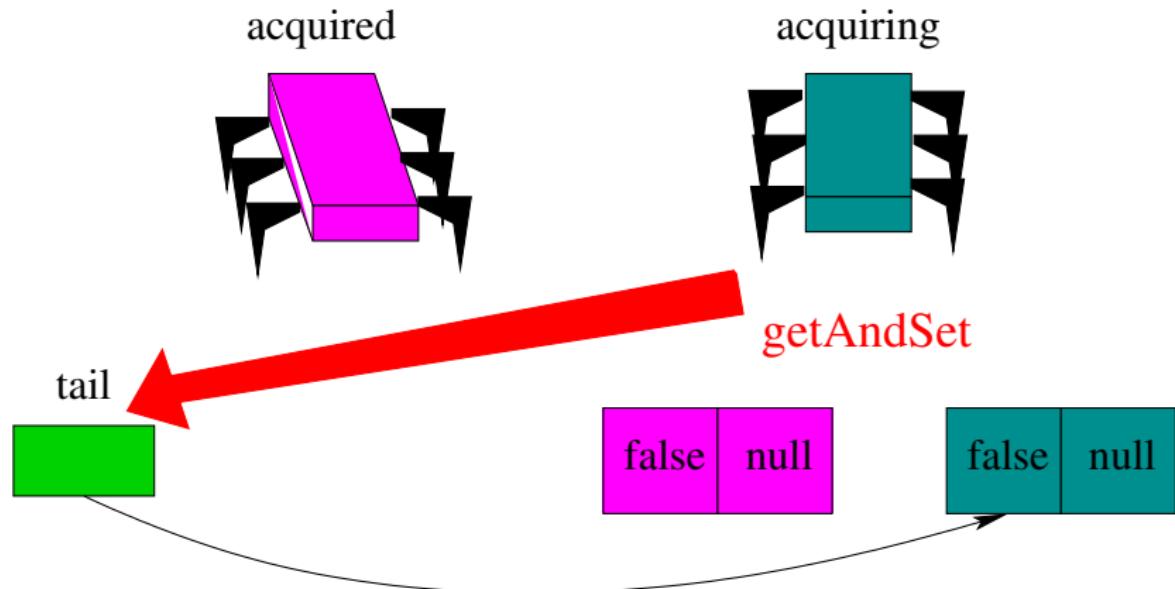
tail



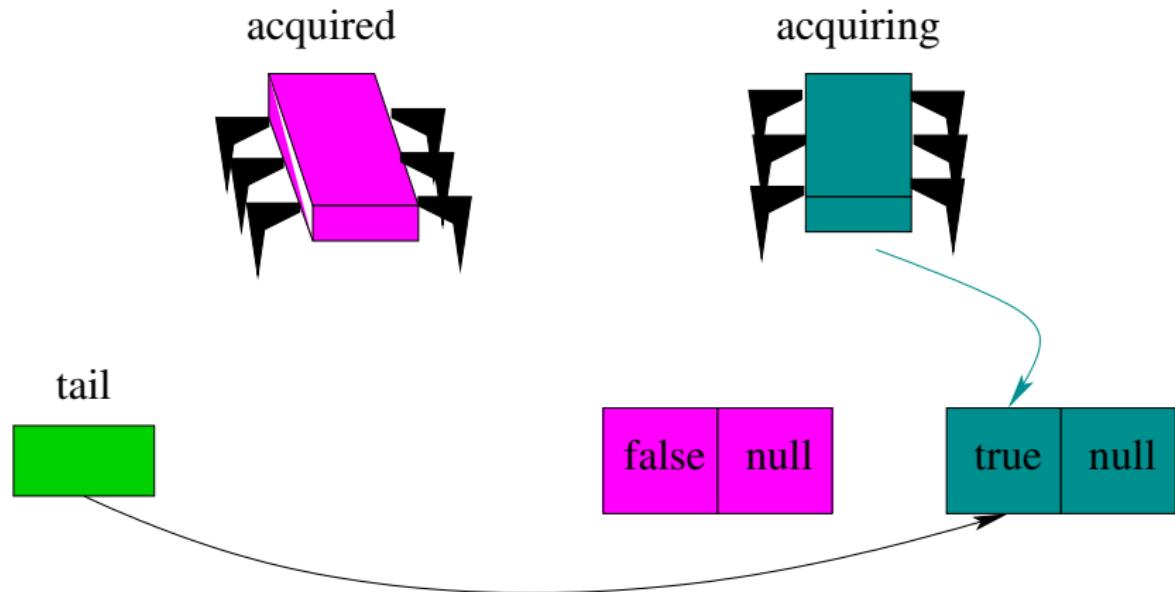
MCS queue lock: Example 1



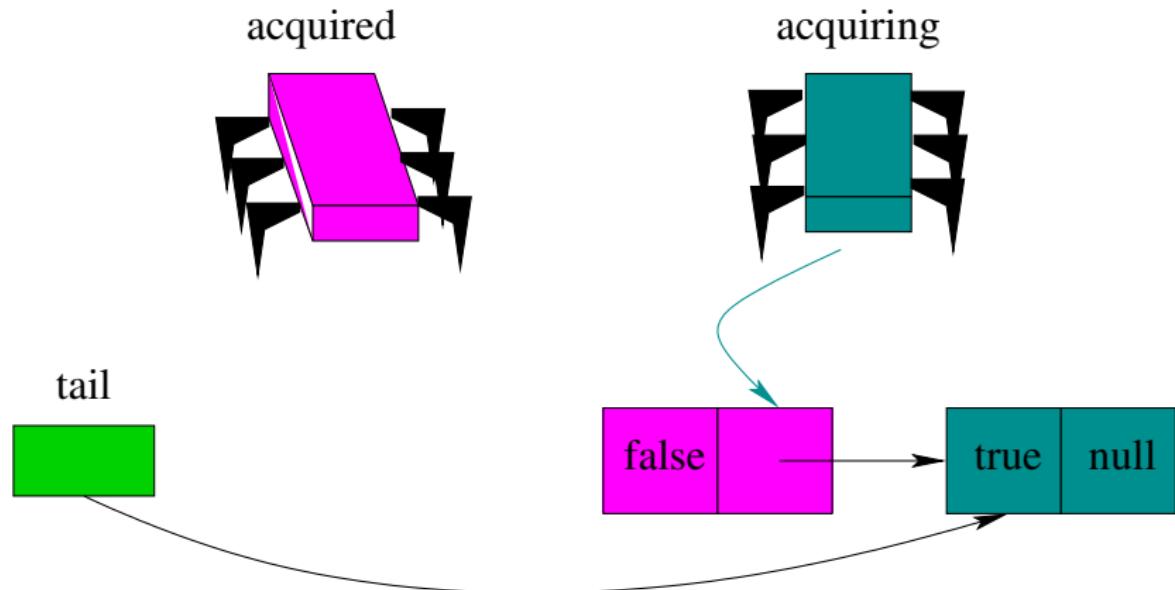
MCS queue lock: Example 1



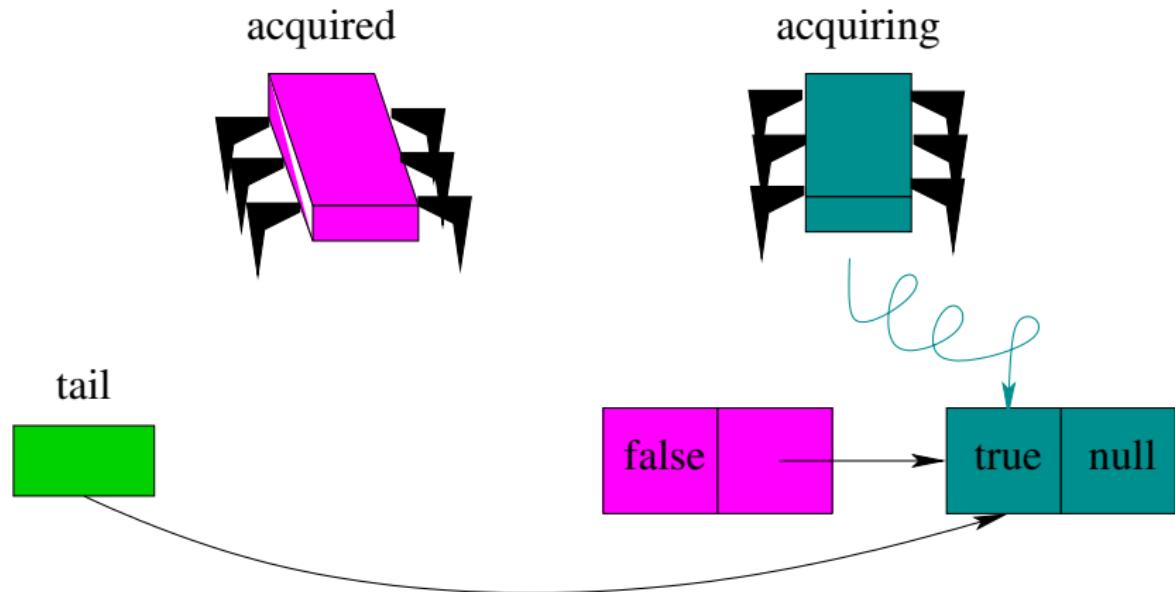
MCS queue lock: Example 1



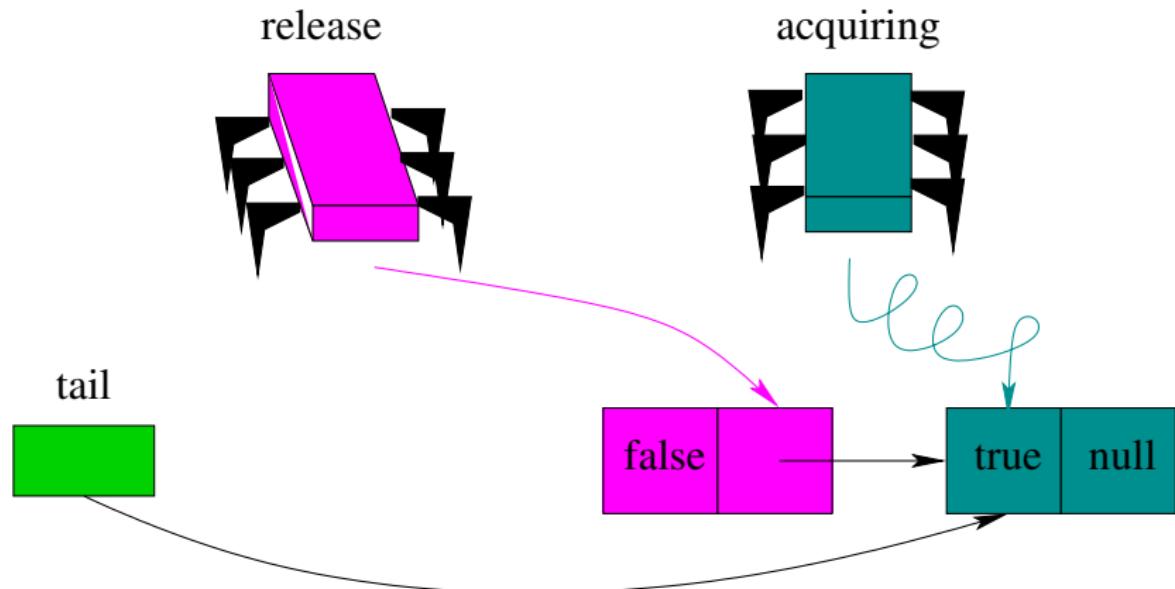
MCS queue lock: Example 1



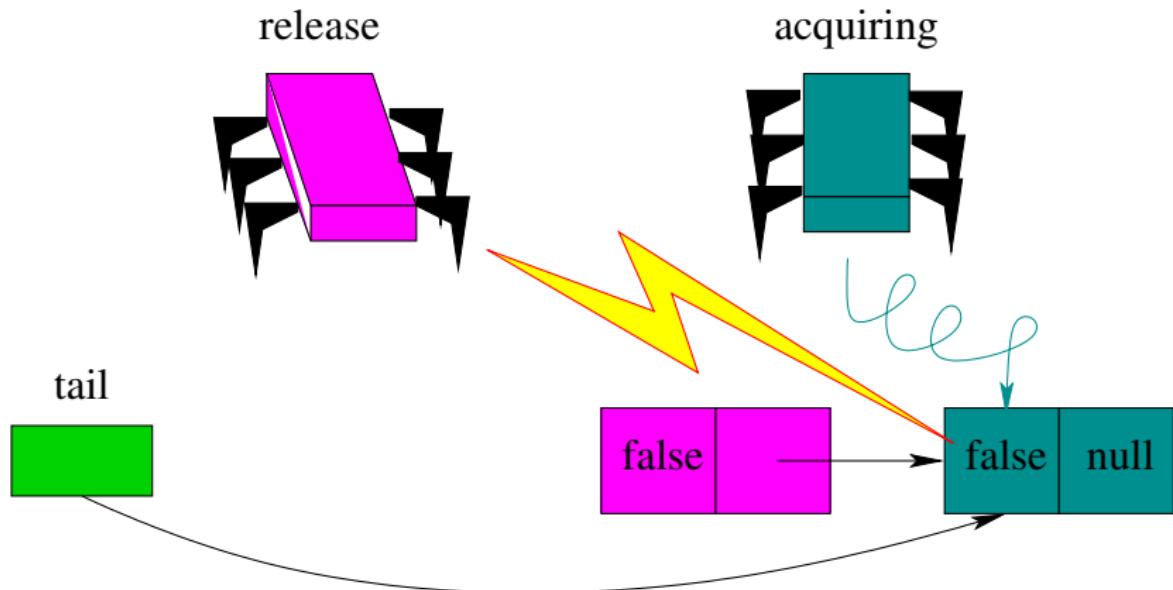
MCS queue lock: Example 1



MCS queue lock: Example 1

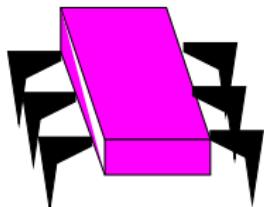


MCS queue lock: Example 1

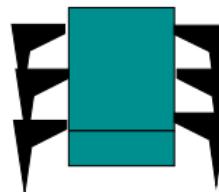


MCS queue lock: Example 1

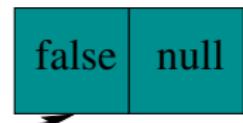
released



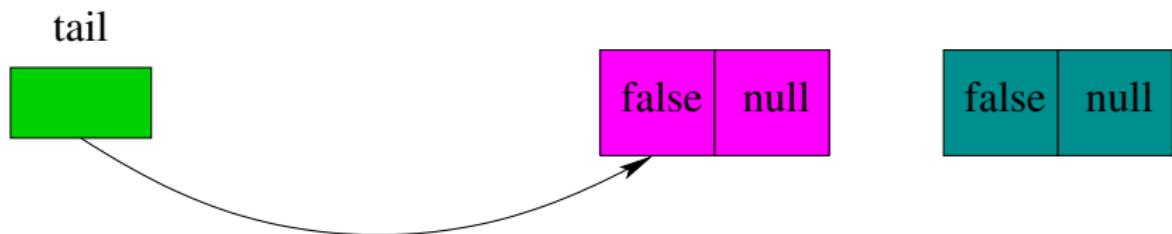
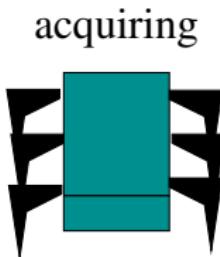
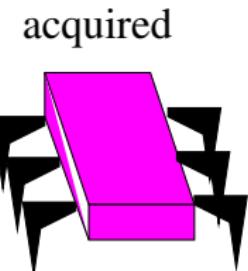
acquired



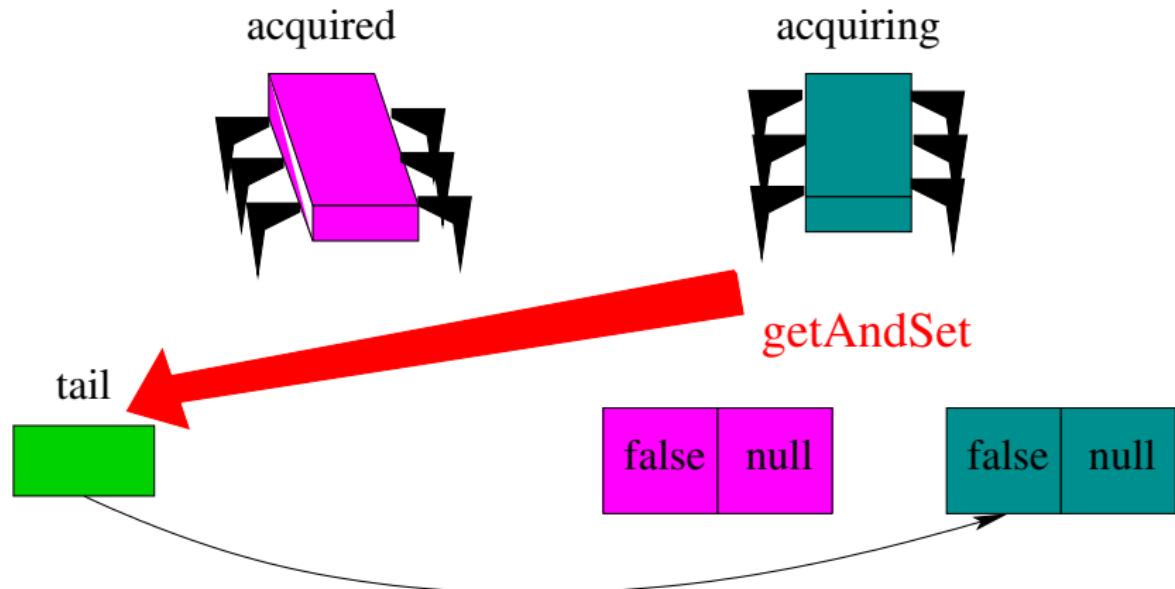
tail



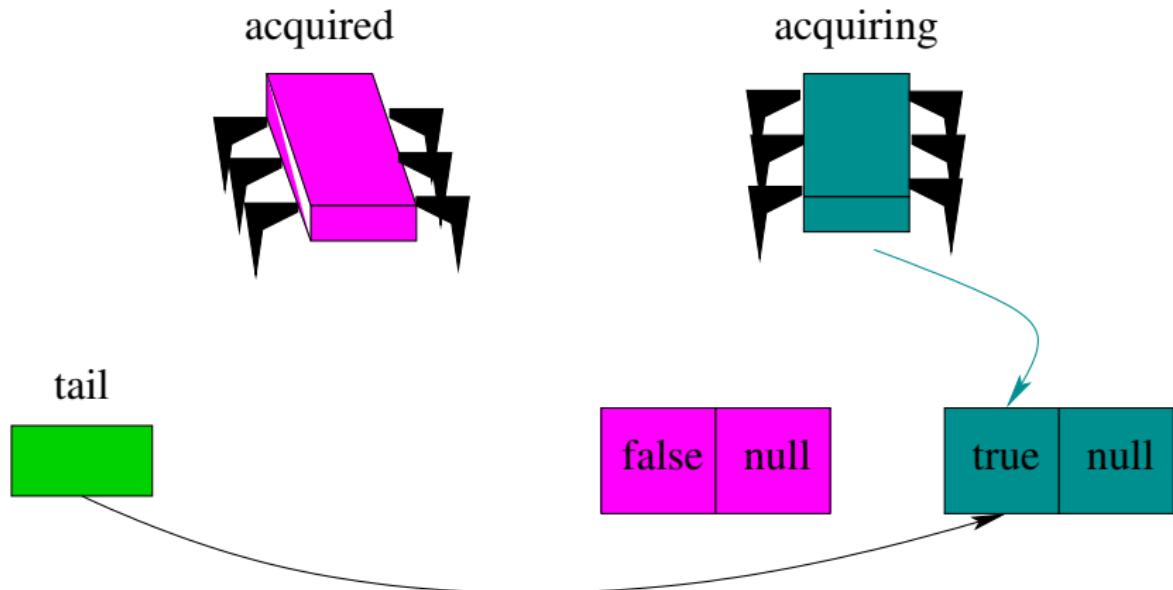
MCS queue lock: Example 2



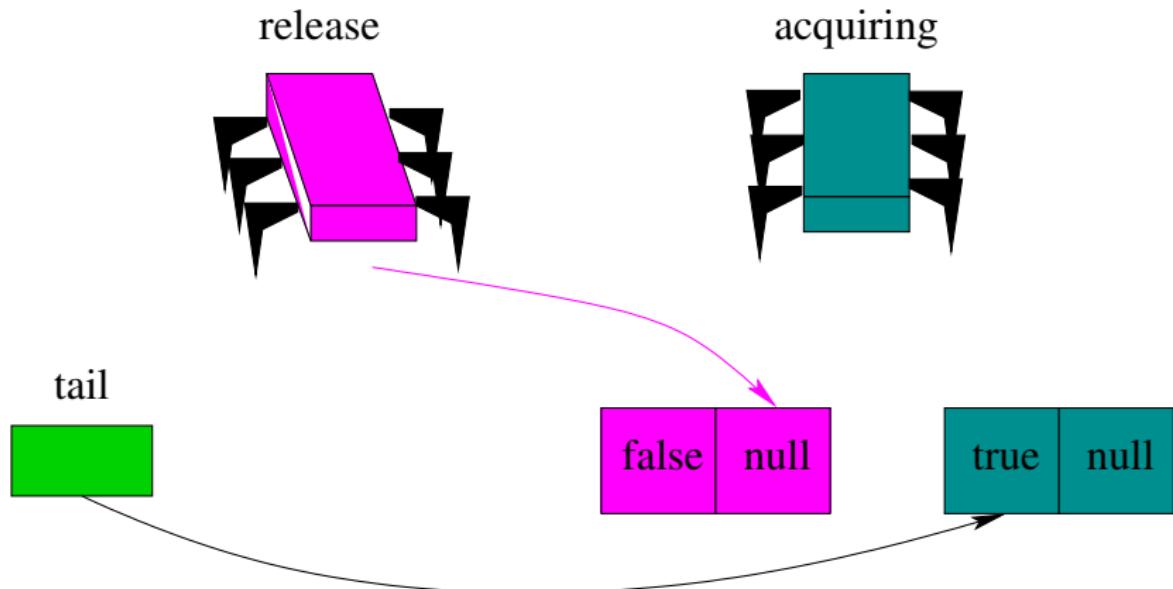
MCS queue lock: Example 2



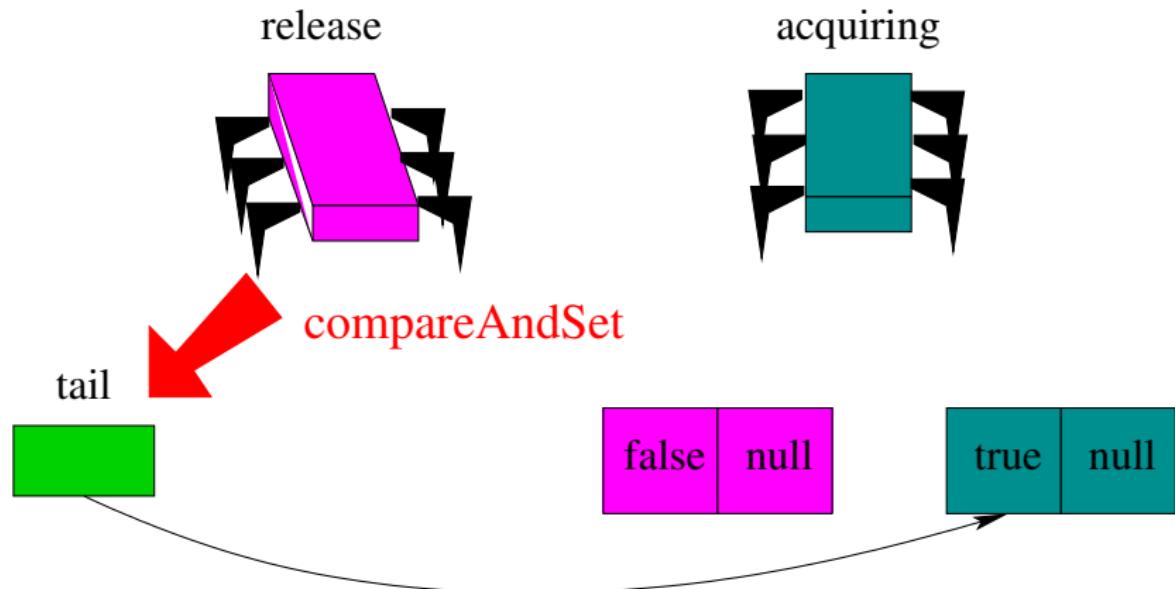
MCS queue lock: Example 2



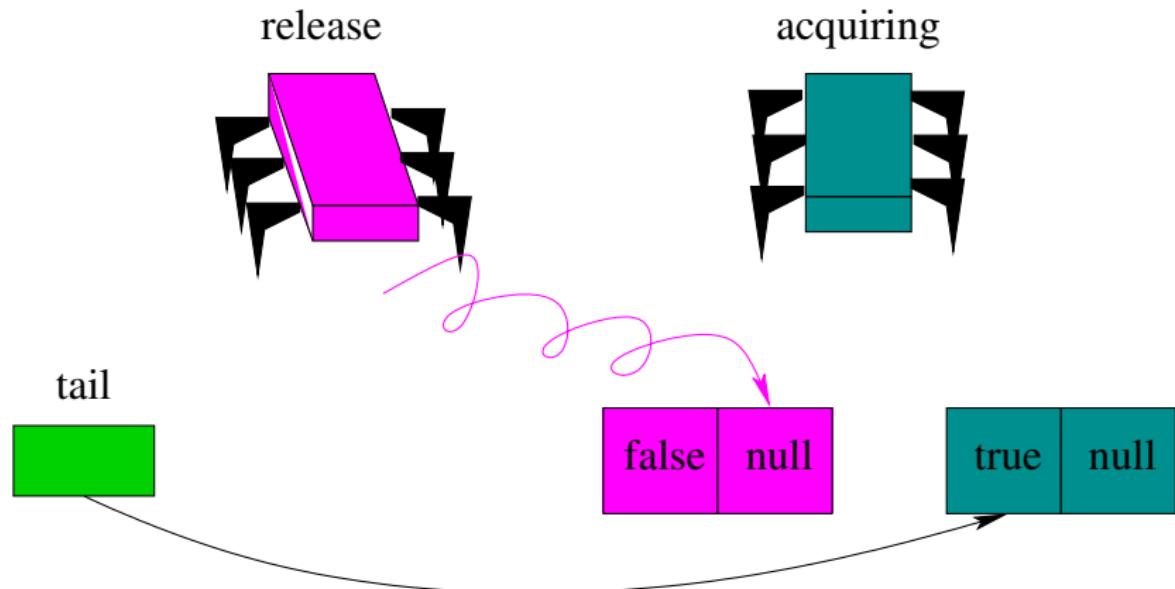
MCS queue lock: Example 2



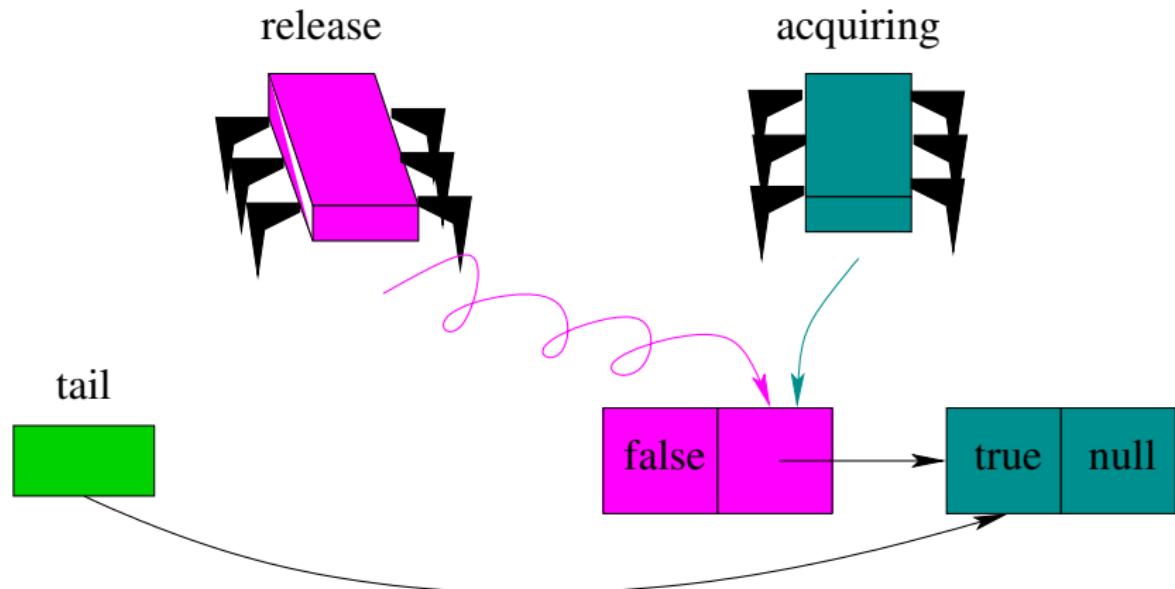
MCS queue lock: Example 2



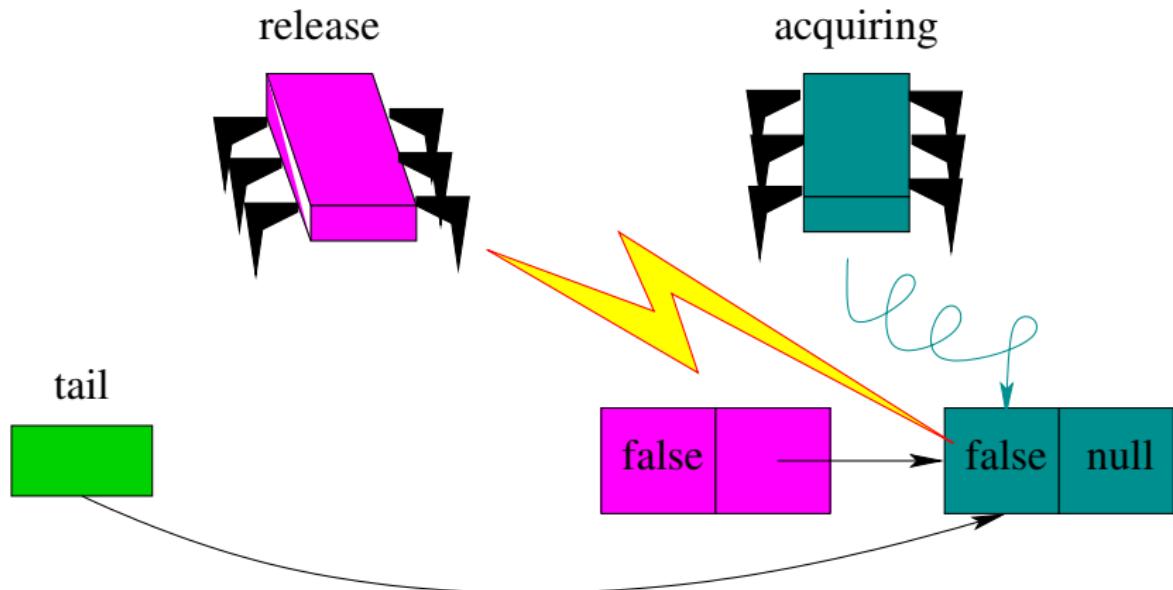
MCS queue lock: Example 2



MCS queue lock: Example 2

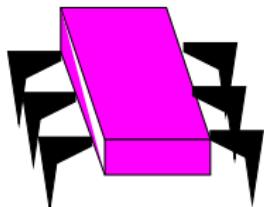


MCS queue lock: Example 2

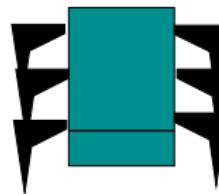


MCS queue lock: Example 2

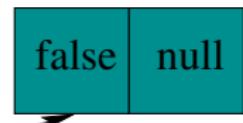
released



acquired



tail



Question

Why is it sensible, regarding performance, to let a node initially contain *false*?

Answer: Most lock requests are granted without contention.

If a thread can take the lock immediately, it doesn't need to spend time on inverting this Boolean value.

If a thread must wait for the lock, it can spend some waiting time on inverting this Boolean value.

MCS queue lock: Performance

The MCS lock has the same good performance and space complexity as the CLH lock.

A thread can reuse its own node.

Its performance doesn't require caches.

The price to pay is a more involved `unlock` method.

CLH queue lock with timeout

With *exponential backoff*, a waiting thread can abandon its attempt to get the lock. But with *queue locks* this isn't so easy.

We extend the CLH lock with **timeouts**.

Again threads wait for the lock in a queue of nodes.

tail points to the most recently added node (initially it is *null*).

The **pred** field in a node of a thread *A* contains a pointer, either:

- ▶ *null*, if *A* is waiting in the queue or is in its critical section;
- ▶ to the node **AVAILABLE**, if *A* left its critical section; or
- ▶ to the node of *A*'s predecessor, if *A* timed out.

A new node needs to be allocated for each lock access.

CLH queue lock with timeout

A thread A that wants the lock creates a node ν containing *null*.

- ▶ A applies `getAndSet(ν)` to tail, to make ν the tail of the queue and get the node of A 's predecessor.
- ▶ If there is no predecessor, A takes the lock immediately.
- ▶ Else A spins on (a cached copy of) the pred field in the node of A 's predecessor until it isn't *null*.

If it points to the node `AVAILABLE`, A takes the lock.

If it points to a `new predecessor node`, A continues to spin on that node's pred field.

CLH queue lock with timeout

A thread that abandons its attempt to get the lock:

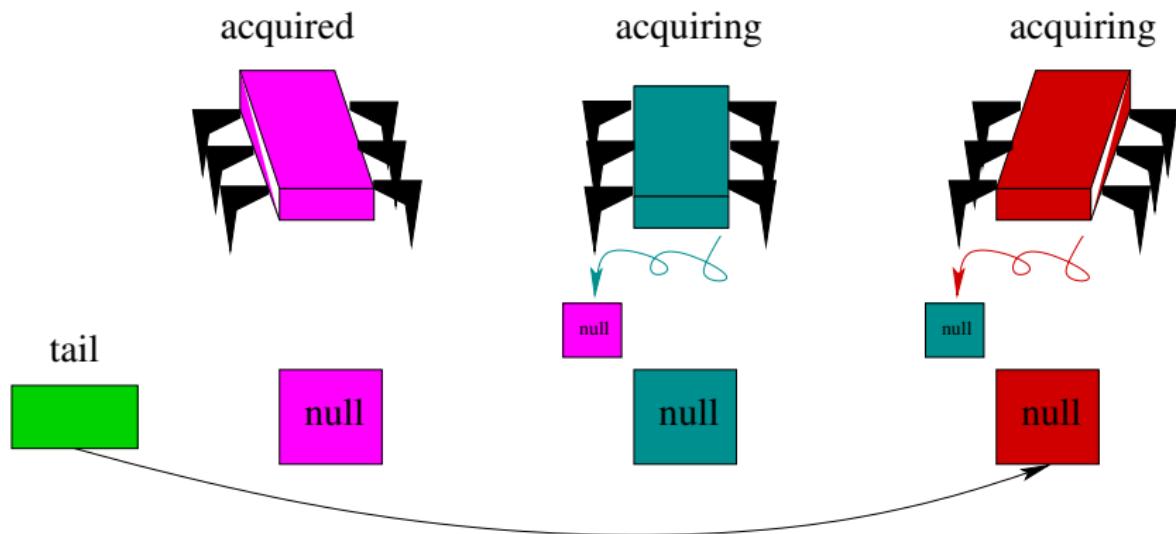
- ▶ sets its `pred` field to its predecessor's `node`, signaling to its successor (if present) that it has a new predecessor.

A thread that releases the lock:

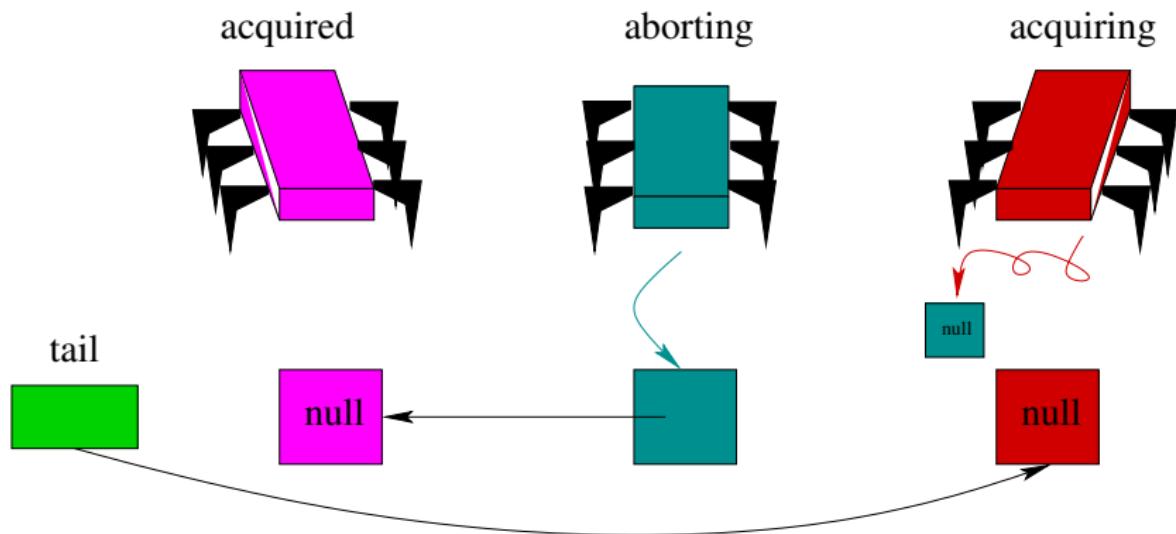
- ▶ applies `compareAndSet(ν , null)` to `tail`, with ν the node of the thread.
- ▶ If this call succeeds, the queue has become empty.
- ▶ If this call doesn't succeed, there is a successor.

Then the thread sets its `pred` field to the node `AVAILABLE`, signaling to its successor that it can take the lock.

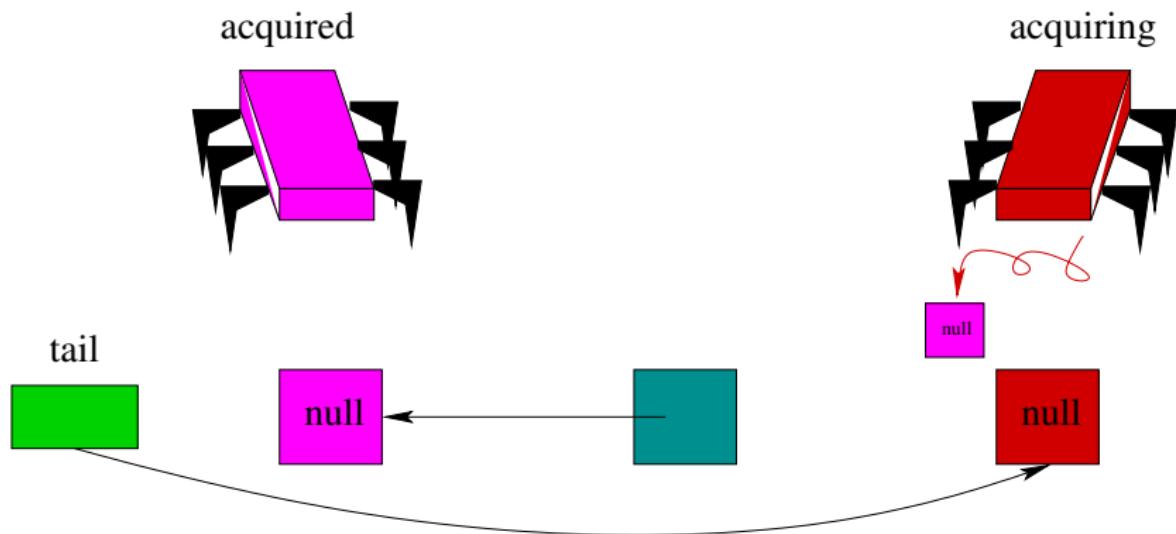
CLH queue lock with timeout: Example



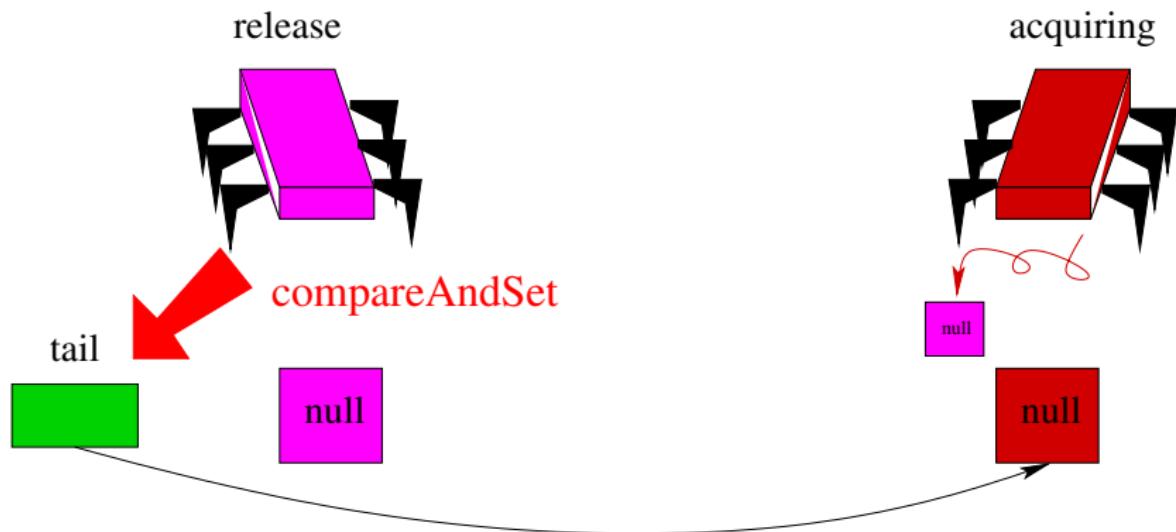
CLH queue lock with timeout: Example



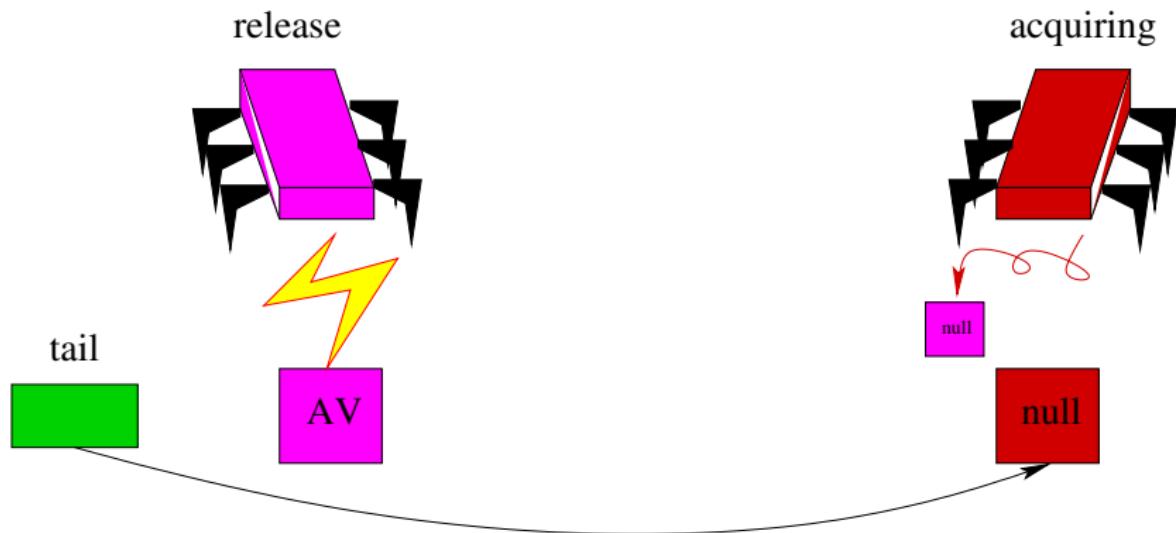
CLH queue lock with timeout: Example



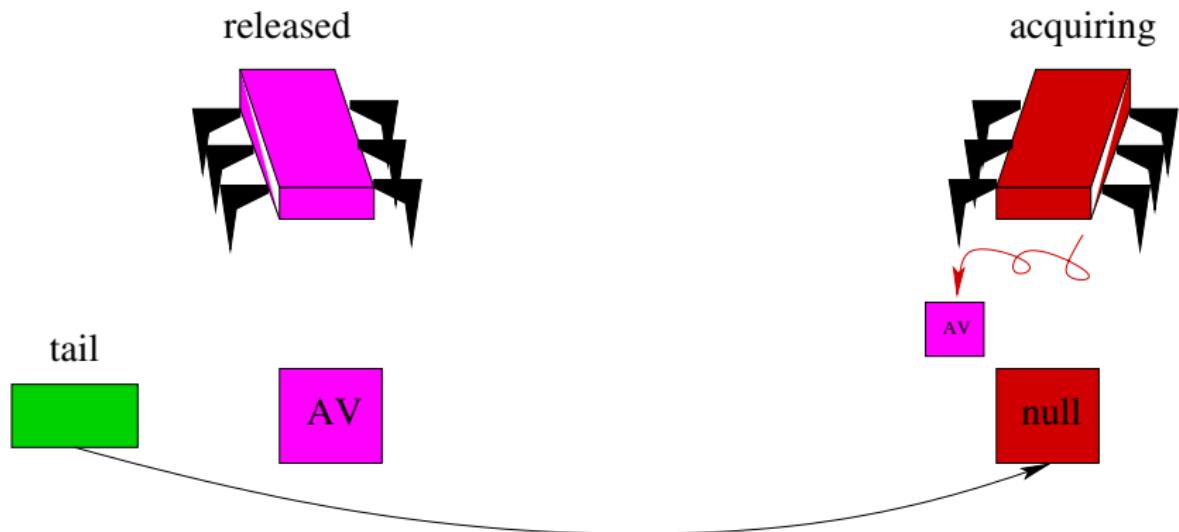
CLH queue lock with timeout: Example



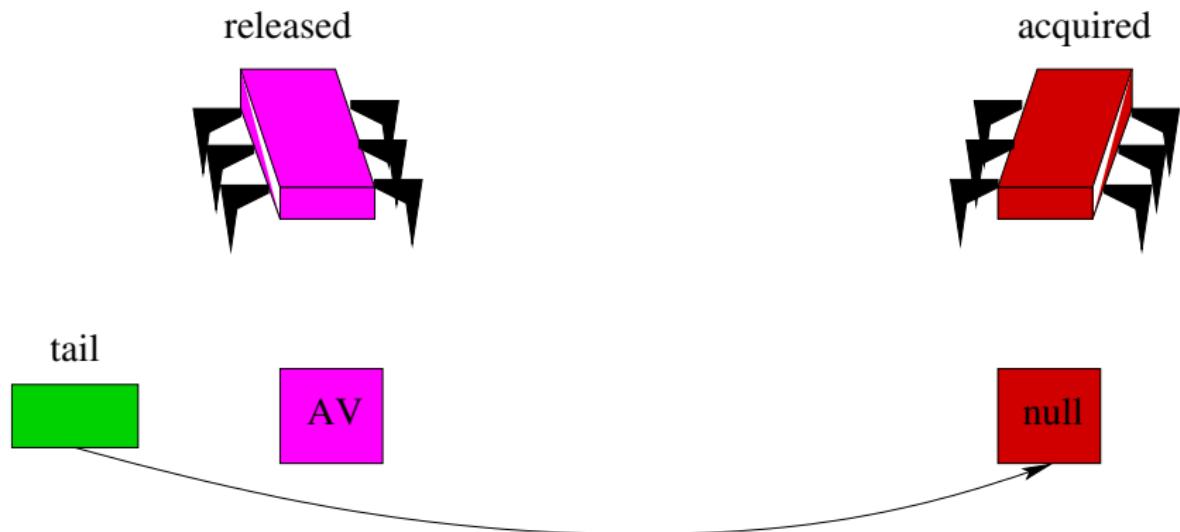
CLH queue lock with timeout: Example



CLH queue lock with timeout: Example



CLH queue lock with timeout: Example



Question

Why does a thread that releases the lock not simply set its pred field to *AVAILABLE* ?

Answer: Most lock accesses are uncontended.

If `compareAndSet(ν ,null)` succeeds, then ν can be removed by a garbage collector.

This lecture in a nutshell

spinning versus backoff

TAS and TTAS lock don't perform well

TTAS lock with exponential backoff

array-based queue lock

false sharing

CLH queue lock

MCS queue lock

CLH queue lock with timeout

Objects manage their locks

Objects better manage their own locks (instead of the threads).

Else threads should be reprogrammed at, or programmed to cope with, changes in datastructures.

Example: Suppose a thread holds the lock of a bounded FIFO queue, and wants to enqueue an item while the queue is full.

Should the method call be blocked, continue, or throw an exception ?

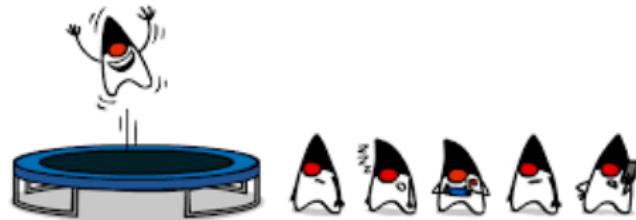
This decision may depend on the internal state of the queue, which is inaccessible to the caller.

Monitors

In Java, a **monitor** is associated with an object.

It combines **data**, **methods** and **synchronization** in one modular package.

Always at most one thread may be executing a method of the monitor.



A monitor provides mechanisms for threads to:

- ▶ temporarily give up exclusive access, until some *condition* is met,
- ▶ or *signal* to other threads that such a condition may hold.

Conditions

While a thread is waiting, say for a queue to become nonempty, the lock on this queue should be released.

Else other threads would never be able to enqueue an item.

In Java, a **Condition** object associated with a lock allows a thread to release the lock temporarily, by calling the `await()` method.

A thread can be awakened:

- ▶ by another thread (that performs `signal()` or `signalAll()`),
- ▶ or because some condition becomes true.

Conditions

An awakened thread must:

- ▶ try to reclaim the lock;
- ▶ when this has happened, retest the property b it is waiting for;
- ▶ if b doesn't hold, release the lock by calling `await()` again.

wrong: `if $\neg b$ condition_name.await()`

correct: `while $\neg b$ condition_name.await()`

($\neg b$ is the negation of the Boolean property b .)

The thread may be woken up if another thread calls
`condition_name.signal()` (which wakes up one thread) or
`condition_name.signalAll()` (which wakes up all threads).

Bounded FIFO queue with locks and conditions: Enqueue

```
final Condition notFull = lock.newCondition();
final Condition notEmpty = lock.newCondition();
public void enq(T x) {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[tail] = x;
        if (++tail == items.length)
            tail = 0;
        count++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

Bounded FIFO queue with locks and conditions: Dequeue

```
public T deq() {  
    lock.lock();  
    try {  
        while (count == 0)  
            notEmpty.await();  
        T y = items[head];  
        if (++head == items.length)  
            head = 0;  
        count--;  
        notFull.signal();  
        return y;  
    } finally {  
        lock.unlock();  
    }  
}
```

Lost-wakeup problem

Condition objects are vulnerable to **lost wakeups**:

A thread may wait forever without realizing that the condition it is waiting for has become true.



Example: Let `enq()` perform `notEmpty.signal()` only if the queue turns from empty to nonempty:

```
if count++ == 0  
    notEmpty.signal();
```

A lost wakeup can occur if multiple dequeuers are waiting.

Only one dequeuer is woken up, even if two elements are enqueued.

Programming practices to avoid lost wakeups:

- ▶ Signal *all* threads waiting for a condition (not just one).
- ▶ Specify a timeout for waiting threads.

Relaxing mutual exclusion

The strict mutual exclusion property of locks is often relaxed.

© Cartoonbank.com



"Aloud!"

Three examples are:

- ▶ **Readers-writers lock:** Allows concurrent readers, while a writer disallows concurrent readers and writers.
- ▶ **Reentrant lock:** Allows a thread to acquire the same lock multiple times, to avoid deadlock.
- ▶ **Semaphore:** Allows at most c concurrent threads in their critical sections, for some given capacity c .

Readers-writers lock: Reader lock

```
class ReadLock implements Lock {  
    public void lock() {  
        lock.lock();  
        try {  
            while writer  
                condition.await();  
            readers++;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Readers-writers lock: Reader lock

```
public void unlock() {  
    lock.lock();  
    try {  
        readers--;  
        if readers == 0  
            condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

Readers-writers lock: Writer lock

```
class WriteLock implements Lock {  
    public void lock() {  
        lock.lock();  
        try {  
            while (writer || readers > 0)  
                condition.await();  
            writer = true;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Readers-writers lock: Writer lock

```
public void unlock() {  
    lock.lock();  
    try {  
        writer = false;  
        condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

The unlock method needs to grab the lock: A thread can only signal or start to await a condition if it owns the corresponding lock.

Readers-writers lock: Writer lock

Question: What is the drawback of this writer lock ?

Answer: A writer can be delayed indefinitely by a continuous stream of readers.

Question: How can this be resolved ?

Answer: Allow one writer to set writer = true if writer == false.

Then other threads can't start to read.

Readers-writers lock: Writer lock

```
class WriteLock implements Lock {  
    public void lock() {  
        lock.lock();  
        try {  
            while writer  
                condition.await();  
            writer = true;  
            while readers > 0  
                condition.await();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

The unlock method is as before.

Question

How can we let a reader only signal to writers?

Answer: Use two condition objects; one for readers and one for writers.

A reader signals to writers if `readers == 0` and `writer == true`.

A writer signals to readers and writers.

Reentrant lock

```
class SimpleReentrantLock implements Lock {  
    public void lock() {  
        int me = ThreadID.get();  
        lock.lock();  
        try {  
            if owner == me  
                { holdCount++; return; }  
            while holdCount != 0  
                condition.await();  
            owner = me;  
            holdCount = 1;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

Reentrant lock

```
public void unlock() {  
    lock.lock();  
    try {  
        holdCount--;  
        if holdCount == 0  
            condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

Semaphore

```
public void acquire() {  
    lock.lock();  
    try {  
        while state == capacity  
            condition.await();  
        state++;  
    } finally {  
        lock.unlock();  
    }  
}
```

Semaphore

```
public void release() {  
    lock.lock();  
    try {  
        state--;  
        condition.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

Actually, `signal()` may be preferable here.

Or only do a `signalAll()` if `state` had the value capacity.

Semaphores in Java

In Java, `Semaphore(int n)` creates a semaphore of capacity `n`.

`acquire(int k)` acquires `k` permits from the semaphore, blocking until all are available.

`release(int k)` releases `k` permits to the semaphore.

Synchronized methods in Java

While a thread is executing a **synchronized method** on an object, other threads that invoke a synchronized method on this object block.

A synchronized method acquires the **intrinsic** lock of the object on which it is invoked, and releases the lock when it returns.

A synchronized method imposes a **memory barrier**:

- ▶ At the start, the cache is invalidated.
- ▶ At completion, modified fields in working memory are written back to shared memory.

Synchronized methods

Synchronized methods are *reentrant*.

Monitors are provided for synchronized methods:

- ▶ `wait()` causes a thread to wait until another thread notifies it of a condition change;
- ▶ `notify()` wakes up one waiting thread;
- ▶ `notifyAll()` wakes up all waiting threads.

Synchronized methods: Drawbacks

Synchronized methods

1. aren't starvation-free
2. are rather coarse-grained
3. can give a false feeling of security
4. may use other locks than one might expect

Synchronized methods: Drawback 3

Example: Instead of using a lock and conditions, we make the `enq()` method on the bounded queue synchronized.

(So `await` and `signal` are replaced by `wait` and `notify`.)

One might expect that other method calls will always see a proper combination of values of the variables `count` and `tail`.

But this is only guaranteed for other *synchronized* methods on queues.

Synchronized methods: Drawback 4

A *static* synchronized method acquires the intrinsic lock of a *class* (instead of an object).

A static synchronized method on an *inner* class only acquires the intrinsic lock of the inner class, and not of the enclosing class.

Synchronized blocks

A **synchronized block** must specify the object that provides the intrinsic lock.

Example:

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

Danger: Nested synchronized blocks may cause deadlock if they acquire locks in opposite order.

Barrier synchronization

Suppose a number of tasks must be completed before an overall task can proceed.

This can be achieved with **barrier synchronization**.

A barrier keeps track whether all threads have reached it.

When the last thread reaches the barrier, all threads resume execution.

Waiting at a barrier resembles waiting to enter a critical section.

It can be based on **spinning** (remote or on a locally cached copy), or on **being woken up**. We discuss barriers based on spinning.



Sense-reversing barrier

The **sense-reversing** barrier consists of:

- ▶ a **counter**, initialized to the barrier size n , and
- ▶ a Boolean **sense** field, initially *false*.

Each thread has a **local sense**, initially *true*.

Each thread that reaches the barrier applies **getAndDecrement** to lower the counter.

Sense-reversing barrier

If the thread *isn't* the last to reach the barrier
(i.e., it decreases the counter to a value > 0), then

- ▶ the thread spins on the barrier's sense field
- ▶ until it matches the thread's local sense.

If the thread *is* the last to reach the barrier
(i.e., it decreases the counter to 0), then

- ▶ the thread resets the counter to n , and
- ▶ reverses the barrier's sense field.

Threads resume execution with reversed local sense,
so that the barrier can be reused.

Sense-reversing barrier: Example

Given three threads *A*, *B* and *C*, initially with local sense *true*.

Initially the barrier's counter has the value 3, and its sense is *false*.

Thread *B* reaches the barrier, applies `getAndDecrement` to the counter, reads counter value 3, and spins on the barrier's sense field.

Thread *A* reaches the barrier, applies `getAndDecrement` to the counter, reads counter value 2, and spins on the barrier's sense field.

Thread *C* reaches the barrier, applies `getAndDecrement` to the counter, and reads counter value 1.

C resets the counter of the barrier to 3, reverses the barrier's sense field to *true*, and leaves the barrier with reversed local sense *false*.

Threads *A* and *B* notice that the sense of the barrier is *true*, and leave the barrier with reversed local sense *false*.

Sense-reversing barrier: Evaluation

Letting all threads spin on the barrier's sense field creates a performance bottleneck.

In **cache-coherent** (symmetric multiprocessing) architectures, threads can spin on a locally cached copy of the barrier's sense field.

In **cacheless** (non-uniform memory access) architectures, *suspending* threads may be a better idea. (See Exercise 199.)

Combining tree barrier

The **combining tree barrier** uses a tree of depth d , where each non-leaf has r children.

This is a barrier for (at most) r^{d+1} threads.

Each node is a **sense-reversing** barrier of capacity r .

Initially these barriers have sense *false*.

To each **leaf**, r threads are assigned.

Each thread has a local sense, initially *true*.

Combining tree barrier

A thread that reaches the barrier, applies `getAndDecrement` to the counter at its leaf.

- ▶ A thread that decreases the counter at a node to a value > 0 , spins on the sense field of this node until it matches its local sense.
- ▶ A thread that decreases the counter at a *non-root* to 0 , moves to its *parent*, and also applies `getAndDecrement` to the counter there.
- ▶ A thread that decreases the counter at the *root* to 0 , *resets the counter* and *reverses the sense* at the root.

Combining tree barrier

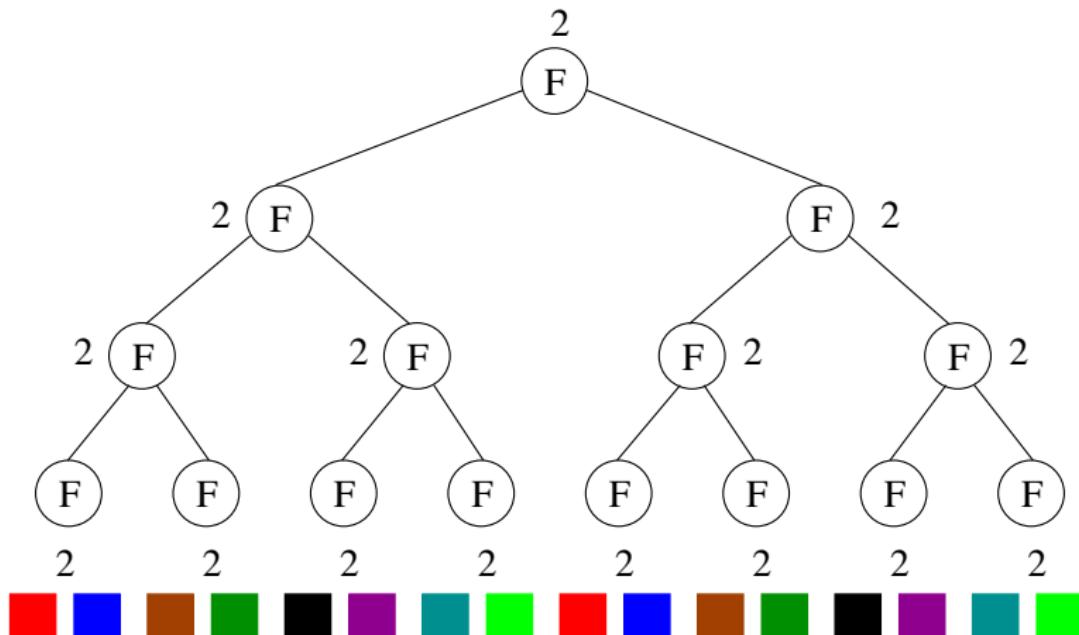
Threads that find the sense field they are spinning reversed, and the thread that reverses the sense at the root,

- ▶ reset the counter, and
- ▶ reverse the sense

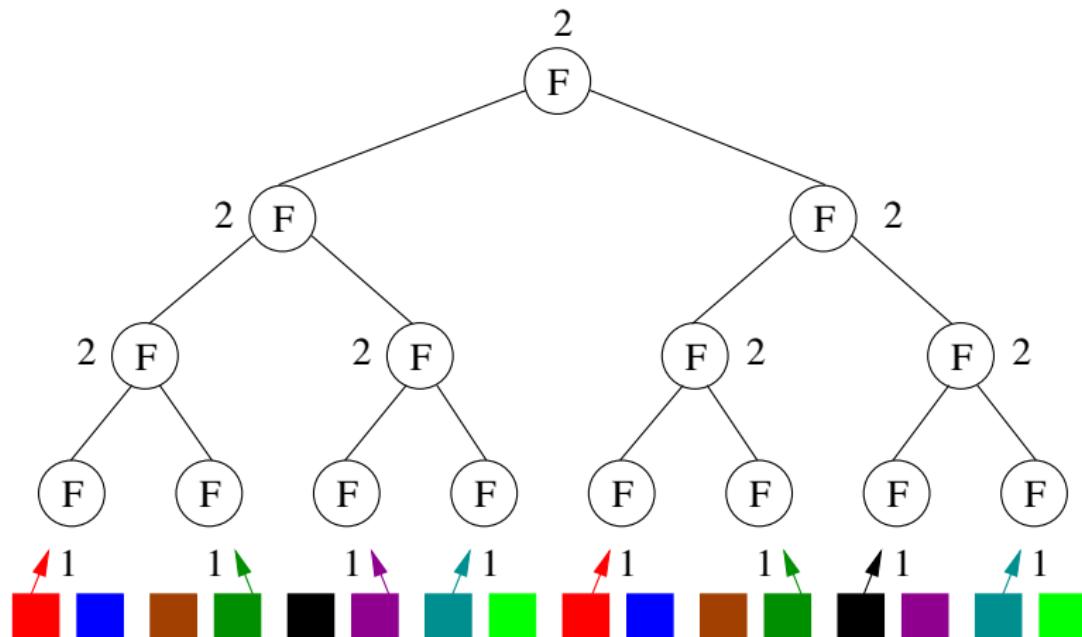
at all nodes they visited before.

Threads resume execution with reversed local sense.

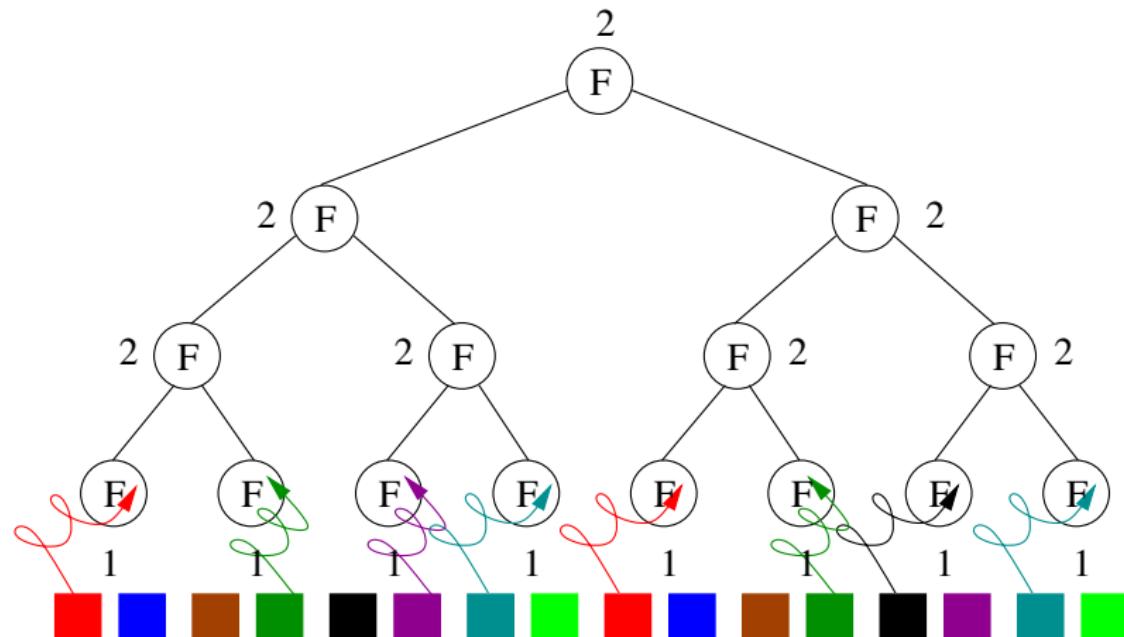
Combining tree barrier: Example



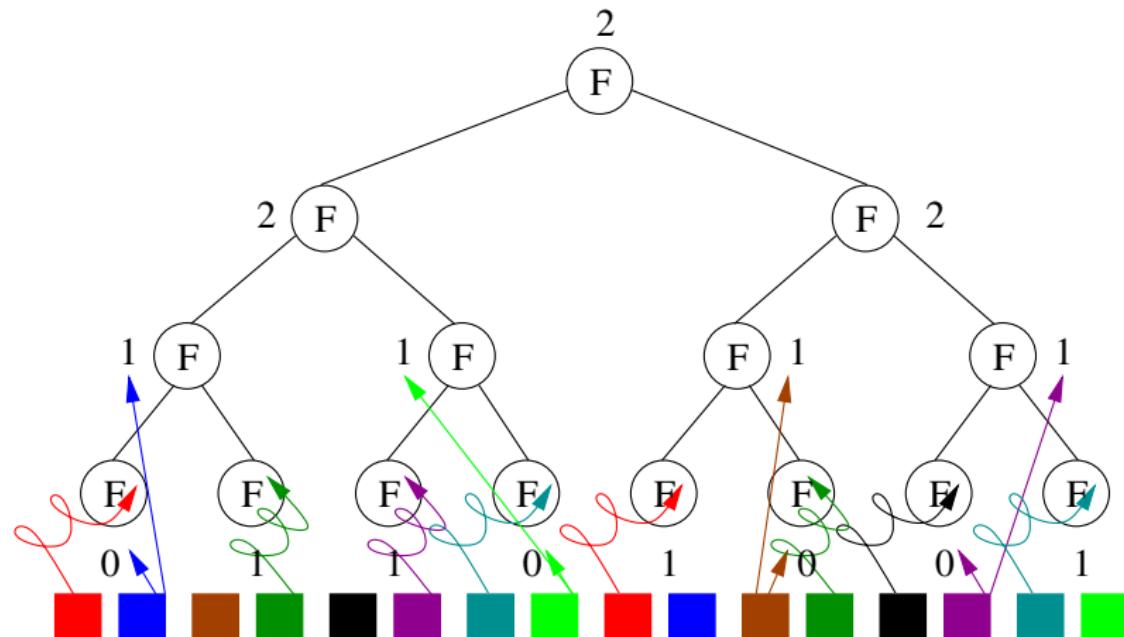
Combining tree barrier: Example



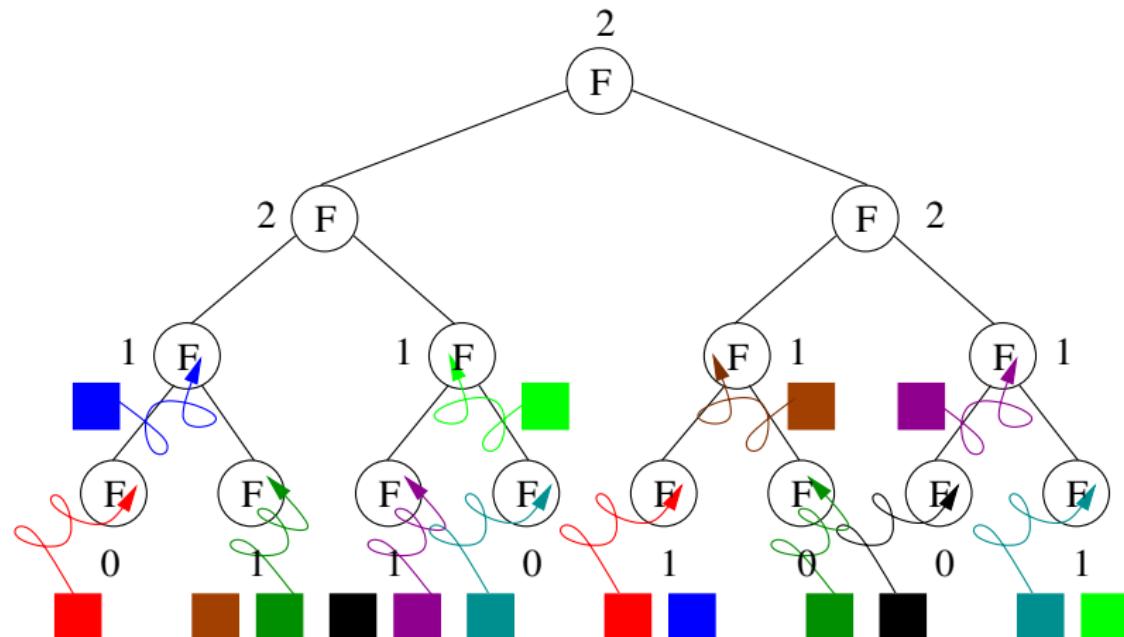
Combining tree barrier: Example



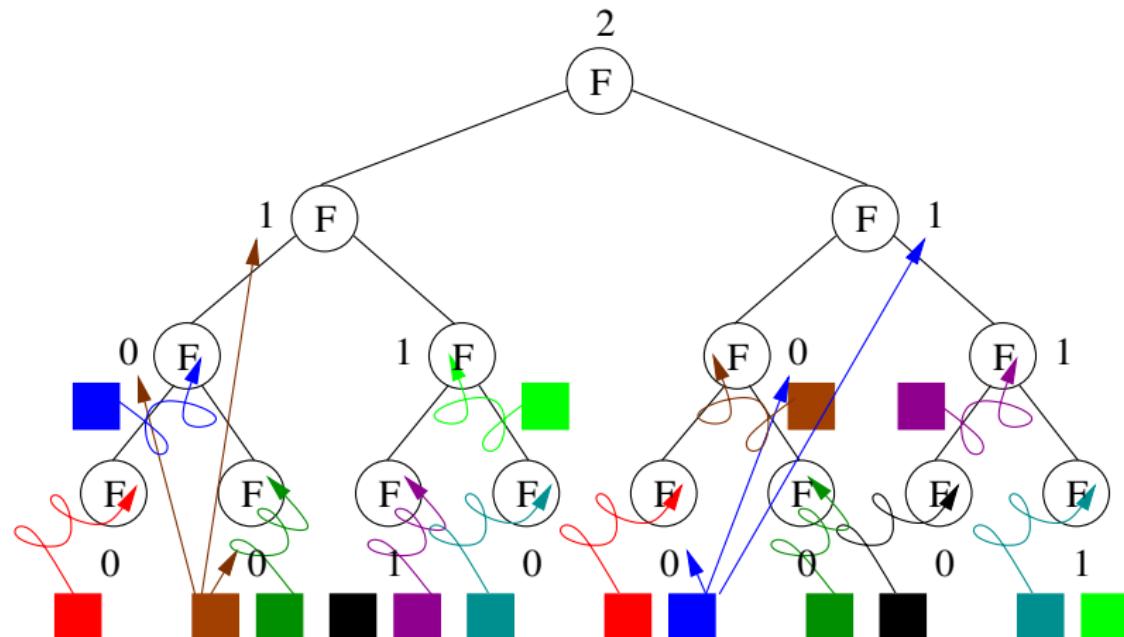
Combining tree barrier: Example



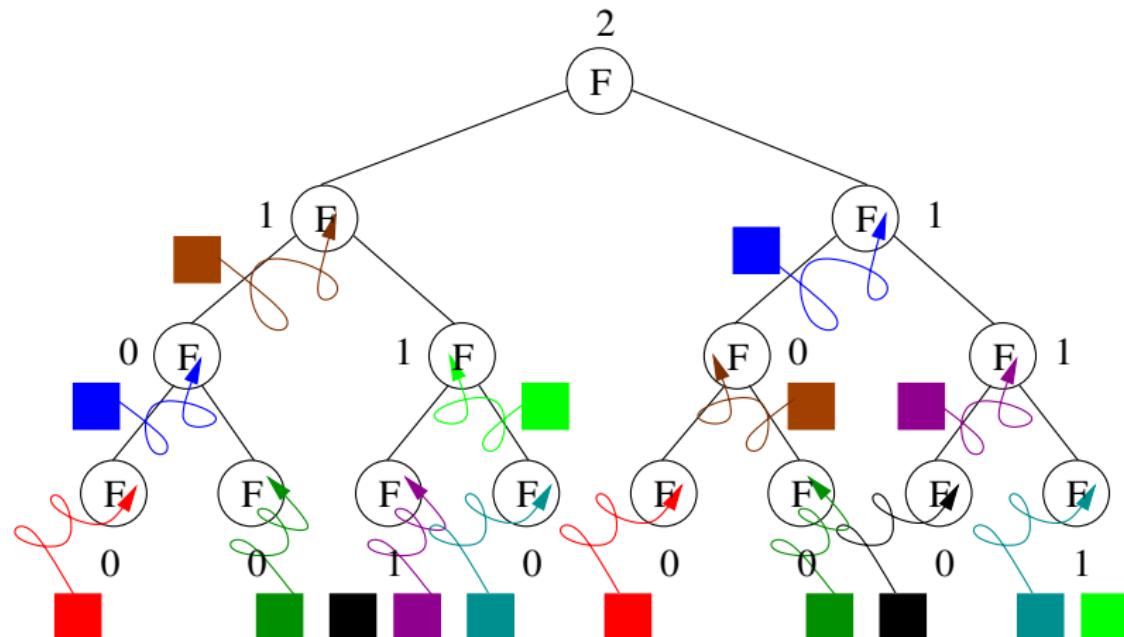
Combining tree barrier: Example



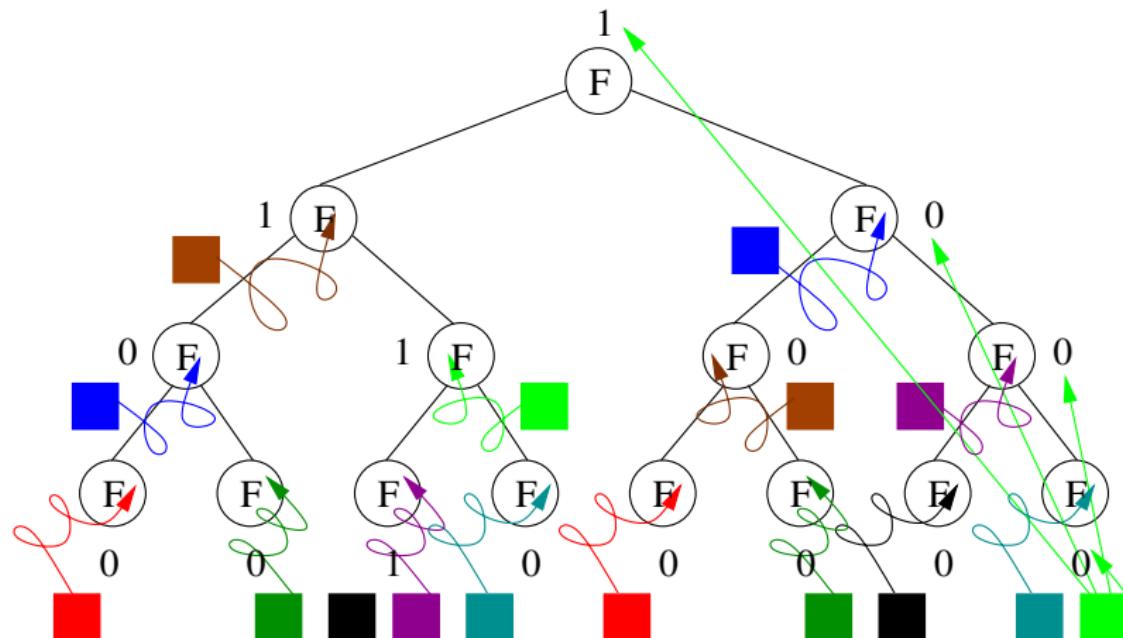
Combining tree barrier: Example



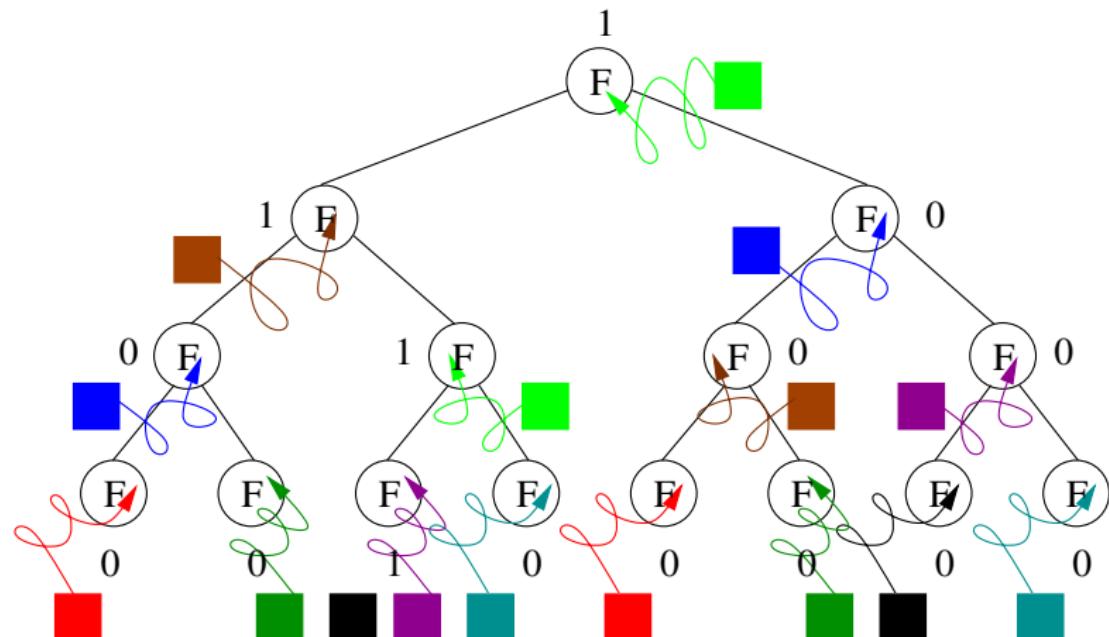
Combining tree barrier: Example



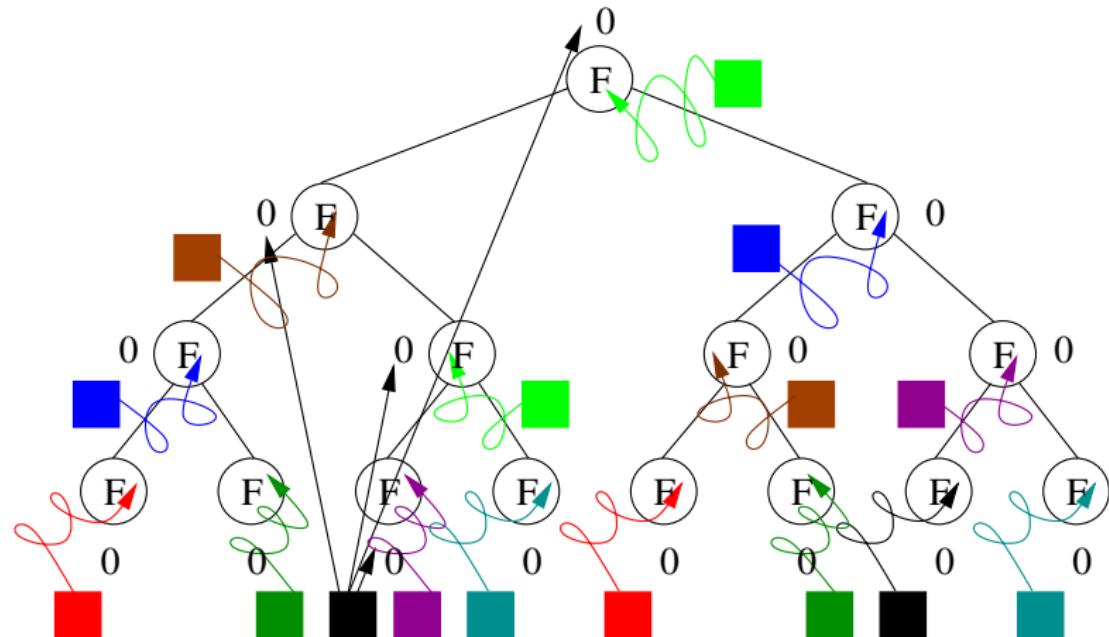
Combining tree barrier: Example



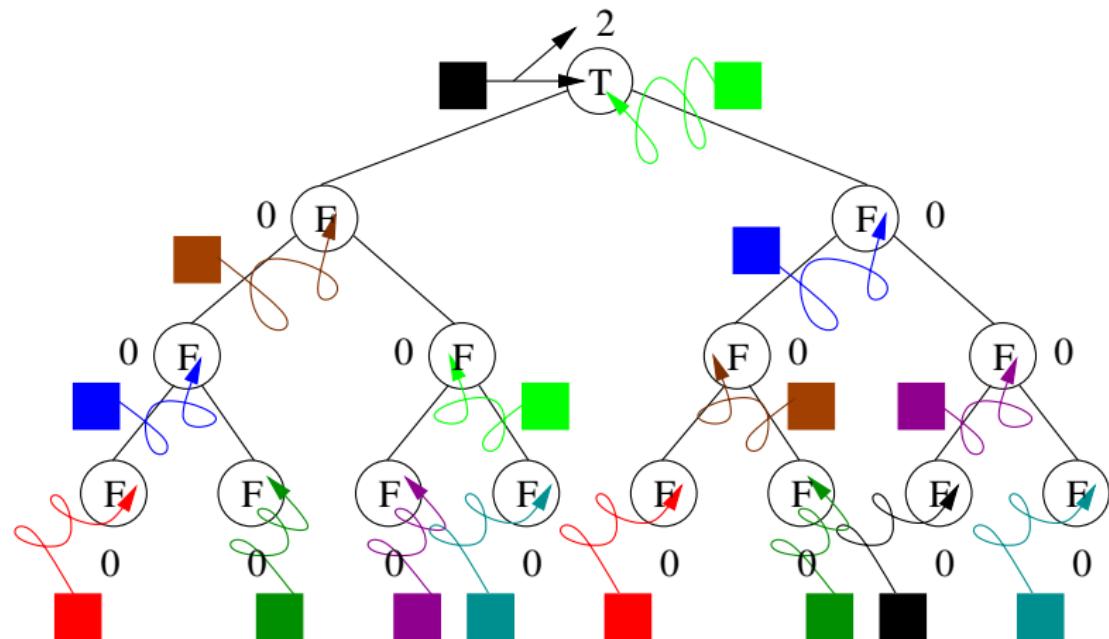
Combining tree barrier: Example



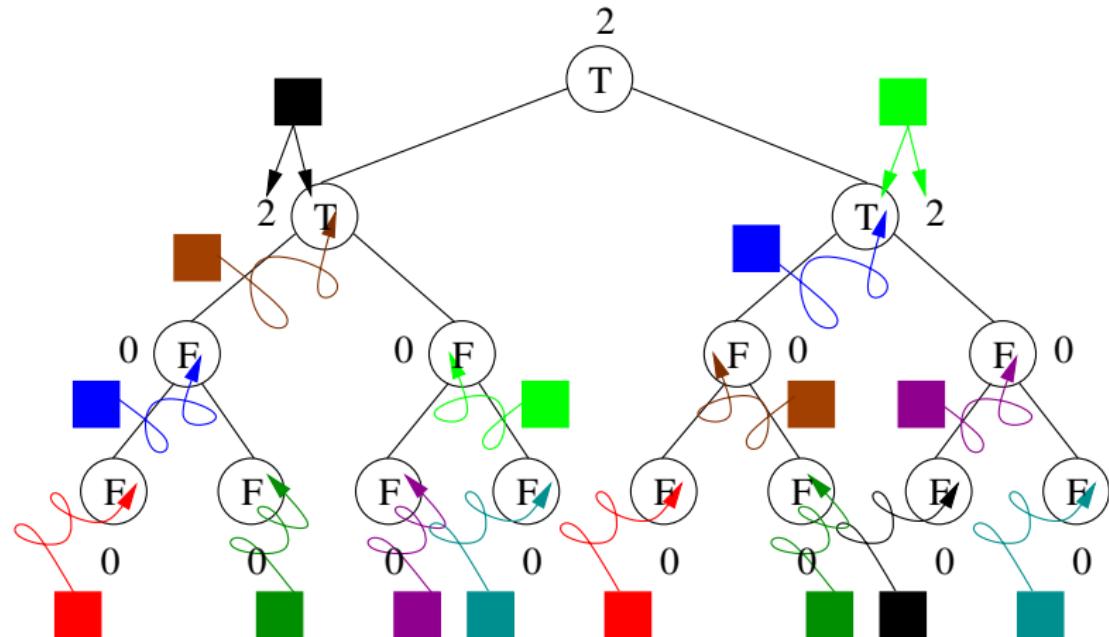
Combining tree barrier: Example



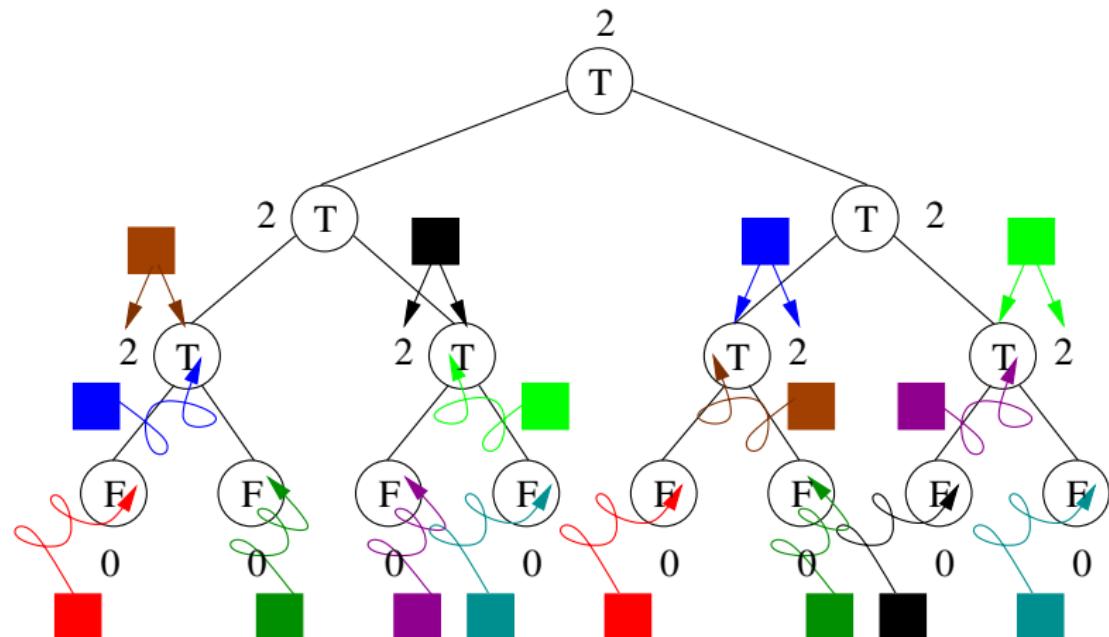
Combining tree barrier: Example



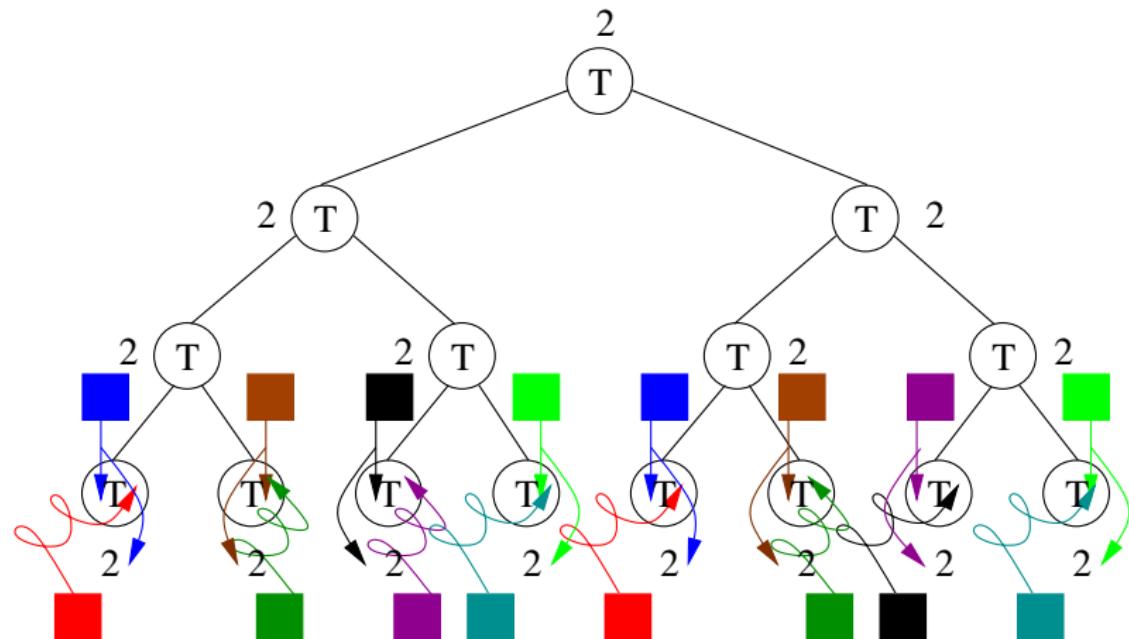
Combining tree barrier: Example



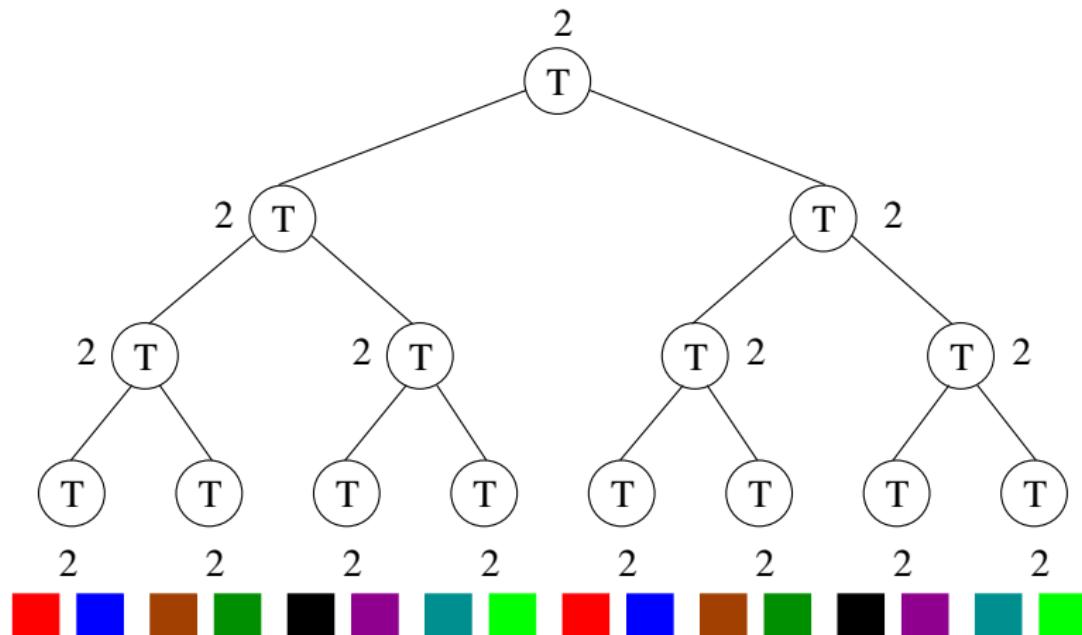
Combining tree barrier: Example



Combining tree barrier: Example



Combining tree barrier: Example



Combining tree barrier: Evaluation

Memory contention is reduced by spreading memory accesses over different nodes.

This is especially favorable for *cacheless* architectures.

In *cache-coherent* architectures, sense fields should be kept at different cache lines, to minimize cache traffic.

Tournament barrier

The **tournament barrier** uses a *binary tree* of depth d .

This a barrier for (at most) 2^{d+1} threads.

Each node is divided into two parts: **active** and **passive**.

Both parts of the node carry a Boolean flag, initially *false*.

The active and passive part of a *non-leaf* have one child each.

To each part of a *leaf*, one thread is assigned.

Threads have a local sense, initially *true*.

Tournament barrier

Suppose a thread reaches the barrier and starts at a leaf, or moves to a parent node.

A *passive* thread reverses the flag of its *active* partner, and spins on its own flag until it equals its local sense.

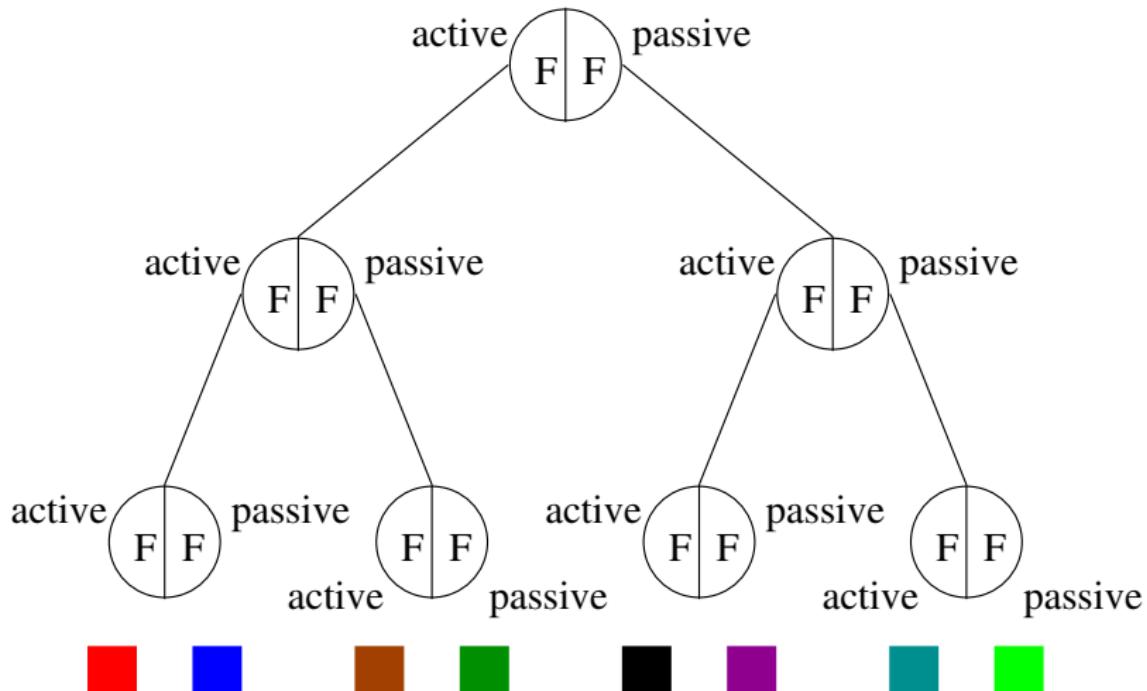
An *active* thread spins on its flag until it is reversed.

- ▶ In case of a *non-root*, it moves to its parent.
- ▶ In case of the *root*, it reverses the flags at the *passive* parts of the nodes it has visited.

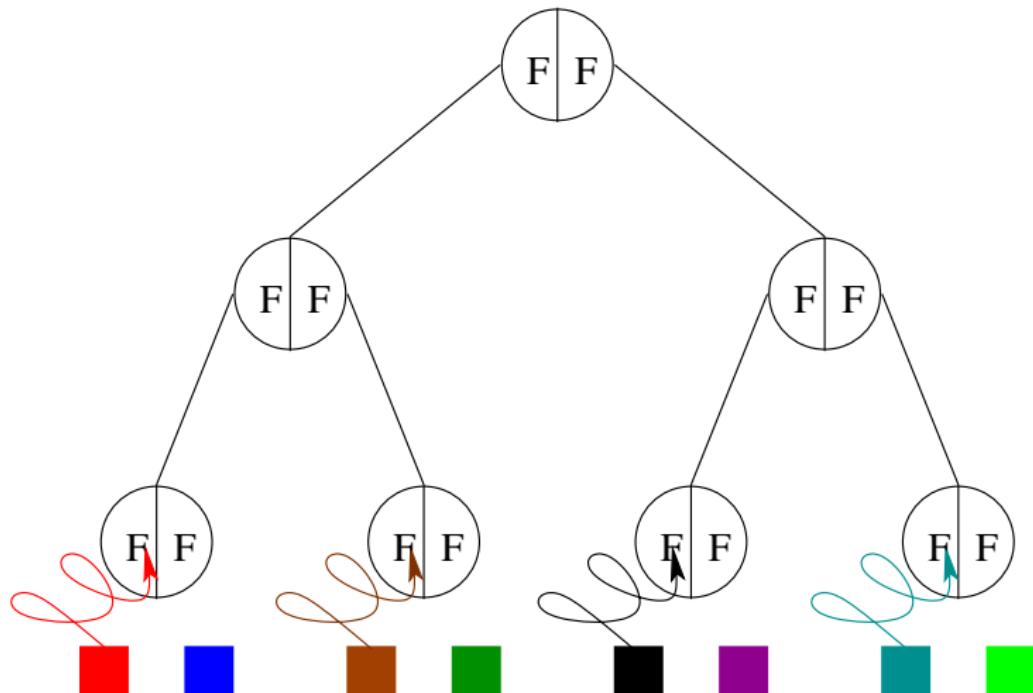
A *passive* thread that finds the flag it is spinning on reversed, reverses the flags at the *passive* parts of the nodes it has visited.

Threads resume execution with reversed local sense.

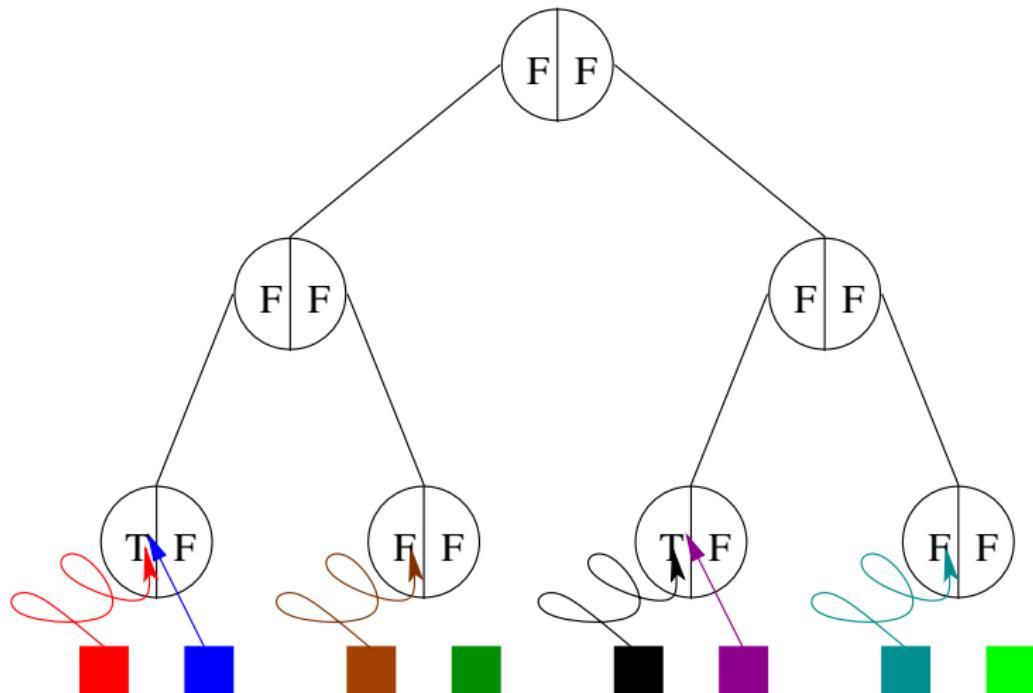
Tournament barrier: Example



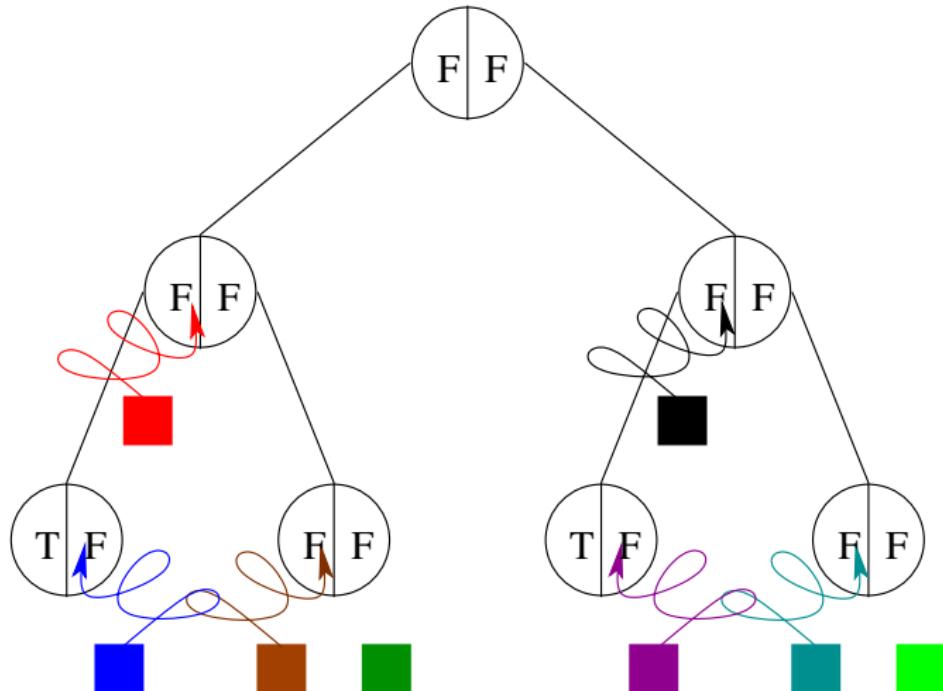
Tournament barrier: Example



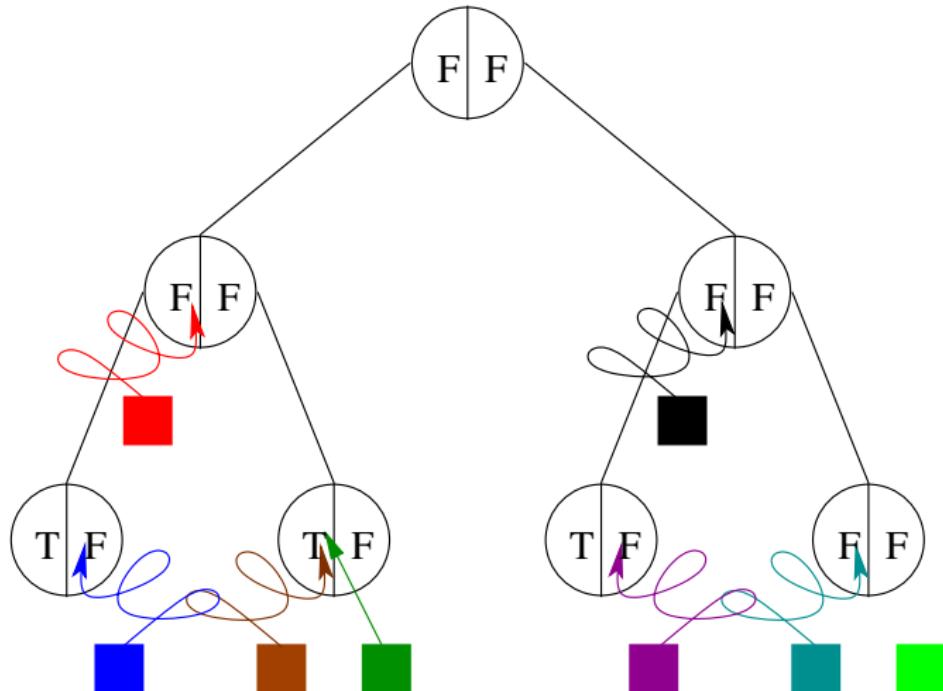
Tournament barrier: Example



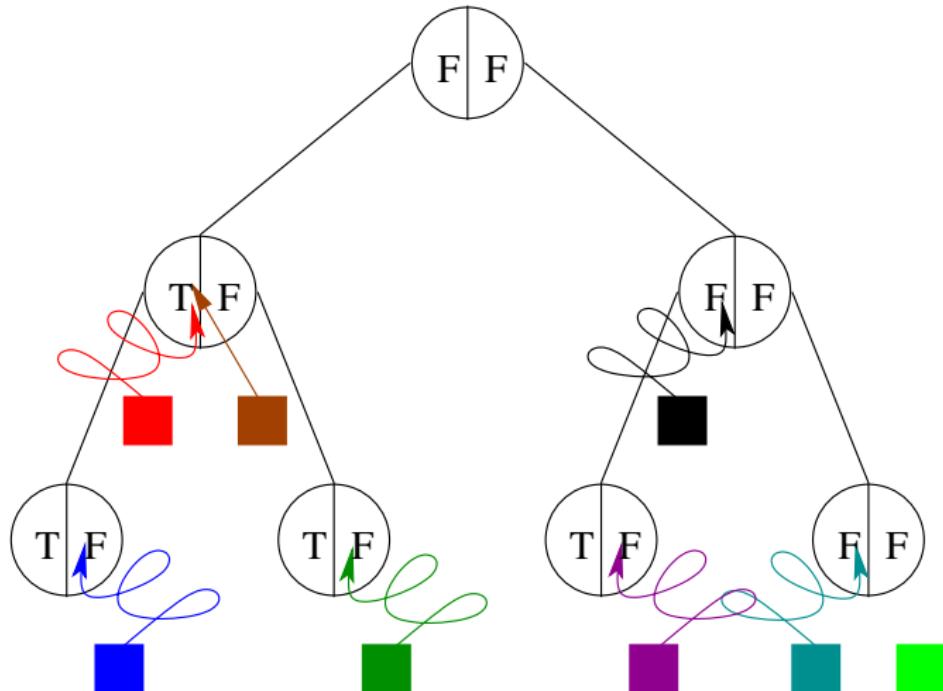
Tournament barrier: Example



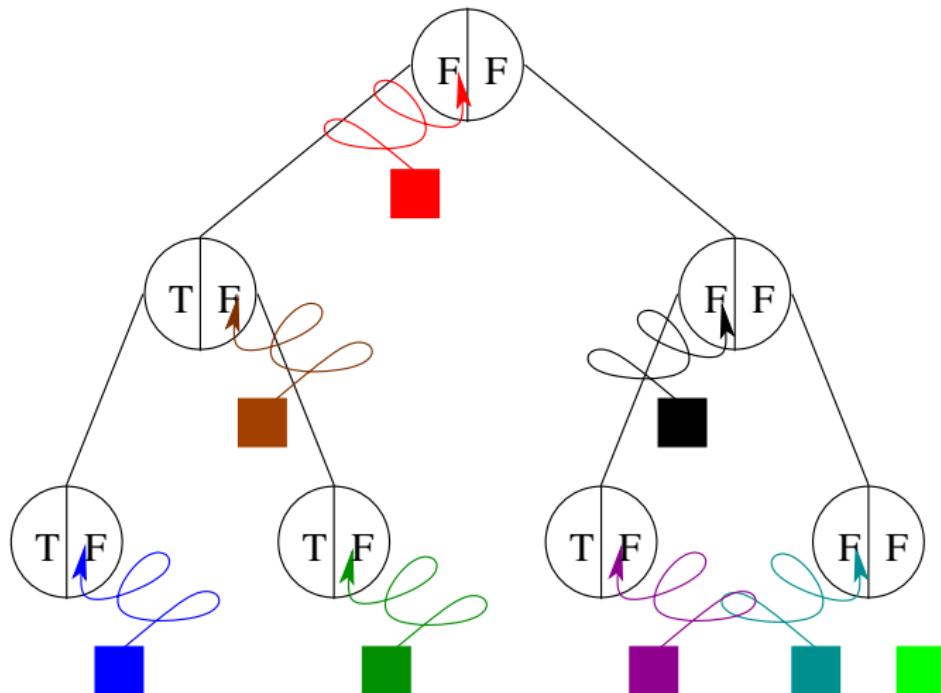
Tournament barrier: Example



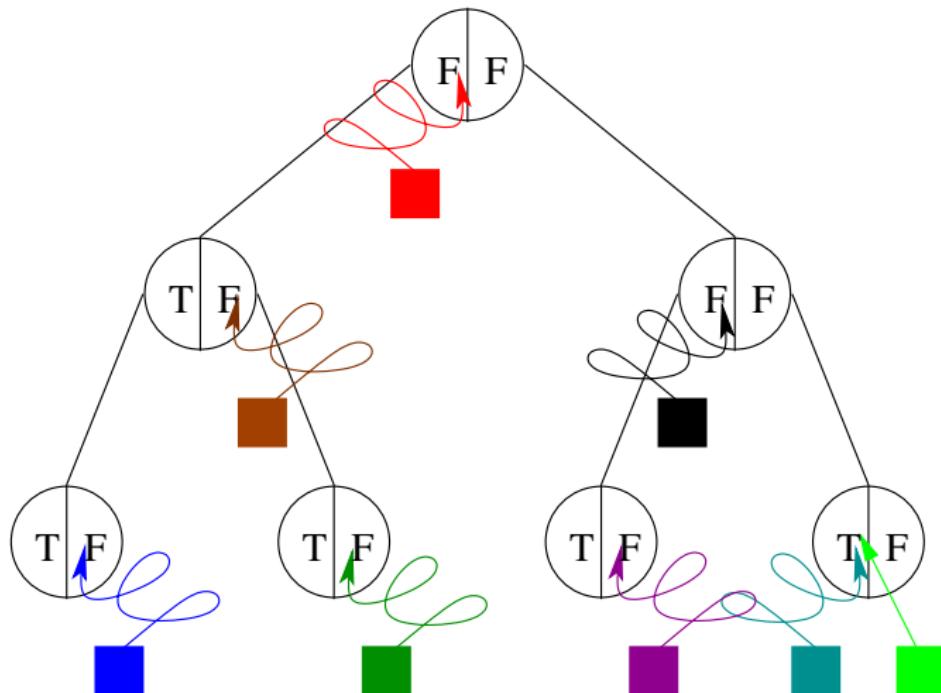
Tournament barrier: Example



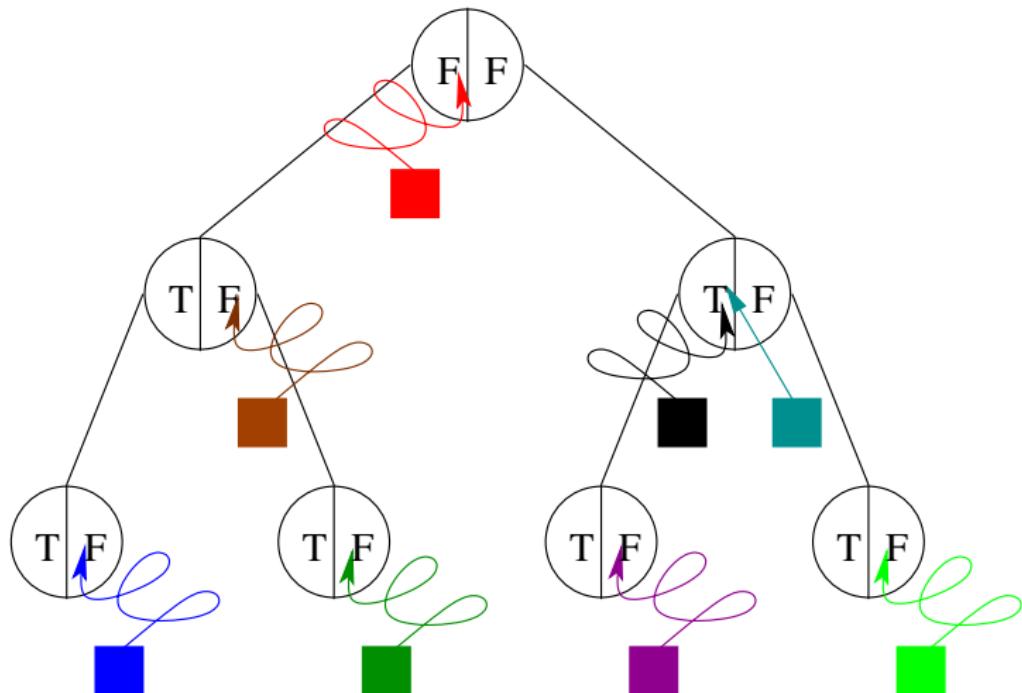
Tournament barrier: Example



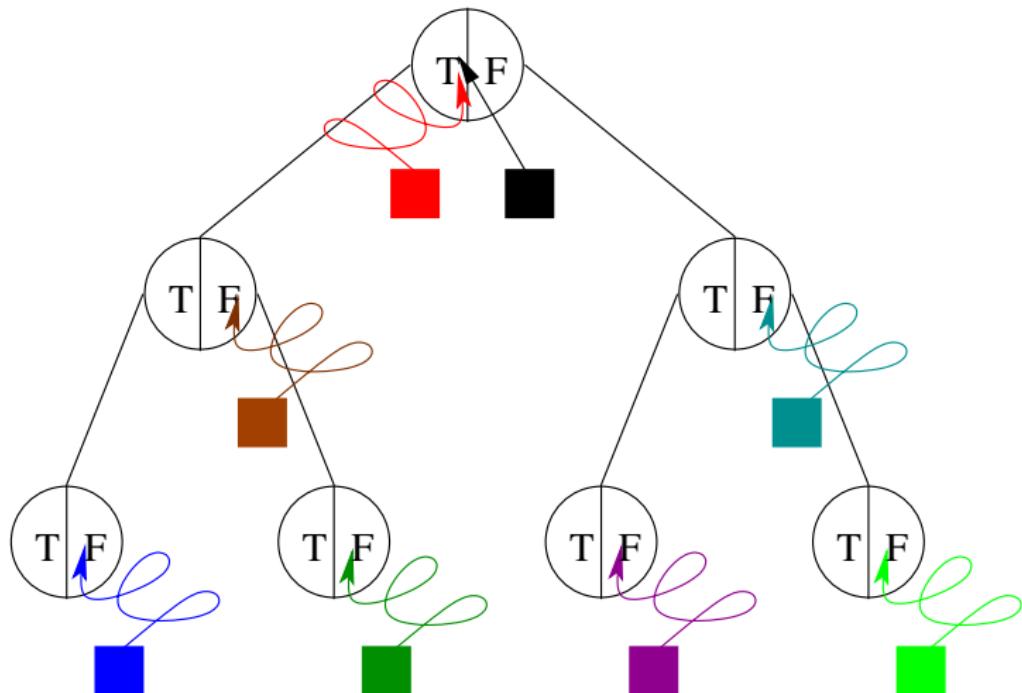
Tournament barrier: Example



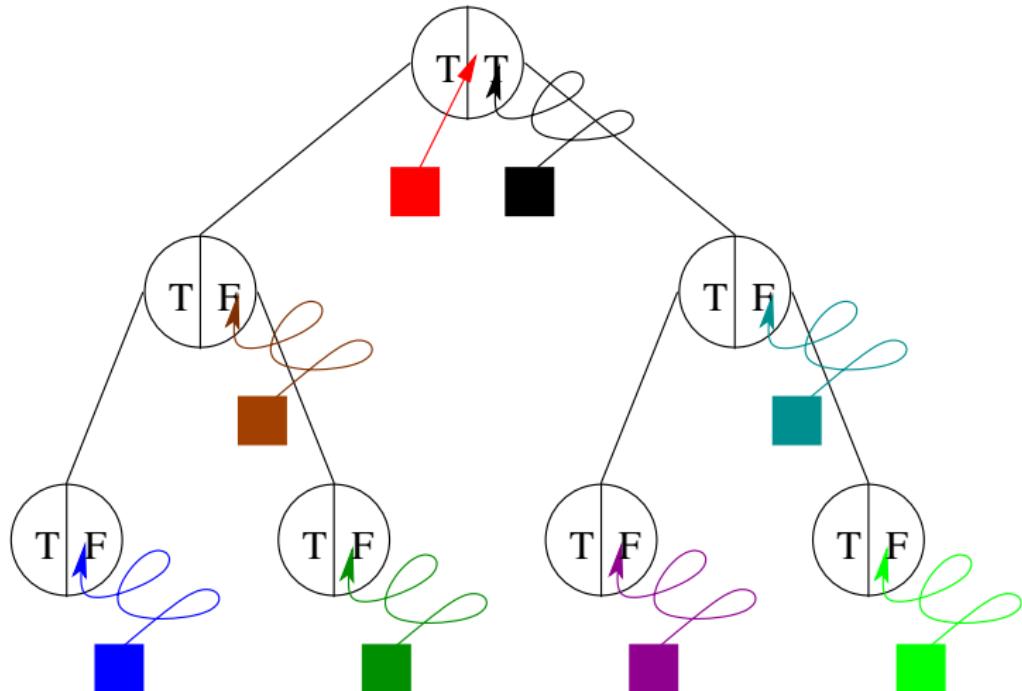
Tournament barrier: Example



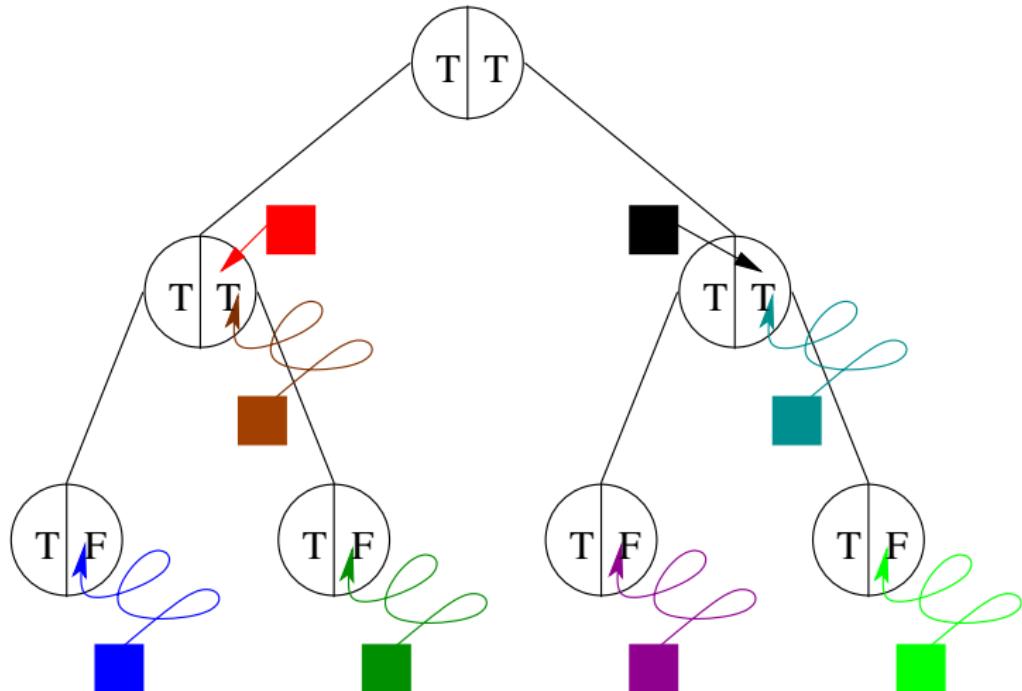
Tournament barrier: Example



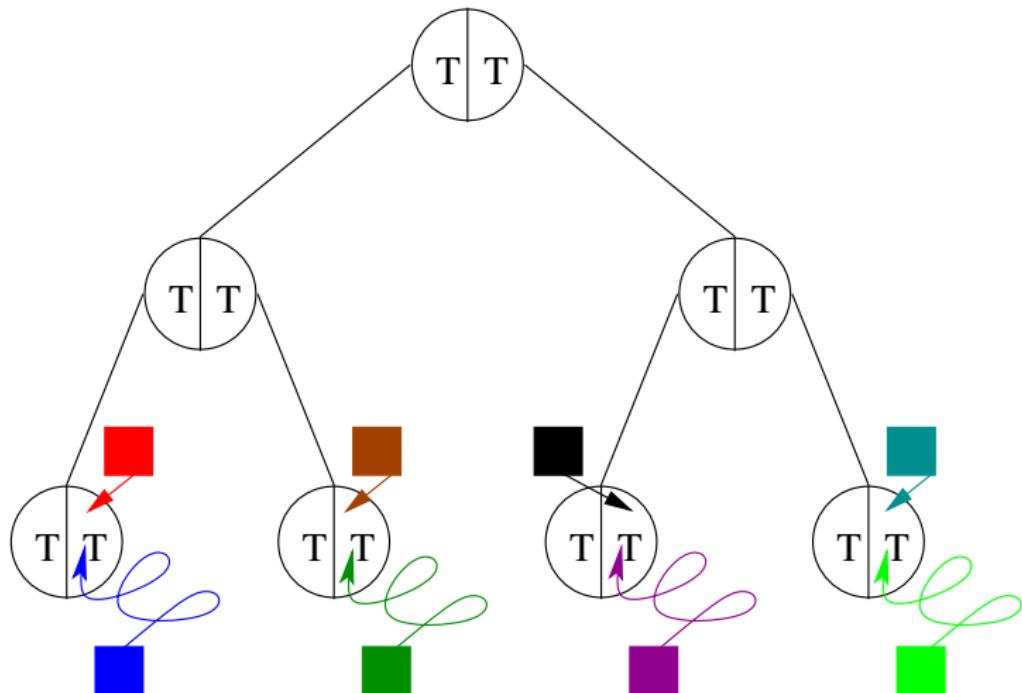
Tournament barrier: Example



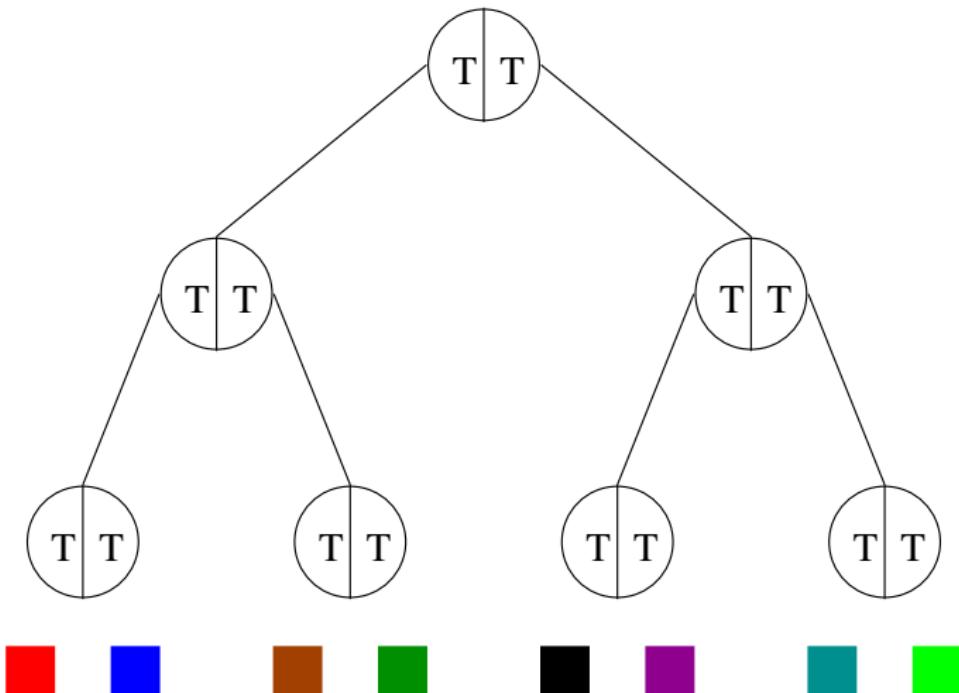
Tournament barrier: Example



Tournament barrier: Example



Tournament barrier: Example

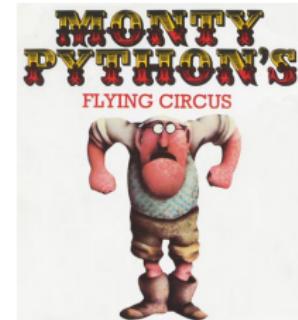


Tournament barrier: Evaluation

Threads spin on a local field, so no memory contention.

There is no shared counter, so no read-modify-write operation is needed to decrease it.

And now for something completely different: The dissemination barrier



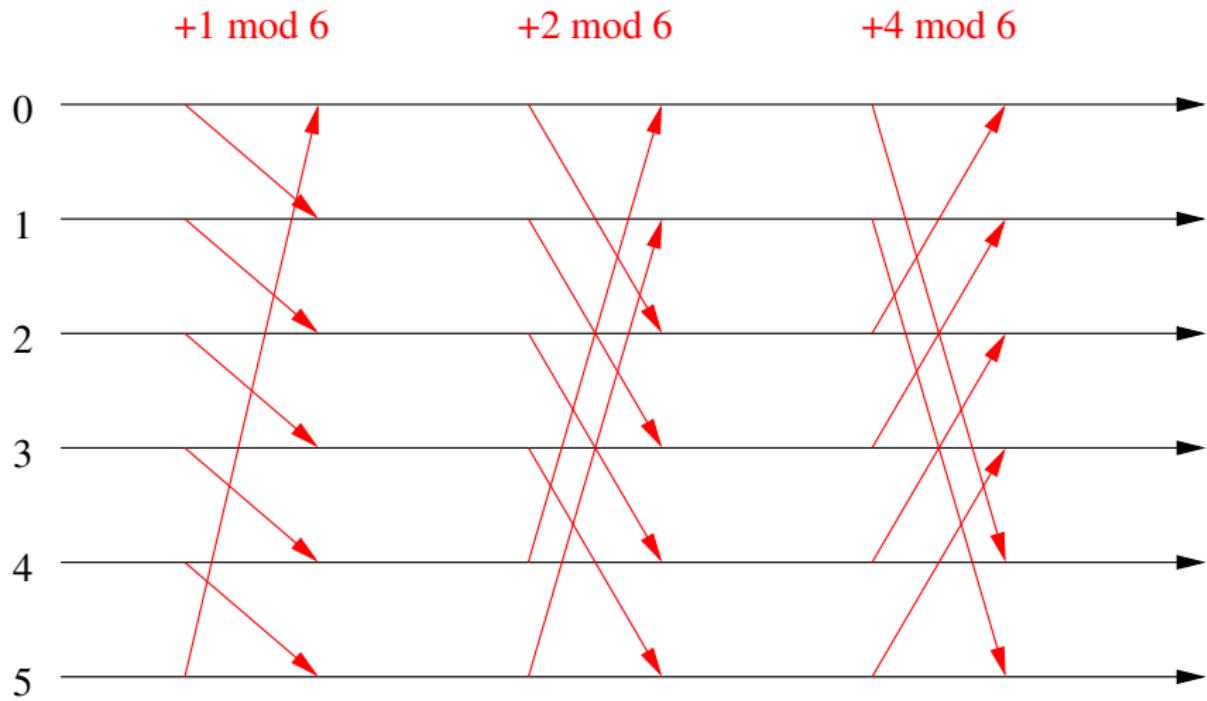
The **dissemination barrier**, for n threads $0, \dots, n - 1$.

In **round** $r \geq 0$, each thread i :

- ▶ notifies thread $i + 2^r \bmod n$, and
- ▶ waits for notification by thread $i - 2^r \bmod n$.

When $\lceil \log_2 n \rceil$ rounds have been completed,
all n threads have reached the barrier.

Dissemination barrier: Example



Dissemination barrier: Correctness

If all n threads have reached the barrier, all rounds can be completed.

Suppose that some thread i hasn't yet reached the barrier.

For convenience we take $n = 2^k$. (See Exercise 207 for general n .)

Thread $i + 1 \bmod n$ hasn't completed round 0.

Threads $i + 2, i + 3 \bmod n$ haven't completed round 1.

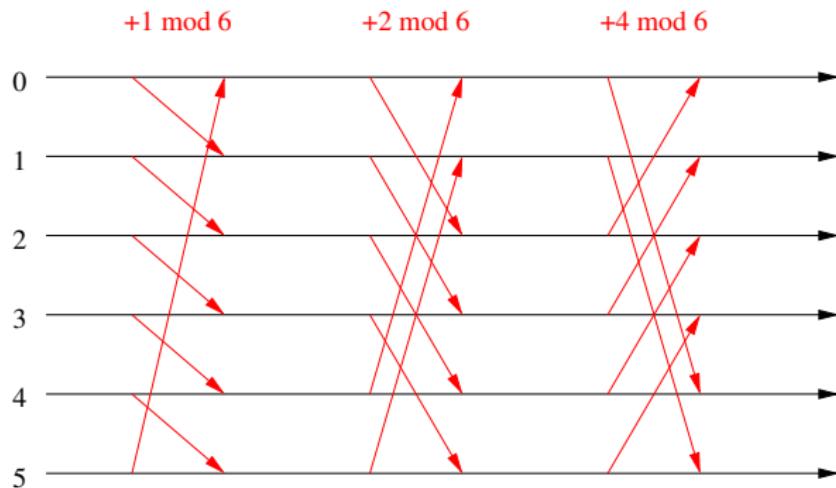
Threads $i + 4, i + 5, i + 6, i + 7 \bmod n$ haven't completed round 2.

...

Threads $i + 2^{k-1}, \dots, i + 2^k - 1 \bmod n$ haven't completed round $k-1$.

So no thread has left the barrier.

Dissemination barrier: Example revisited



Suppose thread 0 hasn't yet reached the barrier.

Thread 1 hasn't yet completed round 0.

Threads 2 and 3 haven't yet completed round 1.

Threads 4 and 5 haven't yet completed round 2.

Dissemination barrier: Evaluation

Threads spin on ($\lceil \log_2 n \rceil$) local fields.

No read-modify-write operation is needed.

Great for lovers of combinatorics.

Not so much fun for those who track memory footprints.



This lecture in a nutshell

monitor (combines data, methods and synchronization)

- ▶ conditions
- ▶ lost-wakeup problem

relaxed mutual exclusion

- ▶ readers-writer lock
- ▶ reentrant lock
- ▶ semaphore

synchronized method / synchronized block

barrier synchronization

- ▶ sense-reversing barrier
- ▶ combining tree barrier
- ▶ tournament barrier
- ▶ dissemination barrier

Synchronization approaches for datastructures

Coarse-grained synchronization, in which each method call locks the entire object, can become a sequential bottleneck.

Four synchronization approaches for concurrent access to an object:

- ▶ **Fine-grained**: Split the object in *components* with their own locks.
- ▶ **Optimistic**: Search *without locks* for a certain component, lock it, *check* if it didn't change during the search, and only then adapt it.
- ▶ **Lazy**: Search without locks for a certain component, lock it, *check* if it isn't *marked*, mark it if needed, and only then adapt it.
- ▶ **Non-blocking**: Avoid locks, by *read-modify-write* operations.

List-based sets

We will show the four techniques on a running example: **sets**.

Consider a **linked list** in which each node has three fields:

- ▶ the actual **item** of interest
- ▶ the **key**, being the item's (unique) hash code
- ▶ **next** contains a reference to the next node in the list

Nodes in the list are sorted in key order.

There are sentinel nodes **head** and **tail**.

head and **tail** carry the *smallest* and *largest* key, respectively.

An implementation of sets based on lists (instead of e.g. trees) will of course never have a very good performance...

List-based sets: Methods

We define three methods:

- ▶ **add(x)**: Add x to the set;
return *true* only if x wasn't in the set.
- ▶ **remove(x)**: Remove x from the set;
return *true* only if x was in the set.
- ▶ **contains(x)**: Return *true* only if x is in the set.

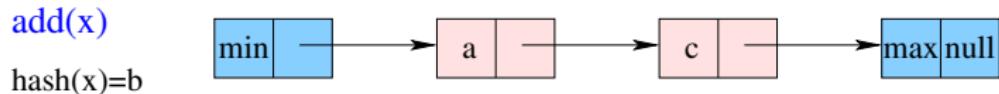
These methods should be *linearizable* in such a way that they act as on a sequential set (on a uniprocessor).

An **abstraction map** maps each *linked list* to the *set* of *items* that reside in a node reachable from *head*.

Coarse-grained synchronization

With **coarse-grained locking**, method calls lock the *entire* list.

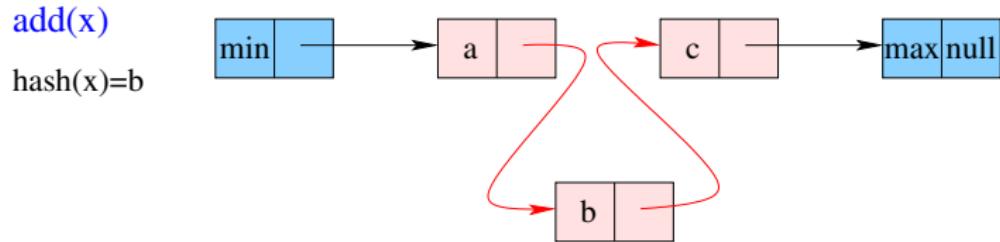
Then they search without contention whether x is in the list.



Coarse-grained synchronization

With **coarse-grained locking**, method calls lock the *entire* list.

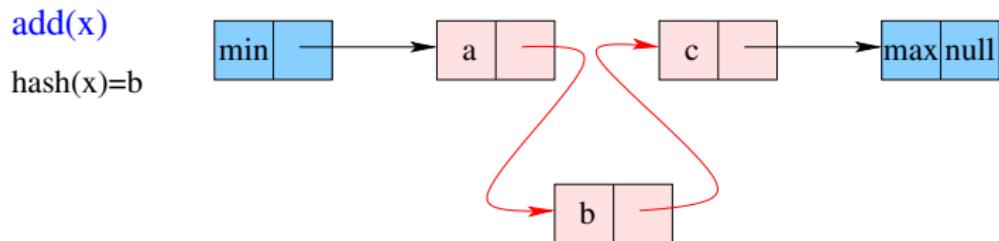
Then they search without contention whether x is in the list.



Coarse-grained synchronization

With **coarse-grained locking**, method calls lock the *entire* list.

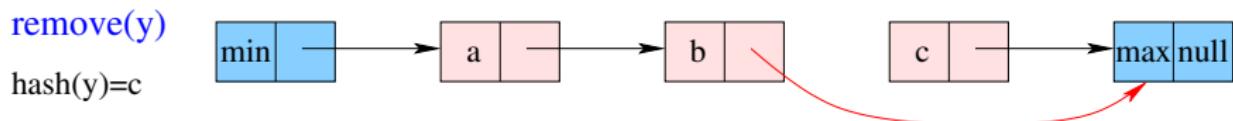
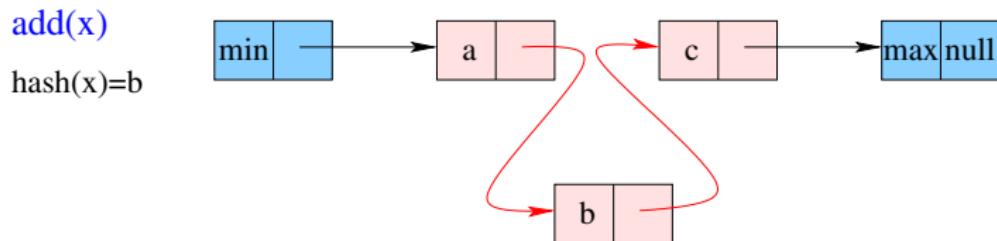
Then they search without contention whether x is in the list.



Coarse-grained synchronization

With **coarse-grained locking**, method calls lock the *entire* list.

Then they search without contention whether x is in the list.



List-based sets: Garbage collection

The implementations of list-based sets described here rely on a garbage collector to recycle memory properly.

Else our implementations would at some points need to be modified.



E.g., an unreachable node may still be traversed, in which case it should not yet be recycled.

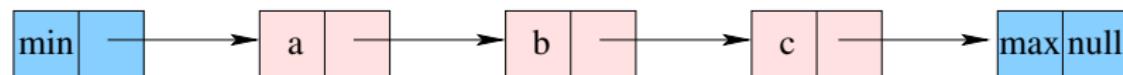
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a **hand-over-hand** fashion.

Example: A search for `c`.



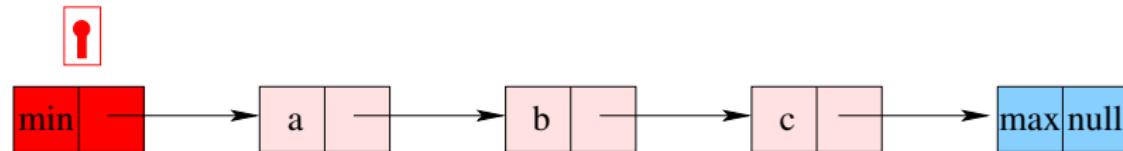
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a **hand-over-hand** fashion.

Example: A search for `c`.



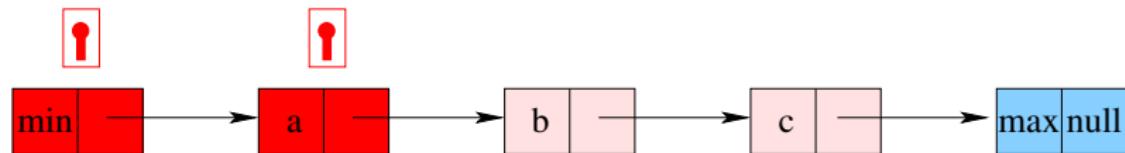
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a **hand-over-hand** fashion.

Example: A search for `c`.



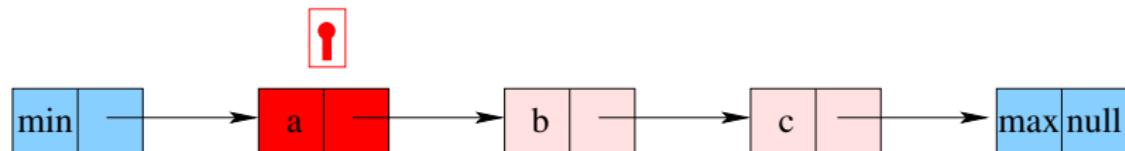
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a **hand-over-hand** fashion.

Example: A search for `c`.



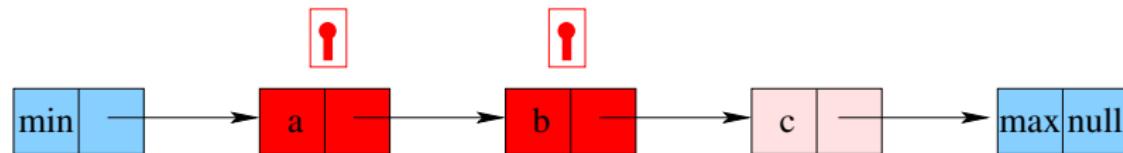
Fine-grained synchronization

Let each node carry its own lock.

`add(x)` and `remove(x)` require locks in ascending key order, until they find `x` or conclude it isn't present.

Threads acquire locks in a **hand-over-hand** fashion.

Example: A search for `c`.

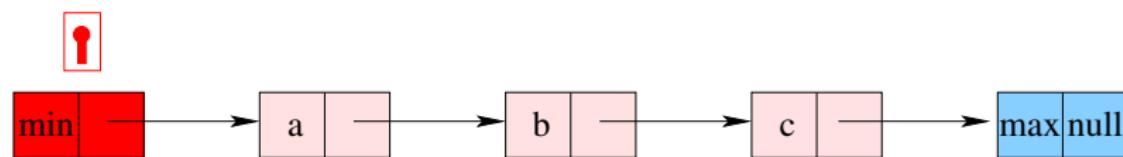


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.

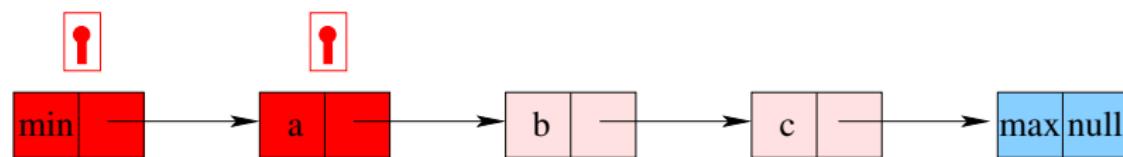


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.

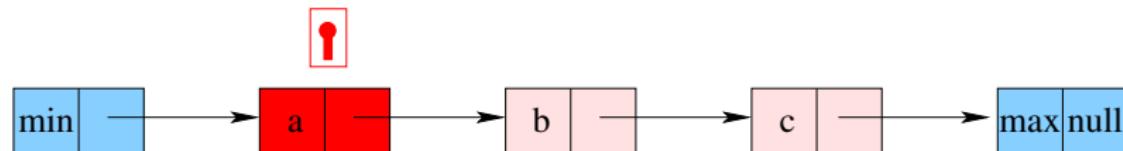


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.

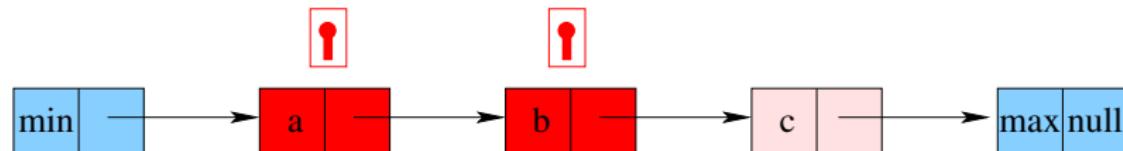


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.

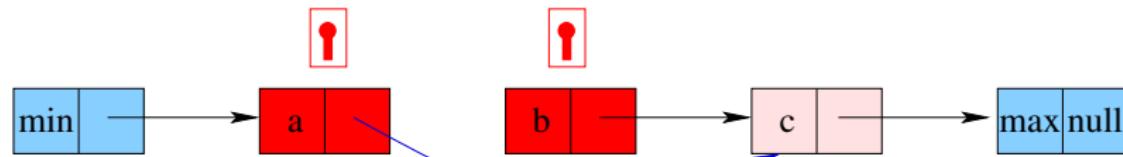


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

Example: We apply `remove(x)` with `hash(x)=b` to the list below.

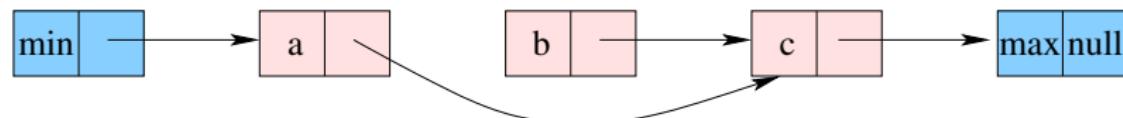


Fine-grained synchronization: Remove

`remove(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it removes this node by redirecting the link of
the **predecessor** to the **successor** of this node,
returns *true*, and releases the locks.
- ▶ or it locks a node with a key *greater than* `hash(x)`;
then it concludes that `x` isn't in the list, returns *false*,
and releases the locks.

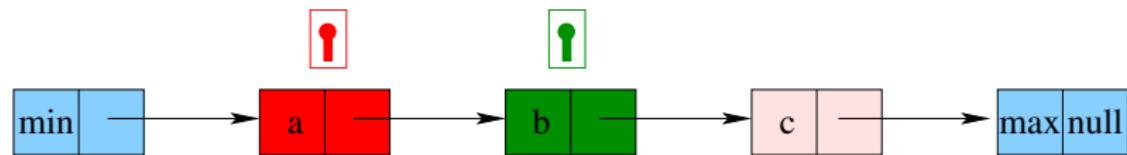
Example: We apply `remove(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: Two locks are needed

If threads would hold only *one* lock (instead of two), this algorithm would be incorrect.

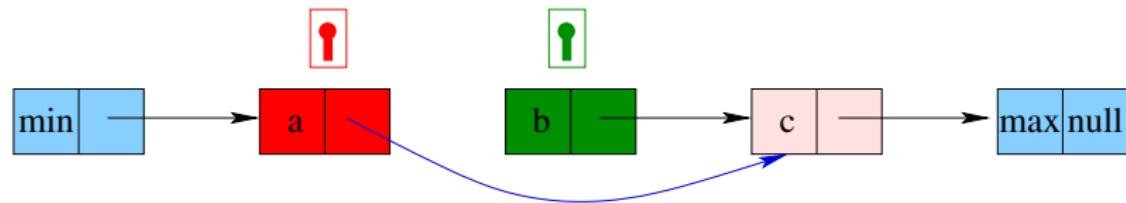
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: Two locks are needed

If threads would hold only *one* lock (instead of two), this algorithm would be incorrect.

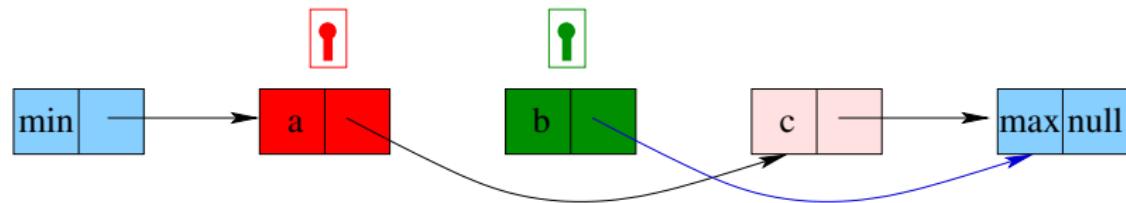
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization: Two locks are needed

If threads would hold only *one* lock (instead of two), this algorithm would be incorrect.

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.

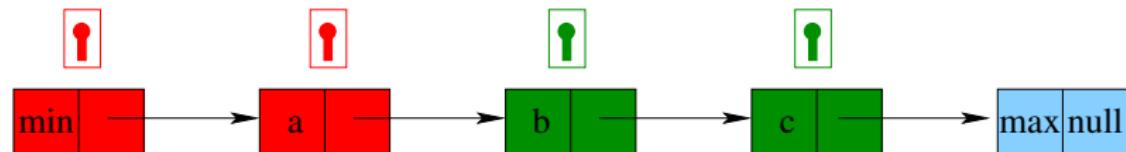


Node c isn't removed !

Fine-grained synchronization

Since threads are required to hold *two locks* at a time, this problem doesn't occur.

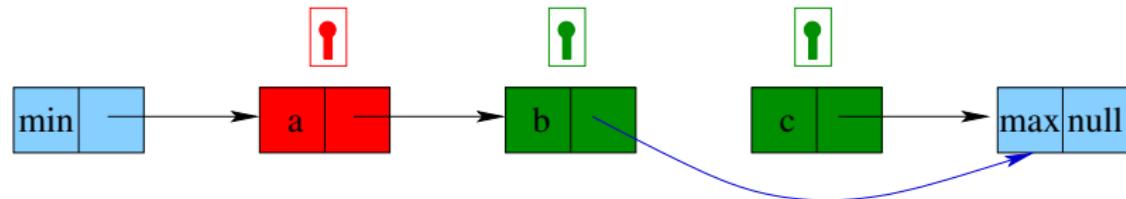
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

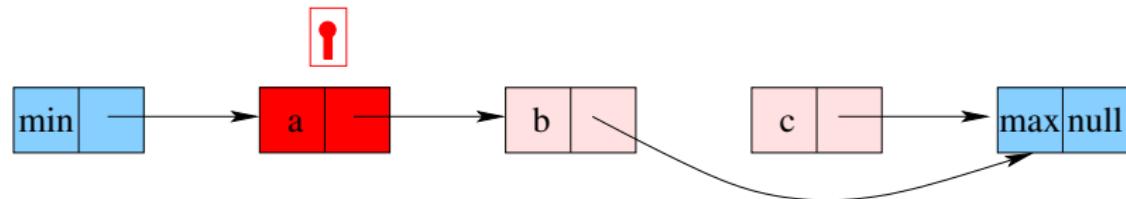
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

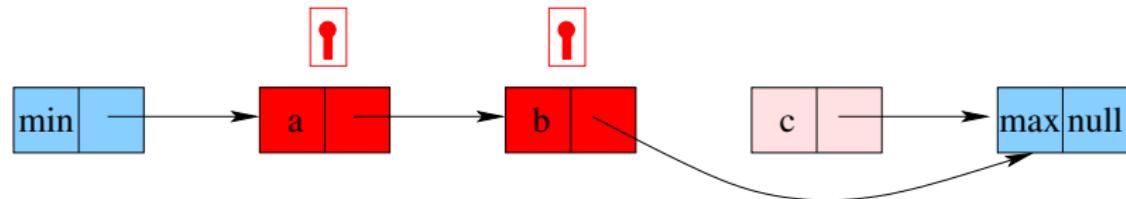
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two* locks at a time, this problem doesn't occur.

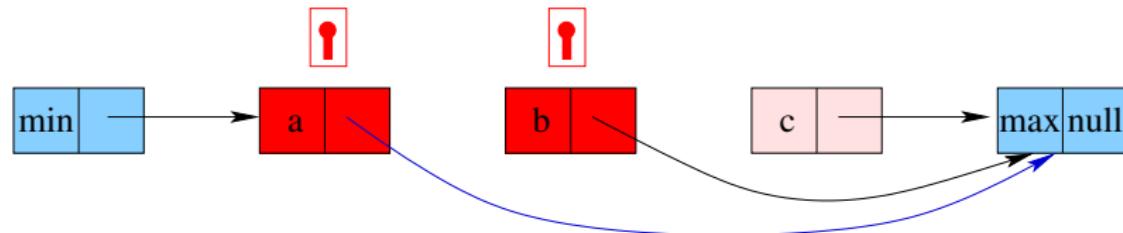
Example: Let two threads concurrently apply `remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.



Fine-grained synchronization

Since threads are required to hold *two locks* at a time,
this problem doesn't occur.

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=b`, and `remove(y)` with `hash(y)=c`.

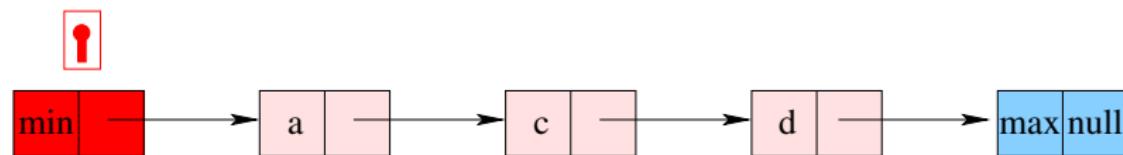


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node with key = `hash(x)` and `next = c`, returns *true*, and releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.

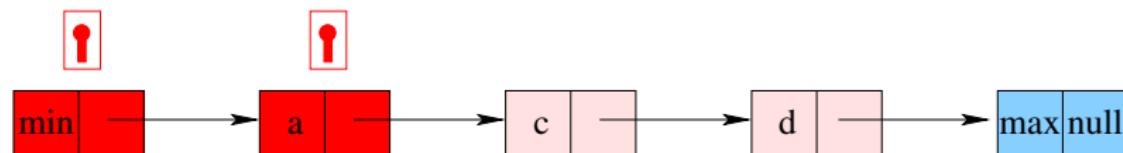


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node with key = `hash(x)` and `next = c`, returns *true*, and releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.

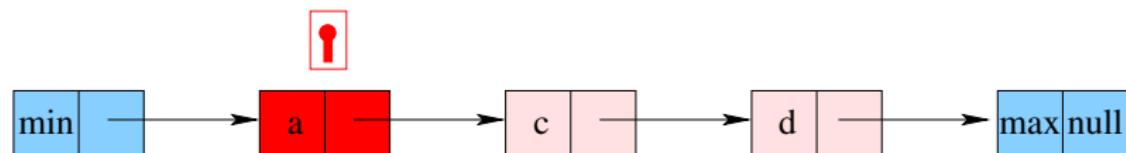


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node with key = `hash(x)` and `next = c`, returns *true*, and releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.

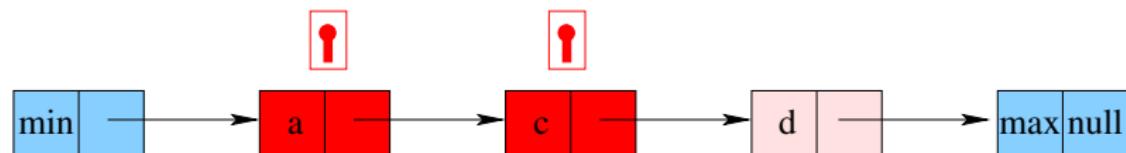


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node with key = `hash(x)` and `next = c`, returns *true*, and releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.

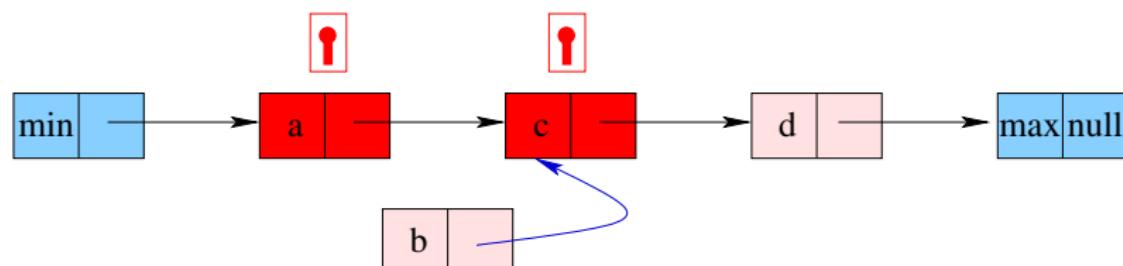


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node
with key = `hash(x)` and `next = c`, returns *true*, and
releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.

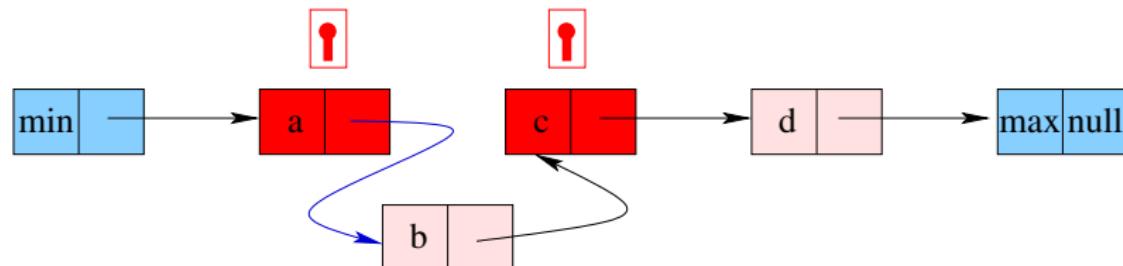


Fine-grained synchronization: Add

`add(x)` continues until:

- ▶ either it locks a node with key `hash(x)`;
then it concludes that x is in the list, returns *false*, and releases the locks.
- ▶ or it locks a node with a key $c > \text{hash}(x)$;
then it redirects the link of the **predecessor of c** to a new node
with key = `hash(x)` and `next = c`, returns *true*, and releases the locks.

Example: We apply `add(x)` with `hash(x)=b` to the list below.



Fine-grained synchronization: Correctness

To **remove** a node, this node and its predecessor must be locked.

So while a node is being removed, this node, its predecessor and its successor can't be removed.

And no nodes can be added between its predecessor and its successor.

Likewise, to **add** a node, this node's predecessor and successor must be locked.

So while a node is being added, its predecessor and successor can't be removed, and no other nodes can be added between them.

Linearization

Recall that a **linearization point** of a method call must be chosen between the moments the call is invoked and returns.

Method calls in an execution are ordered by their linearization points.

Thus the mayhem of concurrent method calls is made sequential.

An execution is linearizable if the method calls act correctly, assuming they were executed by a single thread.

Fine-grained synchronization: Linearization

We say an add or remove call is **successful** if it returns *true*.
Else it is called **unsuccessful**.

The **linearization points** of add and remove:

- ▶ **successful add**: When the predecessor is redirected to the added node.
- ▶ **successful remove**: When the predecessor is redirected to the successor of the removed node.
- ▶ **unsuccessful add and remove**: When it is detected that the call is unsuccessful.

Questions

Question 1: Let a successful `remove` be linearized as explained on the previous slide.

Give an example to show that linearizing an unsuccessful `remove` at the moment it acquires the lock of head would be wrong.

Question 2: Could all method calls be linearized at the moment they acquire the lock of head ?

Answer: Yes !

Fine-grained locking produces a rather sequential implementation...

Fine-grained synchronization: Progress property

The fine-grained synchronization algorithm is **deadlock-free**:

Always the thread holding the “furthest” lock can progress.

The algorithm can be made **starvation-free**, by using for instance the bakery algorithm to ensure that any thread can eventually get the lock of head.

(Since calls can't overtake each other, for performance, calls on large keys should be scheduled first.)

Fine-grained synchronization: Evaluation

Fine-grained synchronization allows threads to traverse the list in parallel.

However, it requires a chain of acquiring and releasing locks, which can be expensive.

And the result is still a rather sequential implementation.

Optimistic synchronization

In **optimistic synchronization**, **add** and **remove** proceed as follows:

- ▶ Search *without locks* for a pair of nodes on which the method call can be performed, or turns out to be unsuccessful.
- ▶ Lock these nodes (*always predecessor before successor*).
- ▶ Check whether the locked nodes are “correct”: meaning that the first locked node
 - ▶ is still reachable from `head`, and
 - ▶ points to the second locked node.
- ▶ If this validation fails, then release the locks and start over.
- ▶ Else, proceed as in fine-grained synchronization.

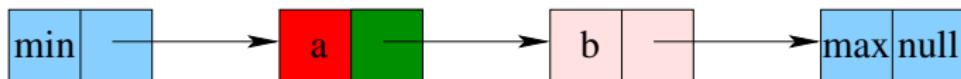
Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



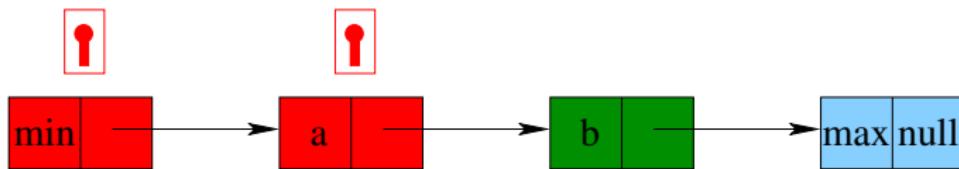
Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply
`remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



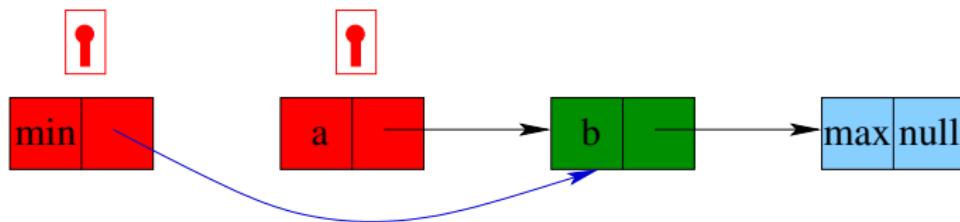
Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



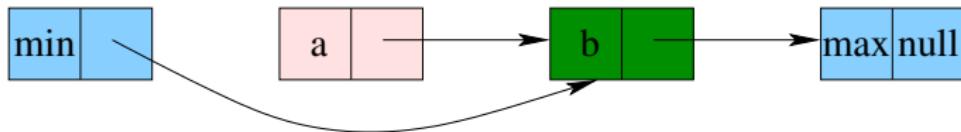
Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



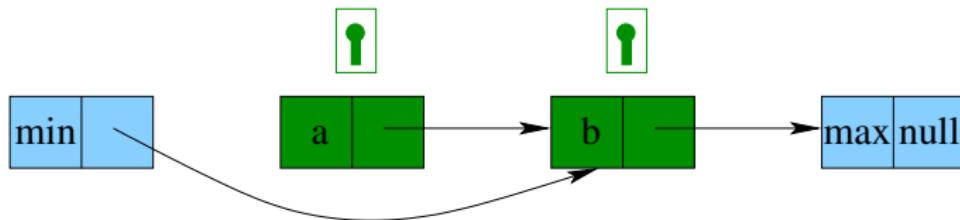
Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Optimistic synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Validation shows that node a isn't reachable from head.

Questions

Give an example to show that it must be checked that the first locked node points to the second locked node.

Give an example to show that validation is needed for the add method.

Optimistic synchronization: Linearization

The **linearization points** of add and remove:

- ▶ **successful add:** When the predecessor is redirected to the added node.
- ▶ **successful remove:** When the predecessor is redirected to the successor of the removed node.
- ▶ **unsuccessful add and remove:** *When validation succeeds (but the call itself is unsuccessful).*

Optimistic synchronization: Progress property

The optimistic synchronization algorithm is **deadlock-free**:

If a validation fails, another thread successfully completed an add or remove.

It is **not starvation-free**:

Validation by a thread may fail an infinite number of times.

Optimistic synchronization: Evaluation

Optimistic synchronization in general requires less locking than fine-grained synchronization.

However, each method call traverses the list at least twice.

Question: How can validation be simplified ?

Lazy synchronization

A bit is added to each node.



If a reachable node has bit 1, it has been *logically* removed
(and will be physically removed).

Lazy synchronization: Remove

In **lazy synchronization**, `remove(x)` proceeds as follows:

- ▶ Search (without locks) for a node c with a key $\geq \text{hash}(x)$.
- ▶ Lock its predecessor p and c itself.
- ▶ Check whether p (1) **isn't marked**, and (2) points to c .
- ▶ If this validation fails, then release the locks and start over.
- ▶ Else, if the key of c is greater than $\text{hash}(x)$, return *false*.

If the key of c equals $\text{hash}(x)$:

- ▶ **mark c ,**
- ▶ redirect p to the successor of c , and
- ▶ return *true*.

Release the locks.

Lazy synchronization: Add

`add(x)` proceeds similarly:

- ▶ Search for a node c with a key $\geq \text{hash}(x)$.
- ▶ Lock its predecessor p and c itself.
- ▶ Check whether p (1) **isn't marked**, and (2) points to c .
- ▶ If this validation fails, then release the locks and start over.
- ▶ Else, if the key of c equals $\text{hash}(x)$, return *false*.

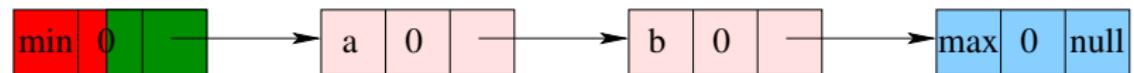
If the key of c is greater than $\text{hash}(x)$:

- ▶ create a node n with key $\text{hash}(x)$, value x , **bit 0**, and link to c ,
- ▶ redirect p to n , and
- ▶ return *true*.

Release the locks.

Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



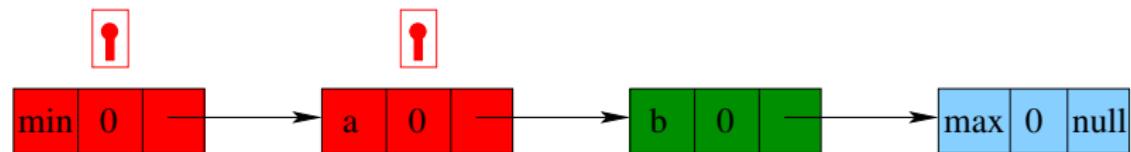
Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



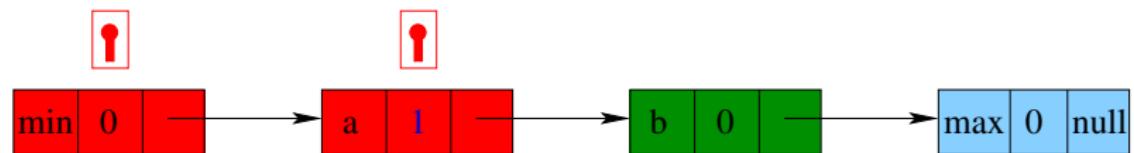
Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



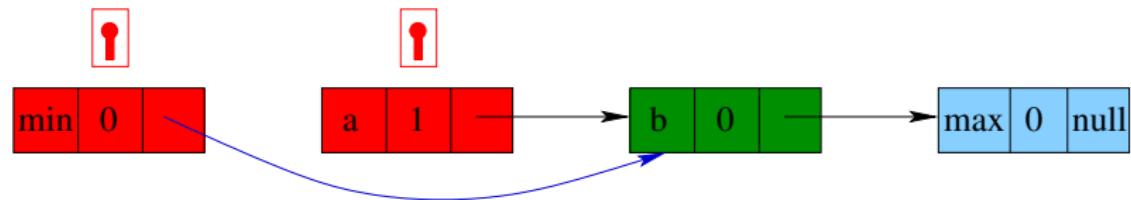
Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



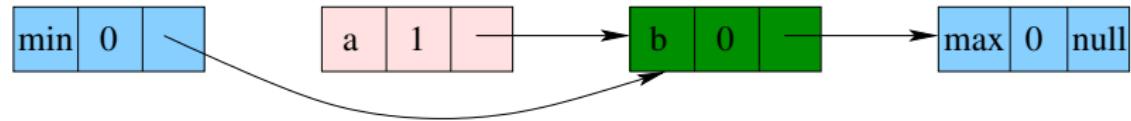
Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



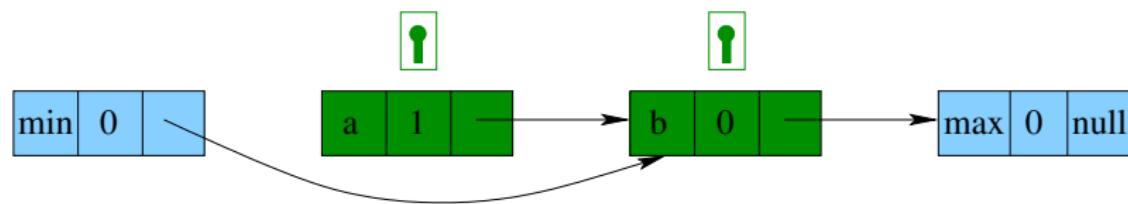
Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Lazy synchronization: Validation is needed

Example: Let two threads concurrently apply `remove(x)` with `hash(x)=a`, and `remove(y)` with `hash(y)=b`.



Validation shows that node a is marked for removal.

Lazy synchronization: Contains



`contains(x)` doesn't require locks:

- ▶ Search for a node with the key $\text{hash}(x)$.
- ▶ If no such node is found, return *false*.
- ▶ If such a node is found, check whether it is marked.
- ▶ If so, return *false*, else return *true*.

Lazy synchronization: Linearization

The **abstraction map** maps each *linked list* to the set of items that reside in an **unmarked** node reachable from **head**.

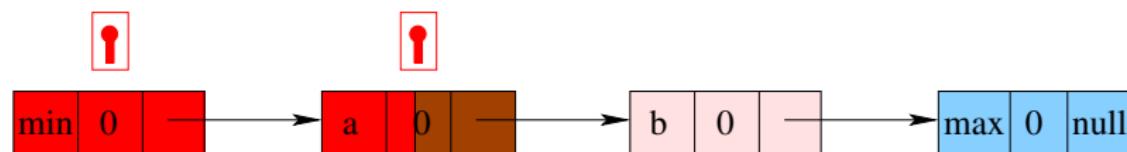
The **linearization points**:

- ▶ **successful add**: When the predecessor is redirected to the added node.
- ▶ **successful remove**: *When the mark is set.*
- ▶ **unsuccessful add and remove**: When validation succeeds.
- ▶ **successful contains**: When the (unmarked) node is found.
- ▶ **unsuccessful contains**: ???

Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

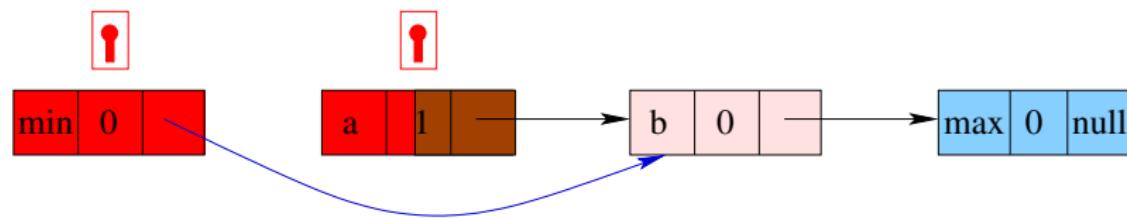
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

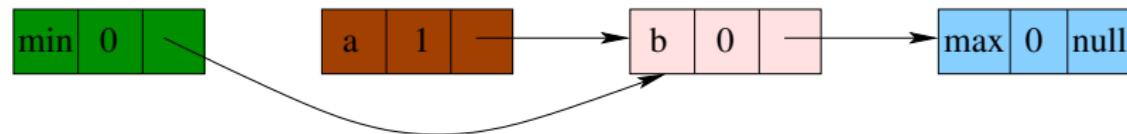
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

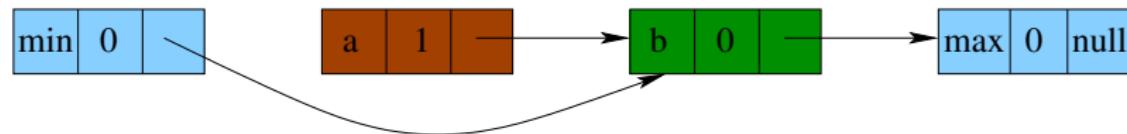
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

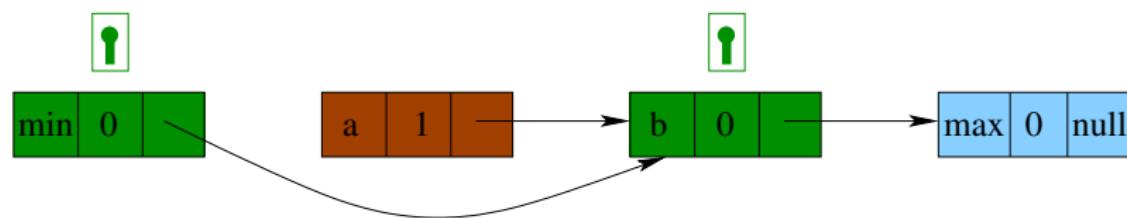
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

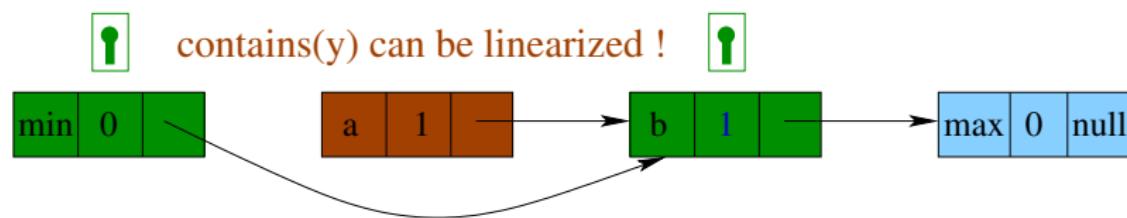
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

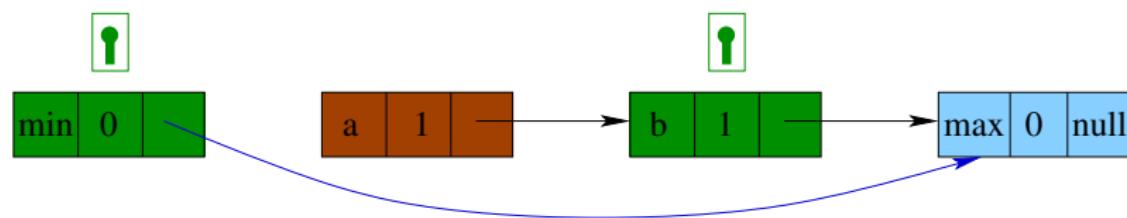
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

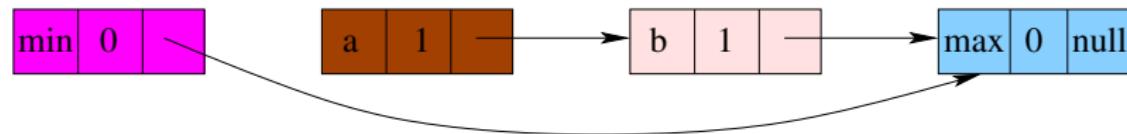
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

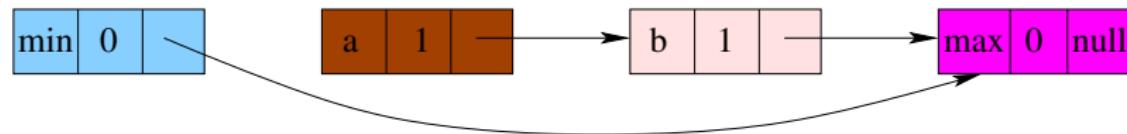
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

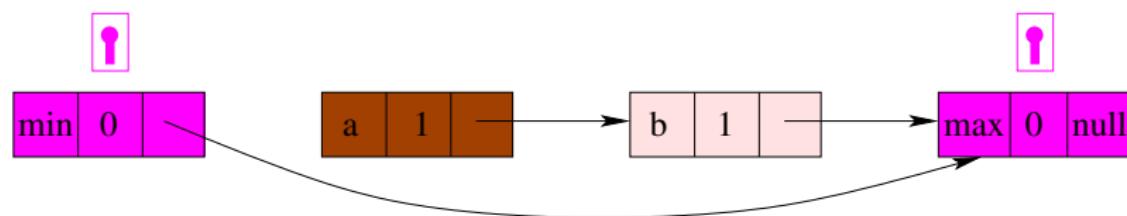
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

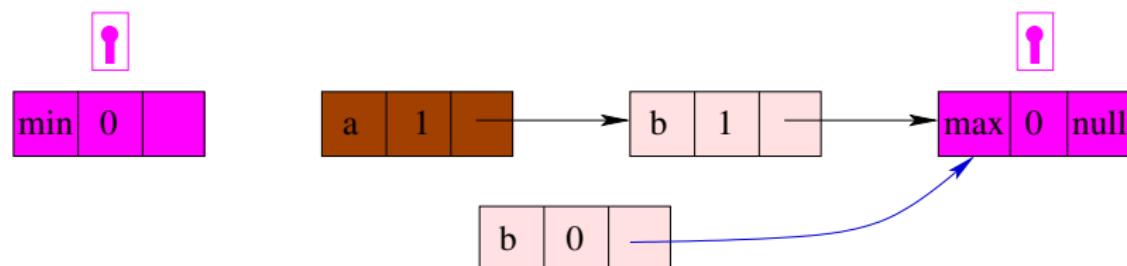
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

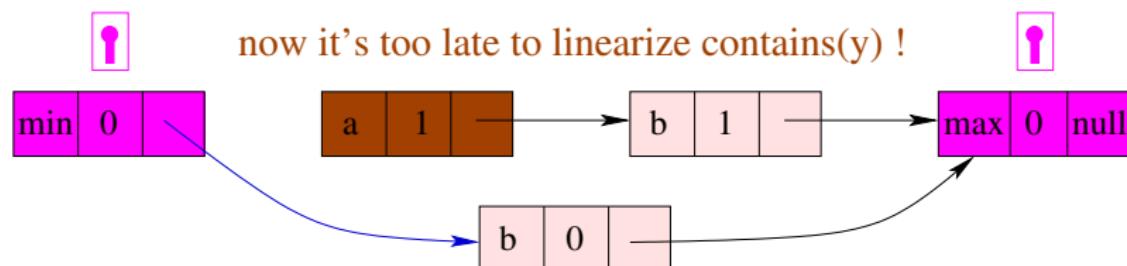
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

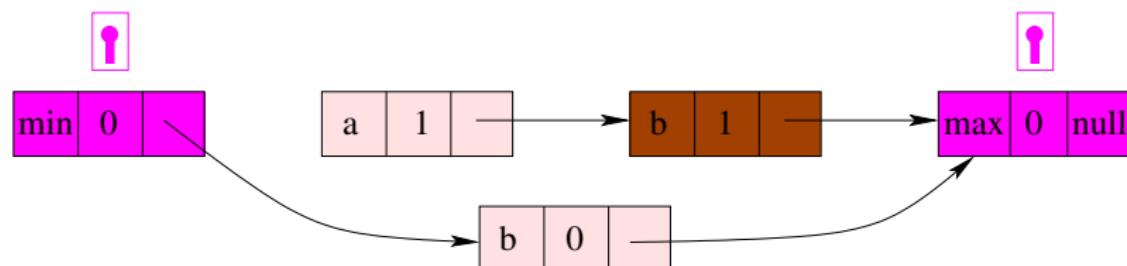
- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

Example: Four methods are applied concurrently:

- ▶ `remove(x)` with `hash(x)=a` and `contains(y)` with `hash(y)=b` are being executed
- ▶ `remove(y)` and `add(y)` are about to be invoked



Lazy synchronization: Linearizing unsuccessful contains

An unsuccessful `contains(x)` can in each execution be linearized at a moment when x isn't in the set.

- ▶ If x isn't present in the set at the moment `contains(x)` is invoked, then we linearize `contains(x)` when it is invoked.
- ▶ Else, a `remove(x)` has its linearization point between the moments when `contains(x)` is invoked and returns.

We linearize `contains(x)` right after the linearization point of such a `remove(x)`.

Lazy synchronization: Progress property

The lazy synchronization algorithm is **not starvation-free**:

Validation of add and remove by a thread may fail an infinite number of times.

However, contains is **wait-free**.

Drawbacks:

- ▶ contended add and remove calls retraverse the list
- ▶ add and remove are still blocking

Lock-free synchronization

We now discuss a *lock-free* implementation of sets.

Again nodes are supplied with a **bit** to mark removed nodes.

`compareAndSet` treats the link and mark of a node as one unit.

For this purpose it employs the **AtomicMarkableReference** class.

`remove()` tries to mark a node `c` with `compareAndSet(s, s, 0, 1)`, where `s` is the successor of `c`.

If successful, it physically removes `c` (again with a `compareAndSet`).

`add()` applies `compareAndSet(c, s, 0, 0)` to the predecessor `p` of the added node `s`, where `c` is the current successor of `p`.

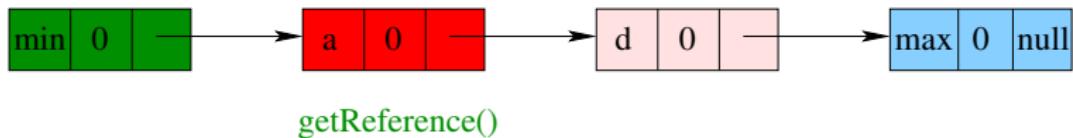
Lock-free synchronization: Example

`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



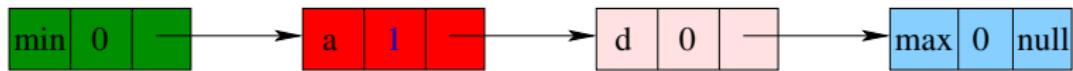
Lock-free synchronization: Example

`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



Lock-free synchronization: Example

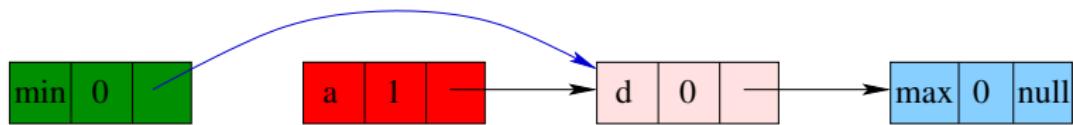
`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`compareAndSet(d,d,0,1)`

Lock-free synchronization: Example

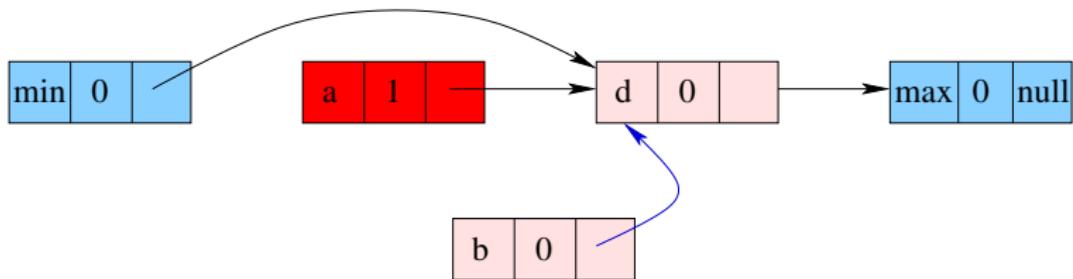
`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`compareAndSet(a,d,0,0)`

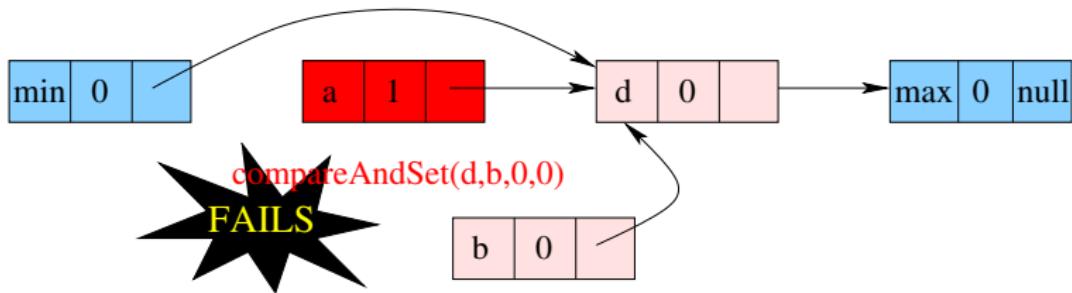
Lock-free synchronization: Example

`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



Lock-free synchronization: Example

`add(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`add(x)` must start over!

AtomicMarkableReference class

`AtomicMarkableReference<T>` maintains:

- ▶ an object reference of type T, and
- ▶ a Boolean mark bit.

An internal object is created, representing a boxed (reference, bit) pair.

These two fields can be updated in one atomic step.

AtomicMarkableReference class: Methods

`boolean compareAndSet(T expectedRef, T newRef,
 boolean expectedMark, boolean newMark)`

Atomically sets reference and mark to newRef and newMark, if reference and mark equal expectedRef and expectedMark.

`boolean attemptMark(T expectedRef, boolean newMark)`

Atomically sets mark to newMark, if reference equals expectedRef.

`void set(T newRef, boolean newMark)`

Atomically sets reference and mark to newRef and newMark.

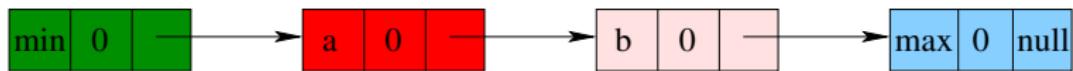
`T get(boolean[] currentMark)` Atomically returns the value of reference and writes the value of mark at place 0 of the argument array.

`T getReference()` Returns the value of reference.

`boolean isMarked()` Returns the value of mark.

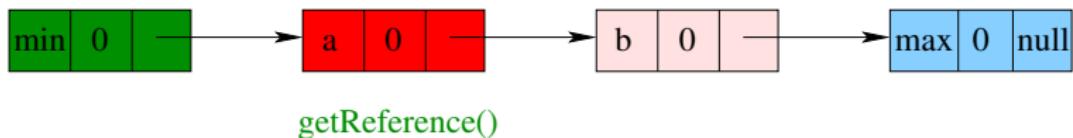
Lock-free synchronization: Example

`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



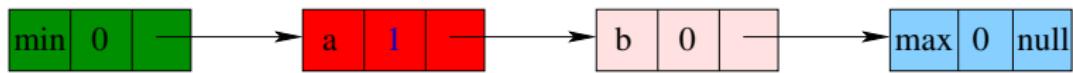
Lock-free synchronization: Example

`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



Lock-free synchronization: Example

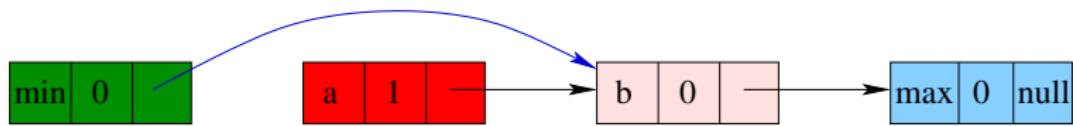
`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`compareAndSet(b,b,0,1)`

Lock-free synchronization: Example

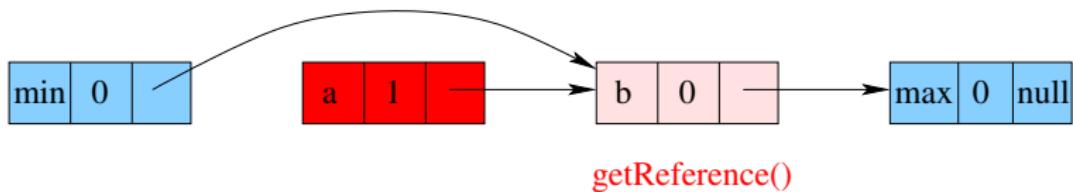
`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`compareAndSet(a,b,0,0)`

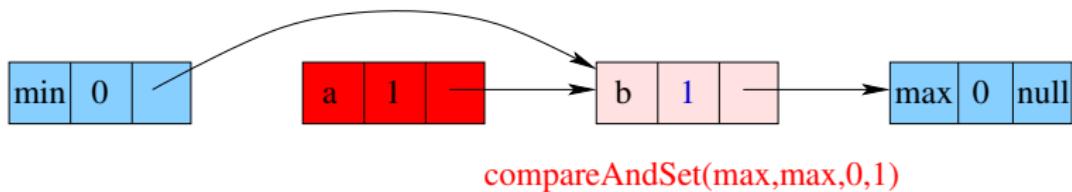
Lock-free synchronization: Example

`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



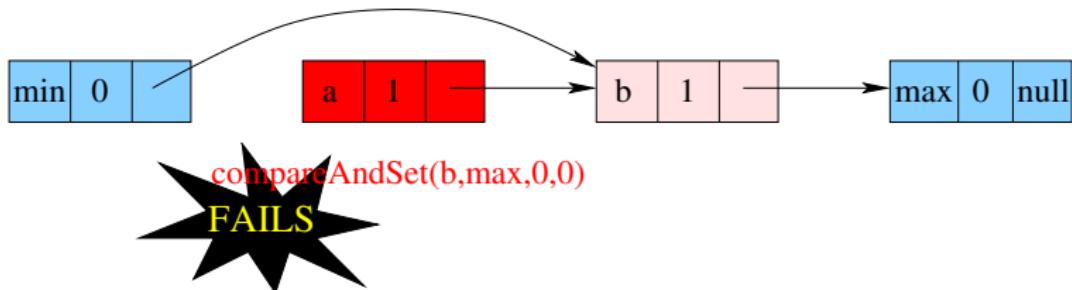
Lock-free synchronization: Example

`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



Lock-free synchronization: Example

`remove(x)` with `hash(x)=b` and `remove(y)` with `hash(y)=a`
are being executed.



`remove(x)` returns *true*, and leaves the physical removal of node b to another thread !!

To make the algorithm lock-free, threads must anyhow help to clean up logically removed nodes.

Lock-free synchronization: Physical removal

Suppose an add or remove call that traverses the list encounters a *marked* node `curr`.

Then it attempts to physically remove `curr`, by applying

`compareAndSet(curr, succ, 0, 0)`

to `curr`'s predecessor `pred`, to redirect it to `curr`'s successor `succ`.

If such an attempt *succeeds*, then the traversal continues at `succ`.

If such an attempt *fails*, then the method call must start over, because it may be traversing an unreachable part of the list.

Lock-free synchronization: Remove

`remove(x)` proceeds as follows:

- ▶ Search for a node c with a key $\geq \text{hash}(x)$ (*reference and mark of a node are read in one atomic step using `get()`.*)

- ▶ During this search, try to physically remove marked nodes, using `compareAndSet`.

If at some point such a physical removal fails, start over.

- ▶ If the key of c is greater than $\text{hash}(x)$, return *false*.

If the key of c equals $\text{hash}(x)$:

- ▶ Apply `getReference()` to obtain the successor s of c .
- ▶ Apply `compareAndSet(s, s, 0, 1)` to try and mark c .
- ▶ If this fails, start over.

Else, apply `compareAndSet(c, s, 0, 0)` to try and redirect the predecessor p of c to s , and return *true*.

Lock-free synchronization: Add

`add(x)` proceeds as follows:

- ▶ Search for a node c with a key $\geq \text{hash}(x)$.
- ▶ During this search, try to physically remove marked nodes, using `compareAndSet`.
If at some point such a physical removal fails, start over.
- ▶ If the key of c equals $\text{hash}(x)$, return *false*.
If the key of c is greater than $\text{hash}(x)$:
 - ▶ Create a node n with key $\text{hash}(x)$, value x , bit 0, and link to c .
 - ▶ Apply `compareAndSet(c, n, 0, 0)` to try and redirect the predecessor p of c to n .
 - ▶ If this fails, start over.
- Else, return *true*.

Lock-free synchronization: Contains

`contains(x)` traverses the list *without cleaning up marked nodes.*

- ▶ Search for a node with the key $\text{hash}(x)$.
- ▶ If no such node is found, return *false*.
- ▶ If such a node is found, check whether it is marked.
- ▶ If so, return *false*, else return *true*.

Lock-free synchronization: Linearization

The linearization points:

- ▶ **successful add**: When the predecessor is redirected to the added node.
- ▶ **successful remove**: When the mark is set.
- ▶ **unsuccessful add(x) and remove(x)**: When the key is found that is equal to, respectively greater than, $\text{hash}(x)$.
- ▶ **successful contains**: When the (unmarked) node is found.
- ▶ **unsuccessful contains(x)**: At a moment when x isn't in the set.

Lock-free synchronization: Progress property

The lock-free algorithm is **lock-free**.

It is **not wait-free**, because list traversal of add and remove by a thread may be unsuccessful an infinite number of times.

`contains` is **wait-free**.

The lock-free algorithm for sets is in the **Java Concurrency Package**.

This lecture in a nutshell

a **linked list** implementation of a **set**

fine-grained synchronization (with hand-over-hand locking)

optimistic synchronization (with validation)

lazy synchronization (with a marker bit)

lock-free synchronization

wait-free contains method

AtomicMarkableReference class

FIFO queue as a linked list

We implement a **FIFO queue** as a *linked list*.

Nodes contain an item and a reference to the next node in the list.

head points to the node of which *the item was dequeued last*.

tail points to the *last node in the list*.

These sentinel nodes initially point to the same dummy node.

We consider three implementations of FIFO queues:

- ▶ bounded, with fine-grained locks
- ▶ unbounded, with fine-grained locks
- ▶ unbounded, lock-free

Fine-grained bounded FIFO queue

Earlier we implemented a **coarse-grained** bounded FIFO queue using an array with locks and conditions.

Now we consider a **fine-grained** bounded FIFO queue.

Enqueuers and dequeuers work at different ends of the queue, so they can require different locks.

capacity denotes the maximum size of the queue.

An enqueue and a dequeue can concurrently write to size, so this is an `AtomicInteger` (i.e., we can apply read-modify-write operations).

Fine-grained bounded FIFO queue: Enqueue

```
public void enq(T x) {  
    boolean mustWakeDequeueuers = false;  
    enqLock.lock();  
    try {  
        while size.get() == capacity  
            notFullCondition.await();  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;  
        if size.getAndIncrement() == 0  
            mustWakeDequeueuers = true;  
    } finally {  
        enqLock.unlock();  
    }  
}
```

Fine-grained bounded FIFO queue: Enqueue

```
if mustWakeDequeueuers {  
    deqLock.lock();  
    try {  
        notEmptyCondition.signalAll();  
    } finally {  
        deqLock.unlock();  
    }  
}  
}
```

Question: Why must we use signalAll instead of signal?

Fine-grained bounded FIFO queue: Dequeue

```
public T deq() {
    boolean mustWakeEnqueuers = false;
    deqLock.lock();
    try {
        while size.get() == 0
            notEmptyCondition.await();
        T y = head.next.value;
        head = head.next;
        if size.getAndDecrement() == capacity
            mustWakeEnqueuers = true;
    } finally {
        deqLock.unlock();
    }
}
```

Fine-grained bounded FIFO queue: Dequeue

```
if mustWakeEnqueuers {  
    enqLock.lock();  
    try {  
        notFullCondition.signalAll();  
    } finally {  
        enqLock.unlock();  
    }  
}  
return y;  
}
```

Questions

Question: Why does head not point to the node the head of the queue ?

Answer: This would complicate the case that the queue becomes empty.

Question: What could go wrong if a dequeuer wouldn't acquire the enqueue lock before signalling notFullCondition ?

(Actually, since notFullCondition is associated to the enqueue lock, this signal can only be sent by a thread holding this lock.)

Fine-grained bounded FIFO queue: No lost wakeups

Answer: A *lost wakeup* could occur if:

- ▶ First an enqueuer reads `size.get() == capacity`.
- ▶ Next a dequeuer performs `size.getAndDecrement()` and calls `notFullCondition.signalAll()`.
- ▶ Next the enqueuer calls `notEmptyCondition.await()`.

This scenario can't occur because a dequeuer must acquire the enqueue lock before signalling `notFullCondition`.

Similarly, the fact that an enqueuer must acquire the dequeue lock before signalling `notEmptyCondition` avoids lost wakeups.

Fine-grained bounded FIFO queue: Linearization

The **abstraction map** maps each *linked list* to the *queue* of items that reside in a node reachable from, but not equal to, head.

(It ignores tail.)

Linearization points:

- ▶ of an `enq()` at `tail.next = e`
- ▶ of a `deq()` at `head = head.next`

Fine-grained unbounded FIFO queue: Enqueue

We now consider an *unbounded* FIFO queue, again with different locks for enqueueers and dequeuers.

We don't employ conditions (and size).

Instead, `deq()` throws an exception in case of an empty queue.

```
public void enq(T x) {  
    enqLock.lock();  
    try {  
        Node e = new Node(x);  
        tail.next = e;  
        tail = e;  
    } finally {  
        enqLock.unlock();  
    }  
}
```

Fine-grained unbounded FIFO queue: Dequeue

```
public T deq() throws EmptyException {
    deqLock.lock();
    try {
        if head.next == null {
            throw new EmptyException();
        }
        T y = head.next.value;
        head = head.next;
    } finally {
        deqLock.unlock();
    }
    return y;
}
```

Fine-grained unbounded FIFO queue: Linearization

Linearization points:

- ▶ of an `enq()` at `tail.next = e`
- ▶ of a `deq()`

on a *nonempty* queue, at `head = head.next`

on an *empty* queue, when `head.next == null` returns *true*

Question

Why is an unsuccessful `deq()` not linearized at `EmptyException()`?

Answer: An `enq()` could be invoked and return between the moments a `deq()` reads `head.next == null` and throws `EmptyException()`.

Lock-free unbounded FIFO queue

We now consider a *lock-free* unbounded FIFO queue.

`enq(x)`

- ▶ builds a new node containing x
- ▶ tries to add the node at the end of the list with `compareAndSet()` (if this fails, retry)
- ▶ tries to advance tail with `compareAndSet()`

`deq()`

- ▶ gets the item in the first node of the list (if it is nonempty)
- ▶ tries to advance head with `compareAndSet()` (if this fails, retry)
- ▶ returns the element

Lock-free unbounded FIFO queue

If an enqueuer, after adding its node, would be solely responsible for advancing tail, then the queue wouldn't be lock-free.

Because the enqueuer might crash between adding its node and advancing tail.

Solution: Enqueuers check whether tail is pointing to the last node, and if not, try to advance it with `compareAndSet()`.

(If an enqueuer has added its node but fails to advance tail, then another thread has already advanced tail.)

Lock-free unbounded FIFO queue: Enqueue

```
public void enq(T x) {  
    Node node = new Node(x);  
    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);  
        }  
    }  
}
```

create new node
retry until enqueue succeeds
is tail the last node?
try add node
try advance tail
try advance tail

Questions

Question: Why can next not be defined as `tail.next.get()` ?

Question: What would be the drawback of omitting the next field (and the check “if `next == null`”) as follows ?

```
if last.next.compareAndSet(null, node)
    tail.compareAndSet(last, node);
    return;
tail.compareAndSet(last, last.next);
```

Answer: A possibly needless application of `compareAndSet()`.

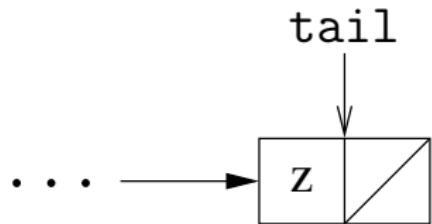
Lock-free unbounded FIFO queue: Optimization

Herlihy and Shavit add a check “if last == tail.get()”:

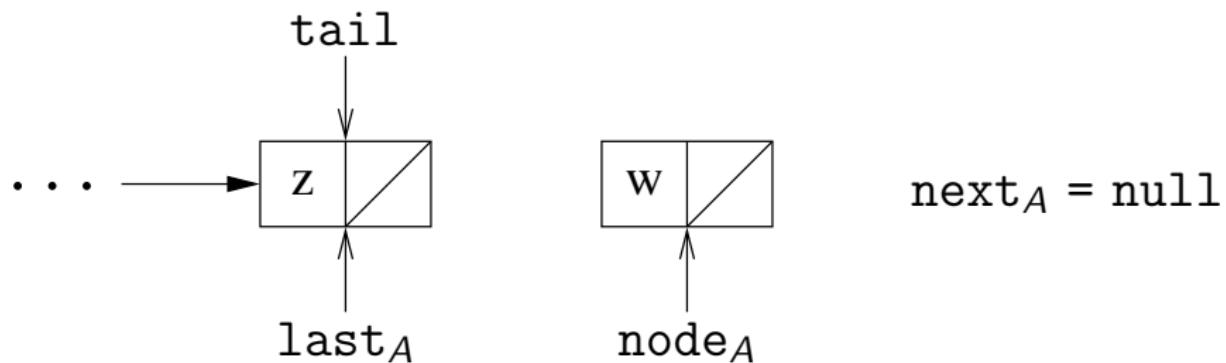
```
Node last = tail.get();
Node next = last.next.get();
if last == tail.get()
```

This avoids a needless compareAndSet operation in case tail was advanced (by another thread) after last = tail.get().

Lock-free unbounded FIFO queue: Example

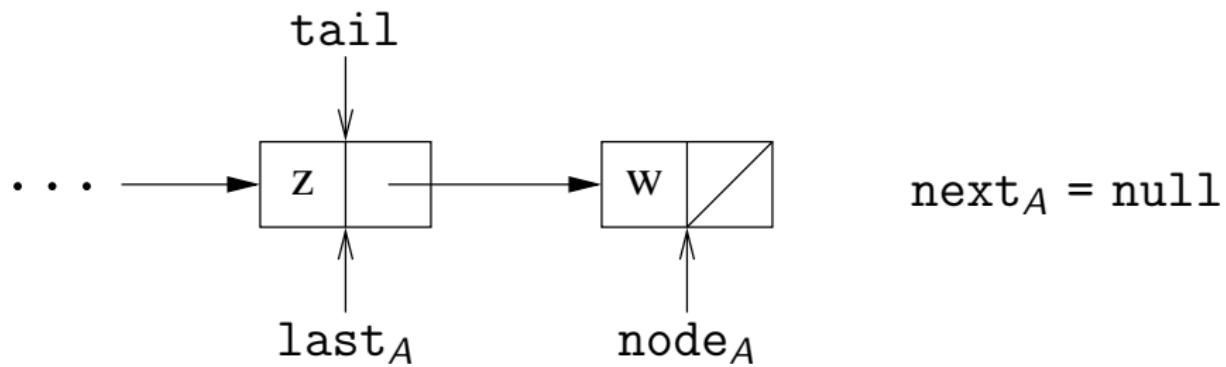


Lock-free unbounded FIFO queue: Example

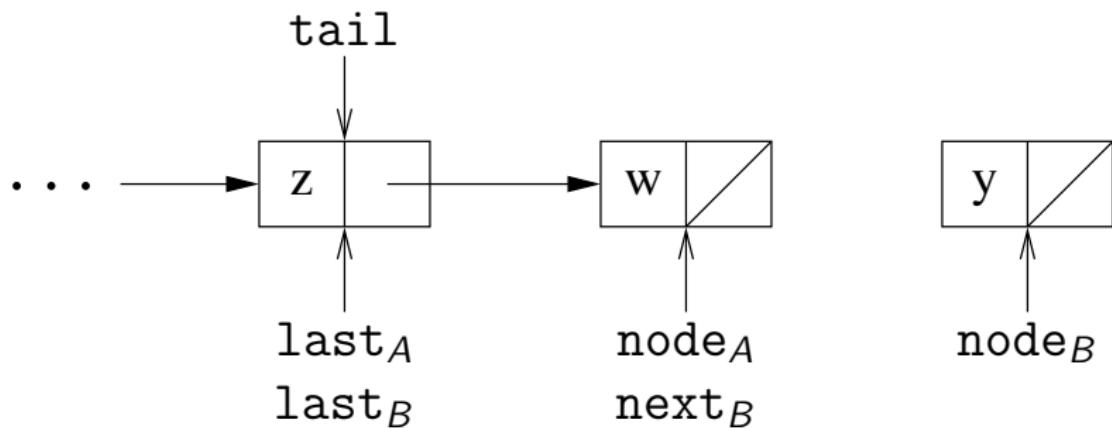


Lock-free unbounded FIFO queue: Example

`lastA.next.compareAndSet(null, nodeA)`

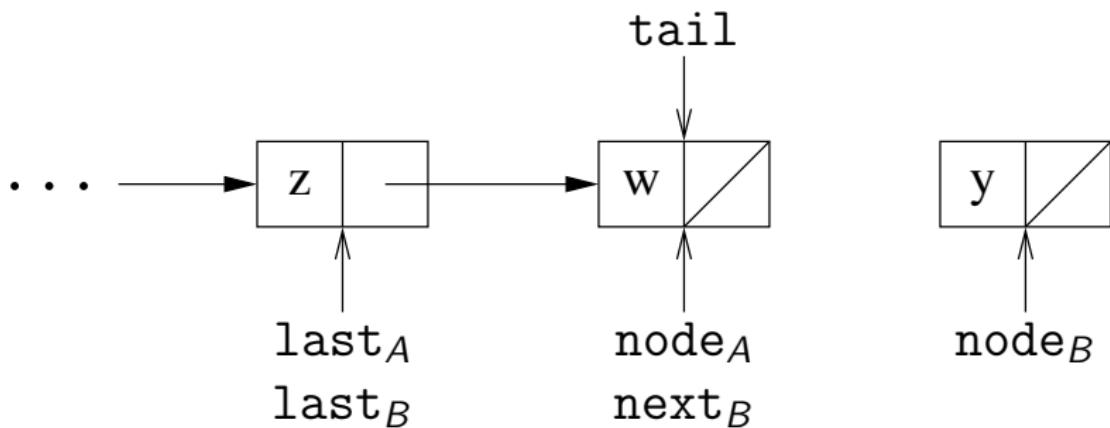


Lock-free unbounded FIFO queue: Example



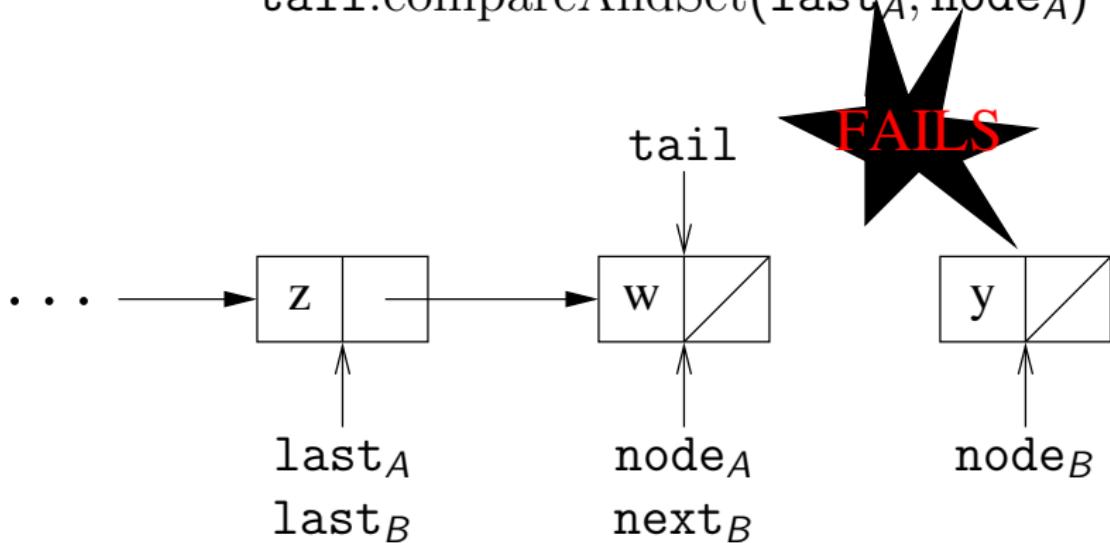
Lock-free unbounded FIFO queue: Example

`tail.compareAndSet(lastB, nextB)`



Lock-free unbounded FIFO queue: Example

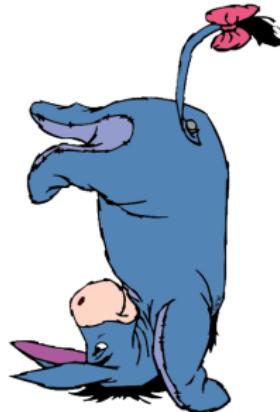
tail.compareAndSet(last_A , node_A)



Lock-free unbounded FIFO queue

While tail is waiting to be advanced, head could overtake tail.

This disallows a dequeuer to free dequeued nodes (which is important for a C implementation).



Solution: If a dequeuer finds that

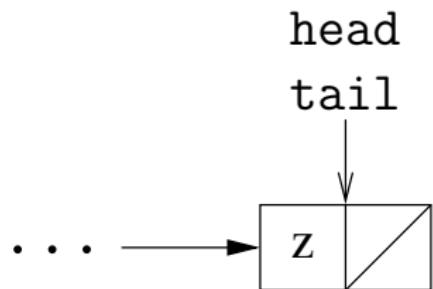
- ▶ head is equal to tail, and
- ▶ the queue is nonempty,

then it tries to advance tail.

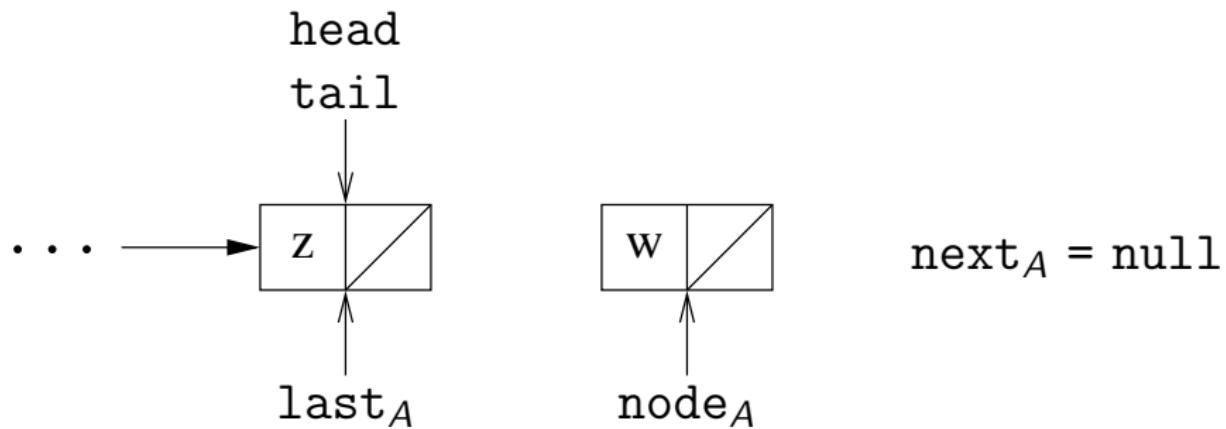
Lock-free unbounded FIFO queue: Dequeue

```
public T deq() throws EmptyException {
    while true {                                retry until dequeue succeeds
        Node first = head.get();
        Node last = tail.get();
        Node next = first.next.get();
        if first == last {                      is the queue empty?
            if next == null {                  is next the last node?
                throw new EmptyException();
            }
            tail.compareAndSet(last, next);      try advance tail
        } else {
            T y = next.value;                 read output value
            if head.compareAndSet(first, next)  try advance head
                return y;
        }
    }
}
```

Lock-free unbounded FIFO queue: Example

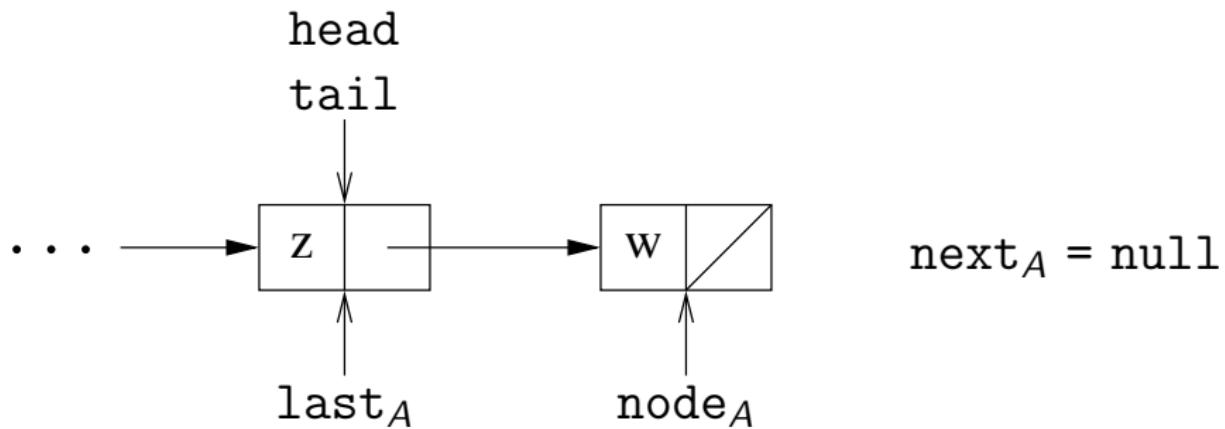


Lock-free unbounded FIFO queue: Example

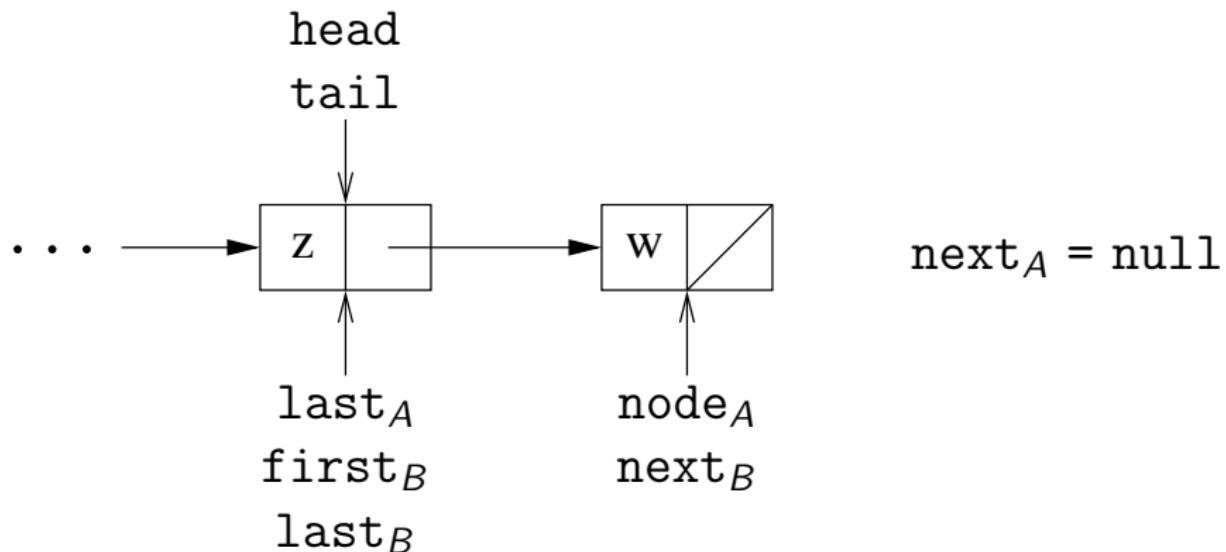


Lock-free unbounded FIFO queue: Example

$\text{last}_A.\text{next}.\text{compareAndSet}(\text{next}_A, \text{node}_A)$

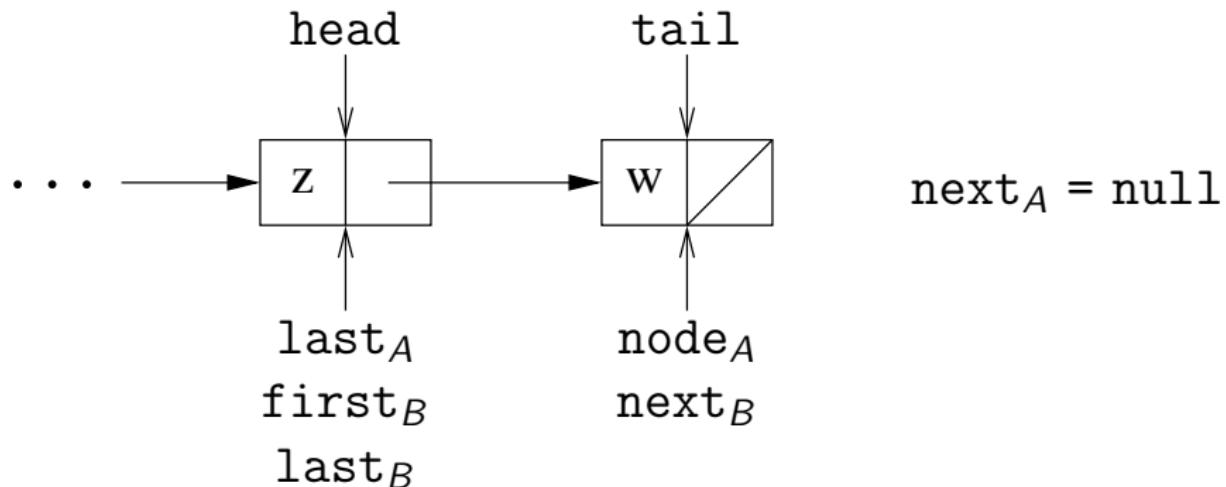


Lock-free unbounded FIFO queue: Example



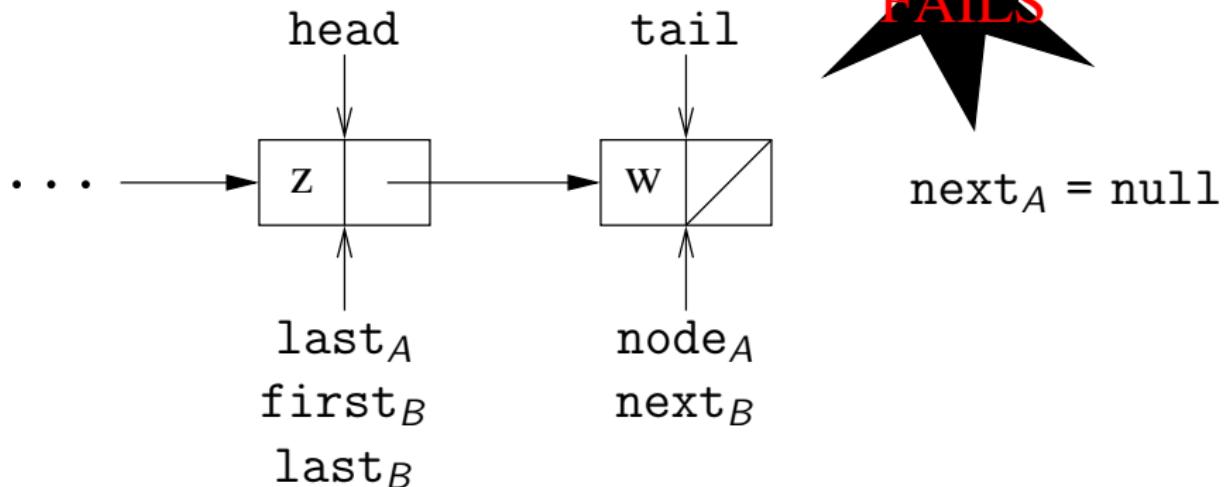
Lock-free unbounded FIFO queue: Example

`tail.compareAndSet(lastB, nextB)`



Lock-free unbounded FIFO queue: Example

`tail.compareAndSet(lastA, nodeA)`



Lock-free unbounded FIFO queue: Linearization

Linearization point of an `enq()`:

- ▶ when `last.next.compareAndSet(next, node)` returns *true*.

Linearization point of a `deq()`:

- ▶ *successful*: When `head.compareAndSet(first, next)` returns *true*.
- ▶ *unsuccessful*: At `next = first.next.get()` in the last try, when an empty exception is thrown.

Question

Why is an unsuccessful `deq()` not linearized at the moment
`if next == null` returns `true`?

Answer: An `enq()` could be invoked and return right after
the moment `deq()` reads the value `null` for `next`.

(Herlihy & Shavit incorrectly linearize it at `EmptyException()`.)

Lock-free unbounded FIFO queue: Evaluation

The unbounded FIFO queue is **lock-free**.

If tail doesn't point to the last node,
an `enq()` or `deq()` will advance it to the last node.

And if tail points to the last node,
an `enq()` or `deq()` only retries if a concurrent call succeeded.

The unbounded FIFO queue is **not wait-free**.

An `enq()` or `deq()` can fail infinitely often,
because of successful concurrent calls.

ABA problem

Often, the motivation for a `compareAndSet(a,c)` is to only write `c` if the value of the register never changed in the meantime (i.e., was always `a`).

However, it may be the case that in the meantime, the value of the register changed to `b`, and then back to `a`.

One could view this as a flaw in the semantics of `compareAndSet()` (blame it on Intel, Sun, AMD).

ABA problem for the lock-free unbounded queue

The programming language C doesn't come with garbage collection, and allows dangling pointers.

In a C implementation of the lock-free unbounded queue, the following scenario might occur:

- ▶ A dequeuer A observes `head` is at node a and the next node is c , and prepares to apply `compareAndSet(a, c)` to `head`.
- ▶ Other threads dequeue c and its successor d ; `head` points to d .
- ▶ Nodes a and c are picked up by the garbage collector.
- ▶ Node a is recycled, and eventually `head` points to a again.
- ▶ Finally dequeuer A applies `compareAndSet(a, c)` to `head`, which erroneously succeeds because `head` points to a .

ABA problem for the lock-free unbounded queue

The Java garbage collector tends to prevent the ABA problem for linked lists.

Also dequeuer *A* keeps a first pointer to the (old) head, so that it can't be reclaimed by the garbage collector.

But we will later see another example (with an array) where the ABA problem pops up in a Java implementation as well.

AtomicStampedReference class

To avoid the ABA problem, a reference can be tagged with an integer, which is increased by 1 at every update.

`AtomicStampedReference<T>` maintains:

- ▶ an object reference of type T, and
- ▶ an integer.

The paper that introduced the lock-free queue

Maged M. Michael & Michael L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, in Proc. PODC, ACM, 1996

describes a C implementation that pairs tail with a stamp.

AtomicStampedReference class: Methods

`boolean compareAndSet(T expectedRef, T newRef,
 int expectedStamp, int newStamp)`

Atomically sets reference and stamp to newRef and newStamp,
if reference and stamp equal expectedRef and expectedStamp.

`boolean attemptStamp(T expectedRef, int newStamp)`

Atomically sets stamp to newStamp, if reference equals expectedRef.

`void set(T newRef, int newStamp)`

Atomically sets reference and stamp to newRef and newStamp.

`T get(int[] currentStamp)` Atomically returns the value of reference
and writes the value of stamp at place 0 of the argument array.

`T getReference()` Returns the value of reference.

`int getStamp()` Returns the value of stamp.

Avoiding the ABA problem in other languages

`Load-Linked()` (read from an address) and `Store-Conditional()` (store a new value if in the meantime the value didn't change) from IBM avoid the ABA problem.

The penalty is that a `Store-Conditional()` may fail even if there is no real need (e.g. a context switch).

In C and C++, a pointer and an integer can be paired in a 64-bits architecture by stealing bits from pointers.

Intel's `CMPXCHG16B` instruction enables a `compareAndSet` on 128 bits.

Lock-free unbounded stack

We model a *stack* as a linked list.

`top` points to the top of the stack (initially it is `null`).

`push()` creates a new node ν , and calls `tryPush()` to

- ▶ set the `next` field of ν to the top of the stack, and
- ▶ try to swing the `top` reference to ν with `compareAndSet()`.

Every time this fails, `push()` retries (using exponential backoff).

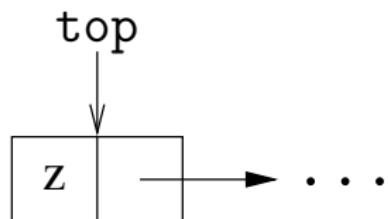
`pop()` calls `tryPop()` to try to swing `top` to the successor of `top` with `compareAndSet()`. (If the stack is empty, an exception is thrown.)

Every time this fails, `pop()` retries (using exponential backoff).

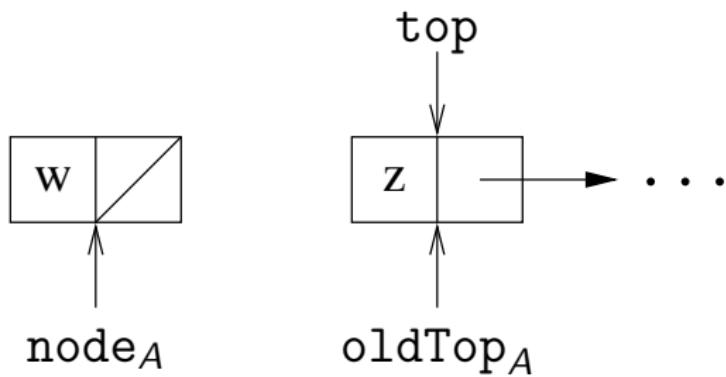
Lock-free unbounded stack: Push

```
protected boolean tryPush(Node node) {  
    Node oldTop = top.get();  
    node.next = oldTop;  
    return top.compareAndSet(oldTop, node);  
}  
  
public void push(T value) {  
    Node node = new Node(value);  
    while true {  
        if tryPush(node)  
            return;  
        else  
            backoff.backoff();  
    }  
}
```

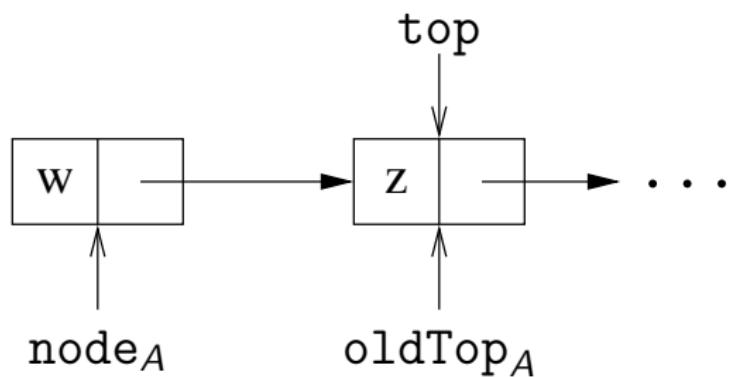
Lock-free unbounded stack: Push example



Lock-free unbounded stack: Push example

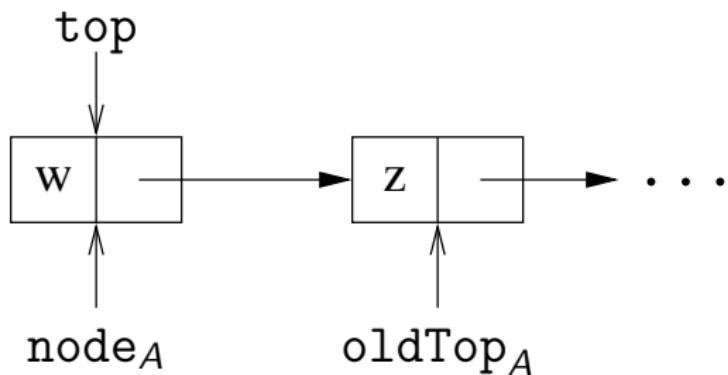


Lock-free unbounded stack: Push example



Lock-free unbounded stack: Push example

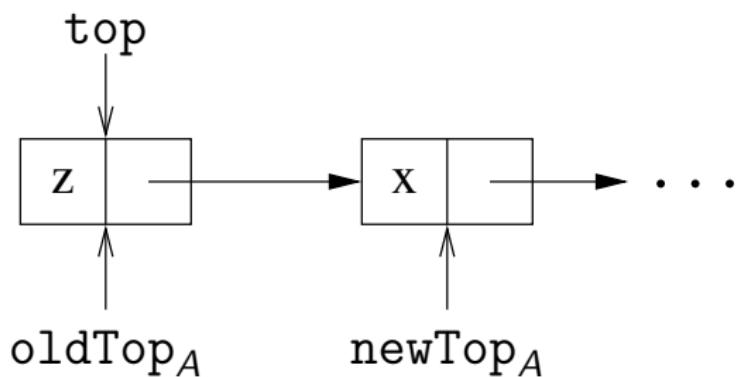
`top.compareAndSet(oldTopA, nodeA)`



Lock-free unbounded stack: Pop

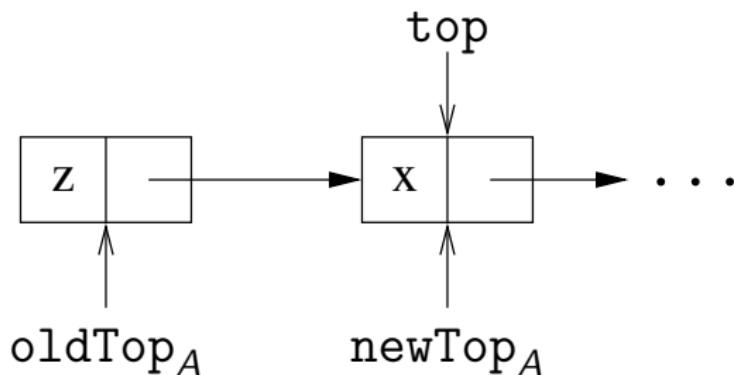
```
protected Node tryPop() throws EmptyException {
    Node oldTop = top.get();
    if oldTop == null
        throw new EmptyException();
    Node newTop = oldTop.next;
    if top.compareAndSet(oldTop, newTop)
        return oldTop;
    else
        return null;
}
public T pop() throws EmptyException {
    while true {
        Node returnNode = tryPop();
        if returnNode != null
            return returnNode.value;
        else
            backoff.backoff(); }
}
```

Lock-free unbounded stack: Pop example

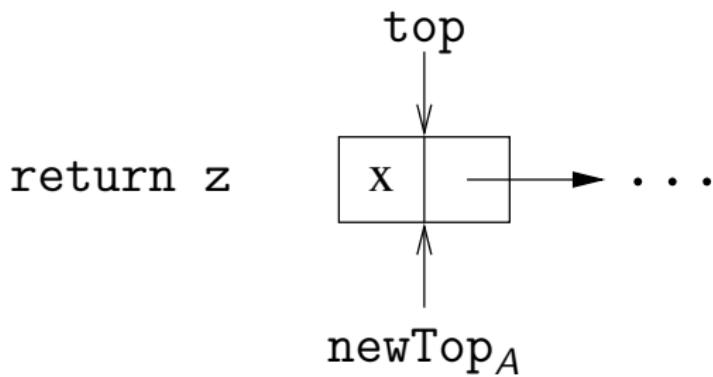


Lock-free unbounded stack: Pop example

`top.compareAndSet(oldTopA, newTopA)`



Lock-free unbounded stack: Pop example



Lock-free unbounded stack: Linearization

The **abstraction map** maps each *linked list* to the *stack* of items that reside in a node reachable from *top*.

Linearization points:

- ▶ of a `push()` when `top.compareAndSet(oldTop, node)` returns *true*
- ▶ of a `pop()`
 - on a *nonempty* stack when `top.compareAndSet(oldTop, newTop)` returns *true*
 - on an *empty* stack at `oldTop = top.get()`, when `oldTop` is assigned the value `null`

Lock-free unbounded stack: Evaluation

The unbounded stack is **lock-free**.

A `push()` or `pop()` only needs to retry if a concurrent call succeeded to modify top.

Although **exponential backoff** helps to alleviate contention, `compareAndSet()` calls on top create a sequential bottleneck.

This can be resolved by an **elimination array**, via which concurrent `push()` and `pop()` calls can be synchronized without referring to the stack.

Elimination array

Suppose a `tryPush()` (or `tryPop()`) fails (due to contention).

Instead of backoff, a random slot of the **elimination array** is claimed with `compareAndSet()`.

There the thread waits, up to a fixed amount of time, for a corresponding call:

- ▶ If a `push()` and `pop()` meet, they can synchronize.
- ▶ If a `push()` and `push()` (or `pop()` and `pop()`) meet, or no other call arrives within the fixed period, the thread reruns `tryPush()` (or `tryPop()`).

The **array size** and **timeout** at the array can be *dynamic*, depending on the level of contention.

Elimination array: Implementation

Thread *A* accessing a slot in the elimination array can meet three states:

EMPTY: No thread is waiting at this slot. *A* sets the state to **WAITING** by a `compareAndSet()`. If it fails, *A* retries.

Else *A* spins until the state becomes **BUSY**. Then another thread accessed the slot. If one called `pop()` and one `push()`, they cancel out. Else *A* retries. In either case, *A* sets the state to **EMPTY**.

If no thread shows up before the timeout, *A* sets the state to **EMPTY** by a `compareAndSet()`. If it fails, some exchanging thread showed up.

WAITING: Another thread *B* is already waiting at the slot.

A sets the state to **BUSY** by a `compareAndSet()`. If it succeeds, *A* checks whether the calls of *A* and *B* cancel out. Else *A* retries.

BUSY: Two other threads are already at the slot. *A* retries.

Elimination array: Linearization + evaluation

Linearization points:

- ▶ A push() or pop() that succeeds on the stack (or throws an exception) is linearized as before.
- ▶ If a push() and pop() cancel out via the elimination array, the push() is linearized right before the pop().

The unbounded stack with an elimination array is **lock-free**.

This lecture in a nutshell

fine-grained bounded / unbounded FIFO queue

lock-free unbounded FIFO queue

ABA problem

AtomicStampedReference class

lock-free unbounded stack

elimination array

Work distribution

Work dealing: An overloaded thread tries to off-load tasks to other threads.

Drawback: If most threads are overloaded, effort is wasted.

Work stealing: An idle thread tries to steal tasks from other threads.

If an attempt to steal a task from a processor fails, the thief tries again at another processor.

(Before a thief tries to steal a task, it usually first gives descheduled threads the opportunity to gain its processor.)



"I'm not asking you to change your spots. I'm just asking you to take out the garbage."



Work-stealing bounded queue (or better: stack)

Each thread maintains a **bounded queue** of tasks to execute.

pushBottom: New tasks are added at the *bottom* of a queue.

popBottom: A thread takes tasks to work on from the *bottom* of its queue.

popTop: A thief tries to steal a task from the *top* of some queue.

bottom: The index of the next available *empty* slot in the queue, where a new task will be placed. (MRSW)

top: The index of the lowest *nonempty* slot in the queue, where a thief can try to steal a task. (MRMW)

When the queue is empty, **bottom** and **top** are 0.

Work-stealing bounded queue

`popBottom` and `pushBottom` don't interfere, as they are performed by the same thread, so they use **atomic reads-writes** on **bottom**.

`popTop`, and `popBottom` if the queue becomes empty, use **compareAndSet** on **top**.

pushBottom: Places a new task at position `bottom` and increases `bottom` by 1 (if `bottom < capacity`).

popTop: Copies task r at position `top` (if `top < bottom`), tries to increase `top` by 1 with `compareAndSet`, and if successful performs r .

Work-stealing bounded queue: popBottom

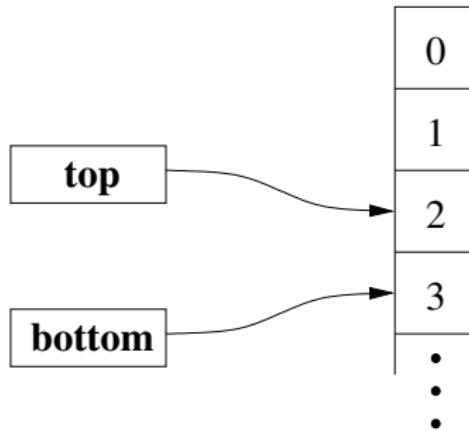
`popBottom`: Decreases bottom by 1 (if `bottom > 0`), and checks whether `top < bottom`.

If yes, the task at position `bottom` is returned, because a clash with a thief is impossible.

If no, the queue becomes empty, and there may be a clash with a thief.

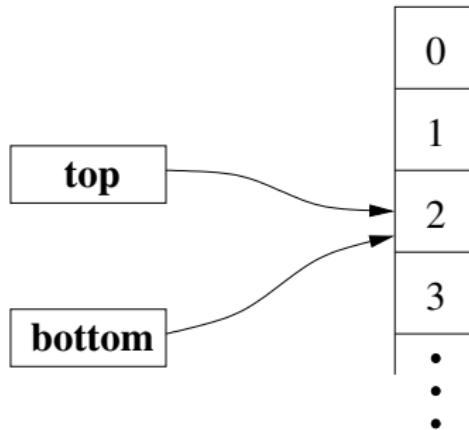
- ▶ `bottom` is set to 0.
- ▶ If it found `top = bottom`, `compareAndSet` is used to try to set `top to 0`. If this succeeds, the task can be returned.
- ▶ If it found `top > bottom` or `compareAndSet fails`, the task was stolen by a thief. Then `top is set to 0` and `null` is returned.

Work-stealing bounded queue: Example 1



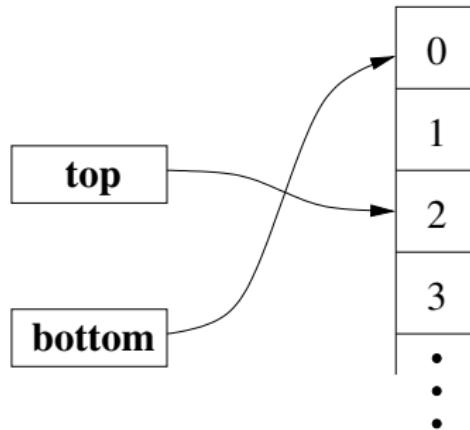
`popTop` finds that `top < bottom`.

Work-stealing bounded queue: Example 1



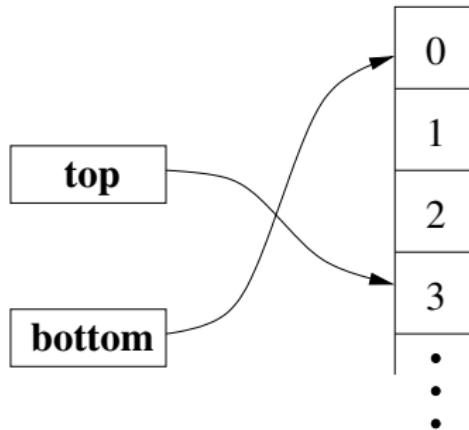
`popBottom` decreases `bottom`, and detects a possible clash with a thief.

Work-stealing bounded queue: Example 1



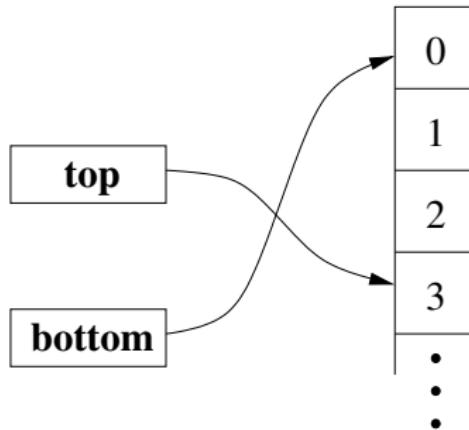
`popBottom` sets `bottom` to 0.

Work-stealing bounded queue: Example 1



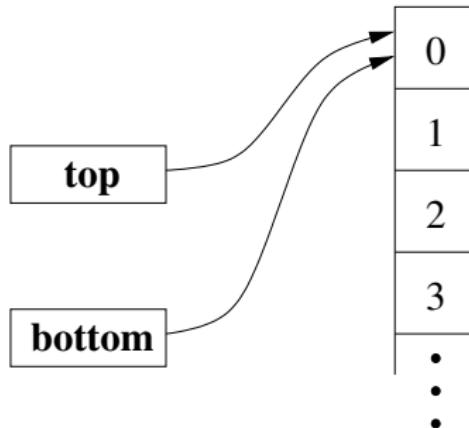
`popTop` increases `top` with `compareAndSet(2,3)`, and returns the task at 2.

Work-stealing bounded queue: Example 1



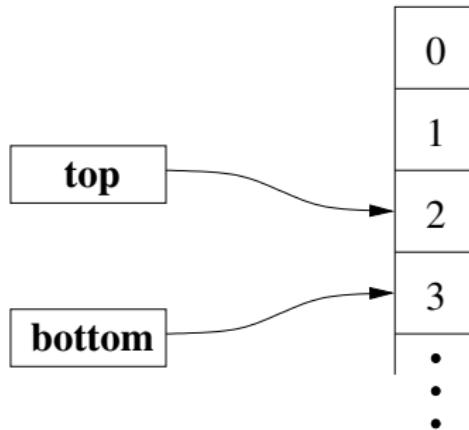
`popBottom` unsuccessfully applies `compareAndSet(2,0)` to `top`.

Work-stealing bounded queue: Example 1



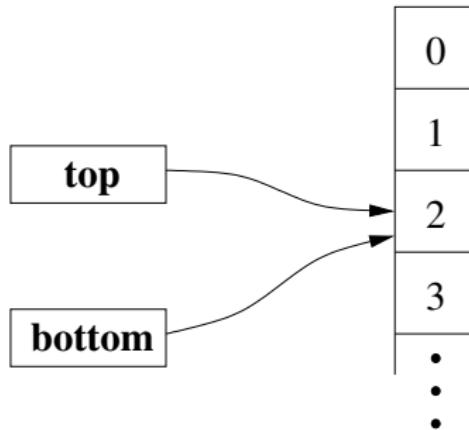
`popBottom` sets `top` to 0, and returns `null`.

Work-stealing bounded queue: Example 2



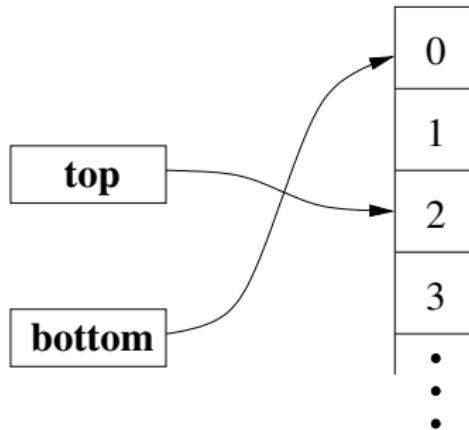
`popTop` finds that `top < bottom`.

Work-stealing bounded queue: Example 2



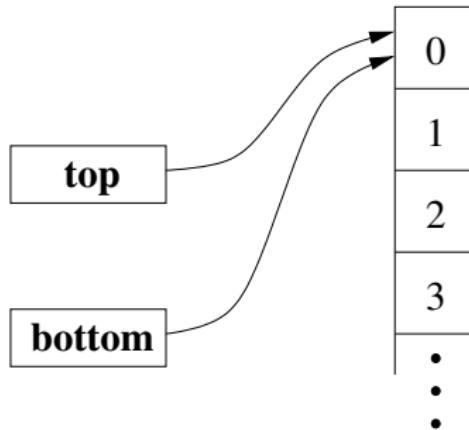
`popBottom` decreases `bottom`, and detects a possible clash with a thief.

Work-stealing bounded queue: Example 2



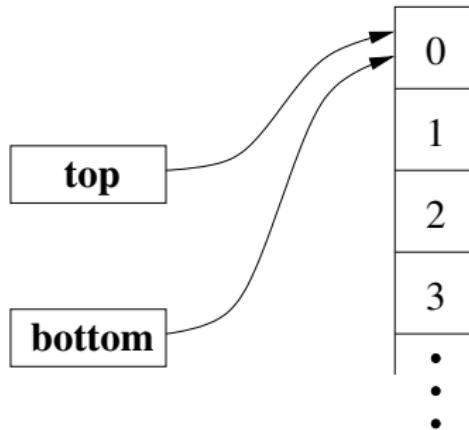
`popBottom` sets `bottom` to 0.

Work-stealing bounded queue: Example 2



`popBottom` sets `top` to 0 with `compareAndSet(2,0)`, and returns the task at 2.

Work-stealing bounded queue: Example 2



`popTop` unsuccessfully applies `compareAndSet(2,3)` to `top`, and returns *null*.

Work-stealing bounded queue: ABA problem

Question: How can the ABA problem arise?

Answer: Consider the following scenario:

- ▶ Thief *A* reads `top = 0` and `bottom > 0` in the queue of *B*, and obtains the task at position 0.
- ▶ *B* removes and executes all tasks in its queue, and replaces them by one or more new tasks.
- ▶ *A* successfully increases `top` by `compareAndSet(0, 1)`.

As a result, *A* steals a task that has already been completed, and removes a task that will never be completed.

Solution: `top` is an `AtomicStampedReference<Integer>`.

Work-stealing bounded queue: bottom is volatile

That top is an `AtomicStampedReference<Integer>` comes with strong synchronization.

A decrement of `bottom` by `popBottom` must also be observed immediately by concurrent thieves.

Else a thief might not observe that the queue is empty, and steal a job that was already popped.

Therefore `bottom` must be **volatile**. (See Exercise 193(1).)

Work-stealing bounded queue: pushBottom

```
public class BDEQueue {  
    volatile int bottom = 0;  
    top = new AtomicStampedReference<Integer>(0, 0);  
    public BDEQueue(int capacity) {  
        Runnable[] tasks = new Runnable[capacity];  
    }  
  
    public void pushBottom(Runnable r)  
        throws QueueFullException {  
        if bottom < capacity then {  
            tasks[bottom] = r;  
            bottom++; }  
        else QueueFullException();  
    }  
}
```

Work-stealing bounded queue: popBottom

```
public Runnable popBottom() {
    if bottom == 0
        return null;
    bottom--;
    Runnable r = tasks[bottom];
    int[] stamp = new int[1];
    int oldTop = top.get(stamp);
    int oldStamp = stamp[0];
    if bottom > oldTop
        return r;
    if bottom == oldTop {
        bottom = 0;
        if top.compareAndSet(oldTop, 0, oldStamp, oldStamp + 1)
            return r; }
    bottom = 0;
    top.set(0, oldStamp + 1);
    return null; }
```

Work-stealing bounded queue: popTop

```
public Runnable popTop() {  
    int[] stamp = new int[1];  
    int oldTop = top.get(stamp);  
    int oldStamp = stamp[0];  
    if bottom <= oldTop  
        return null;  
    Runnable r = tasks[oldTop];  
    if top.compareAndSet(oldTop, oldTop + 1, oldStamp, oldStamp + 1)  
        return r;  
    return null;  
}
```

Questions

In our scenario with the ABA problem, how is the stamp of top increased, although the value of top remains 0 all the time?

Answer: When the queue becomes empty, popBottom sets top (from 0) to 0.

Give a scenario where the queue owner and a thief both write to top with the same stamp value.

Why is this not a problem ?

Work-stealing bounded queue: Linearization

Linearizing an *unsuccessful* `popTop` is tricky.

Herlihy & Shavit linearize it when the queue is found to be empty or `compareAndSet` fails.

This may be too late, if in the meantime a `pushBottom` added a new task and returned.

Work-stealing bounded queue: Linearization

The linearization points:

- ▶ successful `pushBottom`: When `bottom` is incremented.
- ▶ unsuccessful `pushBottom`: When it is found `bottom == capacity`.
- ▶ successful `popBottom`: When it is found `bottom > oldTop`, or when `compareAndSet` succeeds.
- ▶ unsuccessful `popBottom`: When it is found `bottom == 0` or `bottom < oldTop`, or when `compareAndSet` fails.
- ▶ successful `popTop`: When `compareAndSet` succeeds.
- ▶ unsuccessful `popTop`: At a moment the queue is empty, or right after a `popBottom` or concurrent `popTop` that removes the task it wants to steal.

Termination detection barrier

Threads are either **active** or **passive**.

The system has *terminated* when all threads are passive.

A (non-reusable) *barrier* detects termination by means of a **counter** of the number of active threads.

A thread that becomes *passive* performs **getAndDecrement** on the counter.

A thread that becomes *active* again performs **getAndIncrement** on the counter.

Termination detection for the work-stealing queue

A thread with an empty task queue becomes passive.

A passive thread may try to steal a task at another queue.

Before trying to steal a task, it becomes active.

If the theft fails, it becomes passive again.

Questions

Why should a thief become **passive** before trying to steal again ?

Why should a thief become **active** *before* trying to steal a task,
and not after succeeding to steal a task ?

Question

Give an infinite execution in which it is never detected that the system has terminated.

How can this be resolved ?

Answer: Before becoming active to try to steal a task at a queue, the thread first tests whether the queue is nonempty.

Parallelization of matrix operations

Matrix addition and multiplication are *embarrassingly parallel*:

The coefficients of the resulting matrix can be computed individually.

How can we parallelize the computation of the sum or product of two $n \times n$ matrices?

Solution 1: Start n^2 (short-lived) threads; each thread computes a c_{ij} .

Drawback: Creating, scheduling and destroying threads takes a substantial amount of computation and memory.

Solution 2: Create a **pool** of threads, one per processor, that repeatedly obtain an assignment and run the task.

A *thread pool* allows to abstract away from platform-specific details.

Thread pools in Java: Callable

A `Callable<T>` object:

- ▶ calls a `T call()` method, and
- ▶ returns a value of type `T`.

The *executor service* provides:

- ▶ a `Future<T>` interface, which is a *promise* to return the result, when it is ready;
- ▶ a `get()` method which returns the result, blocking if necessary until the result is ready;
- ▶ methods for canceling uncompleted computations, and for testing whether the computation is complete.

Parallelization of Fibonacci numbers

The **Fibonacci numbers** are defined by:

$$Fib(0) = Fib(1) = 1 \quad Fib(n + 2) = Fib(n + 1) + Fib(n)$$

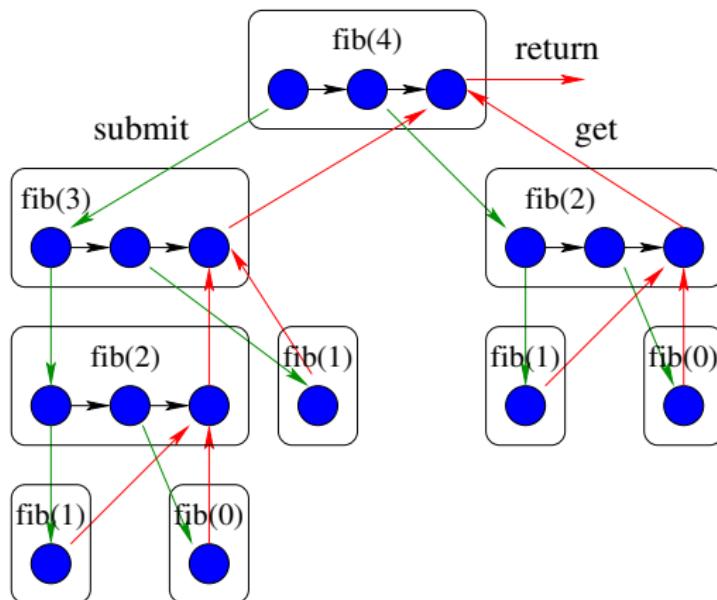
A (very inefficient) implementation of `FibTask(int arg)`, using `Callable<Integer>`:

```
public Integer call() {
    if arg > 1 {
        Future<Integer> left = exec.submit new FibTask(arg-1);
        Future<Integer> right = exec.submit new FibTask(arg-2);
        return left.get() + right.get();
    } else { return 1; }
}
```

Parallelization of Fibonacci numbers

We depict a multithreaded computation as a *directed acyclic graph*.

For instance, the computation of **FibTask(4)** looks as follows:



Thread pools in Java: Runnable

A **Runnable** object returns no result.

(It is typically used in embarrassingly parallel applications.)

The executor service provides:

- ▶ a **Future<?>** interface;
- ▶ a `get()` method which blocks until the result is ready;
- ▶ methods for canceling uncompleted computations, etc.

Runnable was present in Java 1.0; Callable was added in Java 1.5.

Example: Matrix addition and multiplication are implemented with Runnable: Futures are only used to signal when a task is complete.

Executor service

Executor service submissions (like Amsterdam traffic signs) are advisory in nature.



The executor (like an Amsterdam biker) is free to ignore such advice, and may e.g. execute tasks sequentially.

Measuring parallelism

Given a multi-threaded program.

T_P : minimum time (measured in computation steps)
to execute the program on P processors ($1 \leq P \leq \infty$)

T_1 : total amount of work

T_∞ : critical path length

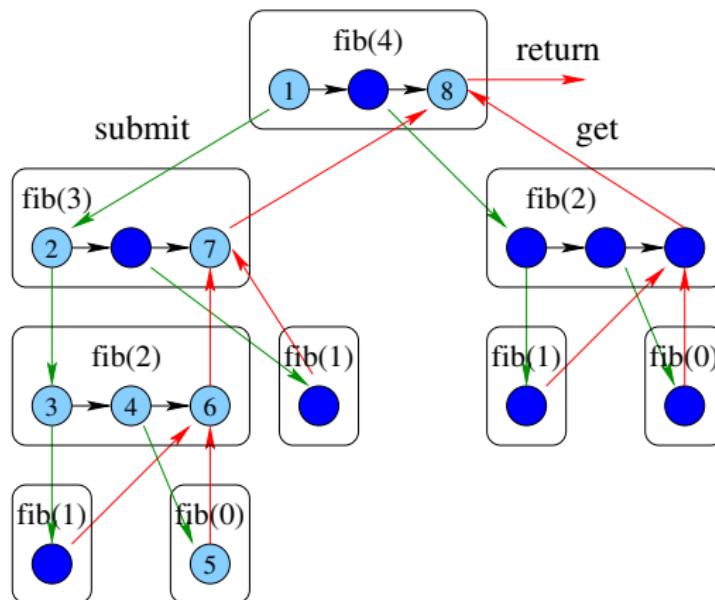
speedup on P processors: T_1/T_P

linear speedup: if $T_1/T_P \in \Theta(P)$

maximum speedup: T_1/T_∞

Parallelization of Fibonacci numbers

The computation of **FibTask(4)** looks as follows:



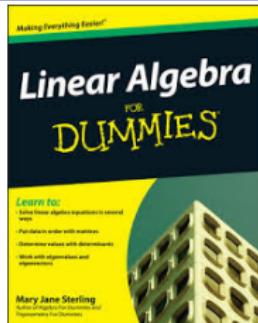
The computation's **work** is 25, and its **critical path** has length 8.

Matrices

Question: What does an $n \times n$ matrix represent?

For example:

$$\begin{pmatrix} 3 & -1 \\ 4 & 2 \end{pmatrix}$$



Answer: A *linear* mapping from an n -dimensional space to itself.

In the example, a mapping from \mathbb{R}^2 to \mathbb{R}^2 .

Question: What do the columns of an $n \times n$ matrix express?

Answer: The images of the n base vectors. In the example:

$$\begin{pmatrix} 3 & -1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \end{pmatrix} \quad \begin{pmatrix} 3 & -1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$$

Matrices

Since the mapping is linear, the images of the n base vectors determine the image of every n -dimensional vector.

Example:

$$\begin{pmatrix} 3 & -1 \\ 4 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ -2 \end{pmatrix} = \begin{pmatrix} 9 \\ 12 \end{pmatrix} + \begin{pmatrix} 2 \\ -4 \end{pmatrix} = \begin{pmatrix} 11 \\ 8 \end{pmatrix}$$

Matrix addition

$$\begin{aligned} & \left(\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} + \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \right) \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} a_{00} \\ a_{10} \end{pmatrix} + \begin{pmatrix} b_{00} \\ b_{10} \end{pmatrix} = \begin{pmatrix} a_{00} + b_{00} \\ a_{10} + b_{10} \end{pmatrix} \end{aligned}$$

$$\left(\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} + \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \right) \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{01} + b_{01} \\ a_{11} + b_{11} \end{pmatrix}$$

Matrix addition

The sum C of two $n \times n$ matrices A and B is given by:

$$c_{ij} = a_{ij} + b_{ij}$$

The total amount of work to compute C is $\Theta(n^2)$.

Because calculating a single c_{ij} takes $\Theta(1)$.

And there are n^2 coefficients c_{ij} .

Matrix multiplication

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} \\ b_{10} \end{pmatrix}$$
$$= \begin{pmatrix} a_{00} \cdot b_{00} + a_{01} \cdot b_{10} \\ a_{10} \cdot b_{00} + a_{11} \cdot b_{10} \end{pmatrix}$$

$$\begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \cdot \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{01} \\ b_{11} \end{pmatrix}$$
$$= \begin{pmatrix} a_{00} \cdot b_{01} + a_{01} \cdot b_{11} \\ a_{10} \cdot b_{01} + a_{11} \cdot b_{11} \end{pmatrix}$$

Matrix multiplication

The product C of two $n \times n$ matrices A and B is given by:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}$$

The total amount of work to compute C is $\Theta(n^3)$.

Because calculating a single c_{ij} takes $\Theta(n)$.

And there are n^2 coefficients c_{ij} .

Parallelization of matrix addition

For simplicity, let n be a power of 2 (say $n = 2^k$).

Matrix addition $A + B$ of $n \times n$ matrices can be split into:

$$\begin{pmatrix} A_{00} + B_{00} & A_{01} + B_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

where the A_{ij} and B_{kl} are $\frac{n}{2} \times \frac{n}{2}$ matrices.

The four sums $A_{ij} + B_{kl}$ can be computed in parallel, and split in turn, until we get 1×1 matrices.

Recall that the total amount of work is $\Theta(n^2)$.

Parallelization of matrix addition

Let $A_P(n)$ be the running time of addition of $n \times n$ matrices on P processors.

Work : $A_1(n) = 4 \cdot A_1(n/2) + \Theta(1)$ if $n \geq 2$

(4 splits, plus constant amount of work to perform the splits)

$$A_1(1) = \Theta(1)$$

$$\begin{aligned} A_1(2^k) &= 4 \cdot A_1(2^{k-1}) + \Theta(1) \\ &= 4^2 \cdot A_1(2^{k-2}) + 4 \cdot \Theta(1) + \Theta(1) \\ &= \dots \\ &= 4^k \cdot A_1(1) + (4^{k-1} + 4^{k-2} + \dots + 1) \cdot \Theta(1) \\ &= \Theta(4^k) = \Theta(2^{2k}) \end{aligned}$$

So $A_1(n) = \Theta(n^2)$.

Parallelization of matrix addition

Critical path length : $A_\infty(n) = A_\infty(n/2) + \Theta(1)$ if $n \geq 2$

$$A_\infty(1) = \Theta(1)$$

$$\begin{aligned} A_\infty(2^k) &= A_\infty(2^{k-1}) + \Theta(1) \\ &= A_\infty(2^{k-2}) + \Theta(1) + \Theta(1) \\ &= \dots \\ &= A_\infty(1) + k \cdot \Theta(1) \\ &= \Theta(k) \end{aligned}$$

So $A_\infty(n) = \Theta(\log n)$.

Maximum speedup: $A_1(n)/A_\infty(n) = \Theta(n^2/\log n)$

Parallelization of matrix multiplication

Matrix multiplication $A \cdot B$ of $n \times n$ matrices can be split into:

$$\begin{pmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{10} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{10} + A_{11} \cdot B_{11} \end{pmatrix}$$

where the A_{ij} and B_{kl} are $\frac{n}{2} \times \frac{n}{2}$ matrices.

The eight products $A_{ij} \cdot B_{kl}$ can be computed in parallel, and split in turn, until we get 1×1 matrices.

Recall that the total amount of work is $\Theta(n^3)$.

Parallelization of matrix multiplication

Let $M_P(n)$ be the running time of multiplication of $n \times n$ matrices on P processors.

Work: $M_1(n) = 8 \cdot M_1(n/2) + 4 \cdot A_1(n/2) = 8 \cdot M_1(n/2) + \Theta(n^2)$

$$\begin{aligned} M_1(2^k) &= 8 \cdot M_1(2^{k-1}) + \Theta(2^{2k}) \\ &= 8^2 \cdot M_1(2^{k-2}) + 8 \cdot \Theta(2^{2k-2}) + \Theta(2^{2k}) \\ &= 8^2 \cdot M_1(2^{k-2}) + \Theta(2^{2k+1}) + \Theta(2^{2k}) \\ &= \dots \\ &= 8^k \cdot M_1(1) + \Theta(2^{3k-1} + 2^{3k-2} + \dots + 2^{2k}) \\ &= \Theta(2^{3k}) \end{aligned}$$

So $M_1(n) = \Theta(n^3)$.

Parallelization of matrix multiplication

Critical path length: $M_\infty(n) = M_\infty(n/2) + A_\infty(n/2) = M_\infty(n/2) + \Theta(\log n)$

$$\begin{aligned}M_\infty(2^k) &= M_\infty(2^{k-1}) + \Theta(k) \\&= M_\infty(2^{k-2}) + \Theta(k-1) + \Theta(k) \\&= \dots \\&= M_\infty(1) + \Theta(2 + 3 + \dots + k) \\&= \Theta(k^2)\end{aligned}$$

So $M_\infty(n) = \Theta(\log^2 n)$.

Maximum speedup: $M_1(n)/M_\infty(n) = \Theta(n^3/\log^2 n)$

This lecture in a nutshell

work-stealing bounded queue

ABA problem

termination detection barrier

thread pools in Java

parallelization of matrix operations

measuring parallelism

What is wrong with locking ?

Performance bottleneck: Many threads may concurrently want the same lock.

Not robust: If a thread holding a lock is delayed or crashes, other threads can't make progress.

Lost wakeup: A waiting thread may not realize when to wake up.

Deadlock: Can for instance occur if two threads attempt to lock the same two objects in different orders.

Not composable: Managing concurrent locks to atomically e.g. delete an item from one table and insert it in another table is essentially impossible without breaking the lock internals.

Hard to use: Already a FIFO queue with fine-grained locking and conditions is non-trivial.

What is wrong with locking ?

Relies on conventions: Nobody really knows how to organize and maintain large systems that rely on locking.

The association between locks and data in the programmer's mind remains implicit, and may be documented only in comments.

Example: A typical comment from the Linux kernel.

When a locked buffer is visible to the I/O layer, BH_Launder is set.

This means before unlocking we must clear BH_Launder [...] and then clear BH_Lock, so no reader can see BH_Launder set on an unlocked buffer and then risk to deadlock.

What is wrong with compareAndSet ?

- ▶ Has a high performance overhead.
- ▶ Delicate to use (e.g. the ABA problem).
- ▶ Operates on a single field.

Often we would want to simultaneously apply compareAndSet on an **array** of fields, where if any one fails, they all do.

(Like the `AtomicStampedReference` class.)

But there is no obvious way to efficiently implement a general `multiCompareAndSet` on conventional architectures.

What is wrong with compareAndSet ?

The lock-free unbounded FIFO queue became complicated because we couldn't in one atomic compareAndSet:

- ▶ add a new node at the end of the list, and
- ▶ advance tail.

To make the queue lock-free, threads had to help a slow enqueuer to advance tail.

Moreover, we had to avoid that head could overtake tail.

The transactional manifesto

What we do now is inadequate to meet the multicore challenge.

A new programming paradigm is needed.

Ungoing research challenge:

- ▶ Develop a *transactional* application programmer interface.
- ▶ Design languages to support this model.
- ▶ Implement the run-time to be fast enough.

Transactions

A **transaction** is a sequence of steps executed by a single thread.

A transaction makes *tentative* changes, and then either:

- ▶ **commits**: its steps take effect at once; or
- ▶ **is aborted**: its effects are rolled back
(usually the transaction is restarted after a while).

A transaction may be aborted by another transaction in case of a **synchronization conflict** on a register.

Transactions are serializable

A transaction allows an *atomic update of multiple locations.*

(This eliminates the need for a `multiCompareAndSet`.)

Transactions are **serializable**:

They appear to execute sequentially, in a one-at-a-time order.

Bounded FIFO queue with locks and conditions: Enqueue

```
public void enq(T x) {  
    lock.lock();  
    try {  
        while (count == items.length)  
            notFull.await();  
        items[tail] = x;  
        if (++tail == items.length)  
            tail = 0;  
        if (count++ == 0)  
            notEmpty.signalAll();  
    } finally {  
        lock.unlock();  
    }  
}
```

Bounded transactional FIFO queue: Enqueue

```
public void enq(T x) {  
    atomic {  
        if count == items.length  
            retry;  
        items[tail] = x;  
        if ++tail == items.length  
            tail = 0;  
        count++;  
    }  
}
```

retry rolls back the enclosing transaction, and restarts it when the object state has changed.

retry is less vulnerable to lost wakeups than await().

Lock-free unbounded FIFO queue: Enqueue

```
public void enq(T x) {  
    Node node = new Node(x);  
    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);  
        }  
    }  
}
```

create new node
retry until enqueue succeeds
is tail the last node?
try add node
try advance tail
try advance tail

Unbounded transactional FIFO queue: Enqueue

```
public void enq(T x) {  
    Node node = new Node(x);  
    atomic {  
        tail.next = node;  
        tail = node;  
    }  
}
```

Synchronized versus atomic blocks

A **synchronized** block acquires a *lock*.

An **atomic** block checks for *synchronization conflicts* on registers.

A **synchronized** block is *blocking* (e.g., nested synchronized blocks that acquire locks in opposite order may cause deadlock).

An **atomic** block is *non-blocking*.

A **synchronized** block is only atomic with respect to other synchronized blocks that *acquire the same lock*.

An **atomic** block is atomic with respect to *all* other atomic blocks.

Transactions are composable

Managing concurrent locks to, e.g., atomically

- ▶ delete an item from one queue q_0 , and
- ▶ insert it in another queue q_1 ,

is impossible without breaking the lock internals.

With transactions we can compose method calls atomically:

```
atomic {
    x = q0.deq();
    q1.enq(x);
}
```

Transactions can wait for multiple conditions

With monitors, we can't wait for one of multiple conditions to become true.

With transactions we can do so:

```
atomic {
    x = q0.deq();
} orElse {
    x = q1.deq();
}
```

If the first block calls `retry`, that subtransaction is rolled back, and the second block is executed.

If that block also calls `retry`, the entire transaction is rerun later.

Nested transactions

Ideally, *nested* transactions are allowed.

A nested transaction can be aborted without aborting its parent.

This way a method can start a transaction and then call another method without caring whether that method starts a transaction.

Hardware transactional memory

Cache coherence protocols already do most of what is needed to implement transactions:

- ▶ avoid and resolve synchronization conflicts
- ▶ buffer tentative changes

MESI cache coherence protocol

Each **cache line** is *marked*:

- ▶ **Modified:** Line has been modified, and must eventually be stored in memory.
- ▶ **Exclusive:** Line hasn't been modified, and no other processor has this line cached (typically used before modifying the line).
- ▶ **Shared:** Line hasn't been modified, and other processors may have this line cached.
- ▶ **Invalid:** Line doesn't contain meaningful data.

If a processor wants to load or store a line, it broadcasts the request over the bus. Other processors and memory listen.

MESI cache coherence protocol

A write to a register may only be performed if its cache line is *exclusive* or *modified*.

If a processor *A* changes a line to *exclusive* mode, other processors *invalidate* corresponding lines in their cache.

When *A* writes to a register, its cache line is set to *modified*.

A processor *B* wanting to read this register broadcasts a request.

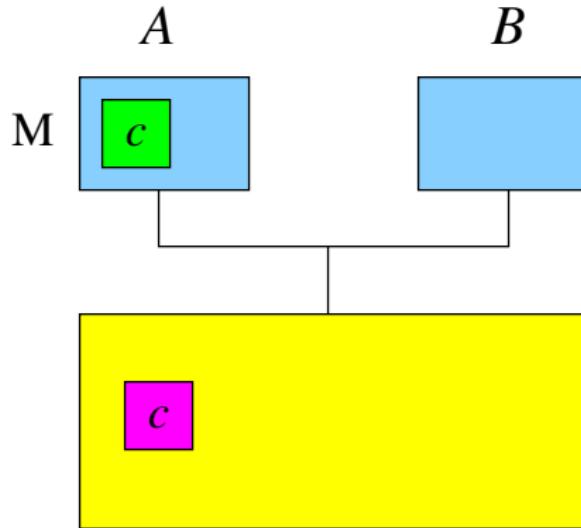
A sends the modified data to both *B* and memory.

The copies at *A* and *B* are *shared*.

If the cache becomes full or at a memory barrier, lines are evicted.

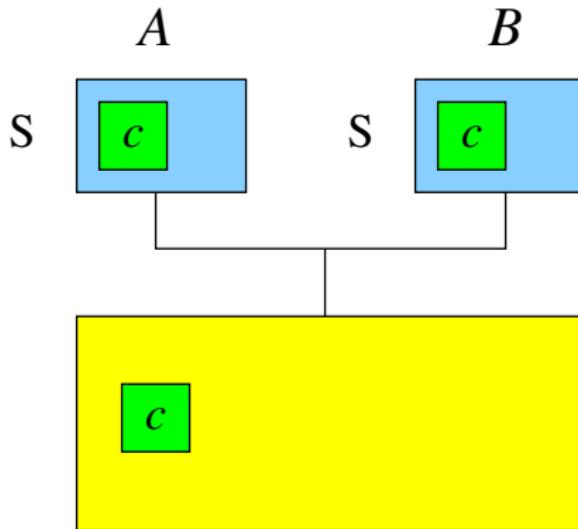
Modified lines are stored in memory when they are invalidated or evicted.

MESI cache coherence protocol: Example



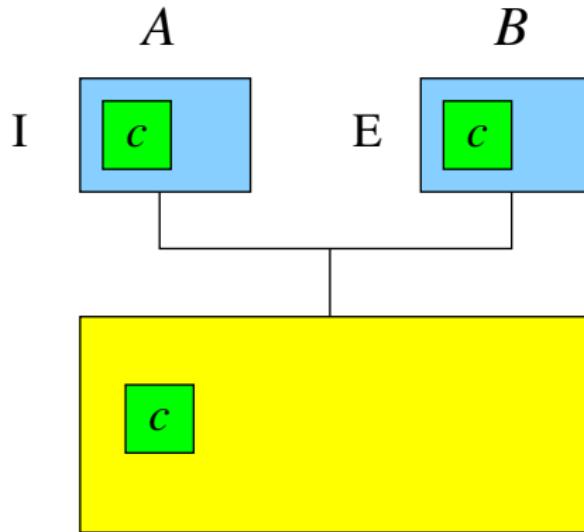
A has line *c* in its cache in *modified* mode.

MESI cache coherence protocol: Example



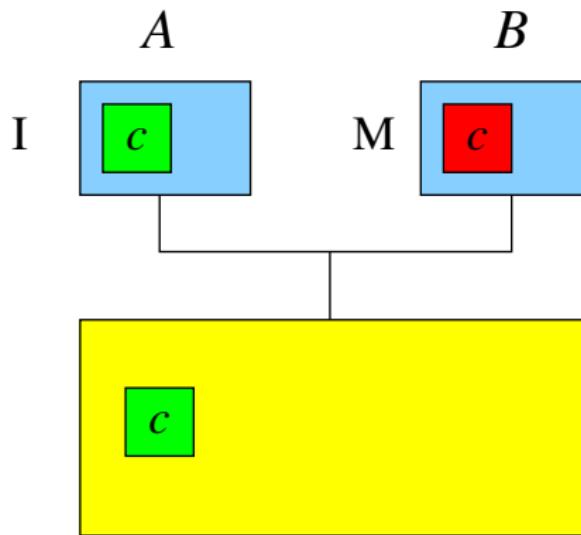
B wants to read from line *c*. *A* sends the data to *B* and memory.
Now *c* is cached at *A* and *B* in *shared* mode.

MESI cache coherence protocol: Example



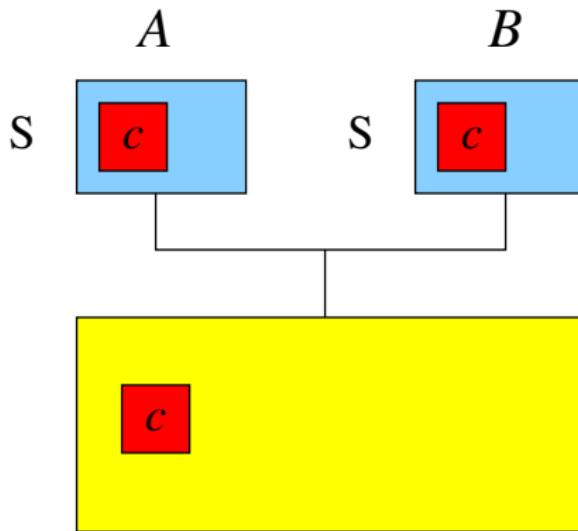
B wants to write to line *c* and changes it to *exclusive*, while *A* sets *c* to *invalid*.

MESI cache coherence protocol: Example



B writes to line *c*, and changes it to *modified*.

MESI cache coherence protocol: Example



A wants to read from *c* and broadcasts a request. *B* sends the modified data to *A* and memory, leaving both copies in *shared* mode.

Transactional cache coherence

A **transactional bit** is added to each cache line.

When a value is placed in a cache line on behalf of a transaction, the line's bit is set. Such a line is called *transactional*.

- ▶ If a transactional line is *invalidated* or *evicted*, the transaction is **aborted**.

Even if the line was modified, it isn't copied to memory.

When a transaction aborts, its transactional lines are *invalidated*.

- ▶ If a transaction finishes with none of its transactional lines invalidated or evicted, it **commits** by *clearing* its transactional bits.

Modified lines are only shared after committing.

Transactional cache coherence: Limitations

- ▶ The **size** of a transaction is limited by the size of the cache.
- ▶ Mostly the cache is cleaned when a thread is descheduled. Then the **duration** of a transaction is limited by the scheduling quantum.

Hardware transactional memory is suited for *small, short* transactions.

- ▶ If a transaction accesses two addresses that map to the same cache line, it may **abort itself**
(in case the second access automatically evicts the first one).
- ▶ There is **no contention management**: Transactions can starve each other by continuously (1) aborting another thread, (2) being aborted by another thread, and (3) restarting.

TSX

Transactional Synchronization Extensions (TSX) adds support for hardware transactional memory to Intel's x86 instruction set architecture.

June 2013, TSX was released on Intel's microprocessors based on the Haswell microarchitecture.

August 2014, Intel announced a bug and disabled TSX on affected CPUs. It was fixed in November 2014.

Software transactional memory

We discuss one possible way to build transactions in software.

A transaction at the start creates an object with as possible states:

- ▶ initially **ACTIVE**
- ▶ after the transaction has committed, **COMMITTED**
- ▶ after the transaction has been aborted, **ABORTED**

A transaction tries to *commit* by applying
`compareAndSet(ACTIVE, COMMITTED)` to its object.

A thread tries to *abort* a transaction by applying
`compareAndSet(ACTIVE, ABORTED)` to the transaction's object.

Atomic objects

Transactions communicate through **atomic objects**, which have three fields:

- ▶ **transaction** points to the transaction that most recently opened the object in *write* mode.
- ▶ **oldObject** points to the *old* object version.
- ▶ **newObject** points to the *new* object version.

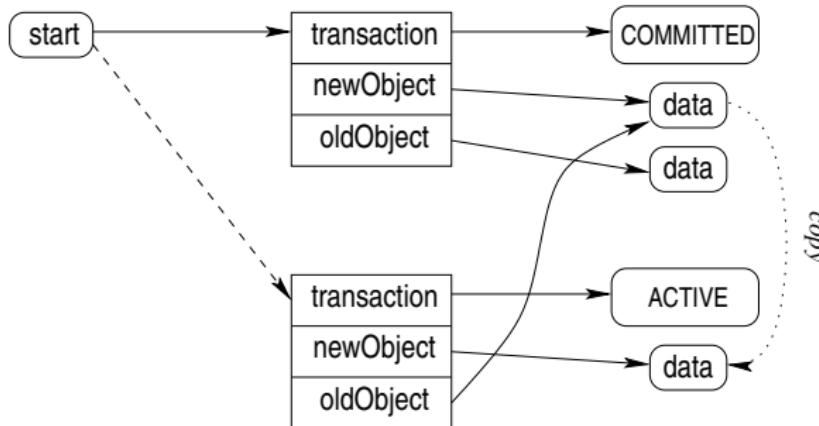
If **transaction** points to a transaction that is:

- ▶ **ACTIVE**, **oldObject** is *current* and **newObject** is *tentative*.
- ▶ **COMMITTED**, **oldObject** is *meaningless* and **newObject** is *current*.
- ▶ **ABORTED**, **oldObject** is *current* and **newObject** is *meaningless*.

Atomic objects: Writes

Suppose a transaction A opens an atomic object in *write mode*.

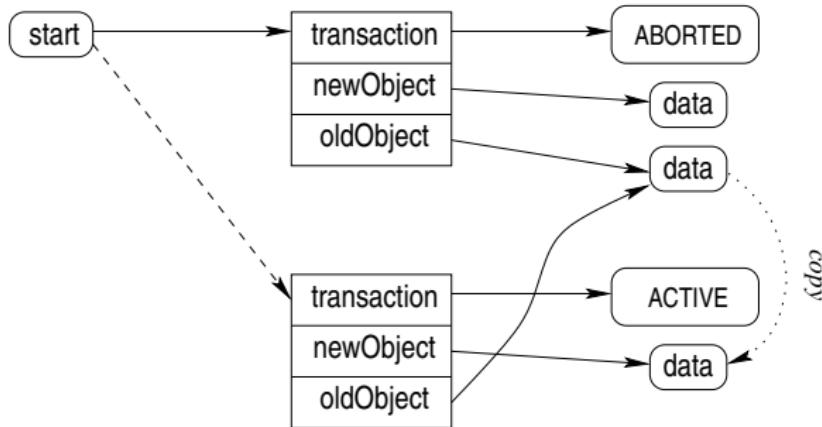
- ▶ If transaction is **COMMITTED**, then `newObject` is current:



By swinging the `start` reference (using `compareAndSet`), the three fields are changed in one atomic step.

Atomic objects: Writes

- If transaction is ABORTED, then oldObject is current:



- If transaction is ACTIVE, there is a *synchronization conflict*.

The conflict is with a thread $B \neq A$. Then A asks a **contention manager** whether it should *abort* B , or *wait* to give B a chance to finish.

Conflict resolution policies

Backoff: A repeatedly backs off, doubling its waiting time up to some max. When this limit is reached, A aborts B .

Priority: Transactions carry a *timestamp*.

If A has an older timestamp than B , A aborts B (otherwise A waits).

A transaction that restarts after an abort keeps its old timestamp, so that it eventually completes.

Greedy: Transactions carry a *timestamp*. If A has an older timestamp than B , or B is waiting for another transaction, A aborts B .

This strategy avoids chains of waiting transactions.

Karma: Transactions keep track of the *amount of work* they did.

If A has done more work than B , A aborts B .

Atomic objects: Reads

Suppose a transaction A opens an atomic object in *read* mode.

If transaction is:

- ▶ **COMMITTED**, A makes a local copy of `newObject`.
- ▶ **ABORTED**, A makes a local copy of `oldObject`.
- ▶ **ACTIVE**, there is a synchronization conflict,
so A consults a contention manager.

Obstruction-freeness

In general, transactions are **not lock-free**.

(This depends on the conflict resolution policy.)

They are always ***obstruction-free***:

If a transaction runs on its own, it is guaranteed to eventually complete.

Software transactional memory: Features

Software transactional memory can be provided with:

- ▶ `retry`
- ▶ `orElse`
- ▶ `nested transactions`

Zombies

A transaction may encounter *inconsistent* states.

Why do we care?

Such inconsistencies could e.g. force a thread into an infinite loop.

Or lead to a division by 0, throw an exception, halt its thread, and possibly crash the application.

Therefore such a transaction, called a **zombie**, should be aborted immediately.



Zombies: Example

Initially x and y have the same value. Thread A performs

```
atomic {  
    x++;  
    y++;  
}
```

Thread B performs

```
atomic {  
    if x != y  
        while true {}  
}
```

If B reads x before A updates it, and y after A commits,
 B would get into an infinite loop.

Validation

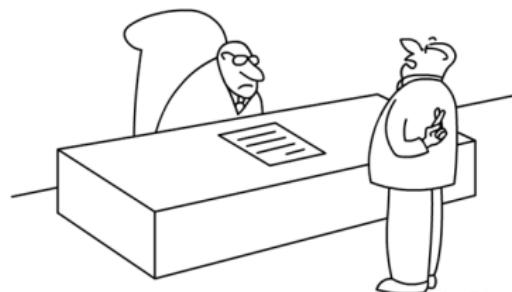
Therefore a transaction must, every time it has read or written to an atomic object, and before it commits, **validate** that:

- ▶ it hasn't been aborted, and
- ▶ the values it read of atomic objects are unchanged.

So transactions keep a log of their reads.

If a read value has changed,
the transaction aborts itself.

This validation procedure guarantees that
transactions only see consistent states.



"Yes sir, you can absolutely trust those numbers"

Question

Why doesn't a transaction need to check that the atomic objects to which it has *written* are unchanged ?

Answer: As long as the transaction is ACTIVE, no other transaction can write to such variables.

I/O operations

When a transaction aborts, its I/O operations should roll back.

Solution: *Buffer* I/O operations.

But this fails if a transaction e.g. writes a prompt and waits for a reply.

Another solution: Only a *privileged* transaction that triumphs over all conflicting transactions and whose read values are shielded from writes by other transactions can perform I/O operations.

The privilege is passed between transactions.

The amount of I/O a program can perform is in any case very limited.

Software transactional memory: Disadvantages

High contention may lead to massive abortion of transactions.

Maintaining a log, validating, and committing transactions is expensive.

Transactions can only perform operations that can be undone (excluding most I/O).

Blocking software transactional memory

In 2006, Robert Ennals advocated *blocking* software transactional memory.

Non-blockingness requires a pointer from an object's header to its data.

The object's header and data tend to be stored in different cache lines.

Storing header and data at the same cache line gives fewer cache misses.

But then data can no longer be updated by swinging a pointer.

Ennals showed that a blocking approach yields a better performance.

Software transactional memory

Software transactional memory
is under development.

It may be the future for
multiprocessor programming.

Some popular software transactional
memory implementations:

- ▶ **SXM**: written in C#, developed by Microsoft Research
- ▶ **Clojure**: a dialect of Lisp



Edsger W. Dijkstra Prize in Distributed Computing 2012:

- ▶ Herlihy & Moss, *Transactional Memory*, 1993
- ▶ Shavit & Touitou, *Software Transactional Memory*, 1997

Learning objectives of the course

Fundamental insight into multicore computing: mutual exclusion, locks, read-modify-write operations, consensus, construction of atomic multi-reader multi-writer registers

Algorithms for multicore computing: spin locks, monitors, barriers, transactional memory

Concurrent datastructures: lists, sets, queues, stacks

Analyzing multicore algorithms: functionality, linearizability, starvation- and wait-freeness, determine efficiency gain of parallelism

Bottlenecks: Amdahl's law, deadlock, lost wakeup, ABA problem

Multicore programming: hands-on experience, experimentation, thread pools in Java, insight into algorithms and datastructures