

# MULTI-CORE PROGRAMMING

Tim Harris

**Microsoft**

# Course overview

- Building shared memory data structures
  - Lists, queues, hashtables, ...
- Why?
  - Used directly by applications (e.g., in C/C++, Java, C#, ...)
  - Used in the language runtime system (e.g., management of work, implementations of message passing, ...)
  - Used in traditional operating systems (e.g., synchronization between top/bottom-half code)
- Why not?
  - Don't think of "threads + shared data structures" as a default/good/complete/desirable programming model
  - It's better to have shared memory and not need it...

# What do we care about?

Is it correct?

What does it mean to be correct?  
e.g., if multiple concurrent threads are using iterators on a shared data structure at the

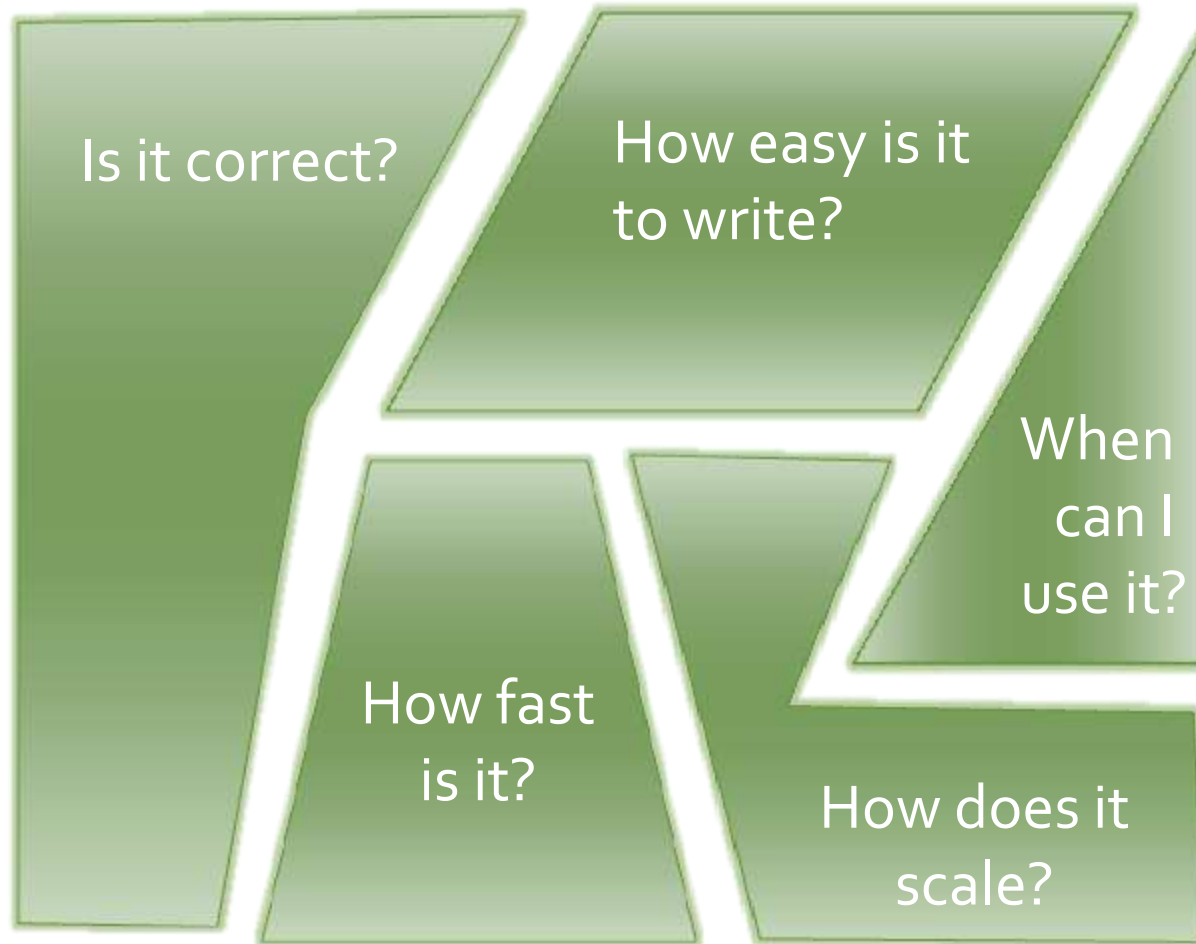
How does performance change as we increase the number of threads? When does the implementation add or avoid synchronization?

the same processes within an er? kind of

Suppose I have a sequential implementation (no concurrency control at all): is the new implementation 5% slower? 5x slower? 100x slower?

runtime system support?

# What do we care about?



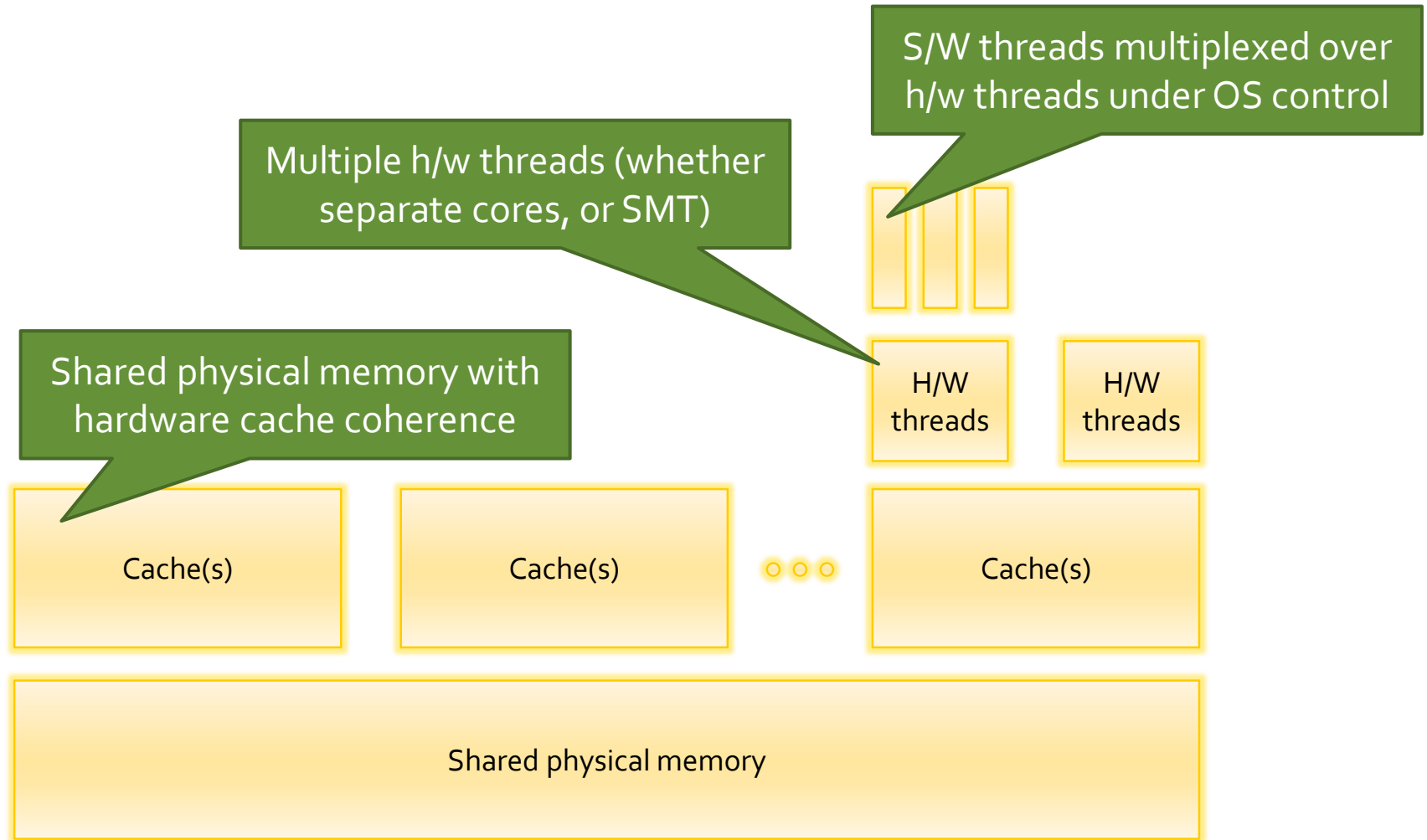
# What should we do?

1. Be explicit about goals and trade-offs
  - A benefit in one dimension often has costs in another
  - Does a perf increase prevent a data structure being used in some particular setting?
  - Does a technique to make something easier to write make the implementation slower?
  - Do we care? It depends on the setting
2. Remember, parallel programming is rarely a recreational activity
  - The ultimate goal is to increase perf (time, or resources used)
  - Does an implementation scale well enough to out-perform a good sequential implementation?

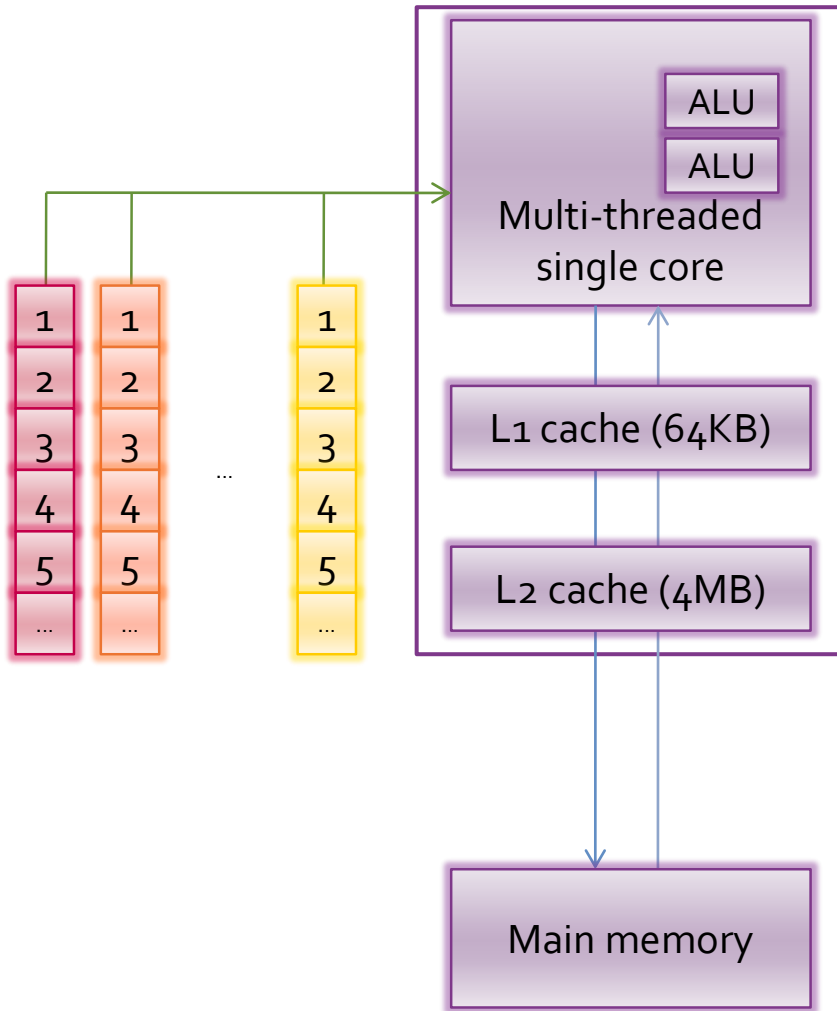
# Suggested reading

- “The art of multiprocessor programming”, Herlihy & Shavit – excellent coverage of shared memory data structures, from both practical and theoretical perspectives
- “Transactional memory, 2<sup>nd</sup> edition”, Harris, Larus, Rajwar – recently revamped survey of TM work, with 350+ references
- “NOrec: streamlining STM by abolishing ownership records”, Dalessandro, Spear, Scott, PPOPP 2010
- “Simplifying concurrent algorithms by exploiting transactional memory”, Dice, Lev, Marathe, Moir, Nussbaum, Olszewski, SPAA 2010

# System model



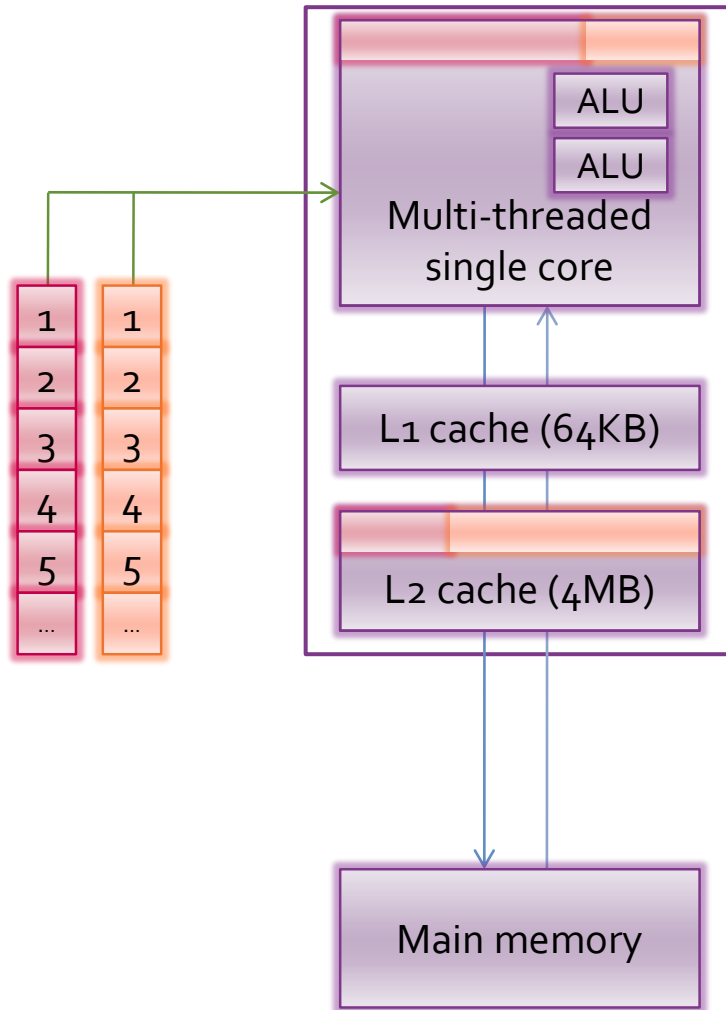
# Multi-threaded h/w



- Multiple threads in a workload with:
  - Poor spatial locality
  - Frequent memory accesses

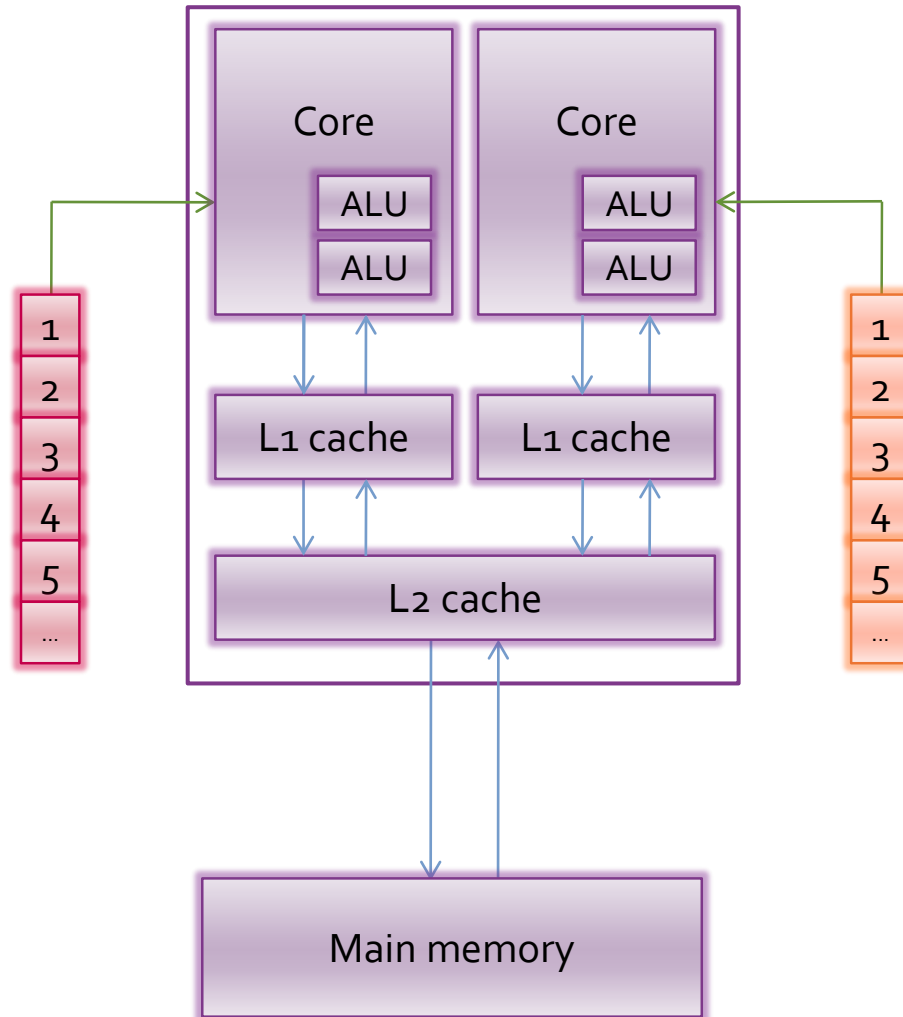


# Multi-threaded h/w

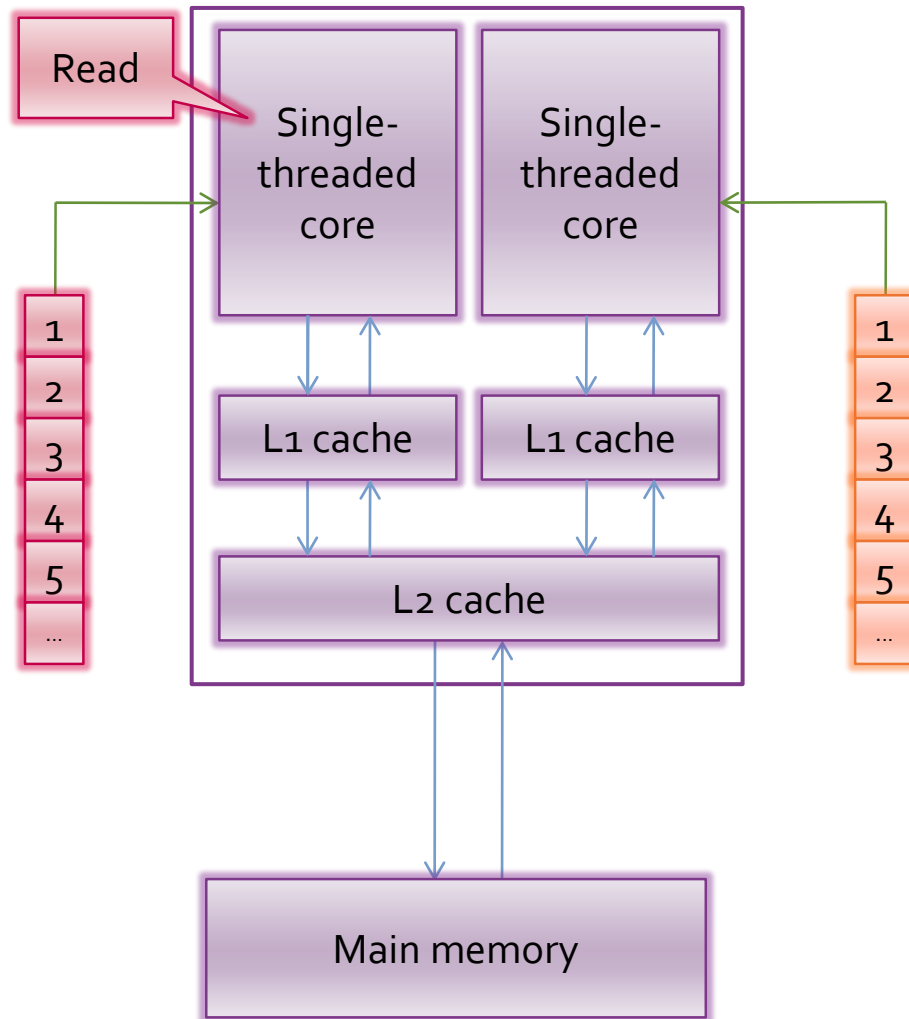


- Multiple threads with synergistic resource needs

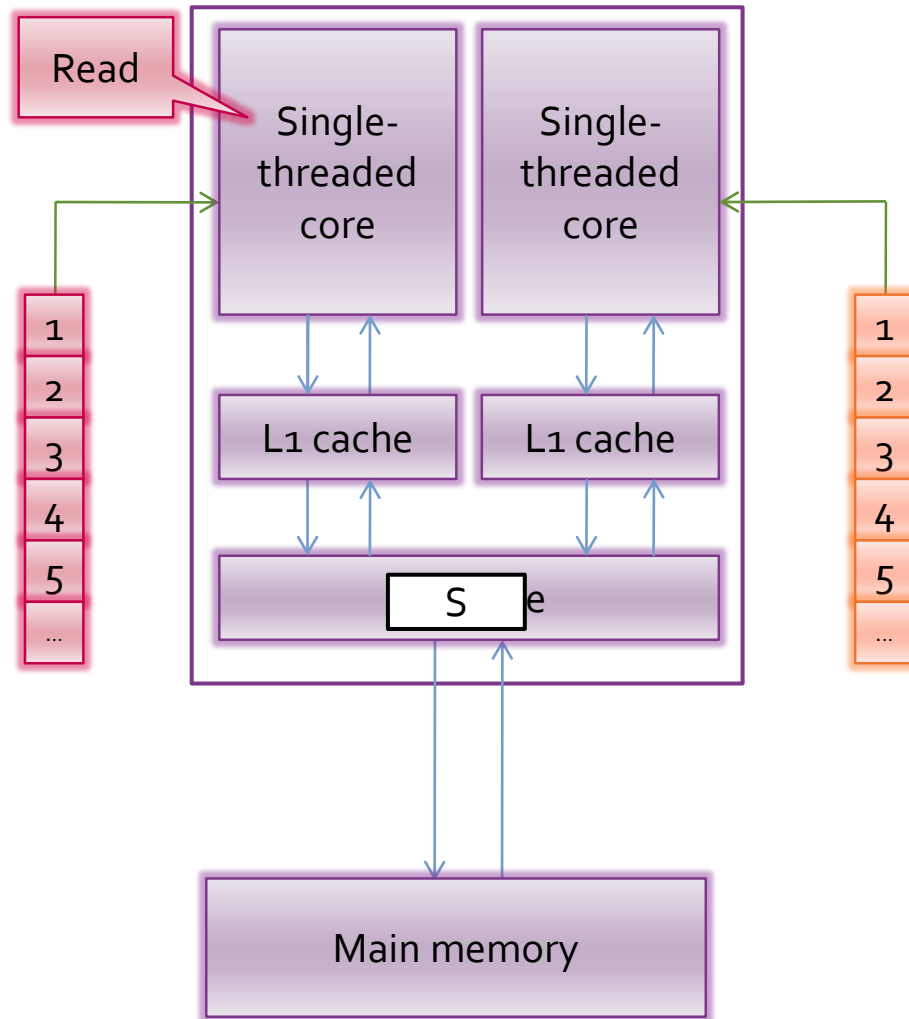
# Multi-core h/w – common L2



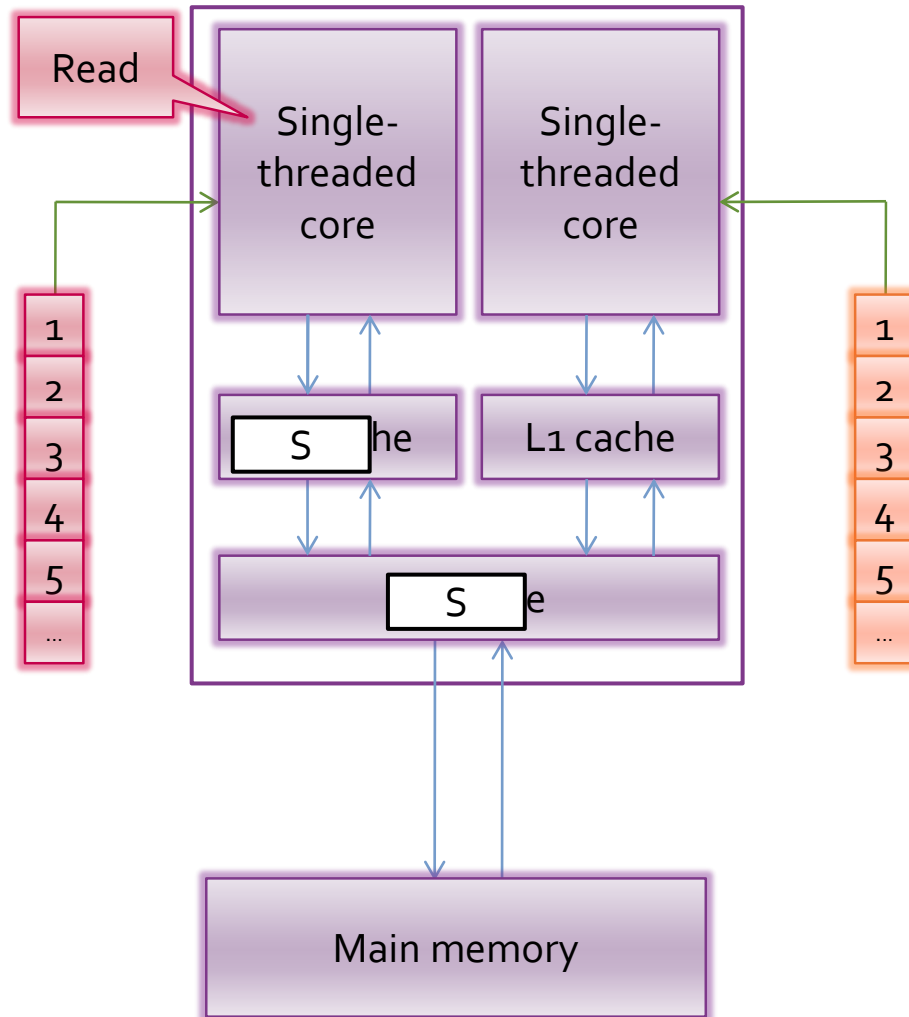
# Multi-core h/w – common L2



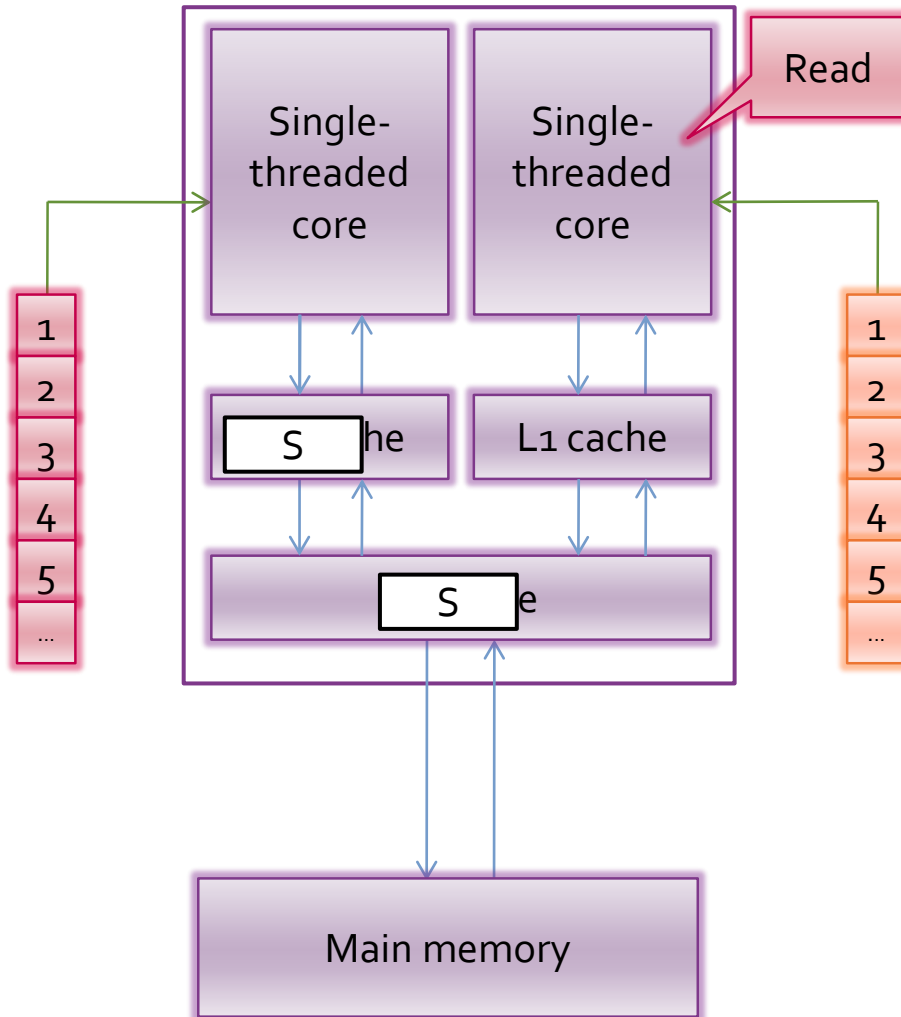
# Multi-core h/w – common L2



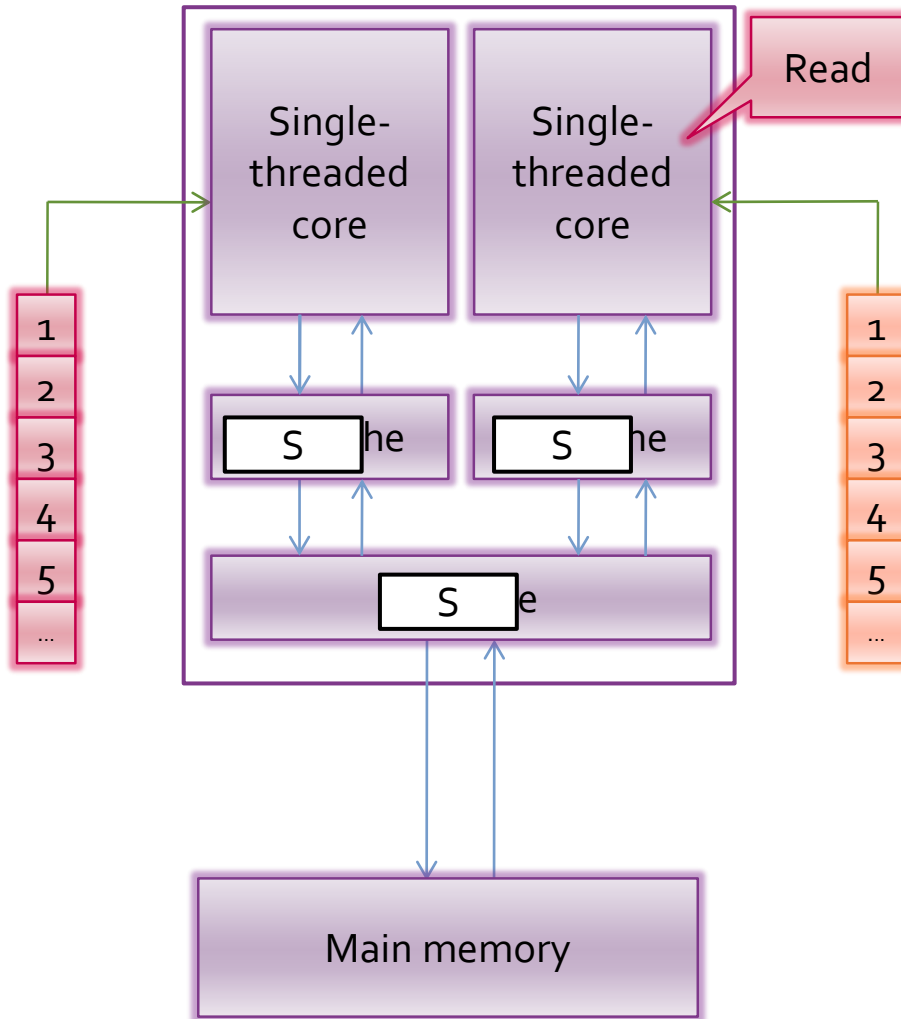
# Multi-core h/w – common L2



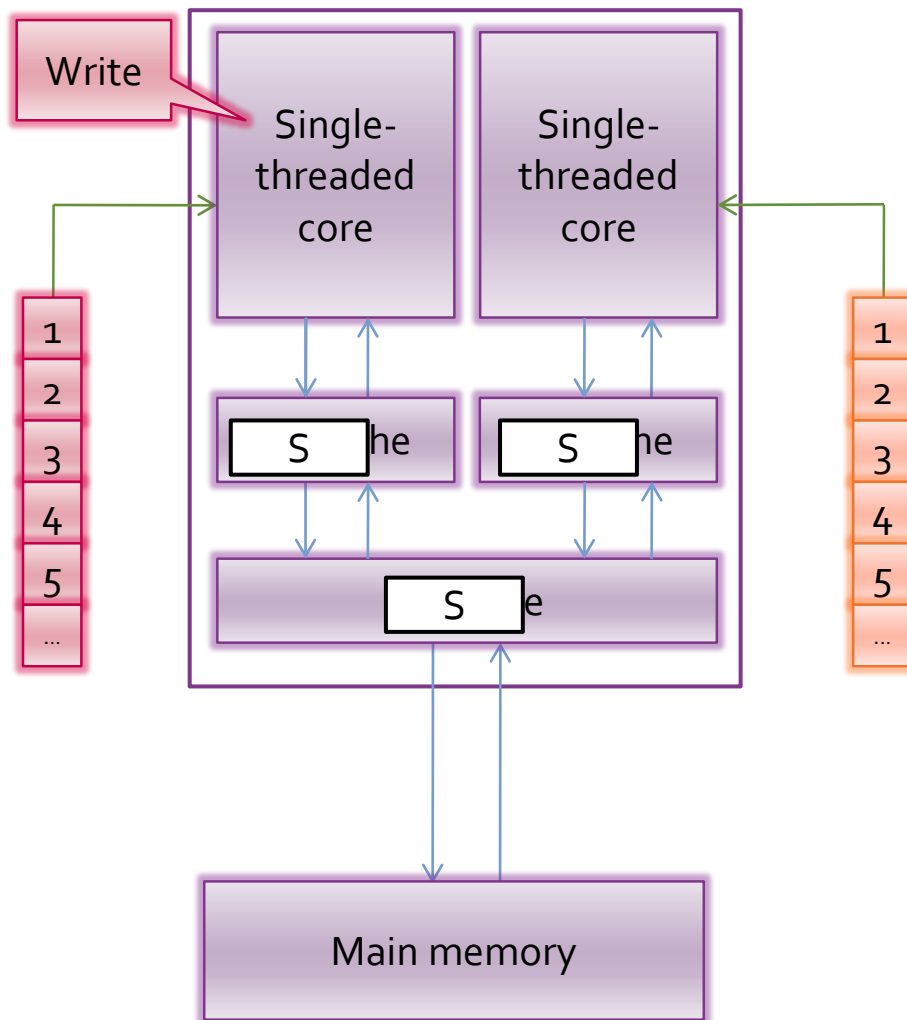
# Multi-core h/w – common L2



# Multi-core h/w – common L2

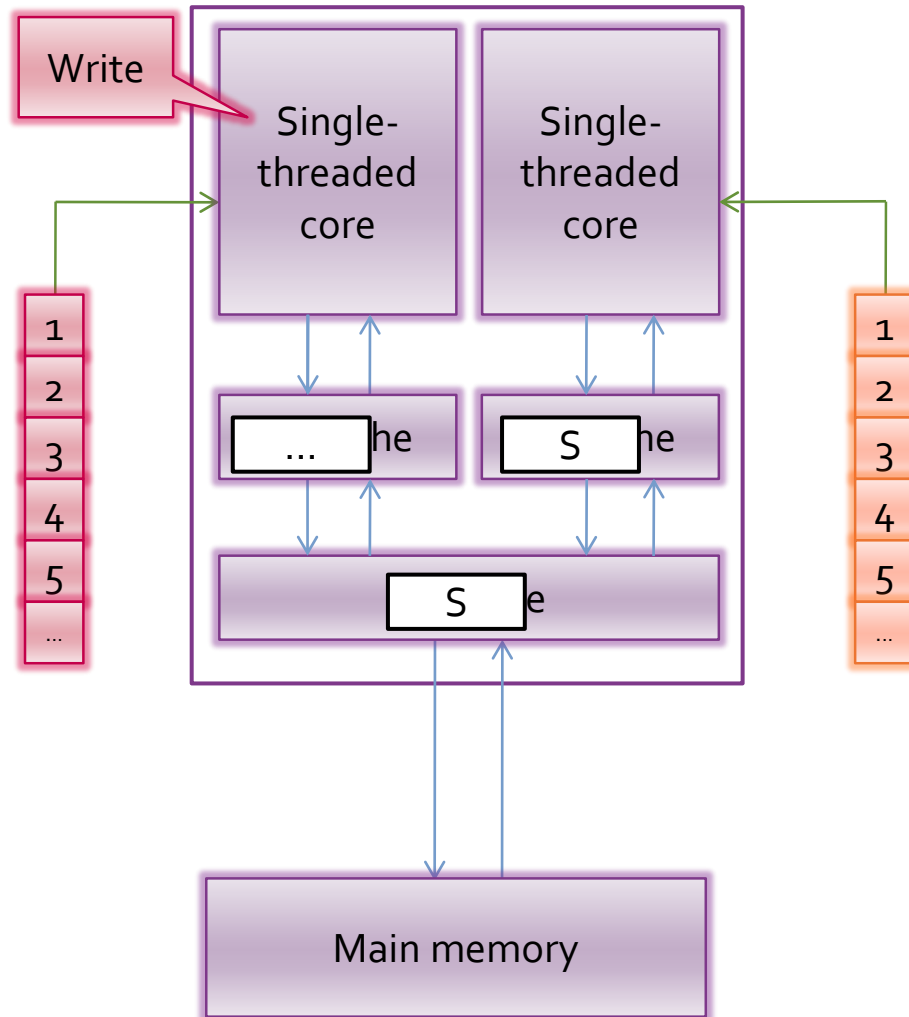


# Multi-core h/w – common L2

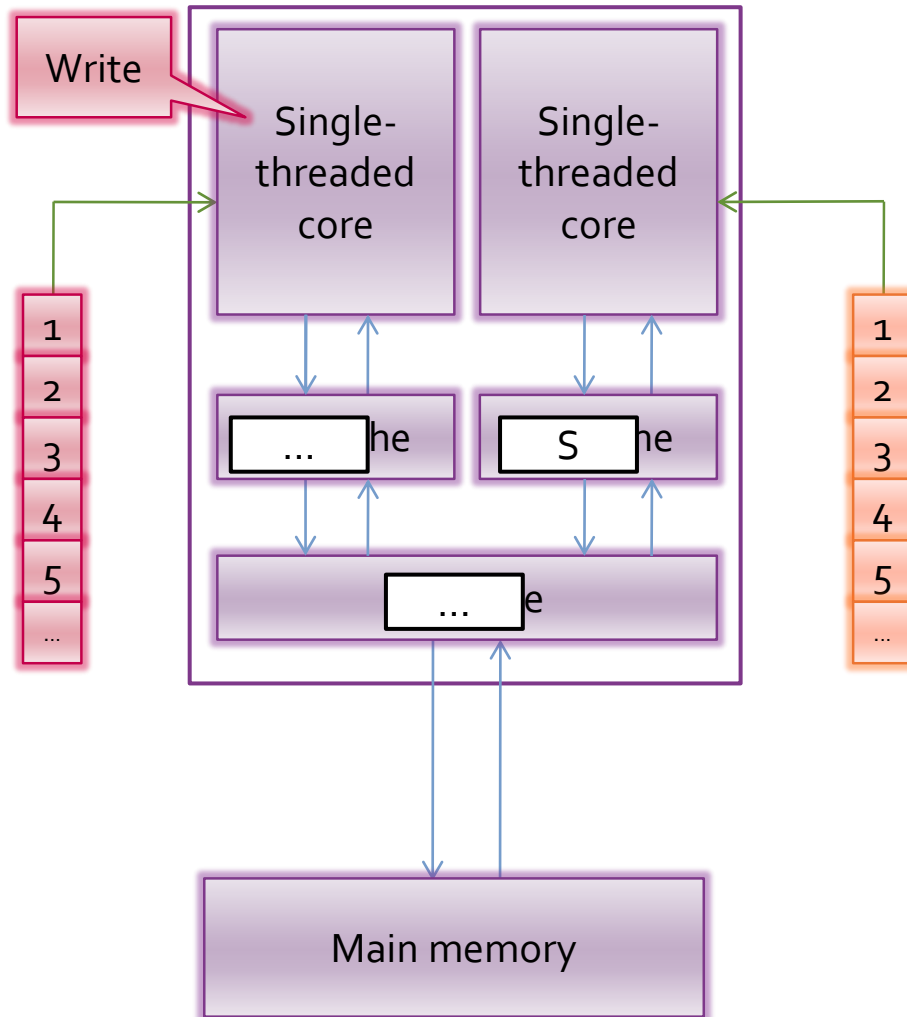




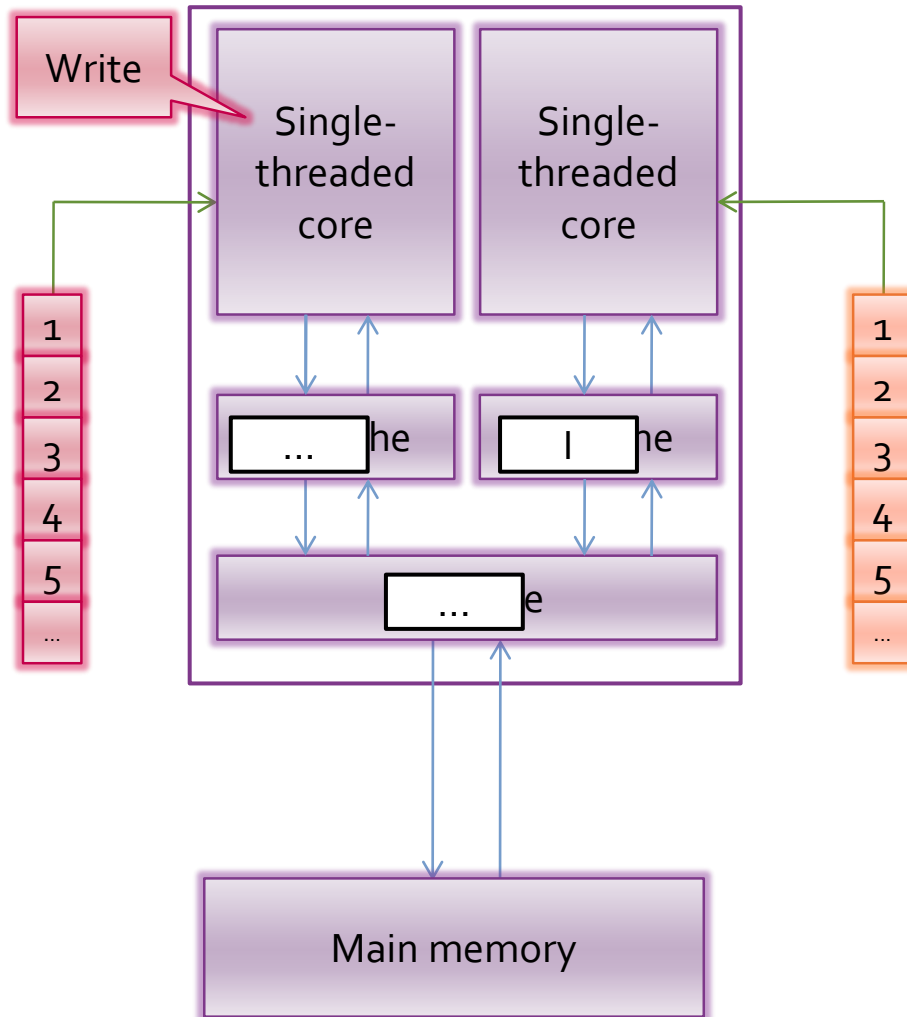
# Multi-core h/w – common L2



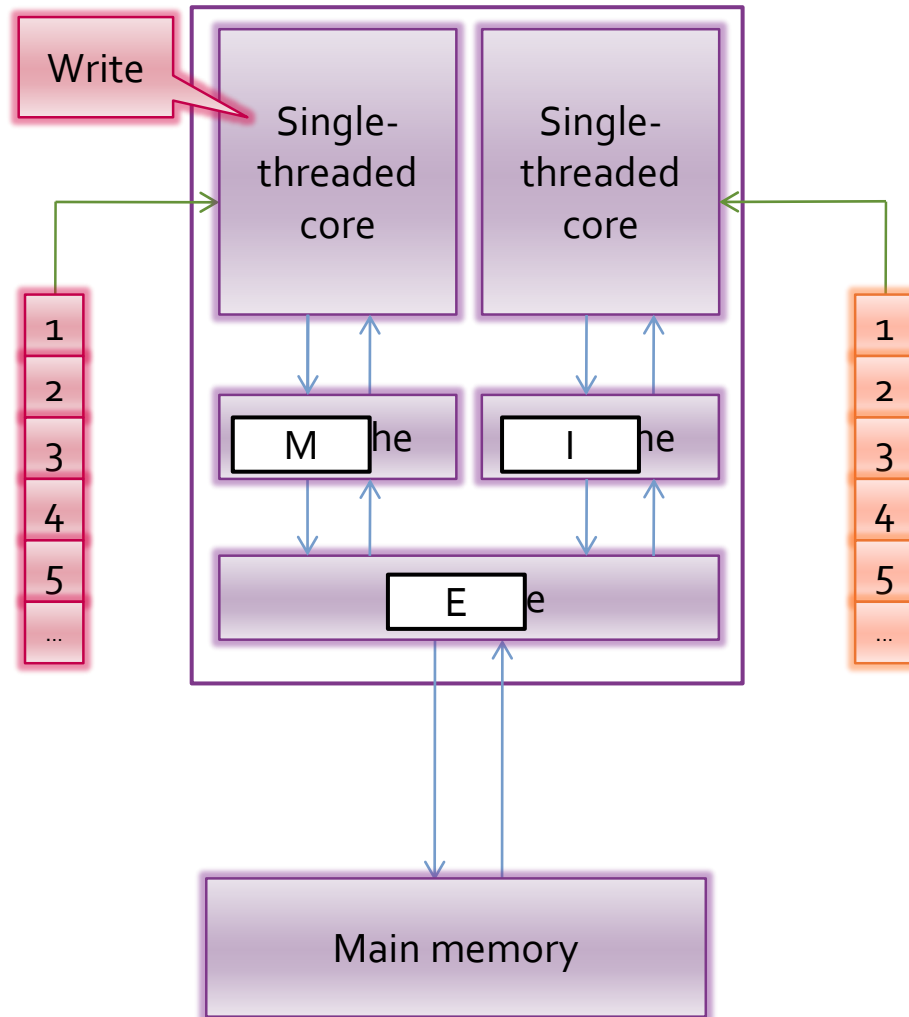
# Multi-core h/w – common L2



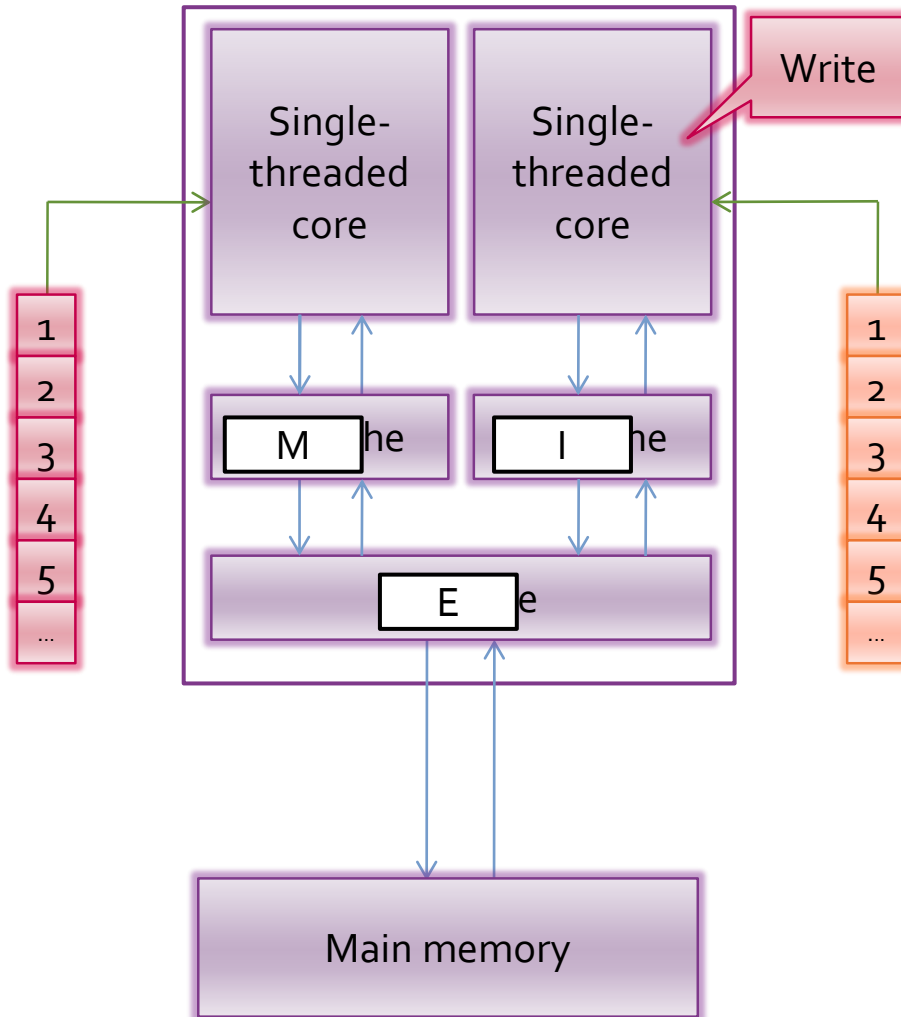
# Multi-core h/w – common L2



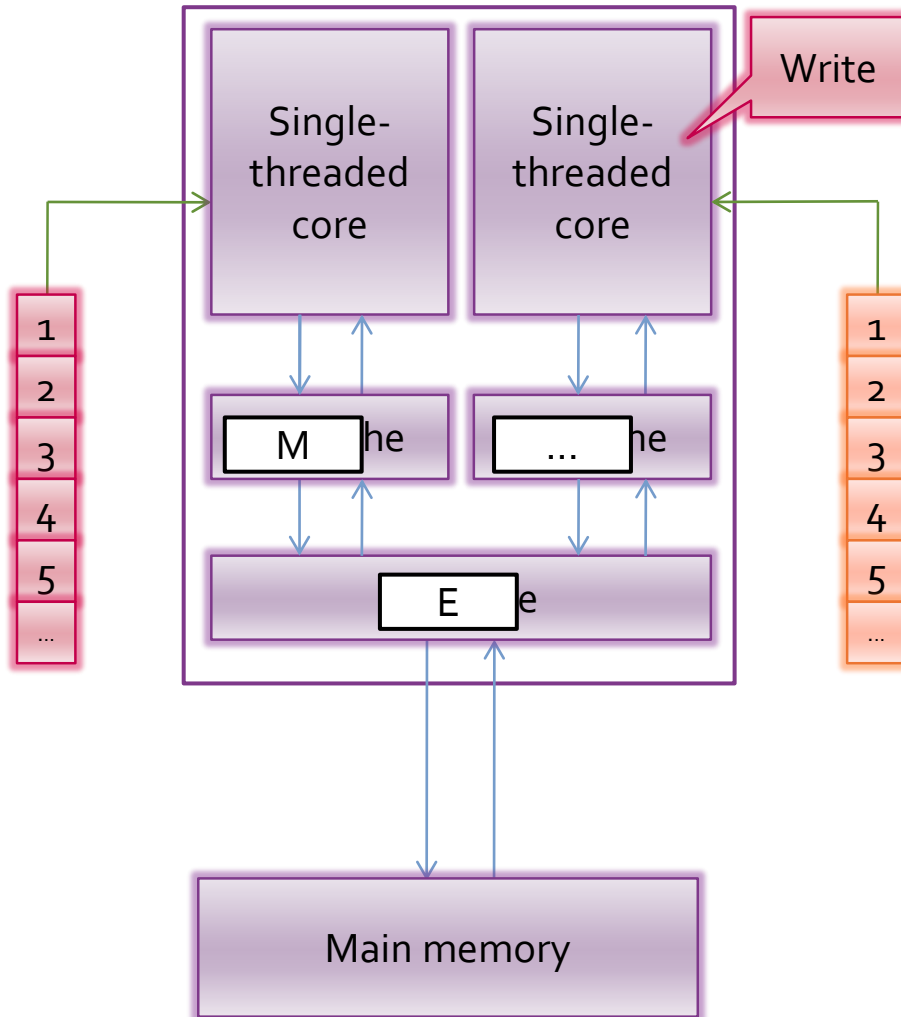
# Multi-core h/w – common L2



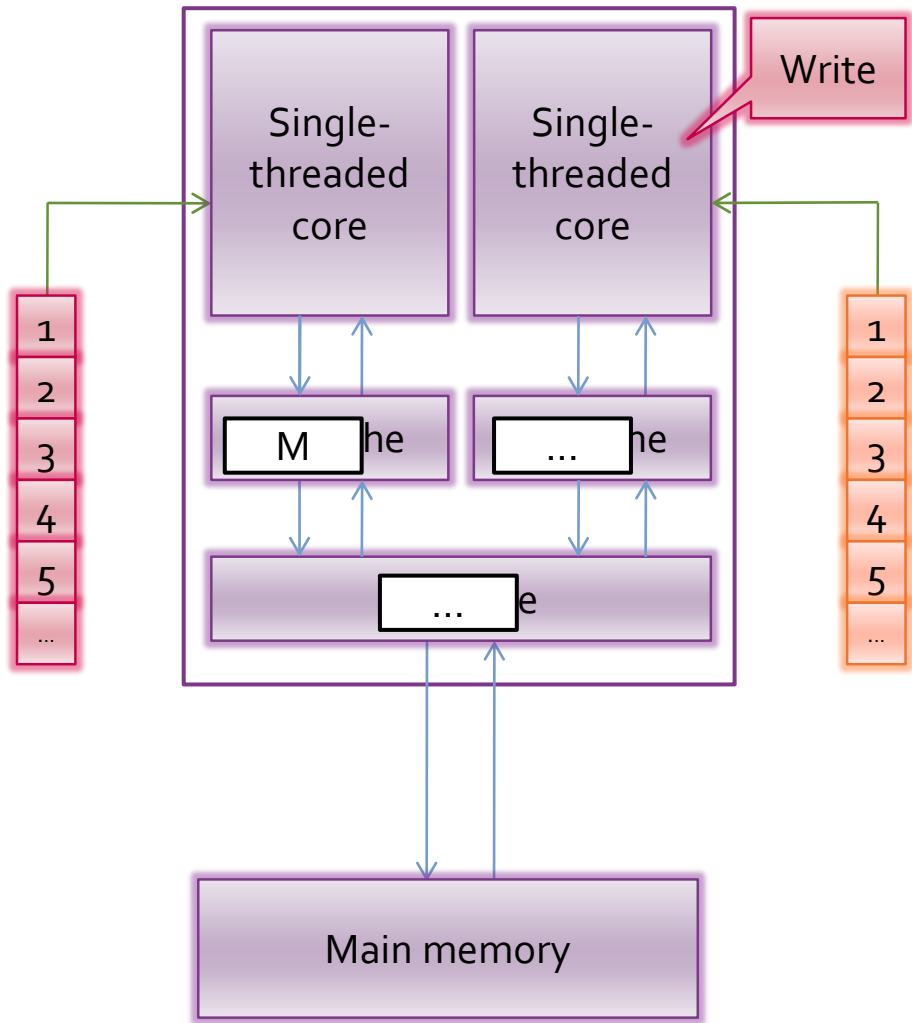
# Multi-core h/w – common L2



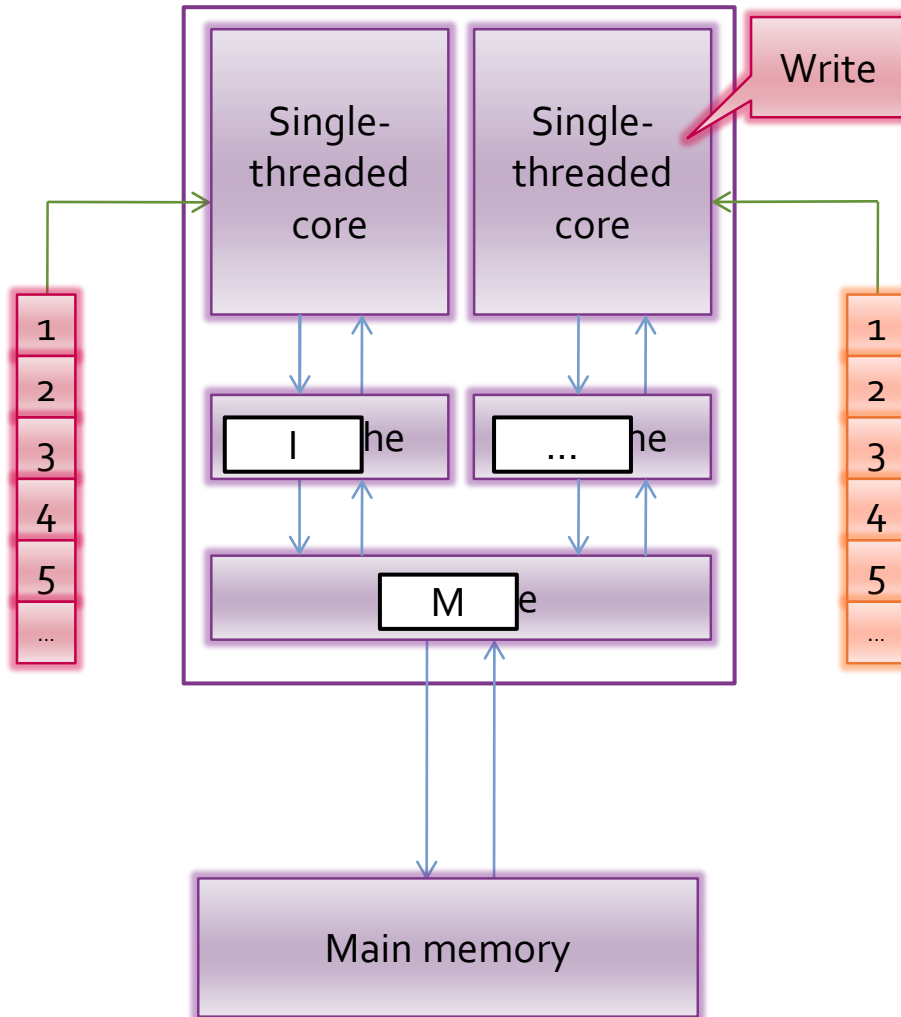
# Multi-core h/w – common L2



# Multi-core h/w – common L2

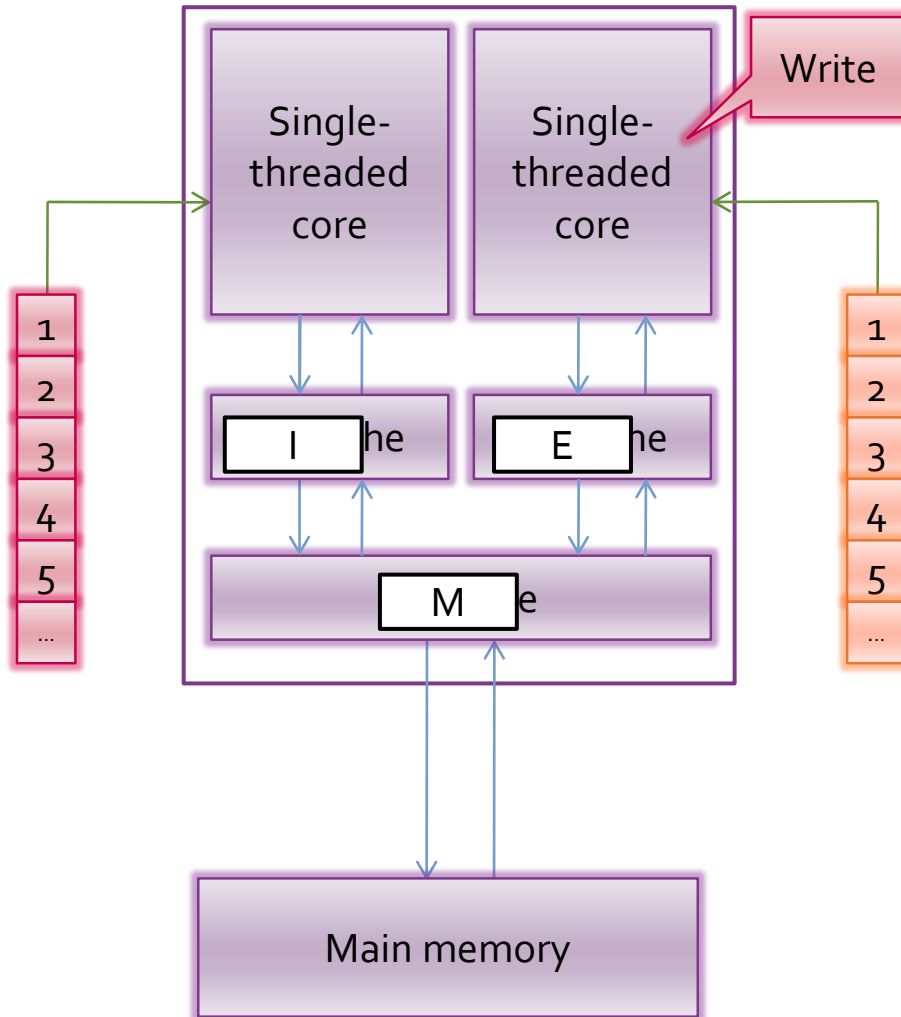


# Multi-core h/w – common L2

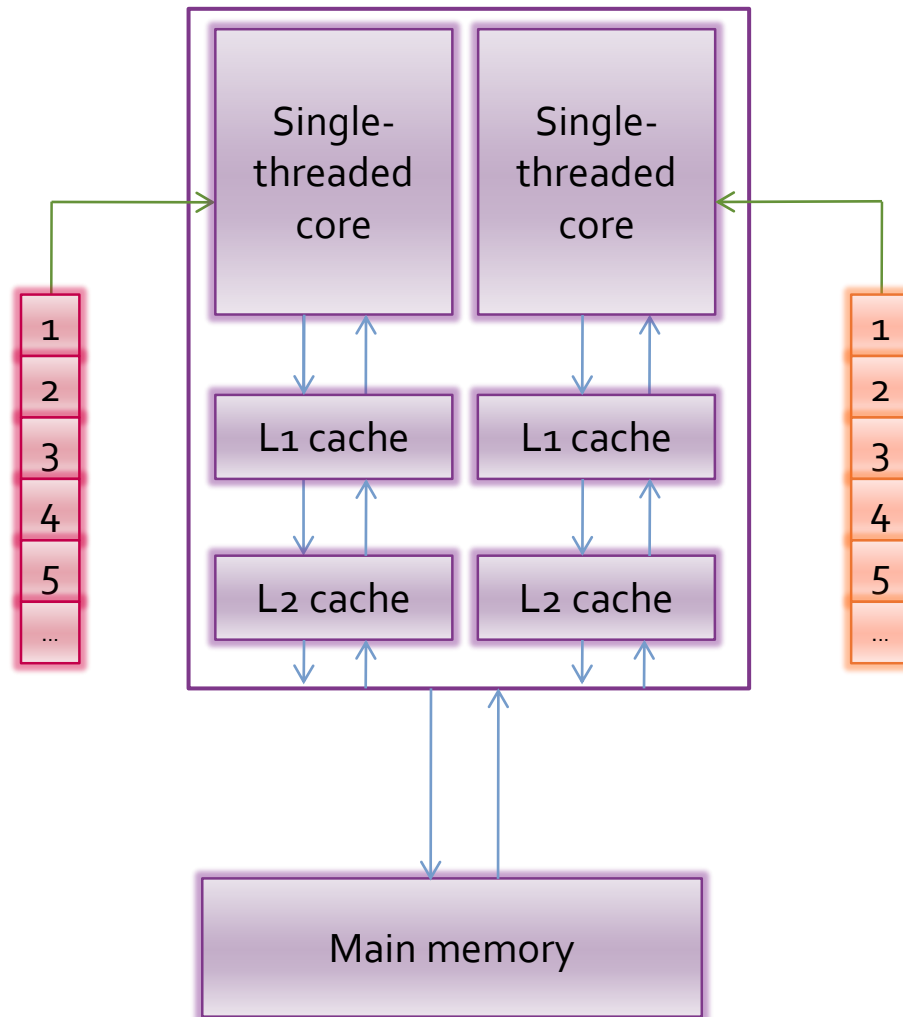




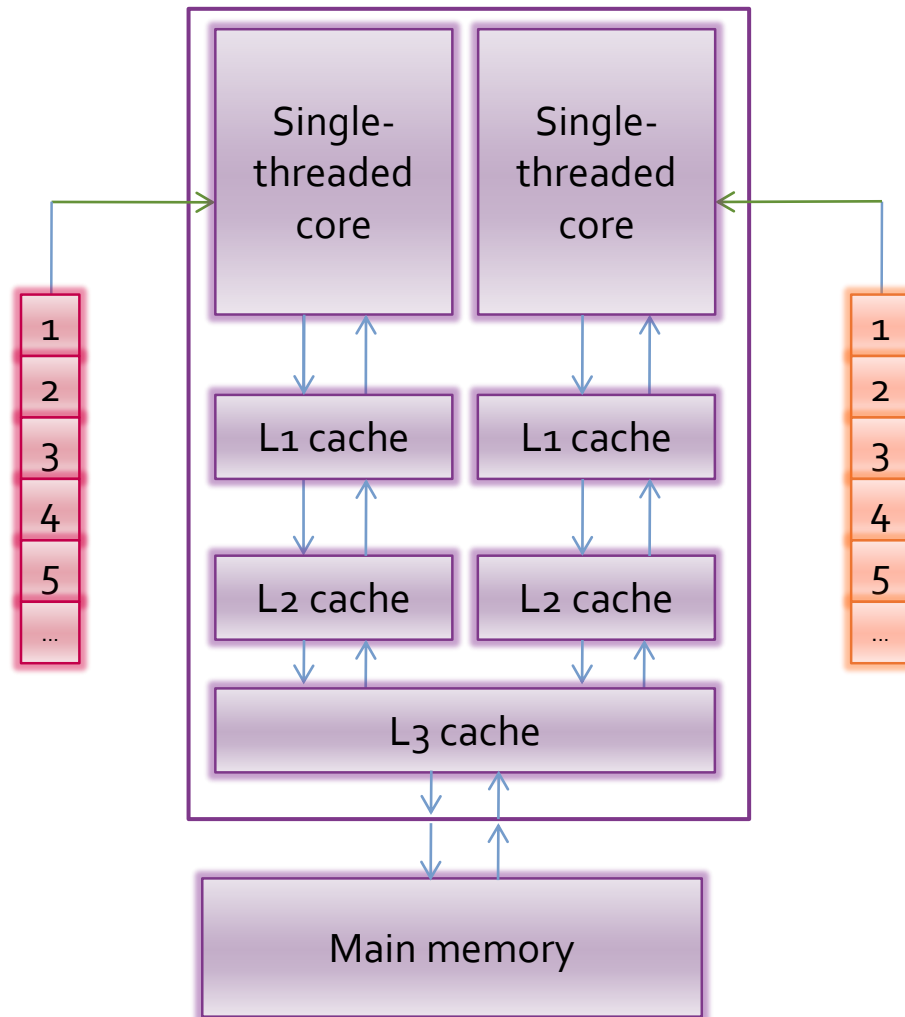
# Multi-core h/w – common L2



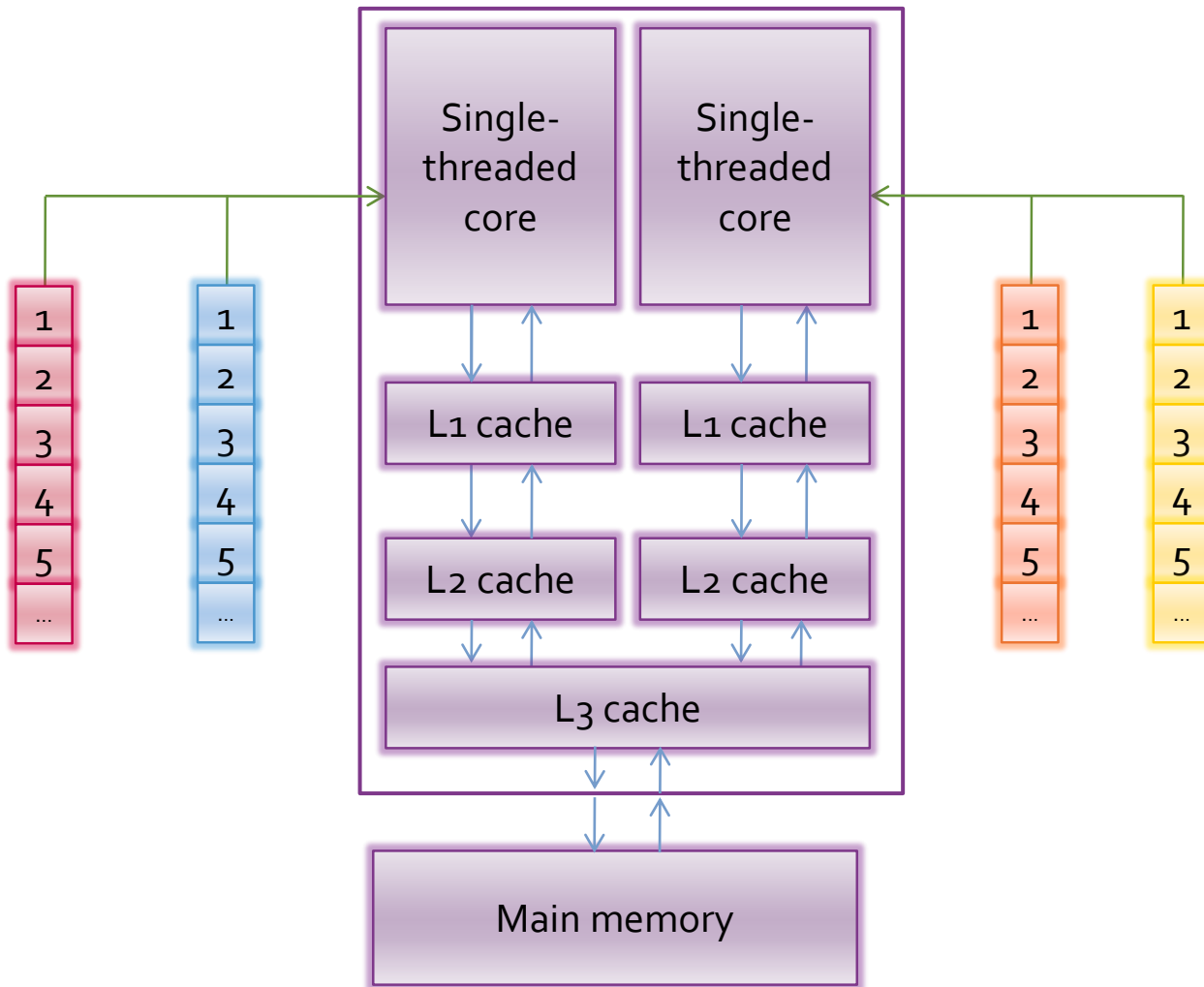
# Multi-core h/w – separate L2



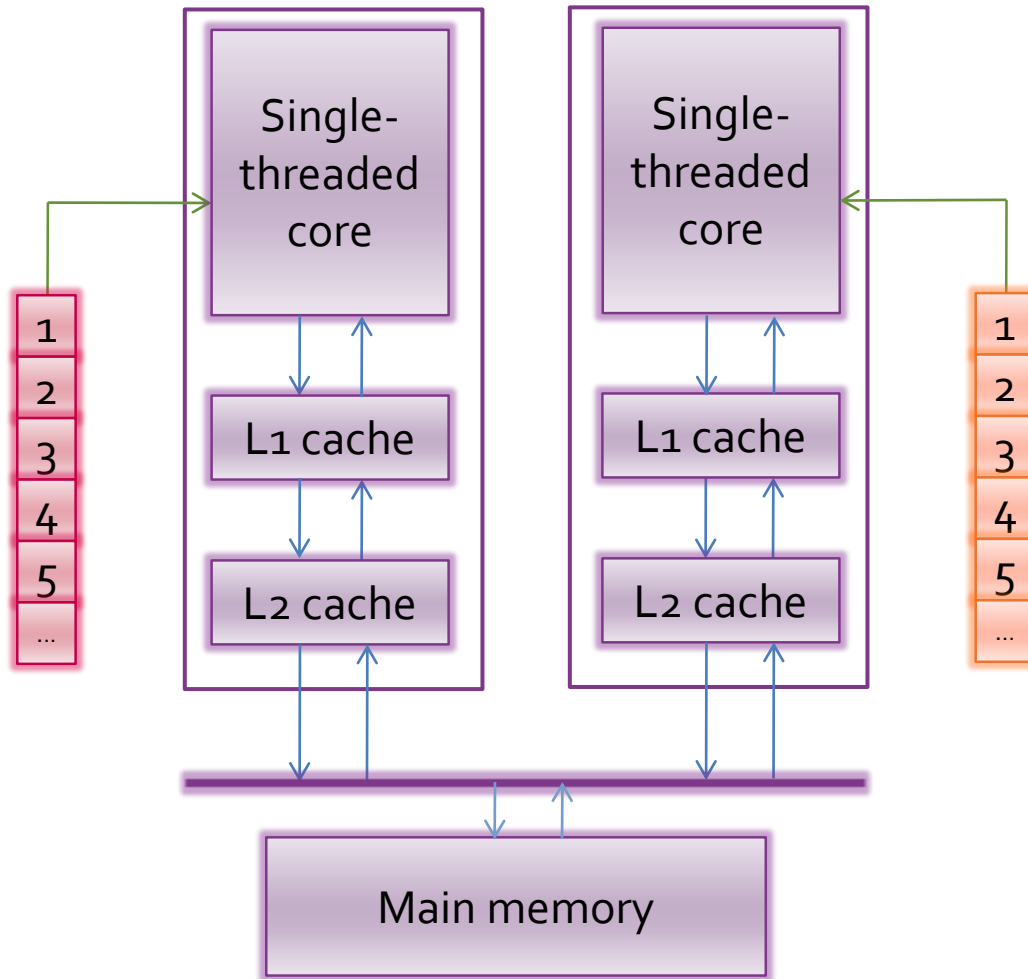
# Multi-core h/w – additional L3



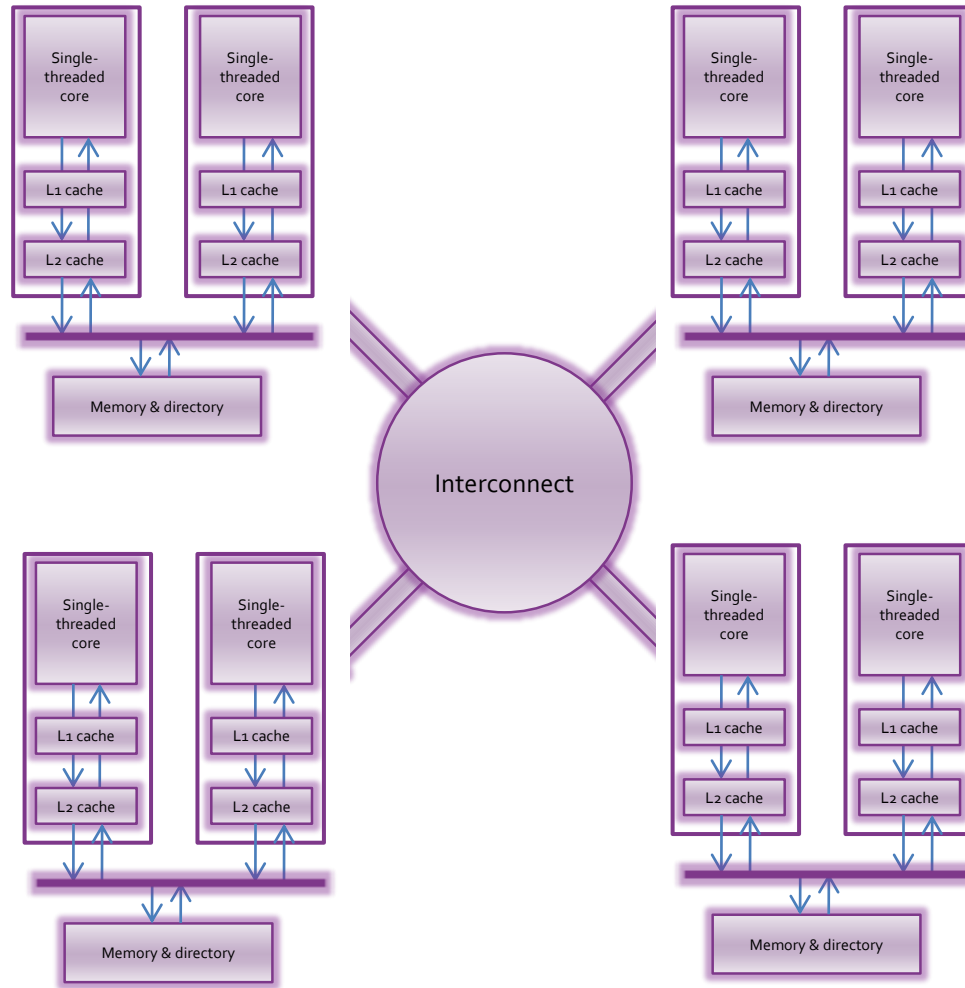
# Multi-threaded multi-core h/w



# SMP multiprocessor



# NUMA multiprocessor



# Three kinds of parallel hardware

- Multi-threaded cores
  - Increase utilization of a core or memory b/w
  - Peak ops/cycle fixed
- Multiple cores
  - Increase ops/cycle
  - Don't necessarily scale caches and off-chip resources proportionately
- Multi-processor machines
  - Increase ops/cycle
  - Often scale cache & memory capacities and b/w proportionately

# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory



# Course overview: structure

- Building locks
  - Test-and-set locks
  - TATAS locks & backoff
  - Queue-based locks
  - Hierarchical locks
  - Reader-writer locks
- Lock-free programming
- Transactional memory

# Test and set (pseudo-code)

```
bool testAndSet(bool *b) {  
    bool result;  
    atomic {  
        result = *b;  
        *b = TRUE;  
    }  
    return result;  
}
```

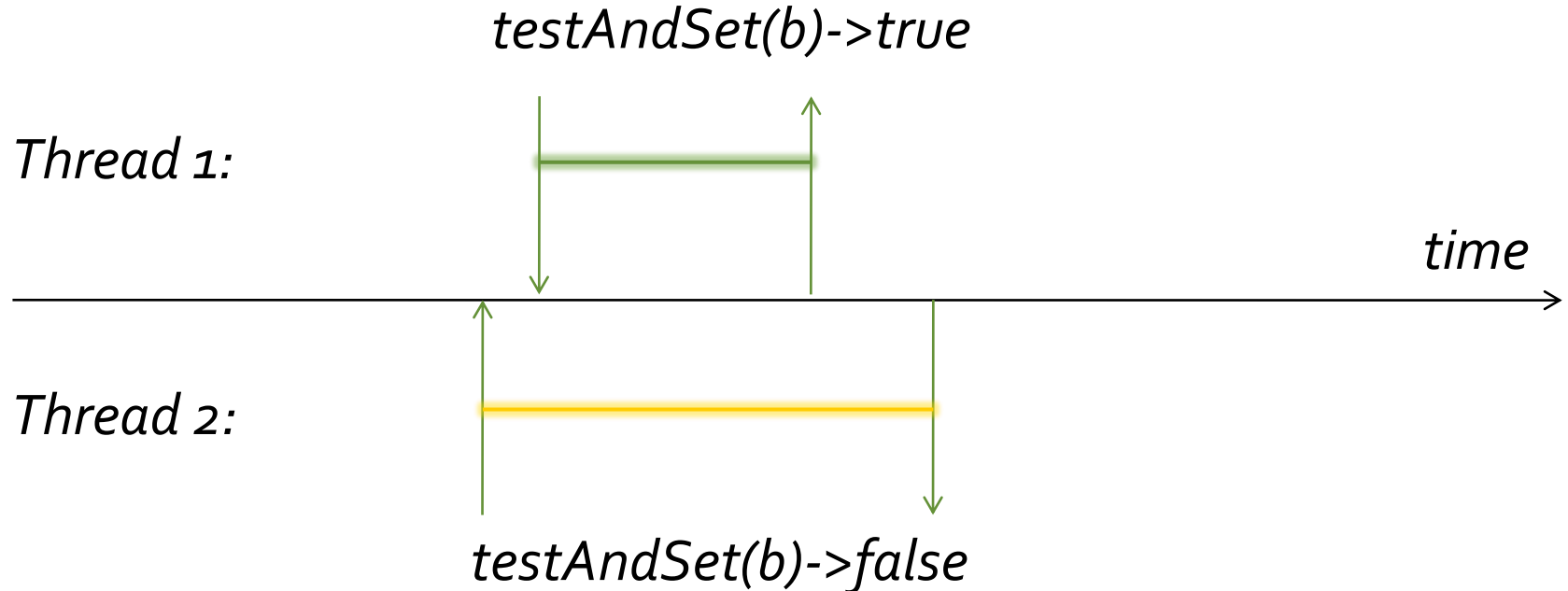
Pointer to a location holding a boolean value (TRUE/FALSE)

Read the current contents of the location b points to...

...set the contents of \*b to TRUE

# Test and set

- Suppose two threads use it at once



# Test and set lock

lock:

FALSE

FALSE => lock available  
TRUE => lock held

```
void acquireLock(bool *lock) {  
    while (testAndSet(lock)) {  
        /* Nothing */  
    }  
}
```

Each call tries to acquire the lock, returning TRUE if it is already held

```
void releaseLock(bool *lock) {  
    *lock = FALSE;  
}
```

NB: all this is pseudo-code, assuming SC memory

# Test and set lock

lock:

TRUE

```
void acquireLock(bool *lock) {  
    while (testAndSet(lock)) {  
        /* Nothing */  
    }  
}
```

```
void releaseLock(bool *lock) {  
    *lock = FALSE;  
}
```

Thread 1



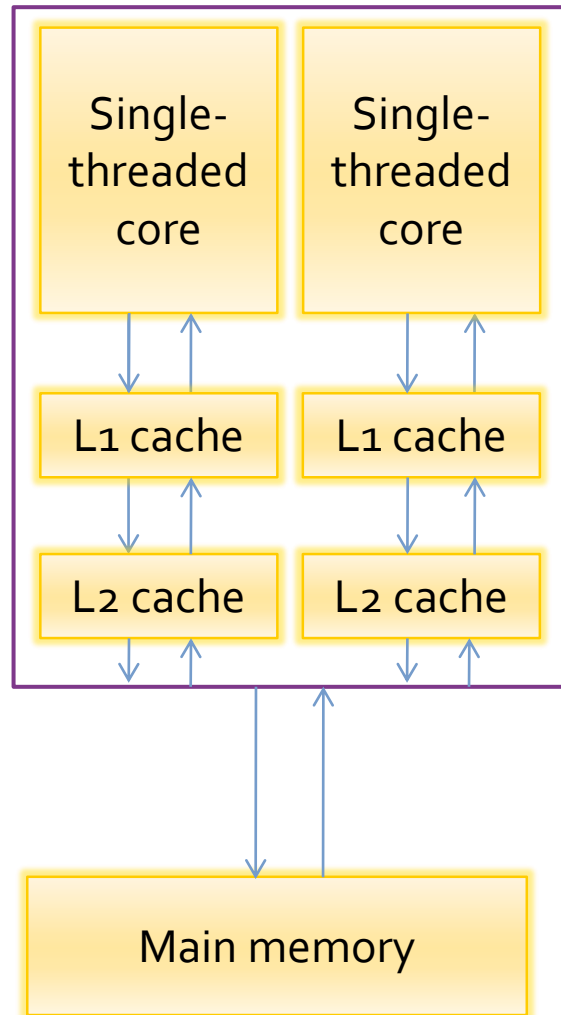
Thread 2



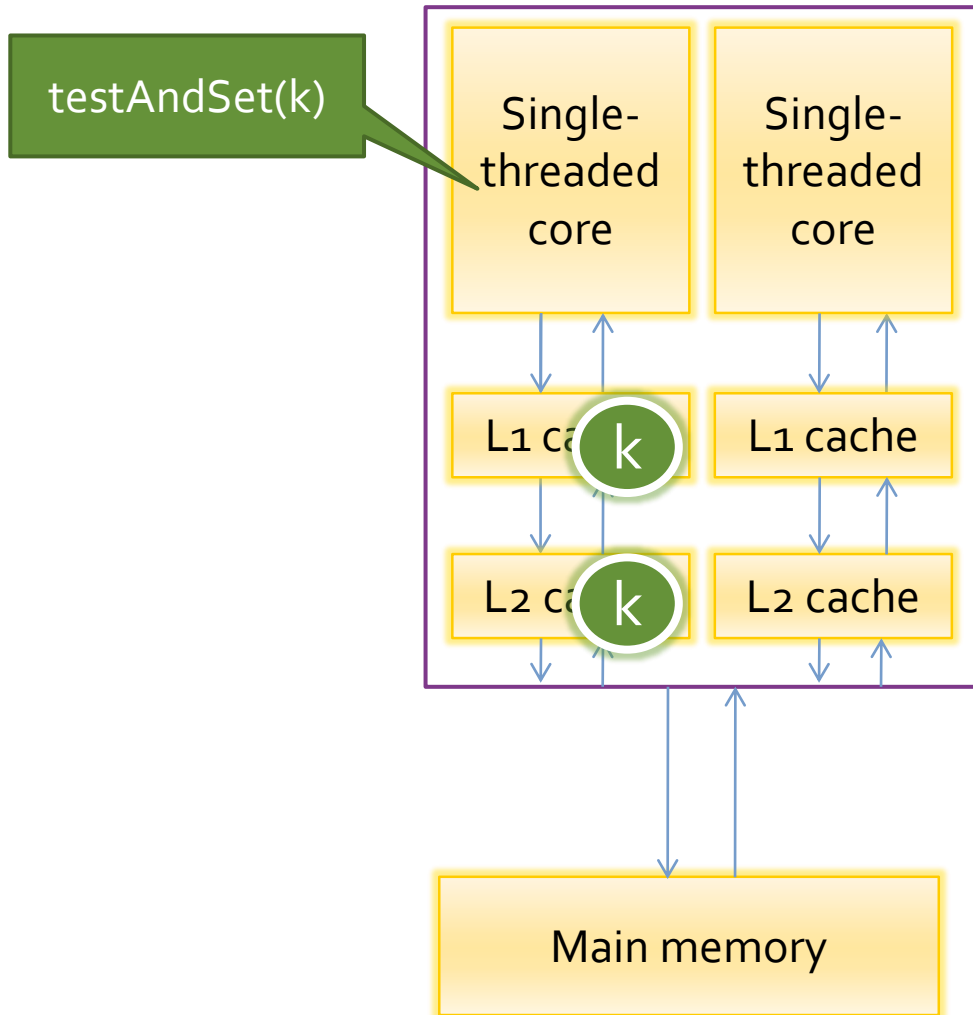
# What are the problems here?

testAndSet  
implementation  
causes contention

# Contention from testAndSet

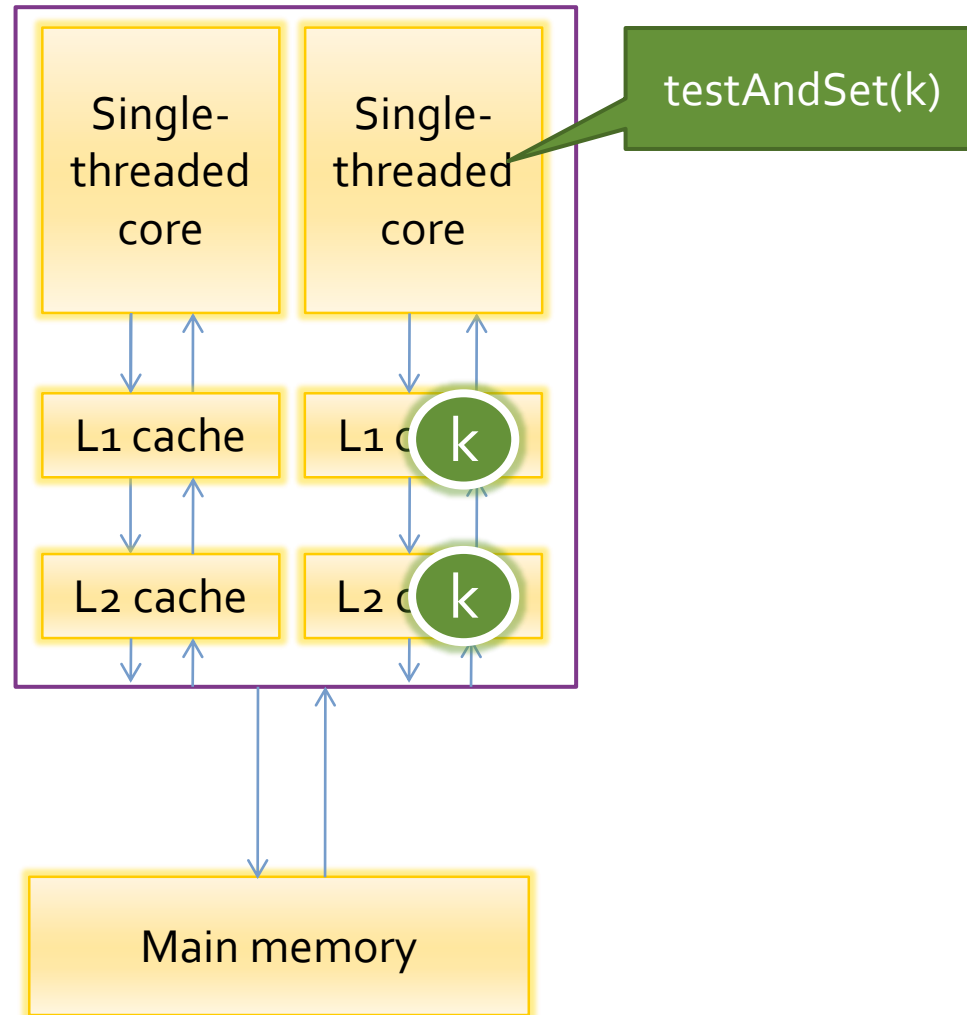


# Multi-core h/w – separate L2

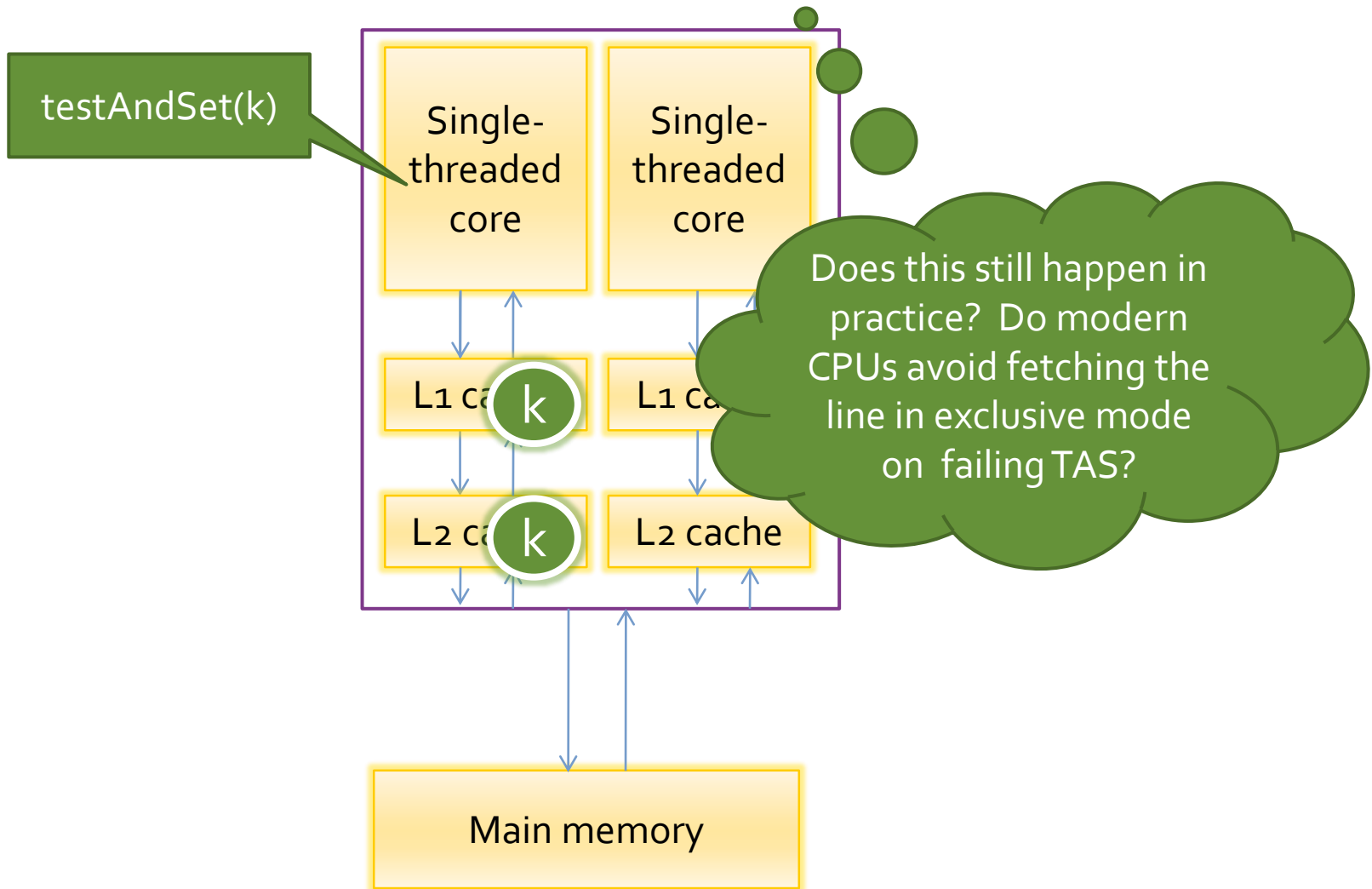




# Multi-core h/w – separate L2



# Multi-core h/w – separate L2



# What are the problems here?

testAndSet  
implementation  
causes contention

No control over  
locking policy

Only supports mutual  
exclusion: not reader-  
writer locking

Spinning may waste  
resources while  
waiting

# General problem

- No logical conflict between two failed lock acquires
- Cache protocol introduces a physical conflict
- For a good algorithm: only introduce physical conflicts if a logical conflict occurs
  - In a lock: successful lock-acquire & failed lock-acquire
  - In a set: successful insert(10) & failed insert(10)

# Course overview: structure

- Building locks
  - Test-and-set locks
  - TATAS locks & backoff
  - Queue-based locks
  - Hierarchical locks
  - Reader-writer locks
- Lock-free programming
- Transactional memory

# Test and test and set lock

lock:

FALSE

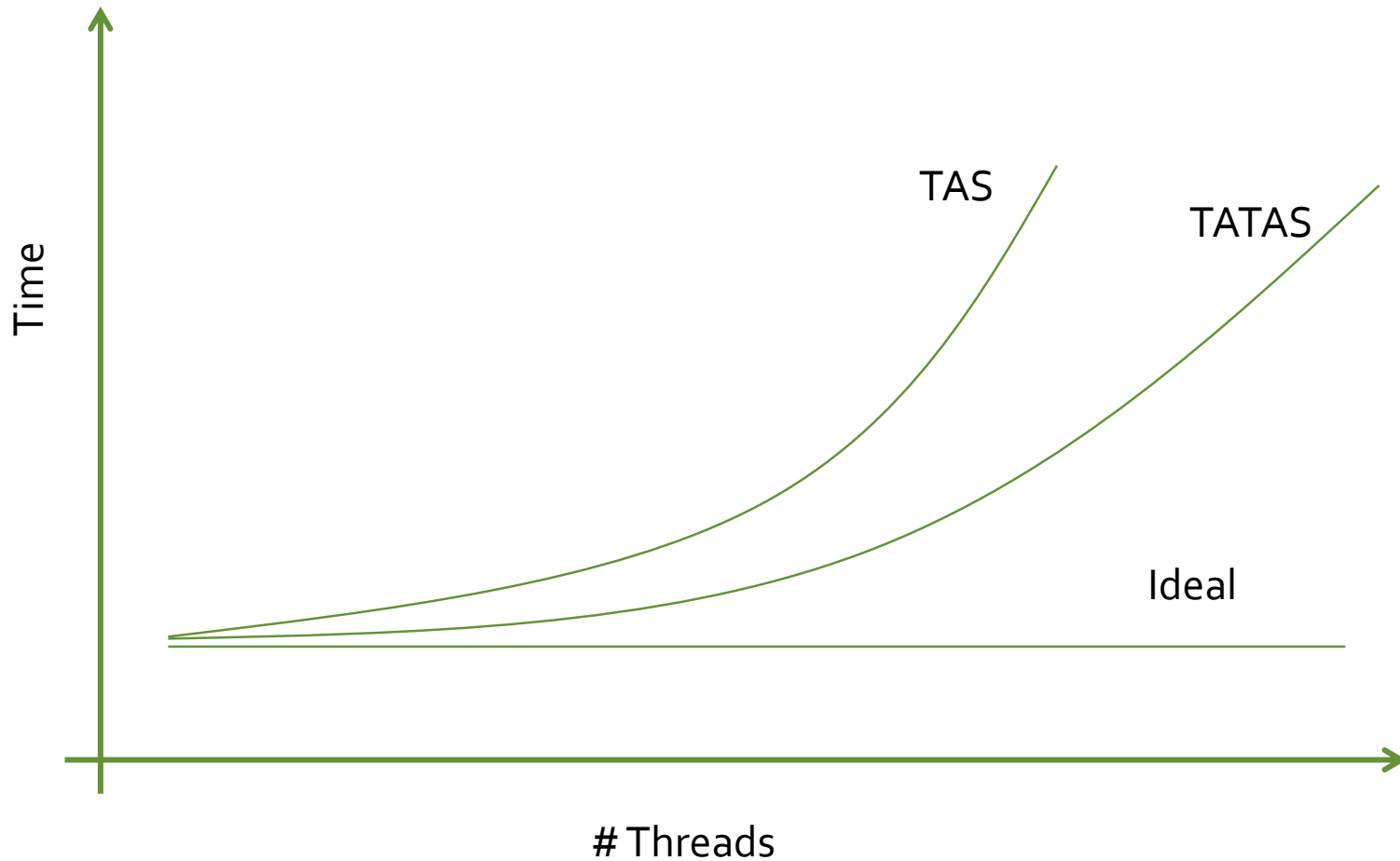
FALSE => lock available  
TRUE => lock held

```
void acquireLock(bool *lock) {  
  do {  
    while (*lock) { }  
  } while (testAndSet(lock));  
}
```

Spin while the lock is  
held... only do  
testAndSet when it is  
clear

```
void releaseLock(bool *lock) {  
  *lock = FALSE;  
}
```

# Performance



Based on Fig 7.4, Herlihy & Shavit, "The Art of Multiprocessor Programming"

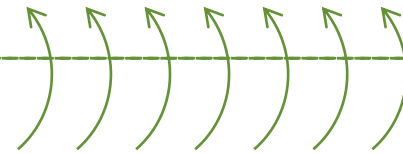
# Stampedes

lock:

TRUE

```
void acquireLock(bool *lock) {  
  do {  
    while (*lock) {}  
  } while (testAndSet(lock));  
}
```

```
void releaseLock(bool *lock) {  
  *lock = FALSE;  
}
```





# Back-off algorithms

1. Start by spinning, watching the lock for  $c$
2. If the lock does not become free, spin locally for  $s$  (*without watching the lock*)

What should “ $c$ ” be?  
What should “ $s$ ” be?

# Time spent waiting “c”

- Lower values:
  - Less time to build up a set of threads that will stampede
  - Less contention in the memory system, if remote reads incur a cost
  - Risk of a delay in noticing when the lock becomes free
- Higher values:
  - Less likelihood of a delay between a lock being released and a waiting thread noticing

# Local spinning time “s”

- Lower values:
  - More responsive to the lock becoming available
- Higher values:
  - If the lock doesn't become available then the thread makes fewer accesses to the shared variable

# Methodical approach

- For a given workload and performance model:
  - What is the best that an oracle could do (i.e. given perfect knowledge of lock demands)?
  - How does a practical algorithm compare with this?
- Look for an algorithm with a bound between its performance and that of the oracle
- “Competitive spinning”

# Rule of thumb

- Spin for a duration that's comparable with the shortest back-off interval
- Exponentially increase the per-thread back-off interval (resetting it when the lock is acquired)
- Use a maximum back-off interval that is large enough that waiting threads don't interfere with the other threads' performance

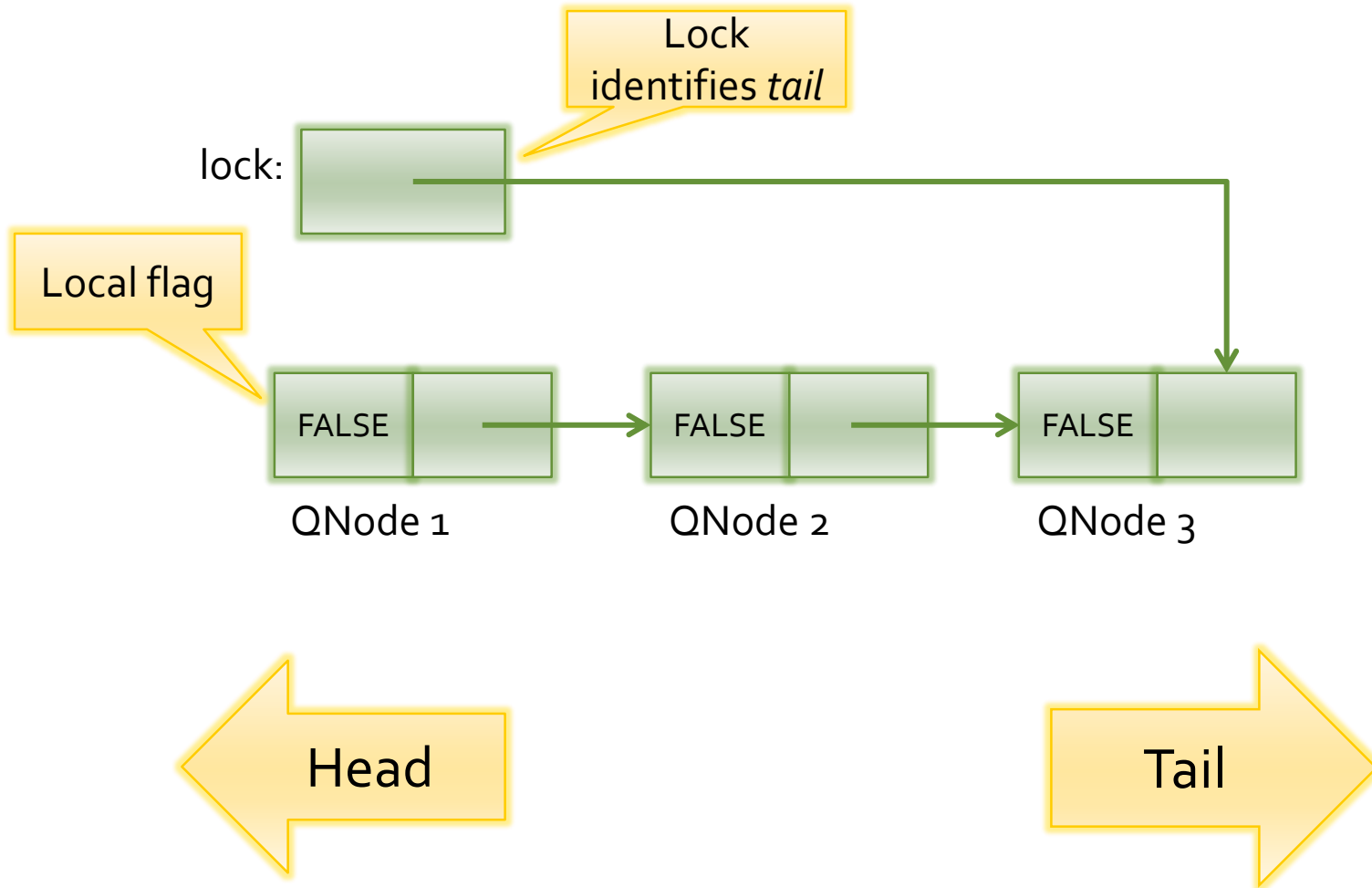
# Course overview: structure

- Building locks
  - Test-and-set locks
  - TATAS locks & backoff
  - Queue-based locks
  - Hierarchical locks
  - Reader-writer locks
- Lock-free programming
- Transactional memory

# Queue-based locks

- Lock holders queue up: immediately provides FCFS behavior
- Each spins *locally* on a flag in their queue entry: no remote memory accesses while waiting
- A lock release wakes the next thread directly: no stampede

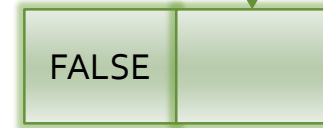
# MCS locks





# MCS lock acquire

lock:



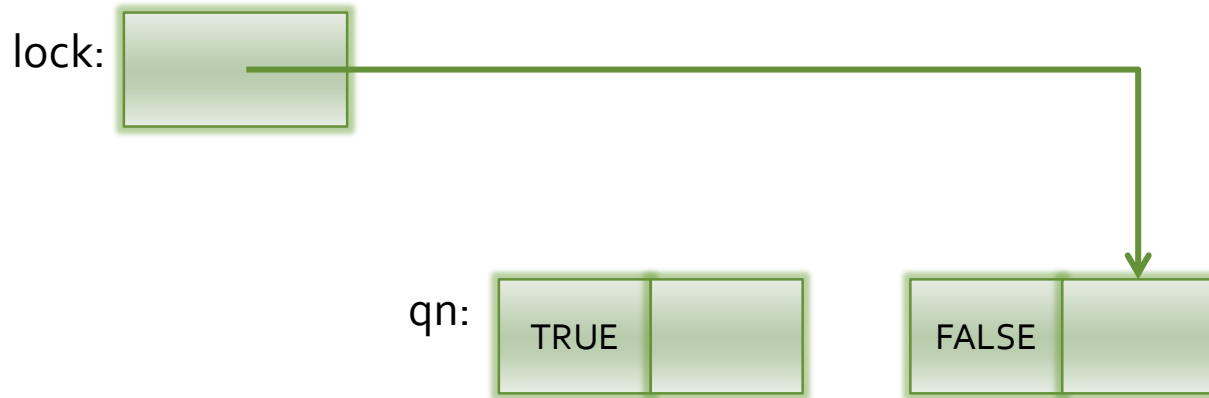
```
void acquireMCS(mcs *lock, QNode *qn) {
    QNode *prev;
    qn->flag = false;
    qn->next = NULL;
    while (true) {
        prev = lock->tail;
        if (CAS(&lock->tail, prev, qn)) break;
    }
    if (prev != NULL) {
        prev->next = qn;
        while (!qn->flag) { } // Spin
    }
}
```

Find previous  
tail node

Atomically replace  
"prev" with "qn" in  
the lock itself

Add link within  
the queue

# MCS lock release



```
void releaseMCS(mcs *lock, QNode *qn) {  
    if (lock->tail = qn) {  
        if (CAS(&lock->tail, qn, NULL)) return;  
    }  
    while (qn->next == NULL) { }  
    qn->next->flag = TRUE;  
}
```

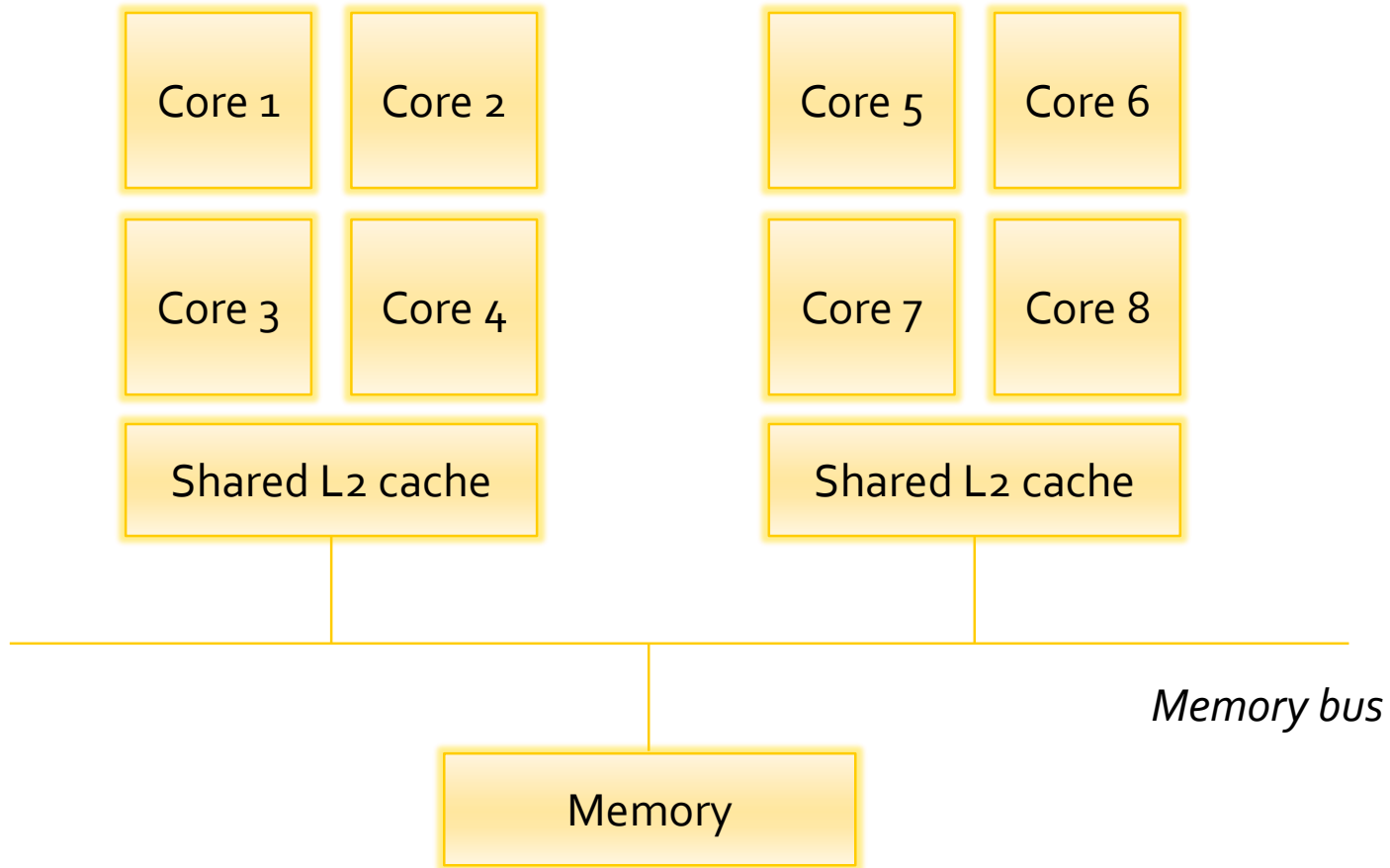
If we were at the tail  
then remove us

Wait for next lock holder  
to announce themselves;  
signal them

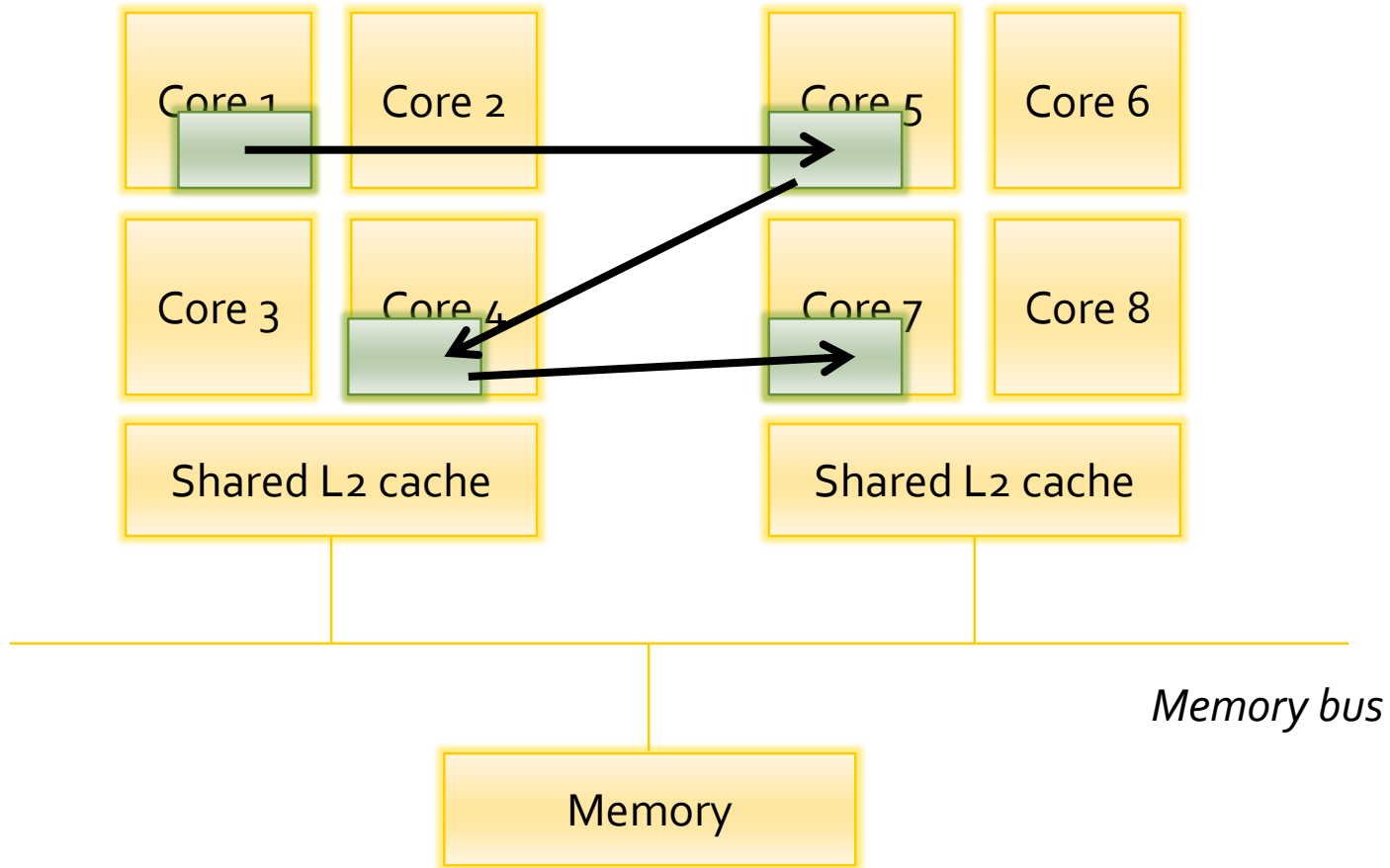
# Course overview: structure

- Building locks
  - Test-and-set locks
  - TATAS locks & backoff
  - Queue-based locks
  - Hierarchical locks
  - Reader-writer locks
- Lock-free programming
- Transactional memory

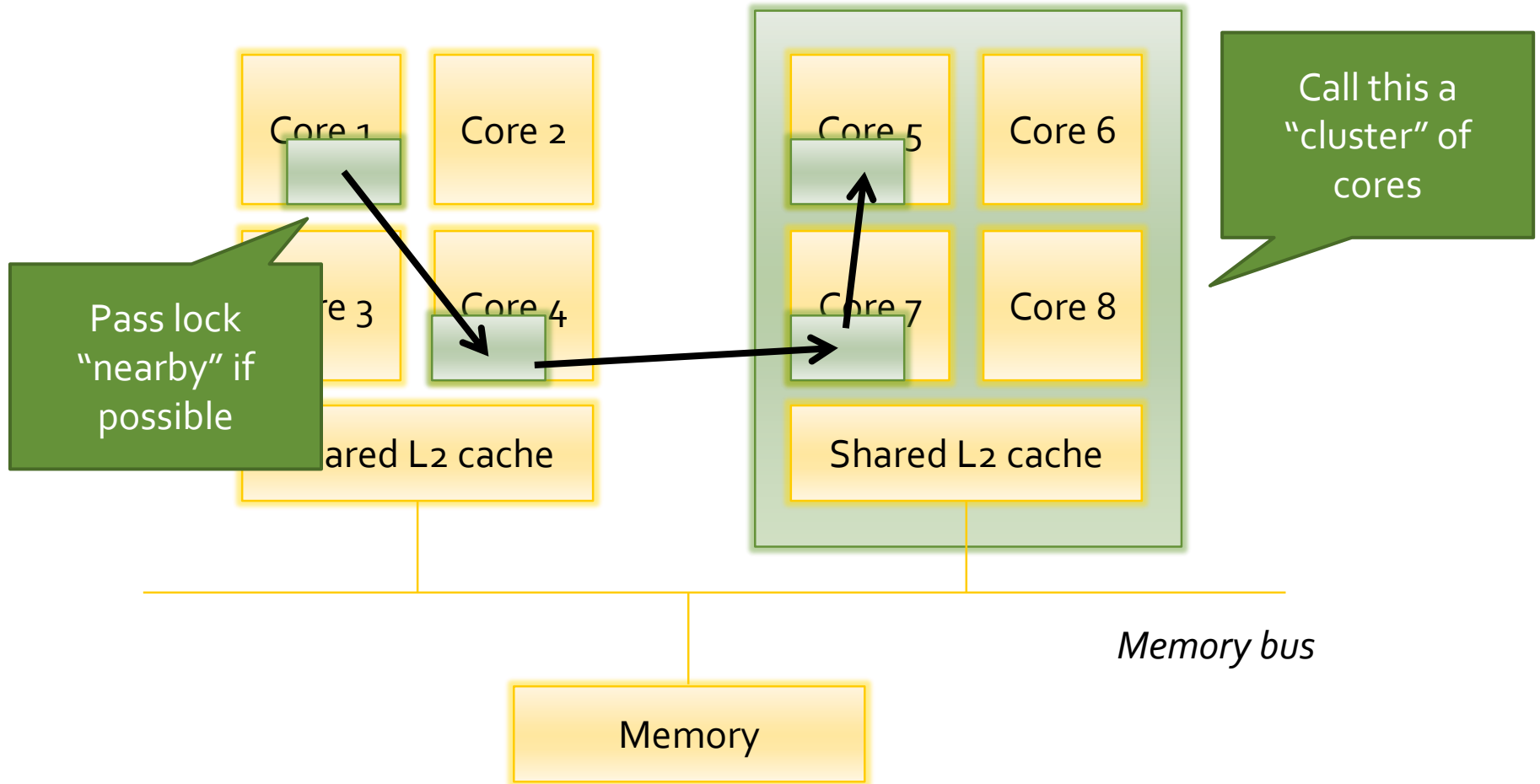
# Hierarchical locks



# Hierarchical locks



# Hierarchical locks



# Hierarchical TATAS with backoff

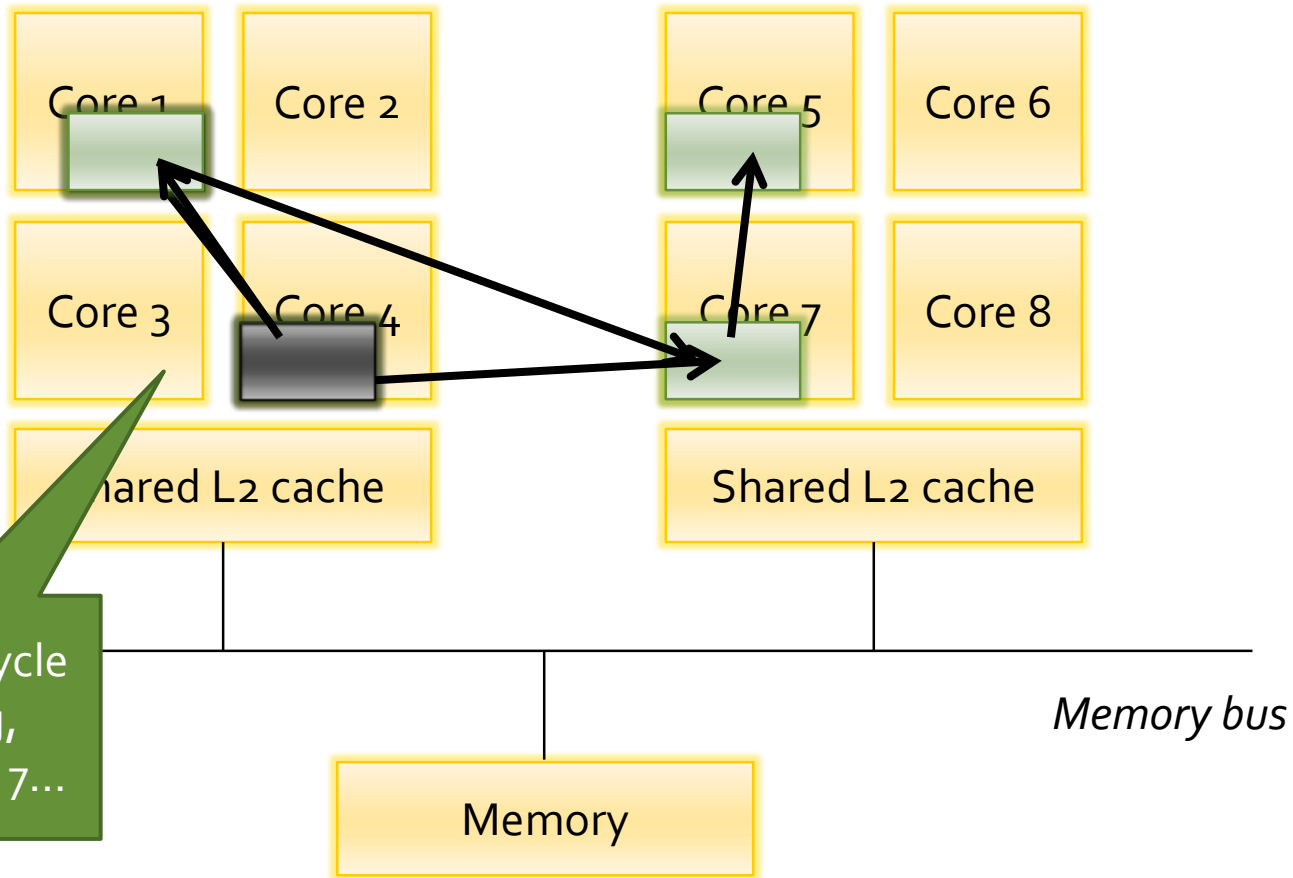
lock:

-1

-1 => lock available  
n => lock held by cluster n

```
void acquireLock(bool *lock) {  
  do {  
    holder = *lock;  
    if (holder != -1) {  
      if (holder == MY_CLUSTER) {  
        BackOff(SHORT);  
      } else {  
        BackOff(LONG);  
      }  
    }  
  } while (!CAS(lock, -1, MY_CLUSTER));  
}
```

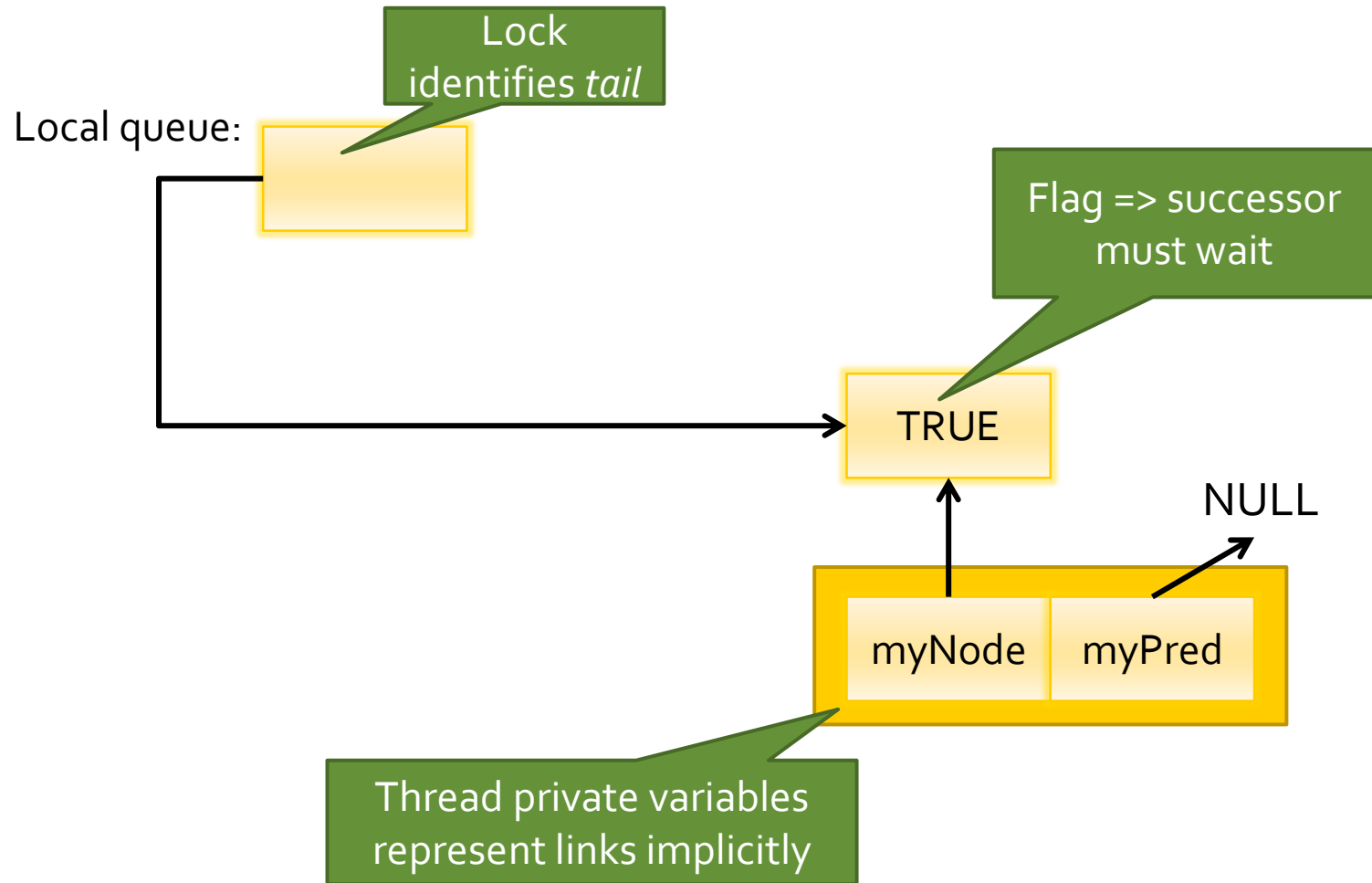
# Hierarchical locks



Avoid this cycle repeating, starving 5 & 7...

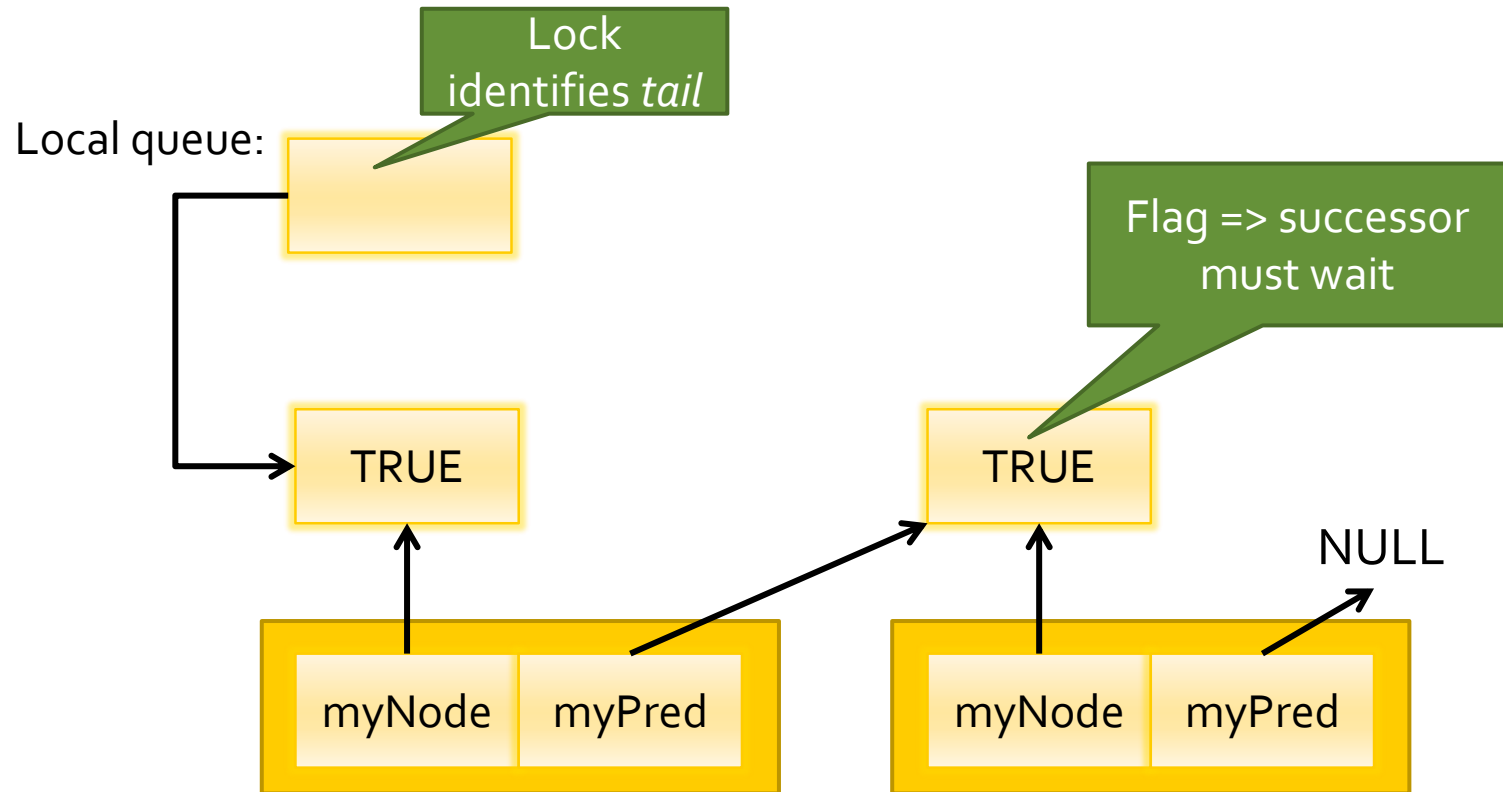


# Hierarchical CLH queue lock

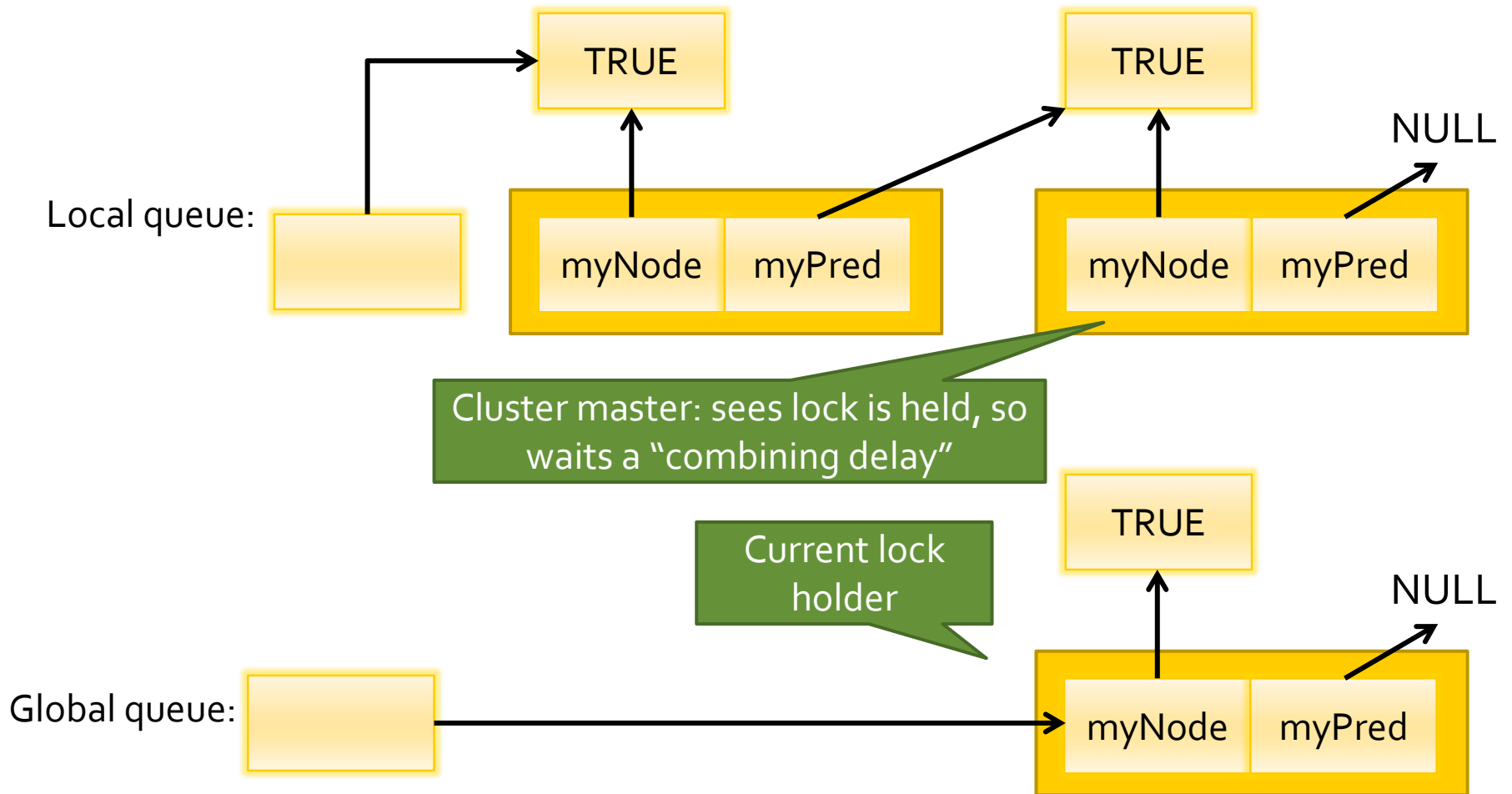


Based on hierarchical CLH lock of Luchangco, Nussbaum, Shavit

# Hierarchical CLH queue lock

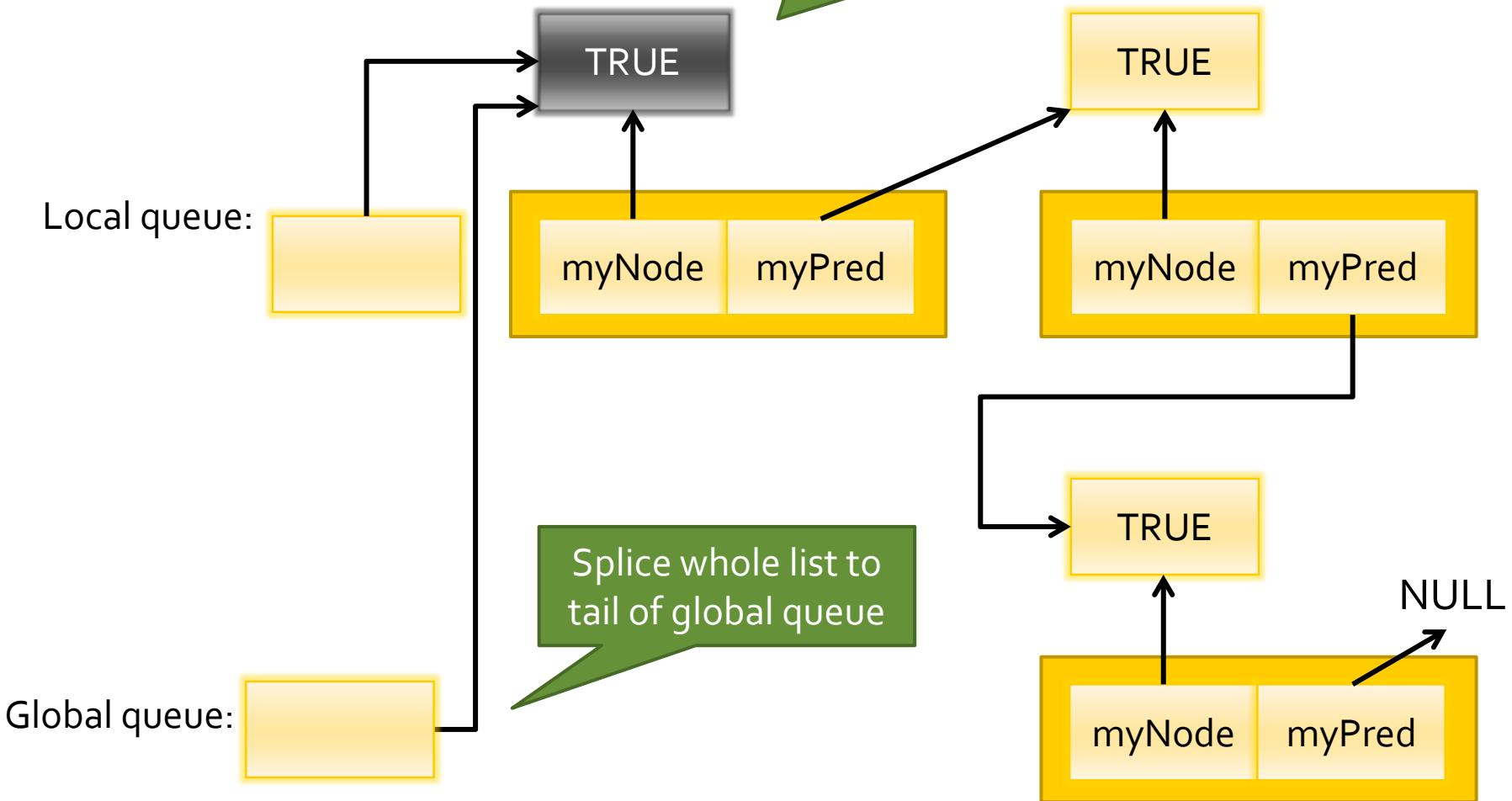


# Hierarchical CLH queue lock



# Hierarchical CLH

Set "Tail When Spliced" flag: next local queue entry will be a new cluster master



# Course overview: structure

- Building locks
  - Test-and-set locks
  - TATAS locks & backoff
  - Queue-based locks
  - Hierarchical locks
  - Reader-writer locks
- Lock-free programming
- Transactional memory

# Reader-writer locks (TATAS-like)

lock:

0

-1 => Locked for write  
0 => Lock available  
+n => Locked by n readers

```
void acquireWrite(int *lock) {  
    do {  
        if ((*lock == 0) &&  
            (CAS(lock, 0, -1))) {  
            break;  
        } while (1);  
    }  
}
```

```
void acquireRead(int *lock) {  
    do {  
        int oldVal = *lock;  
        if ((oldVal >= 0) &&  
            (CAS(lock, oldVal, oldVal+1))) {  
            break;  
        } } while (1);  
}
```

```
void releaseWrite(int *lock) {  
    *lock = 0;  
}
```

```
void releaseRead(int *lock) {  
    FADD(lock, -1); // Atomic fetch-and-add  
}
```

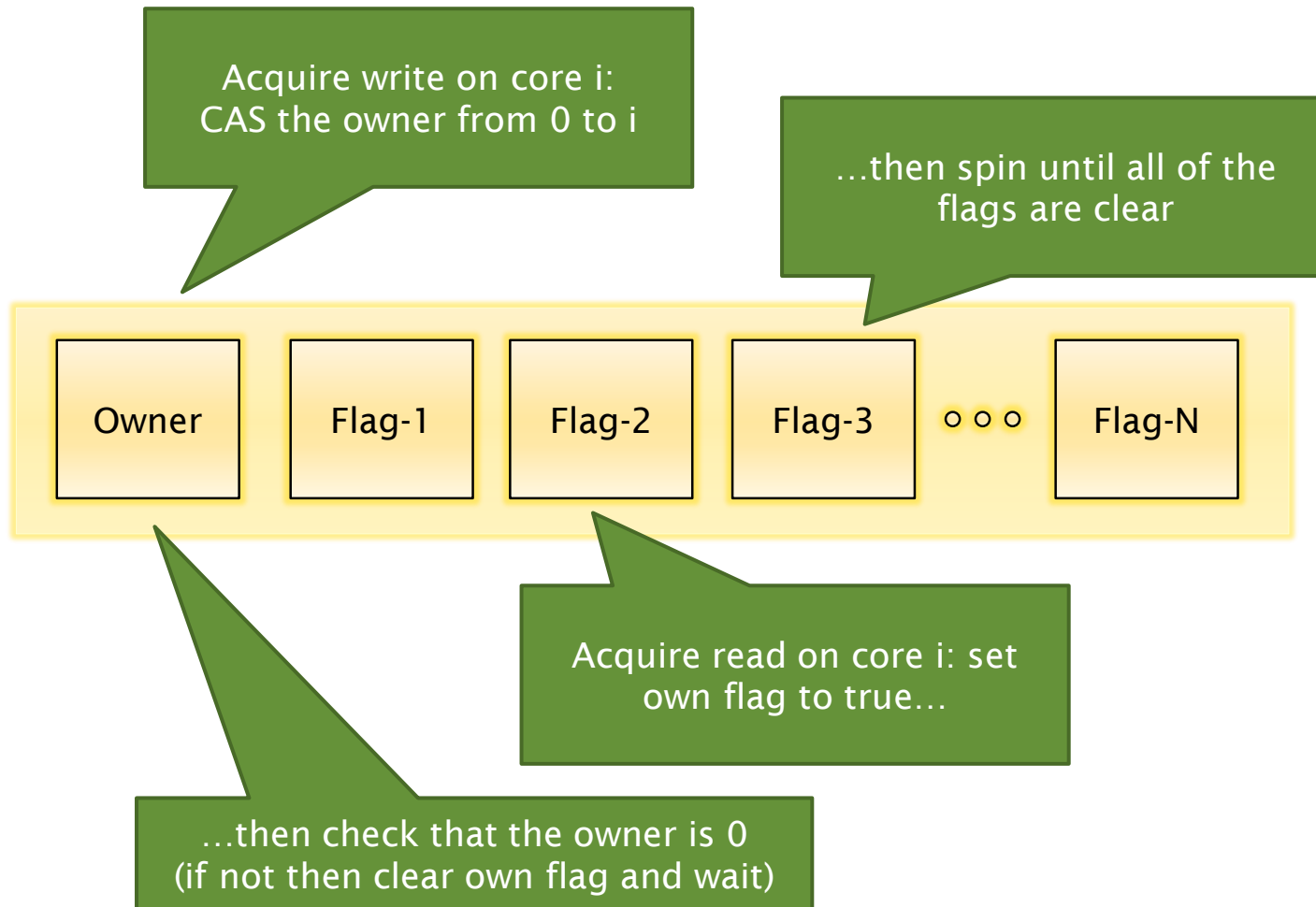
# The problem with readers

```
int readCount() {  
    acquireRead(lock);  
    int result = count;  
    releaseRead(lock);  
    return result;  
}
```

```
void incrementCount() {  
    acquireWrite(lock);  
    count++;  
    releaseWrite(lock);  
}
```

- Each acquireRead fetches the cache line holding the lock in exclusive mode
  - Again: acquireRead are not logically conflicting, but this introduces a physical conflict
- The time spent managing the lock is likely to vastly dominate the actual time looking at the counter
- Many workloads are read-mostly...

# Keeping readers separate





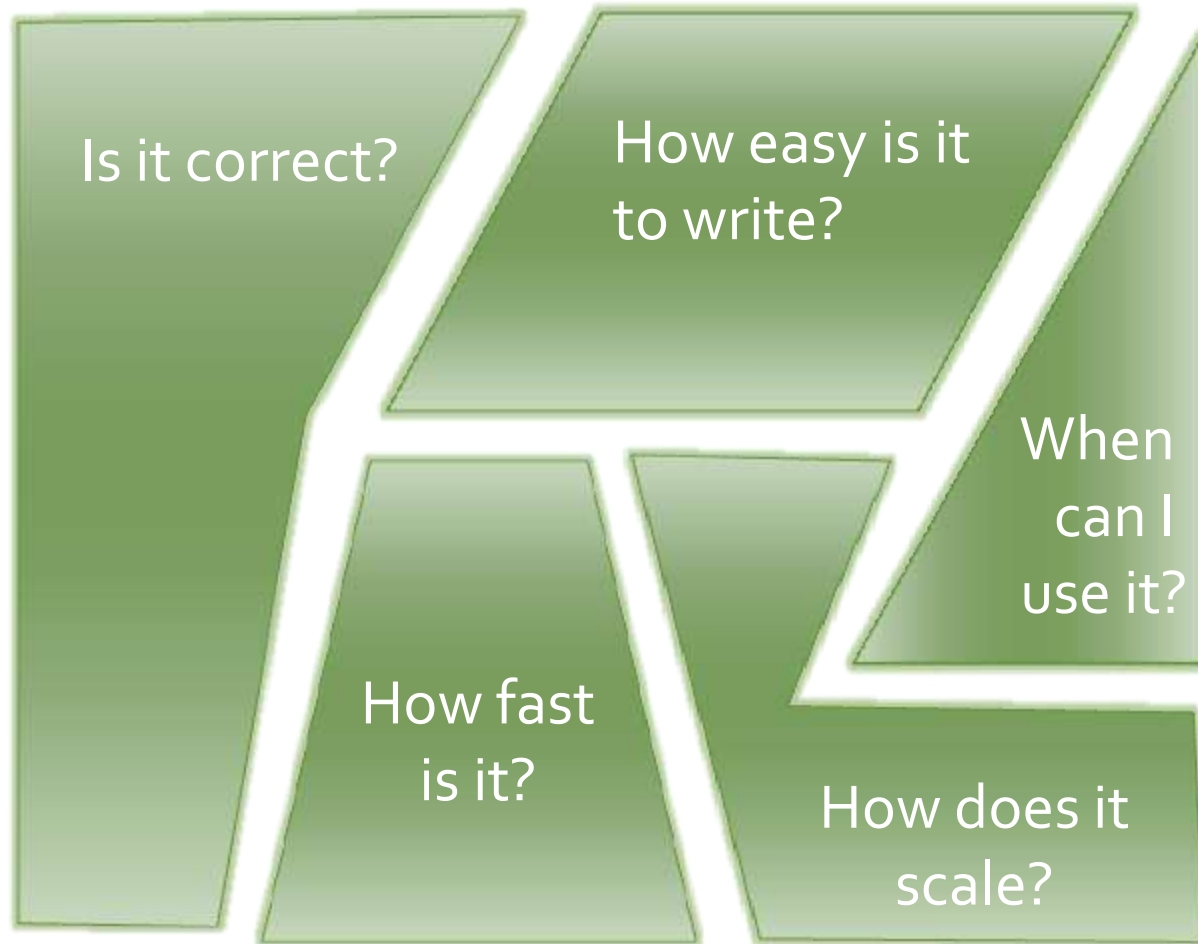
# Keeping readers separate

- With care, readers do not need to synchronize with other readers
  - Extend the flags to be whole cache lines
  - Pack multiple locks flags for the same thread onto the same line
  - Exploit the cache structure in the machine: Dice & Shavit's TLRW byte-lock
- If “N” threads is very large..
  - Dedicate the flags to specific important threads
  - Replace the flags with ordinary multi-reader locks

# Read-Copy-Update (RCU)

- Use locking to serialize updates (typically)
  - ...but allow readers to operate concurrently with updates
- Ensure that readers don't go wrong if they access data mid-update
  - Have data structures reachable via a single root pointer: update the root pointer rather than updating the data structure in-place
  - Ensure that updates don't affect readers – e.g., initializing nodes before splicing them into a list, and retaining “next” pointers in deleted nodes
  - Exact semantics offered can be subtle (ongoing research direction)
- Memory management problems common with lock-free data structures

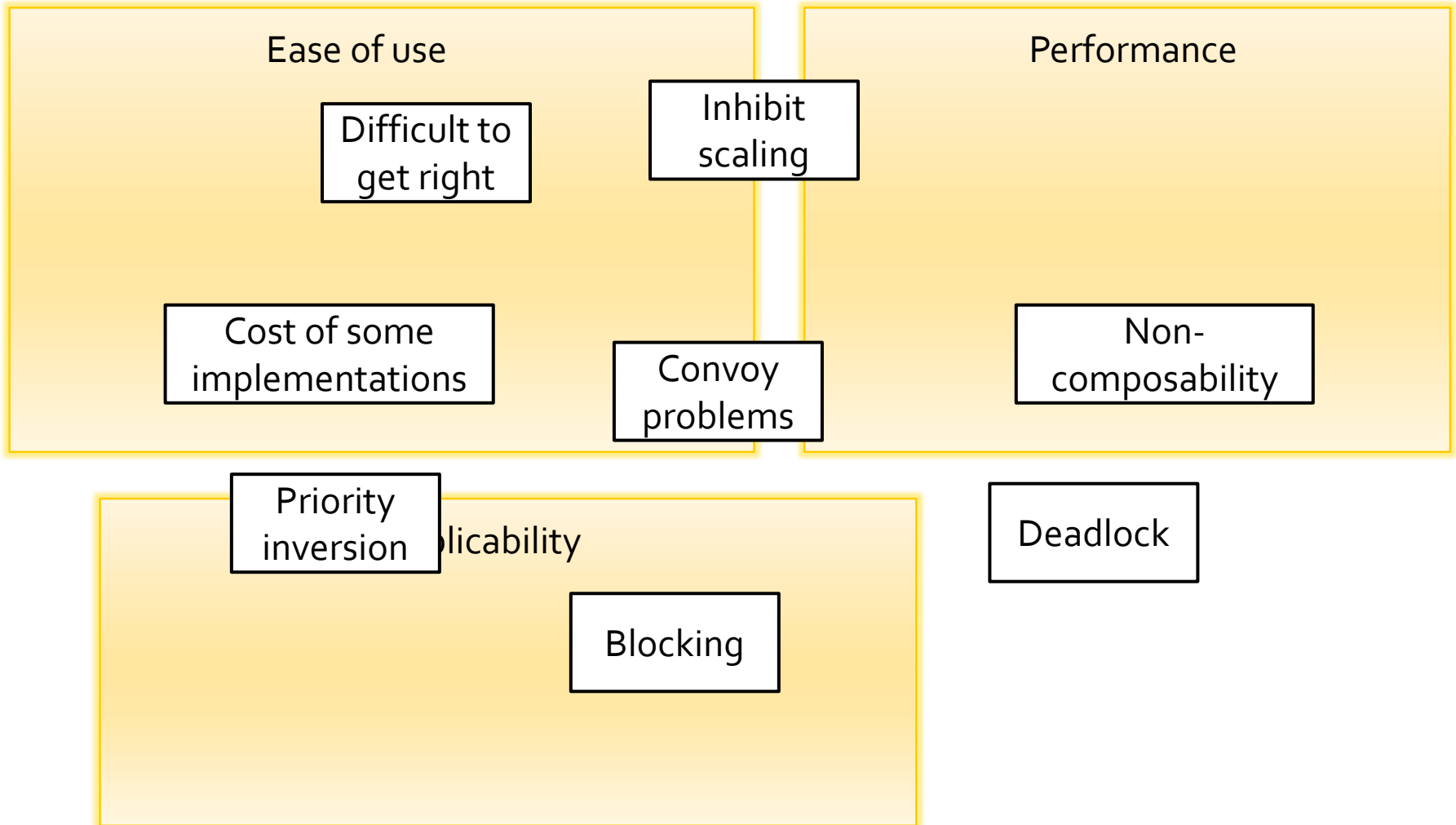
# What do we care about?



# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# What do people say is wrong with locks?



# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# What we're building

- A set of integers, represented by a sorted linked list
- `find(int) -> bool`
- `insert(int) -> bool`
- `delete(int) -> bool`

# The building blocks

- `read(addr) -> val`
- `write(addr, val)`
- `cas(addr, old-val, new-val) -> bool`

(I'll assume that memory is sequentially consistent, and ignore allocation / de-allocation for the moment)



# Searching a sorted list

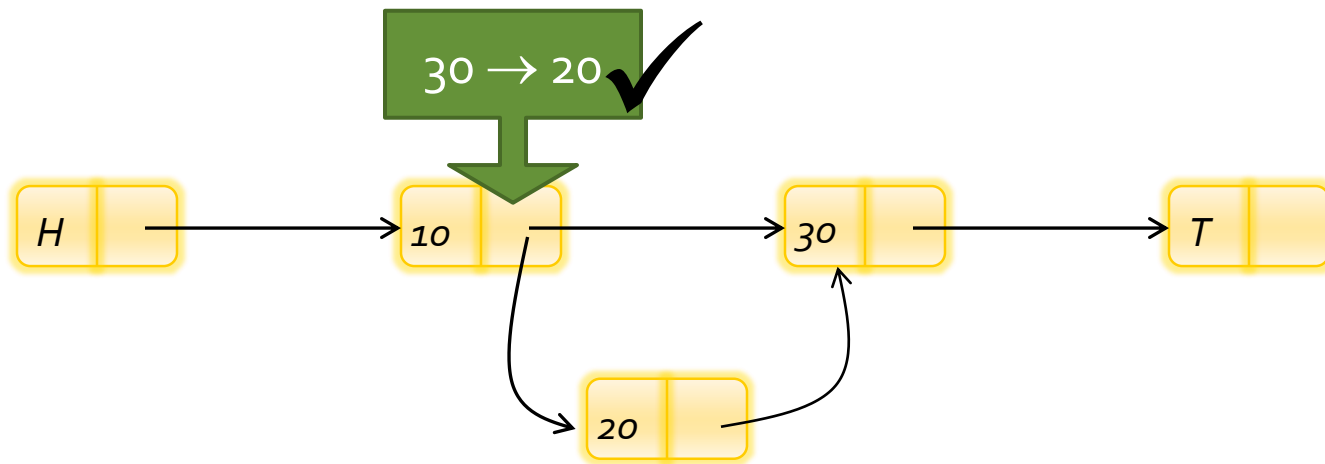
- `find(20):`



`find(20) -> false`

# Inserting an item with CAS

- `insert(20):`

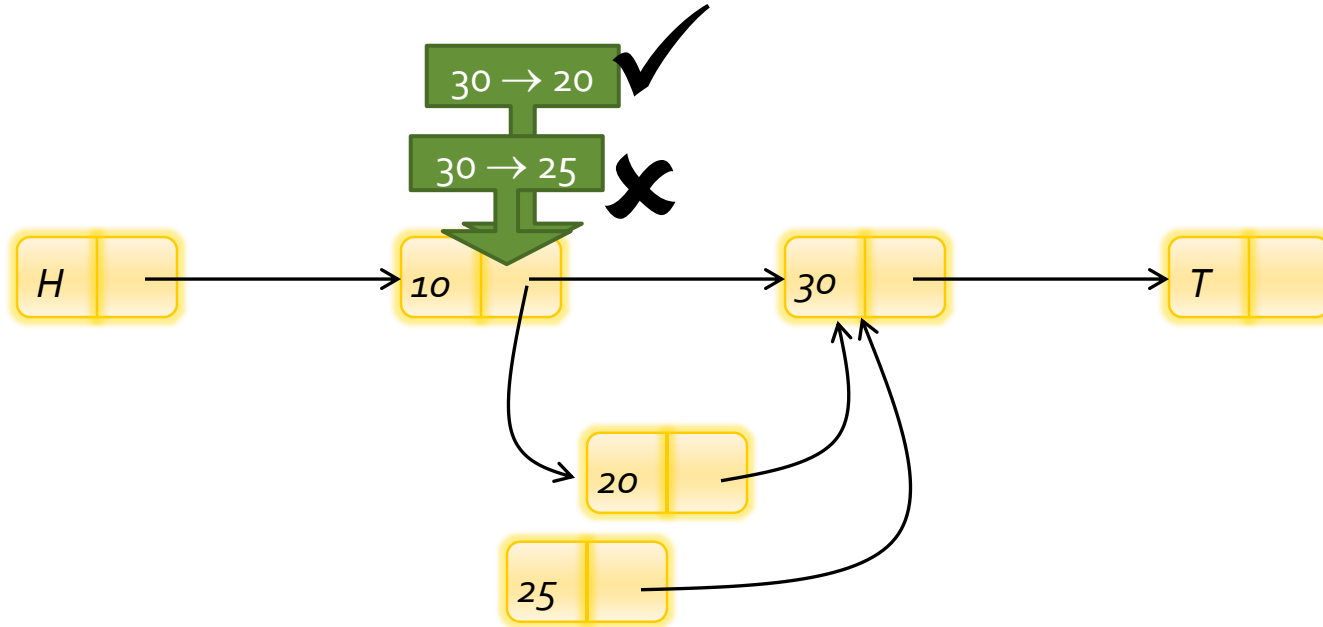


`insert(20) -> true`

# Inserting an item with CAS

■ insert(20):

• insert(25):



# Searching and finding together

▪ `find(20) -> false`

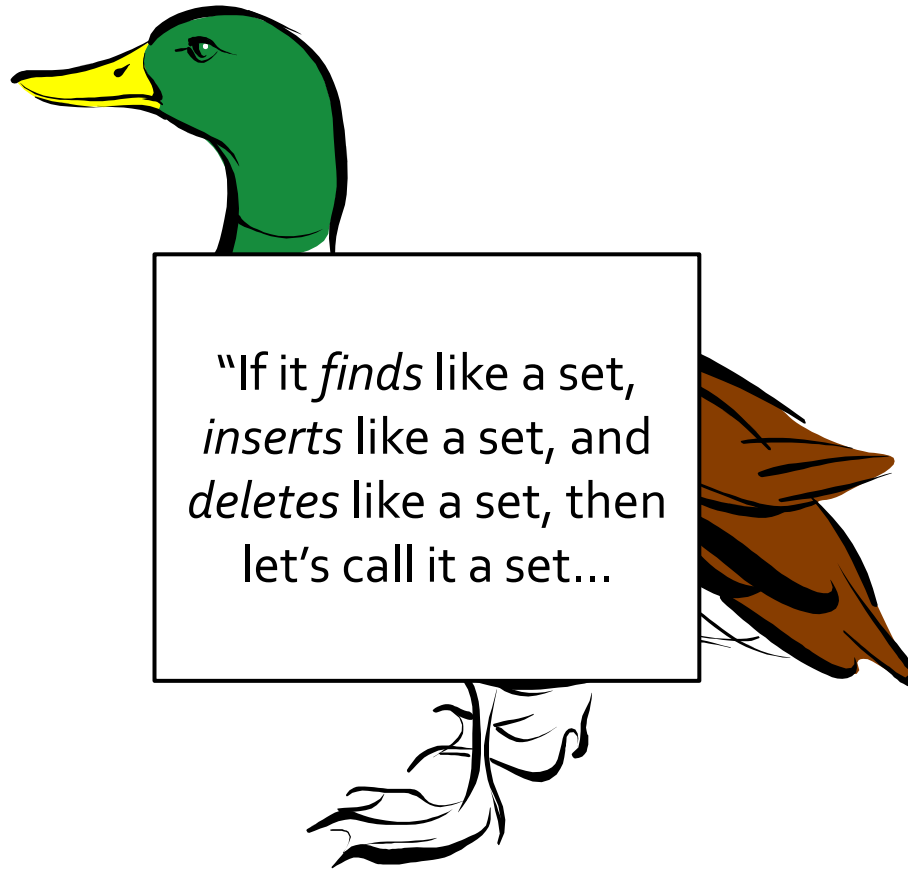
• `insert(20) -> true`

This thread saw 20  
was not in the set...

...but this thread  
succeeded in putting  
it in!

- Is this a correct implementation of a set?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

# Correctness criteria



# Sequential specification

- Ignore the list for the moment, and focus on the set:

*Sequential:* we're only considering one operation on the set at a time

*Specification:* we're saying what a set does, not what a list does, or how it looks in memory

find(int) -> bool

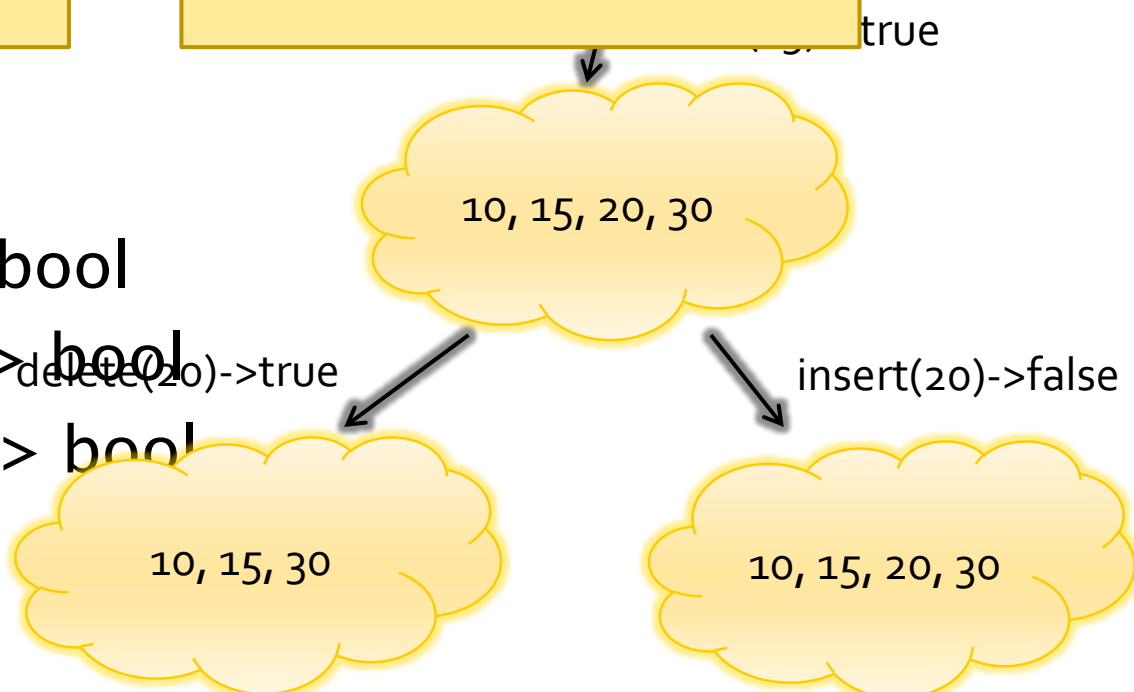
insert(int) -> bool

delete(int) -> bool

10, 15, 20, 30

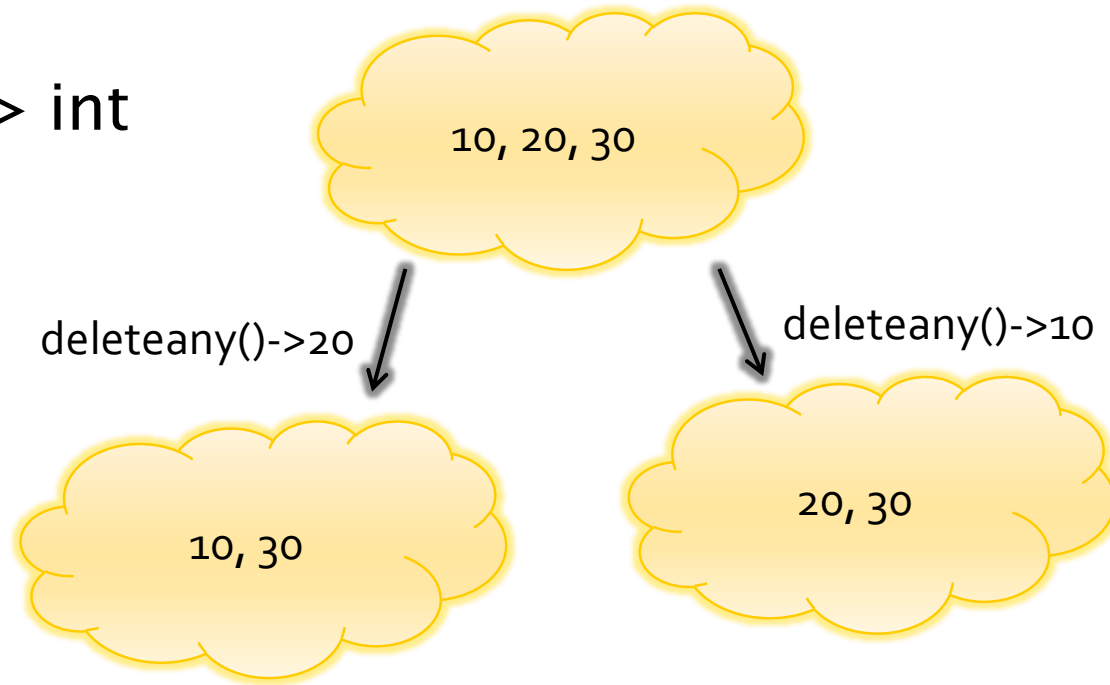
10, 15, 30

10, 15, 20, 30



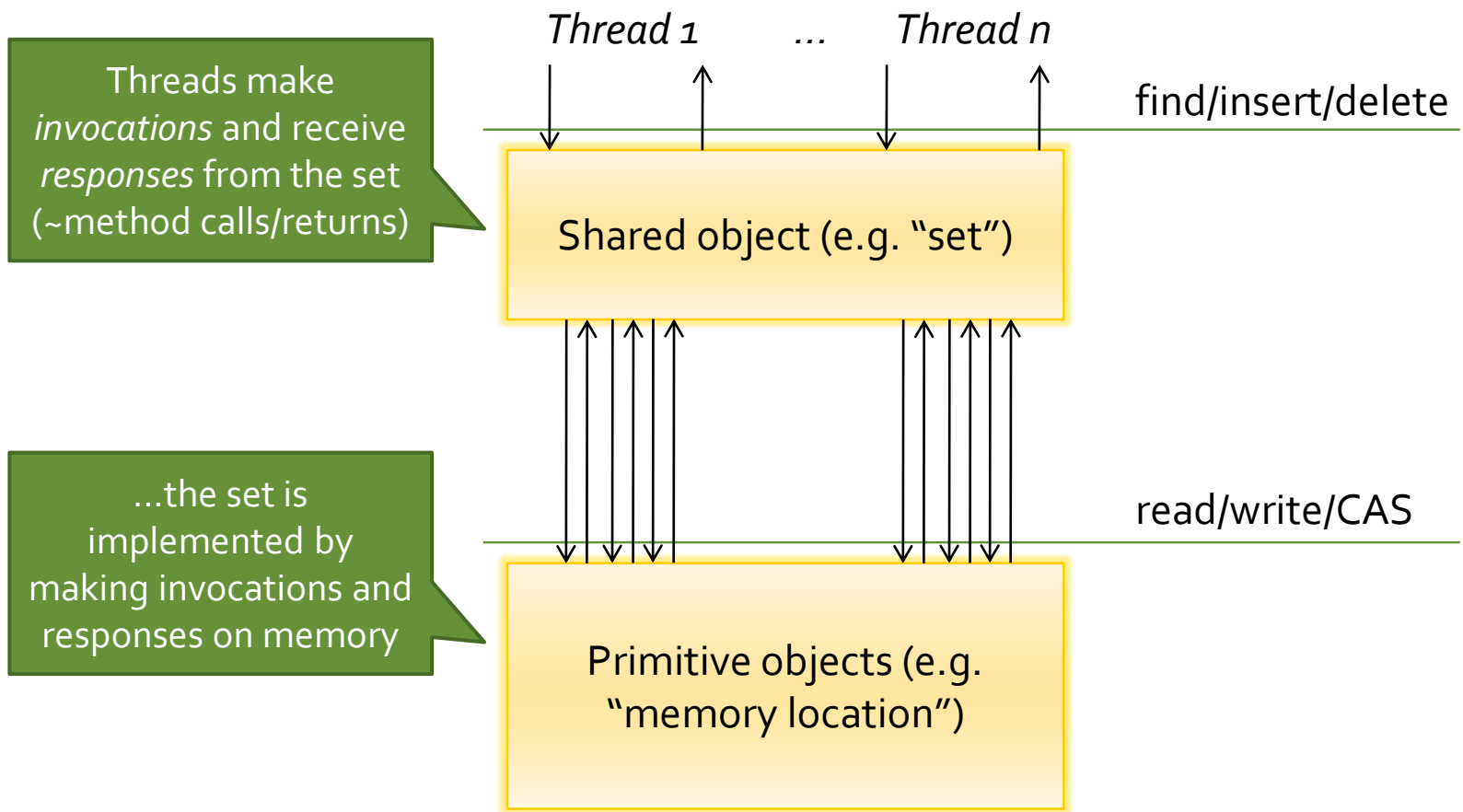
# Sequential specification

deleteany() -> int



This is still a *sequential* spec... just not a *deterministic* one

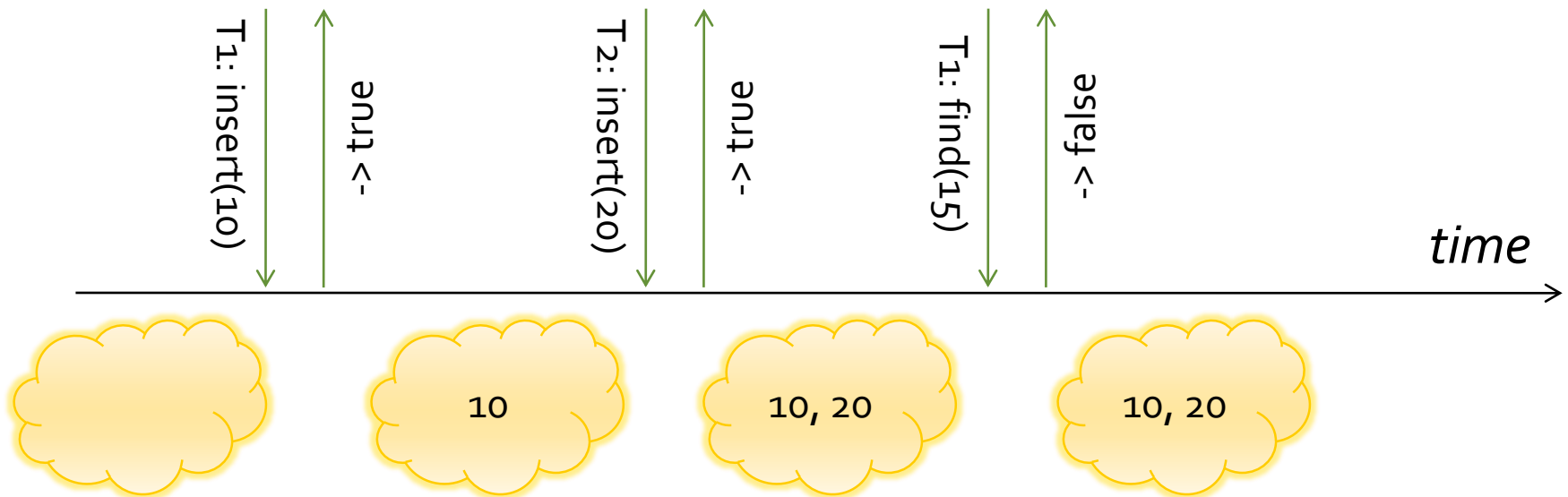
# System model





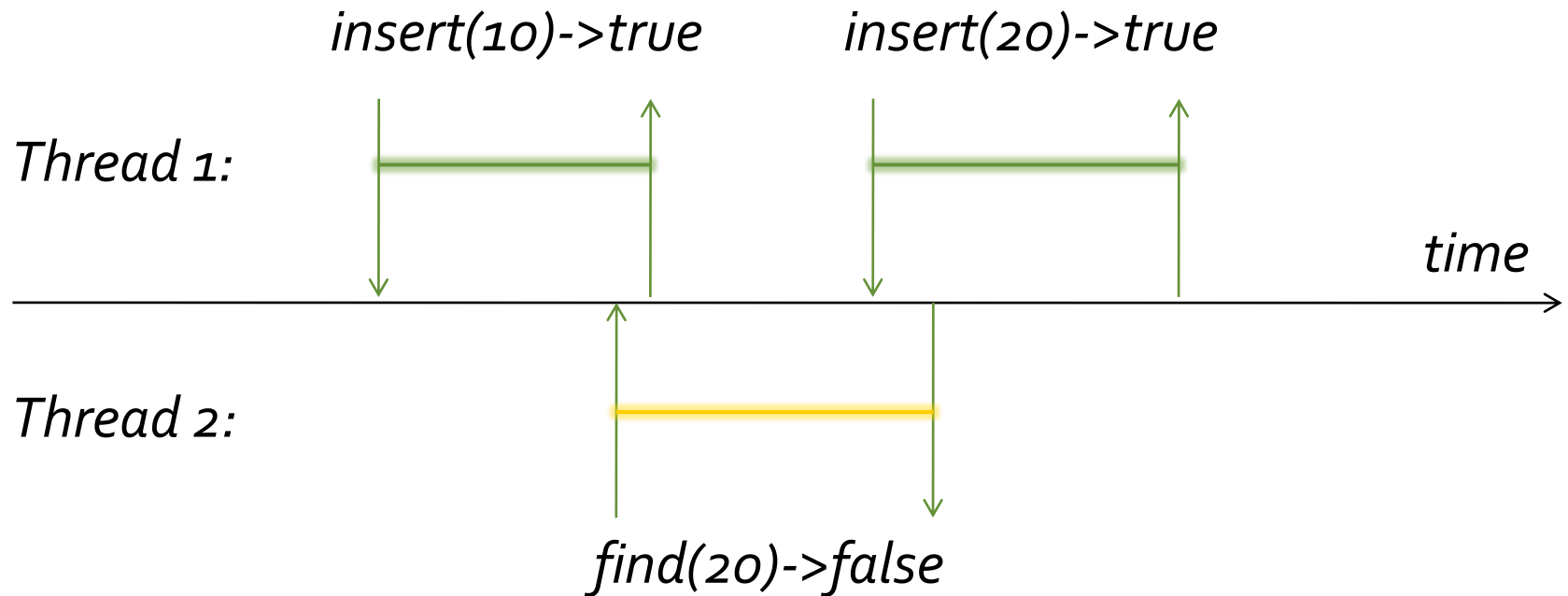
# High level: sequential history

- No overlapping invocations:



# High level: concurrent history

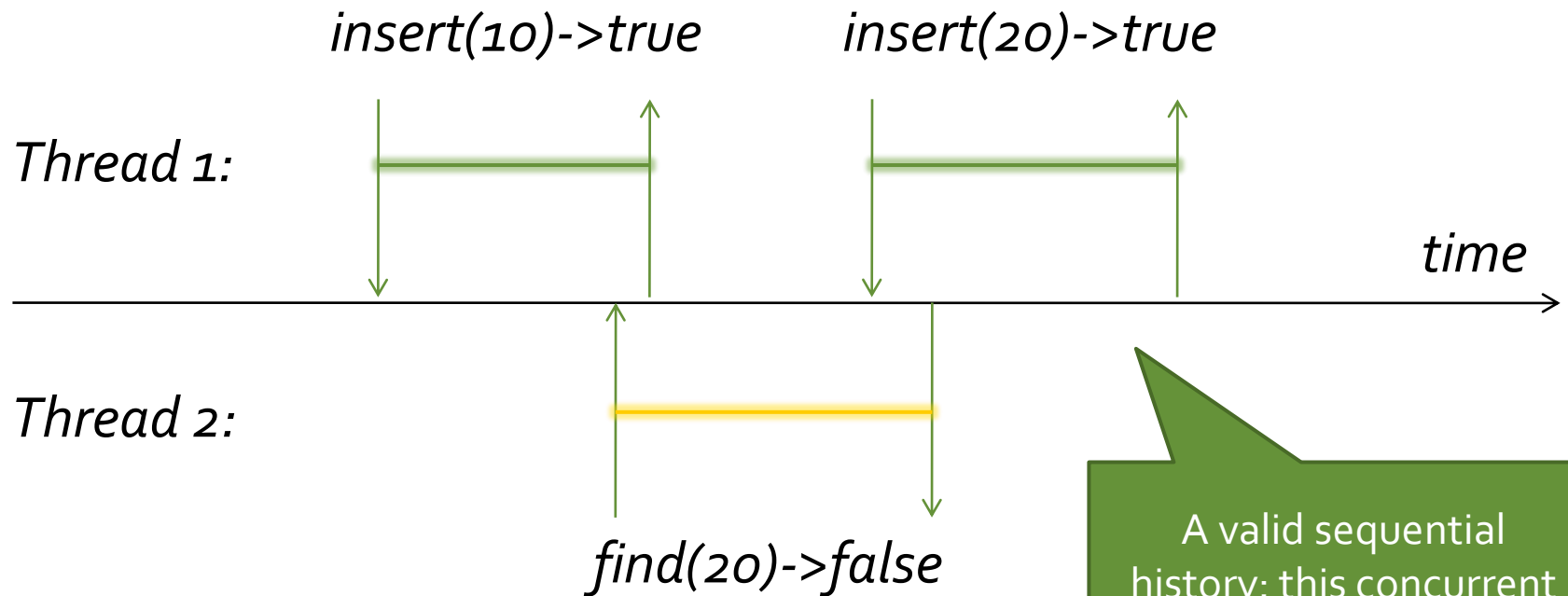
- Allow overlapping invocations:



# Linearizability

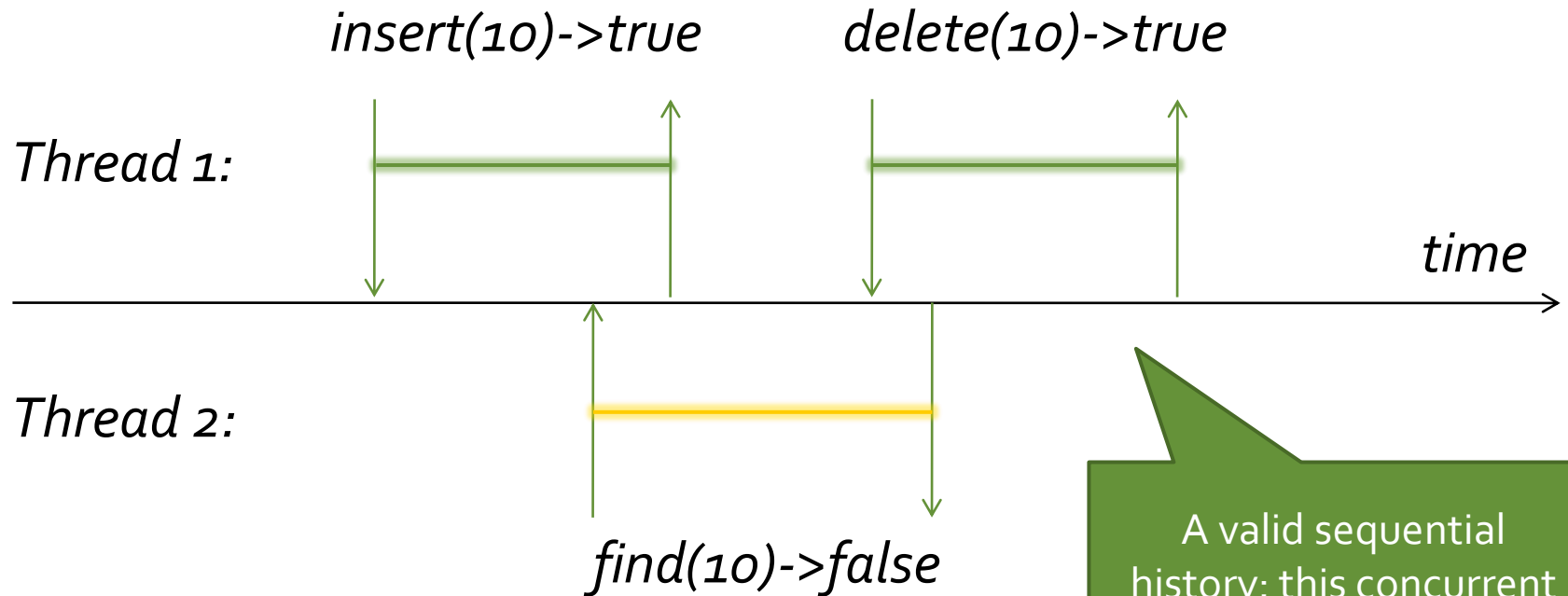
- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?

# Example: linearizable



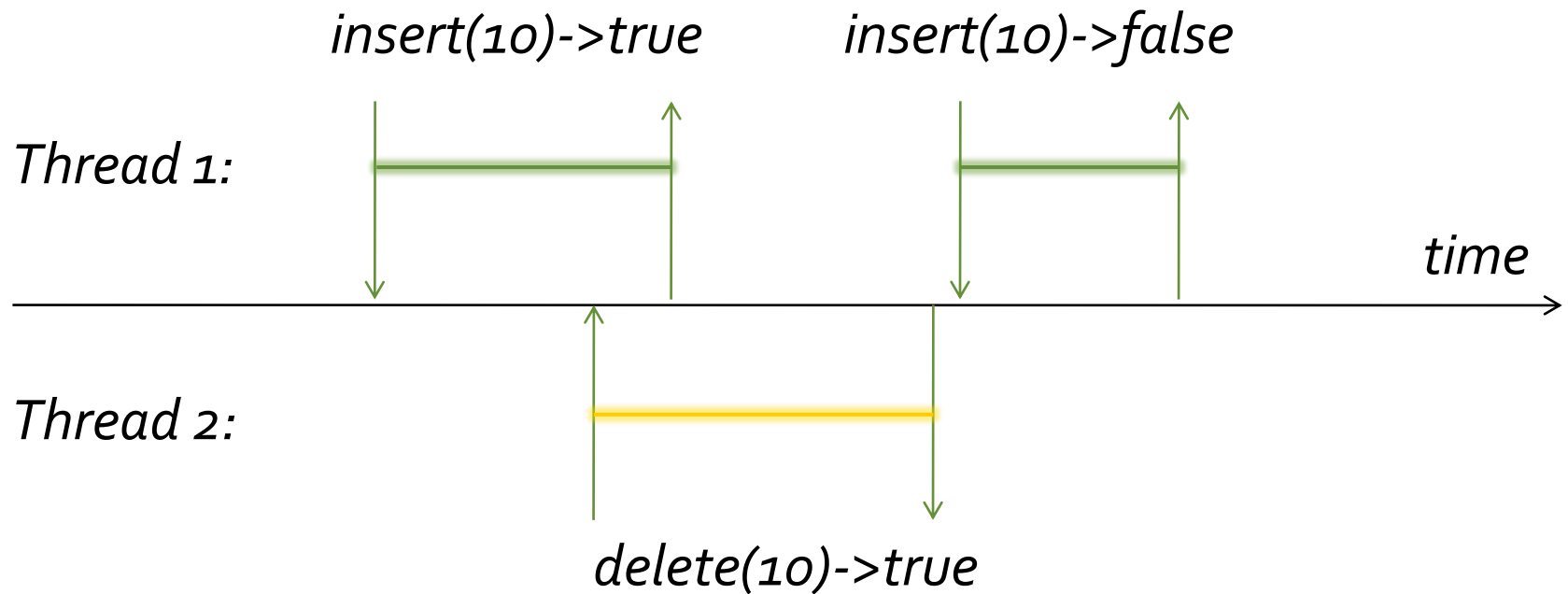
A valid sequential history: this concurrent execution is OK

# Example: linearizable



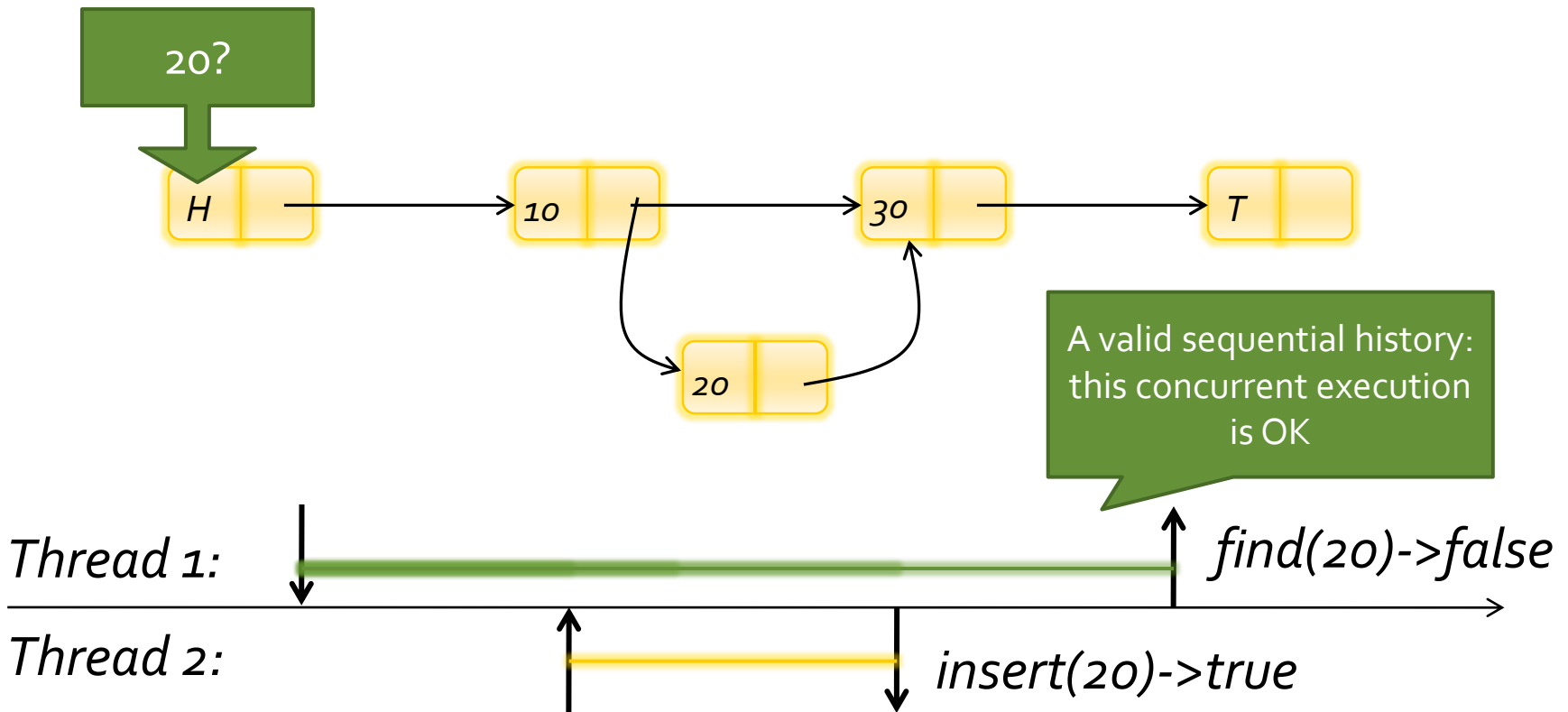
A valid sequential history: this concurrent execution is OK

# Example: not linearizable



# Returning to our example

- `find(20) -> false`
- `insert(20) -> true`

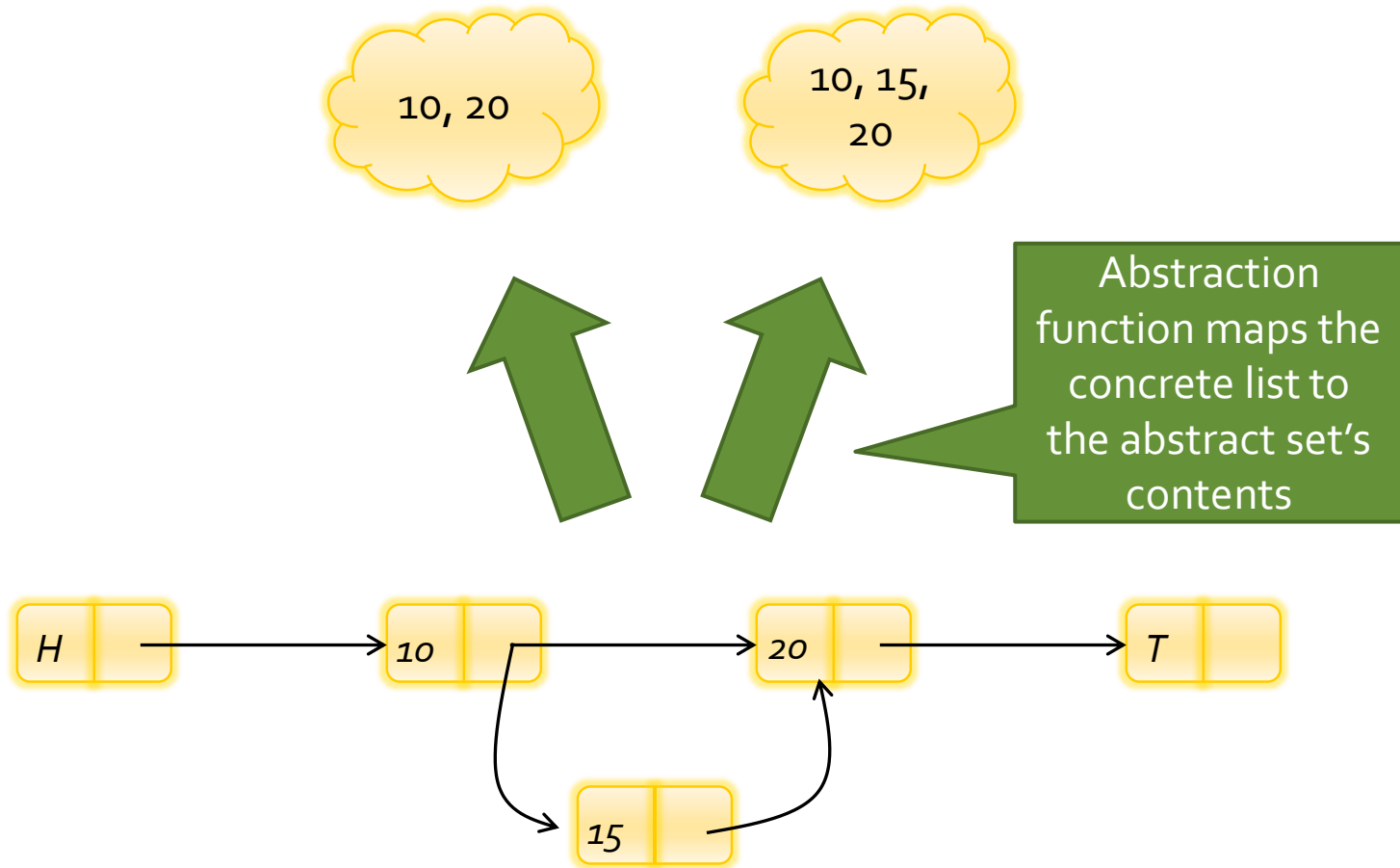


# Recurring technique

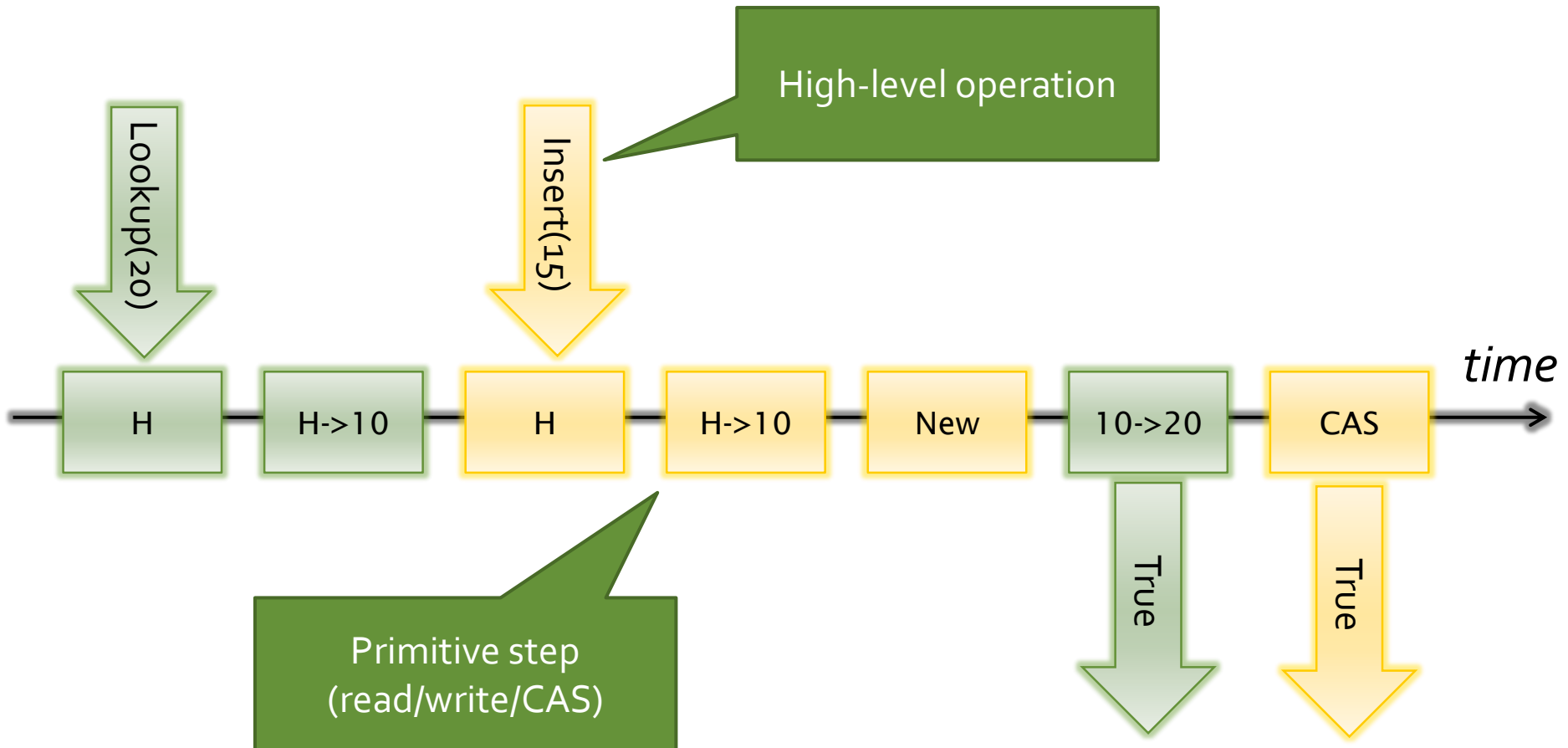
- For updates:
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads:
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction



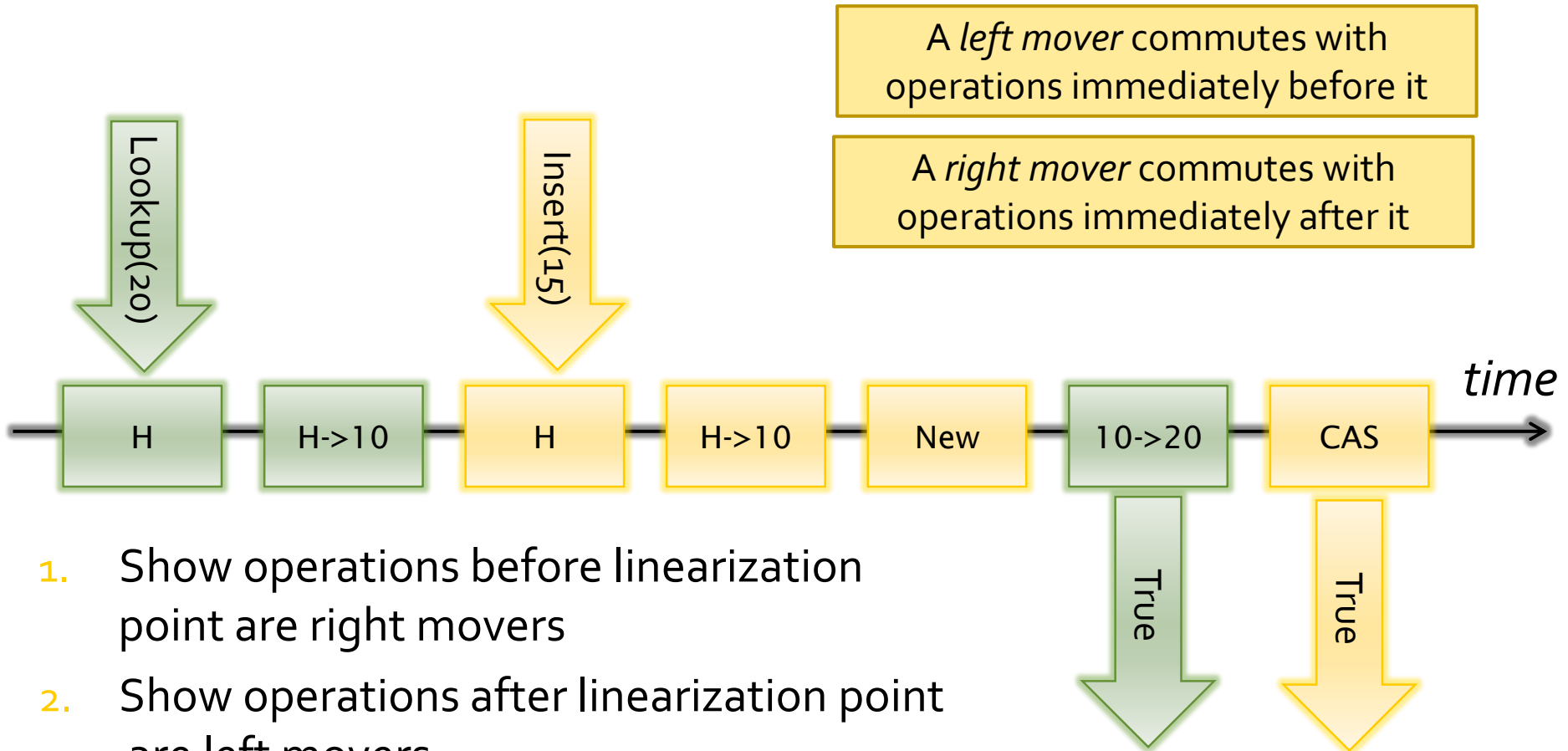
# Correctness (informal)



# Correctness (informal)

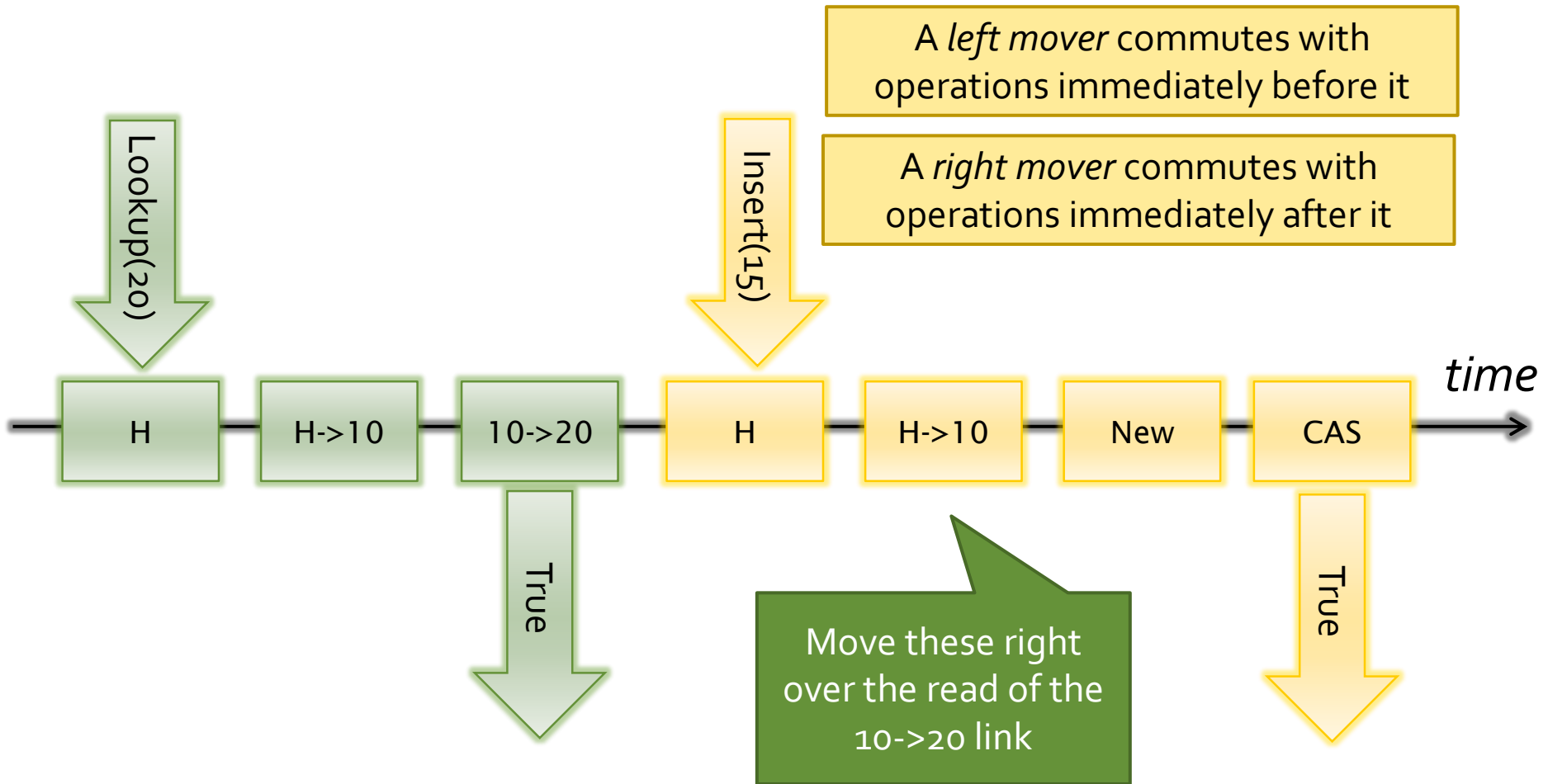


# Correctness (informal)



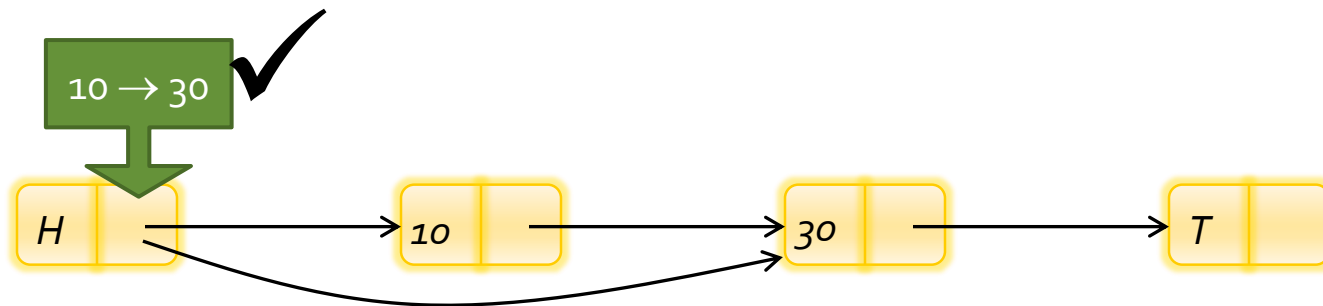
1. Show operations before linearization point are right movers
2. Show operations after linearization point are left movers
3. Show linearization point updates abstract state

# Correctness (informal)



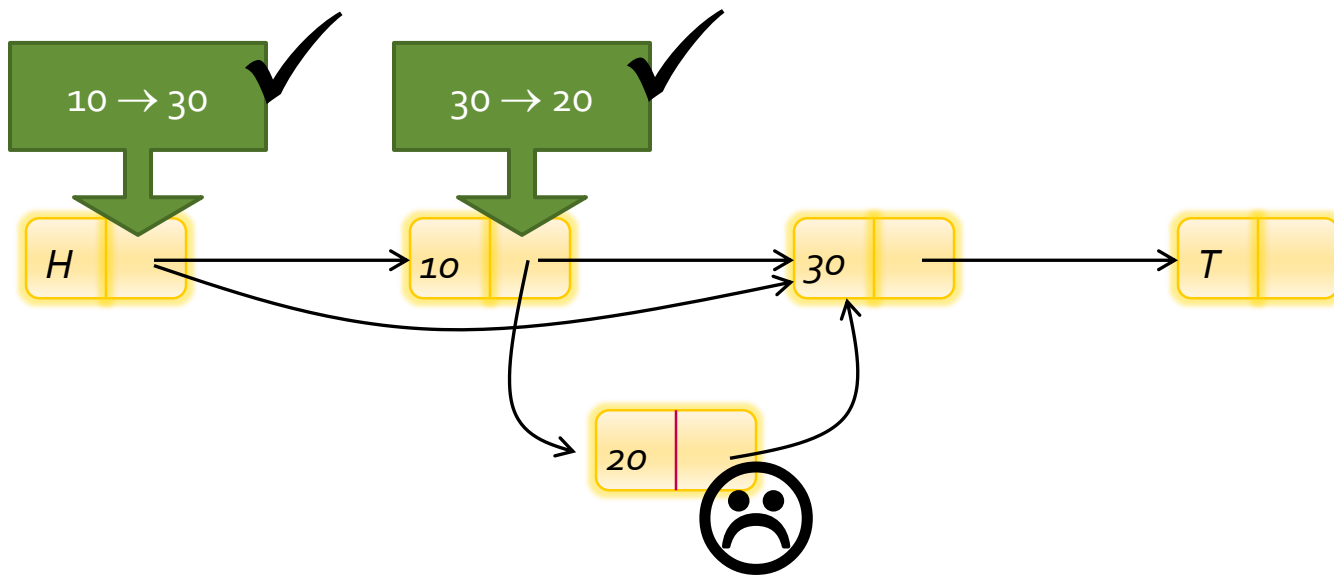
# Adding "delete"

- First attempt: just use CAS  
delete(10):



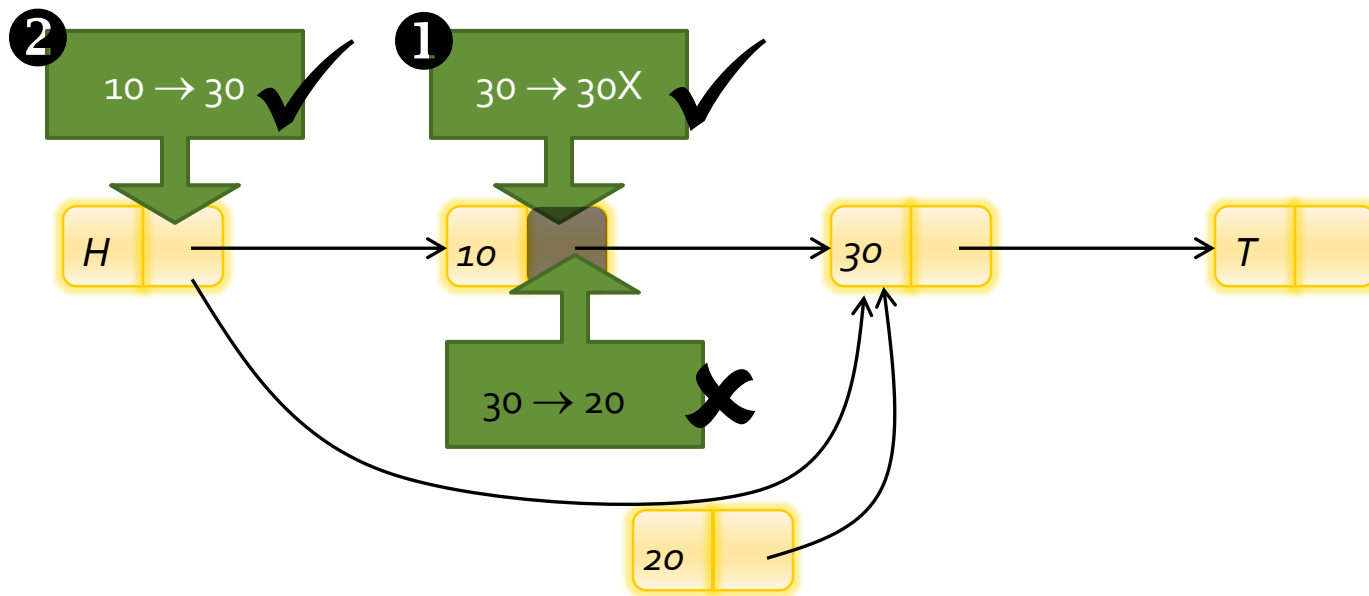
# Delete and insert:

- delete(10) & insert(20):



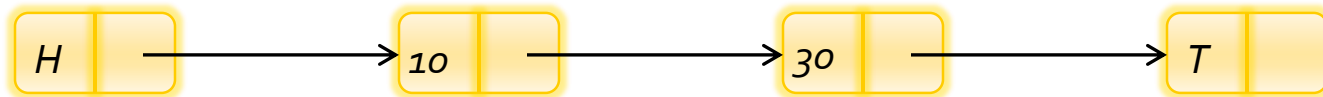
# Logical vs physical deletion

- Use a 'spare' bit to indicate logically deleted nodes:

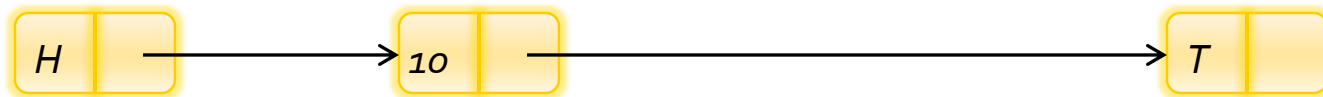


# Delete-greater-than-or-equal

- DeleteGE(int x) -> int
  - Remove "x", or next element above "x"

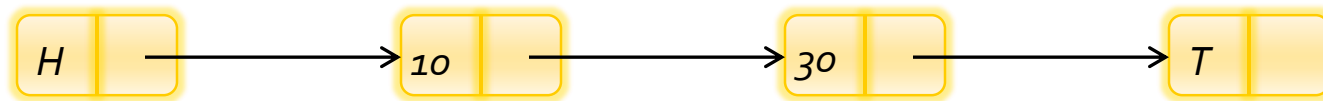


- DeleteGE(20) -> 30





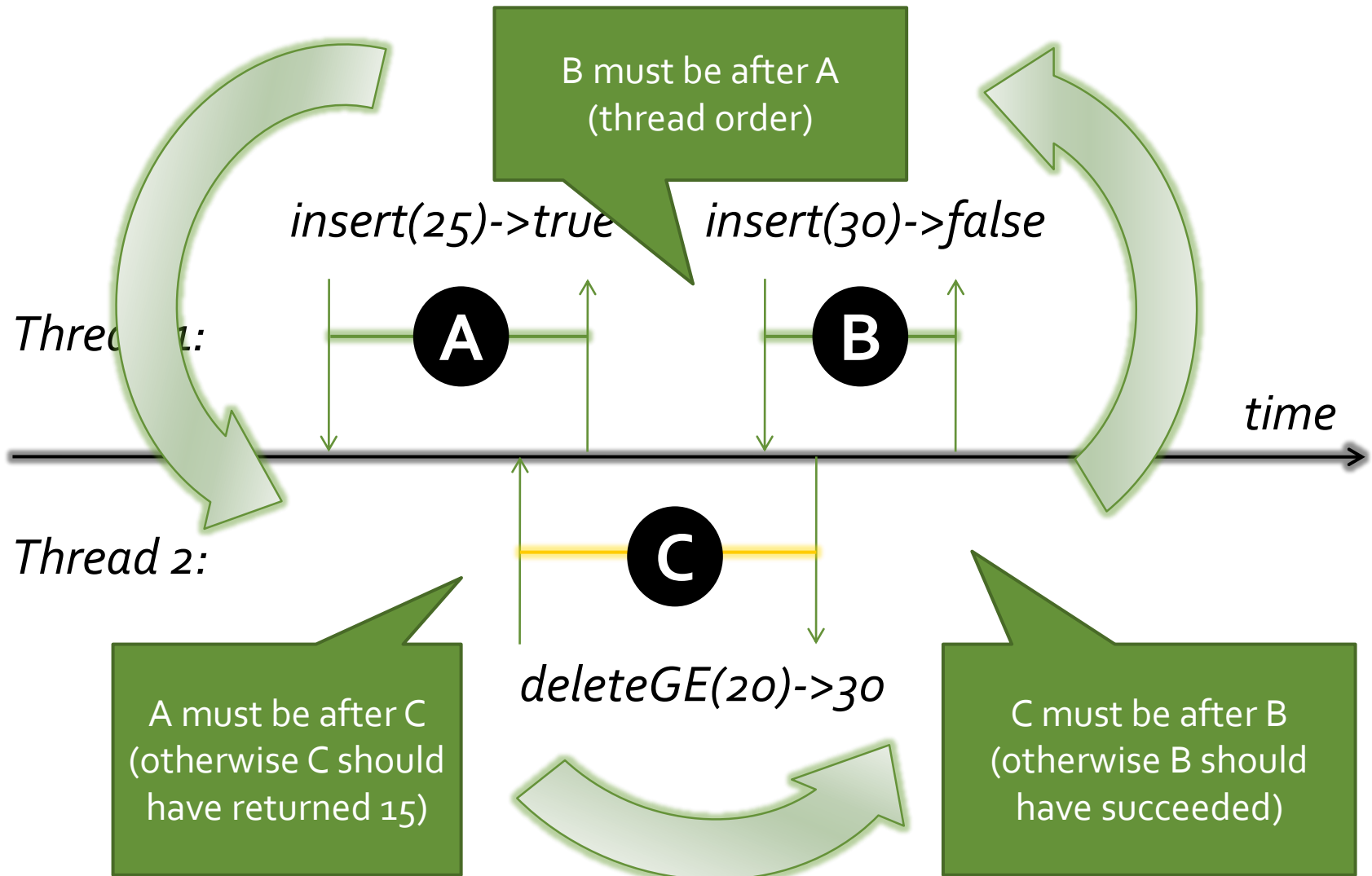
# Does this work: DeleteGE(20)



1. Walk down the list, as in a normal delete, find 30 as next-after-20

2. Do the deletion as normal: set the mark bit in 30, then physically unlink

# Delete-greater-than-or-equal



# How to realise this is wrong

- See operation which determines result
- Consider a delay at that point
- Is the result still valid?
  - Delayed read: is the memory still accessible (more of this next week)
  - Delayed write: is the write still correct to perform?
  - Delayed CAS: does the value checked by the CAS determine the result?

# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# Progress: is this a good “lock-free” list?

```
static volatile int MY_LIST = 0;

bool find(int key) {

    // Wait until list available
    while (CAS(&MY_LIST, 0, 1) == 1) {
    }

    ...

    // Release list
    MY_LIST = 0;
}
```

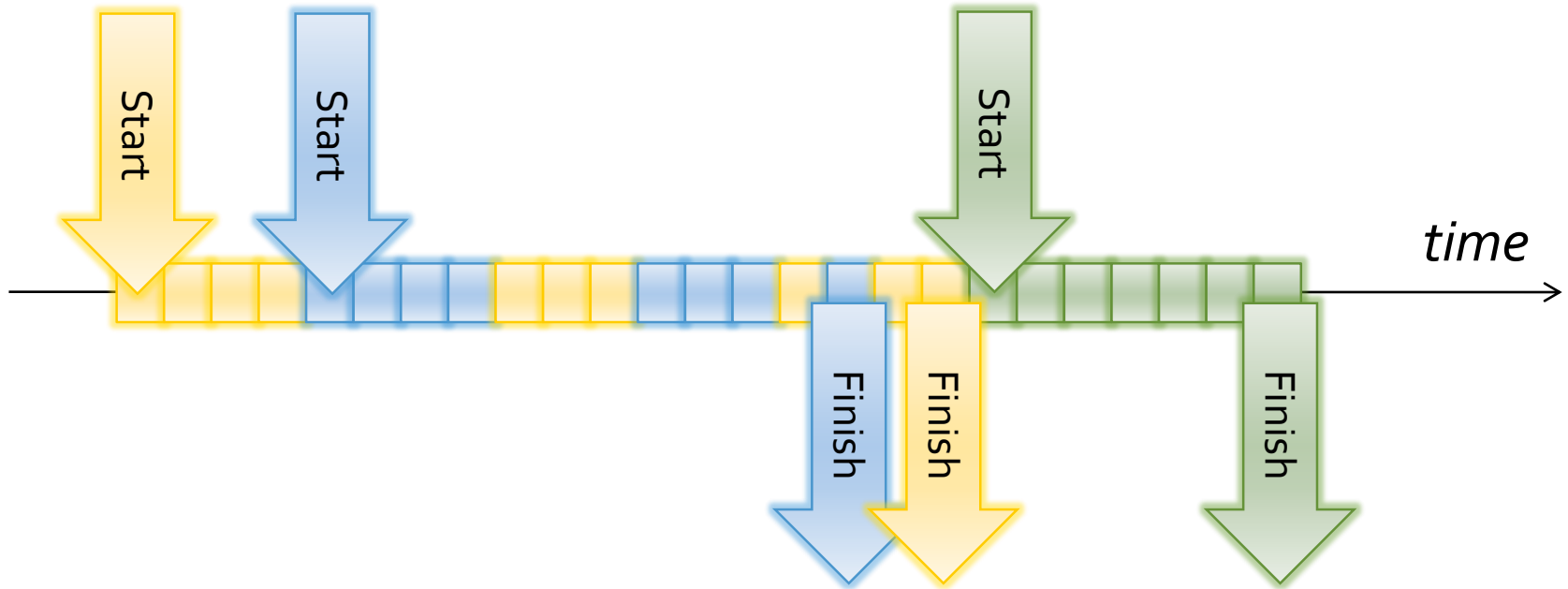
OK, we're not calling  
pthread\_mutex\_lock... but  
we're essentially doing the  
same thing

# “Lock-free”

- A specific kind of *non-blocking* progress guarantee
- Precludes the use of typical locks
  - From libraries
  - Or “hand rolled”
- Often mis-used informally as a synonym for
  - Free from calls to a locking function
  - Fast
  - Scalable

# Wait-free

- A thread finishes its own operation if it continues executing steps



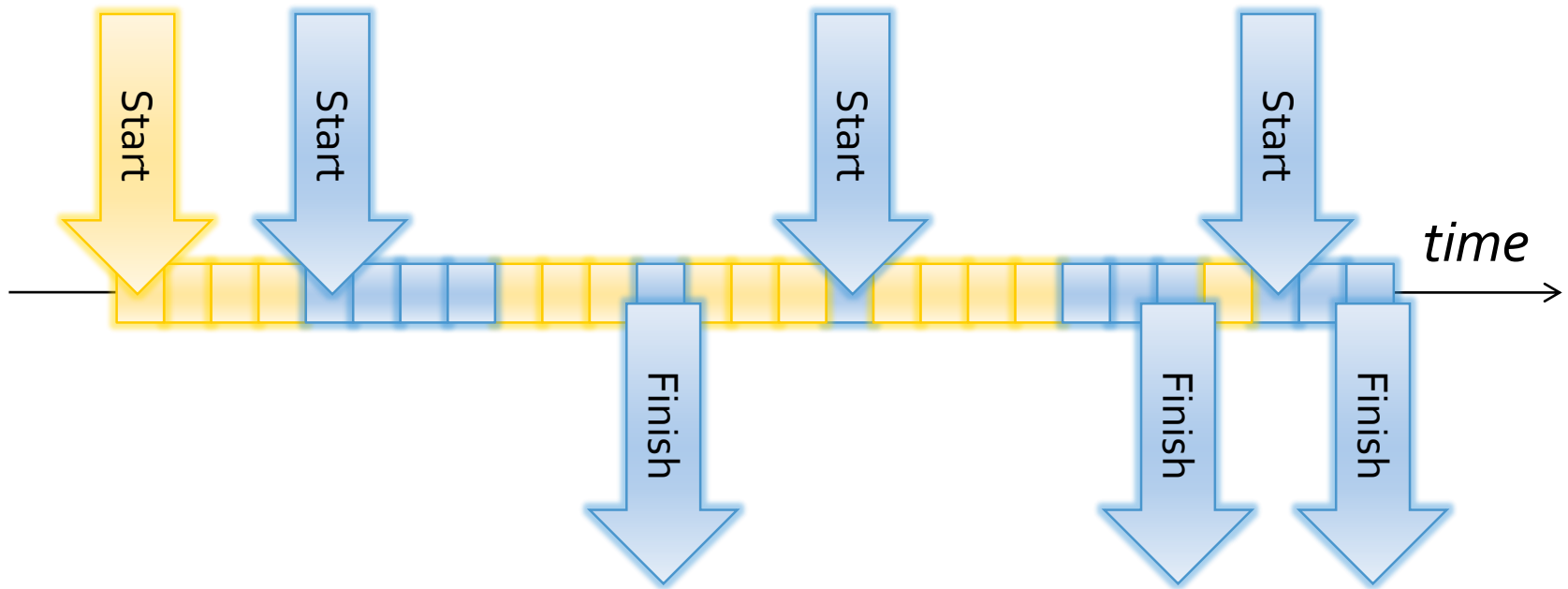
# Implementing wait-free algorithms

- General construction techniques exist (“universal constructions”)
- In practice, often used in hybrid settings (e.g., wait-free find)
- Queuing and helping strategies: everyone ensures oldest operation makes progress
- Niches, e.g., bounded-wait-free in real-time systems



# Lock-free

- Some thread finishes its operation if threads continue taking steps



# A (poor) lock-free counter

```
int getNext(int *counter) {  
    while (true) {  
        int result = *counter;  
        if (CAS(counter, result, result+1)) {  
            return result;  
        }  
    }  
}
```

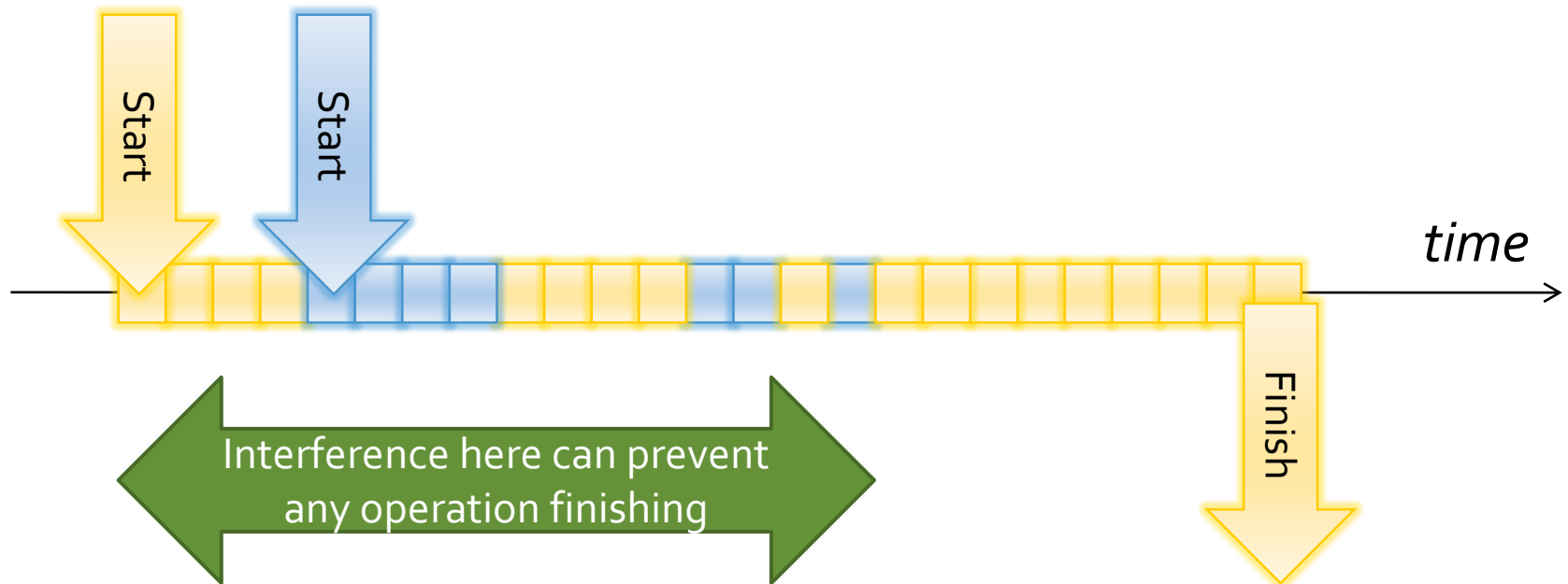
Not wait free: no  
guarantee that any  
particular thread will  
succeed

# Implementing lock-free algorithms

- Ensure that one thread (A) only has to repeat work if some other thread (B) has made “real progress”
  - e.g., insert(x) starts again if it finds that a conflicting update has occurred
- Use helping to let one thread finish another’s work
  - e.g., physically deleting a node on its behalf

# Obstruction-free

- A thread finishes its own operation if it runs in isolation



# A (poor) obstruction-free counter

```
int getNext(int *counter) {  
    while (true) {  
        int result = LL(counter);  
        if (SC(counter, result+1)) {  
            return result;  
        }  
    }  
}
```

Weak load-linked (LL)  
store-conditional (SC): LL  
on one thread will prevent  
an SC on another thread  
succeeding

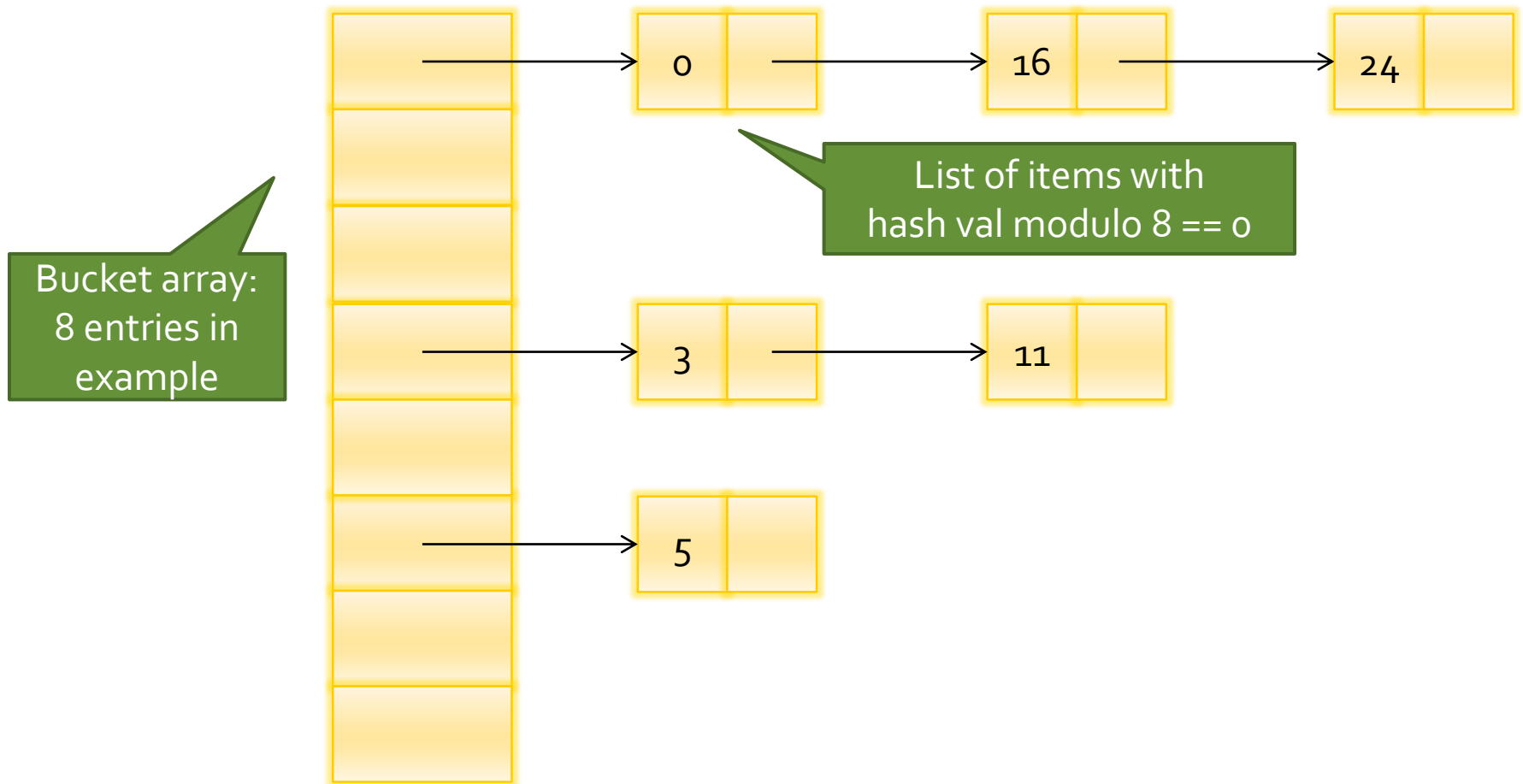
# Building obstruction-free algorithms

- Ensure that none of the low-level steps leave a data structure “broken”
- On detecting a conflict:
  - Help the other party finish
  - Get the other party out of the way
- Use *contention management* to reduce likelihood of live-lock

# Course overview: structure

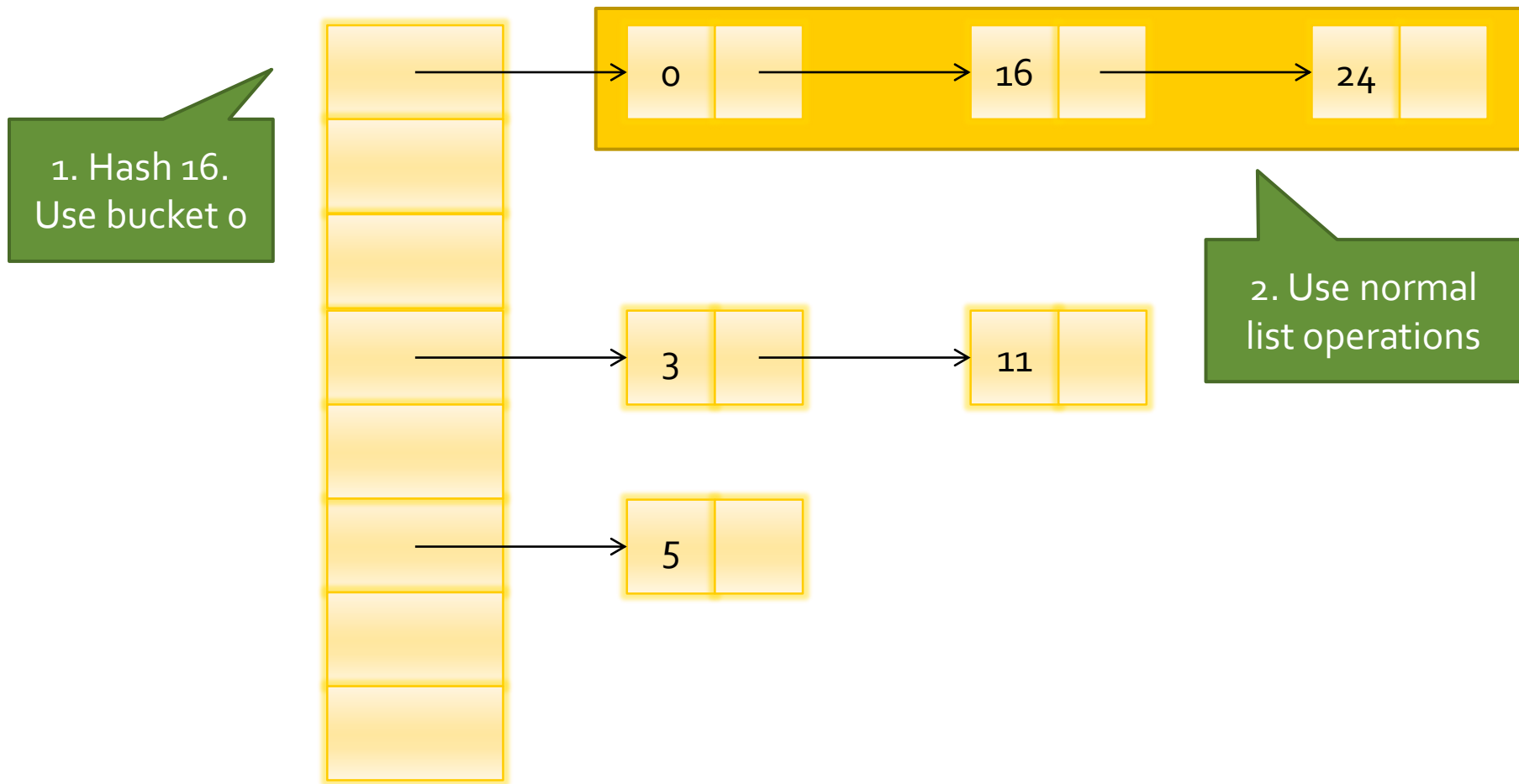
- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# Hash tables

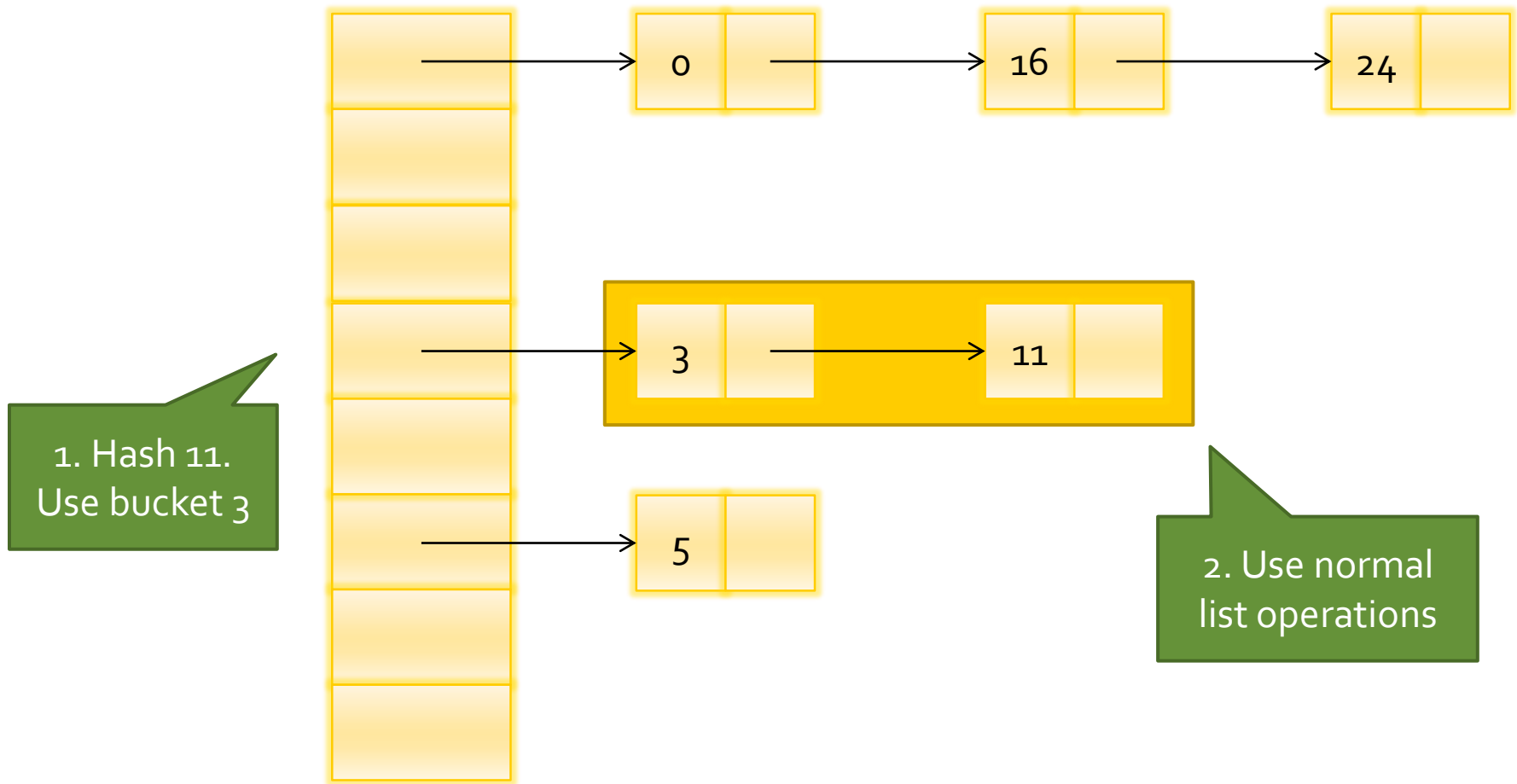




# Hash tables: Contains(16)



# Hash tables: Delete(11)



# Lessons from this hashtable

- Informal correctness argument:
  - Operations on different buckets don't conflict: no extra concurrency control needed
  - Operations appear to occur atomically at the point where the underlying list operation occurs
- (Not specific to lock-free lists: could use whole-table lock, or per-list locks, etc.)

# Practical difficulties:

- Key-val
- Popu
- Itera
- Resi

Options to consider when implementing a “difficult” operation:

Relax the semantics  
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted  
(e.g., lock the whole table for resize)

Design a clever implementation  
(e.g., split-ordered lists)

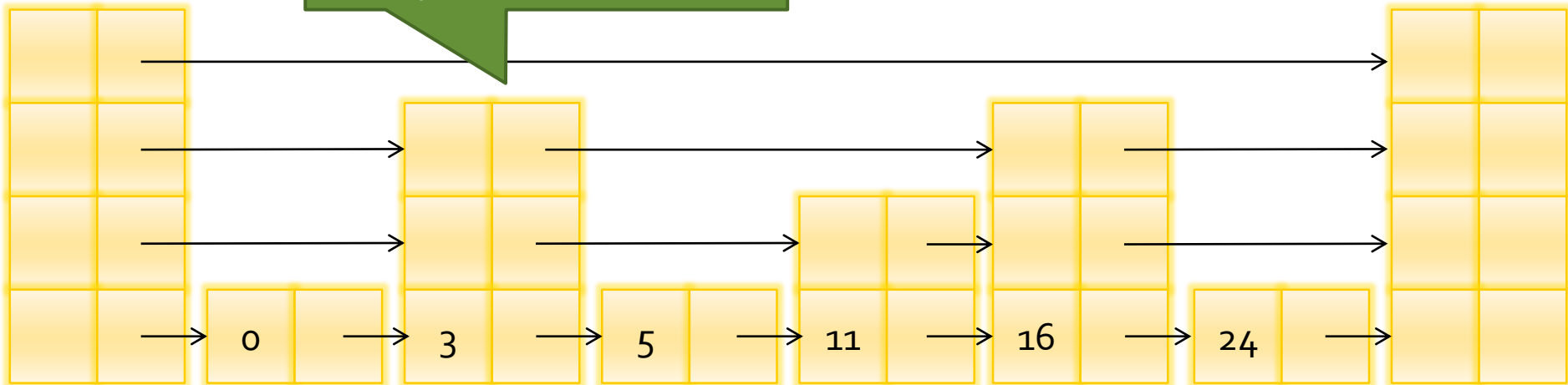
Use a different data structure  
(e.g., skip lists)

# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# Skip lists

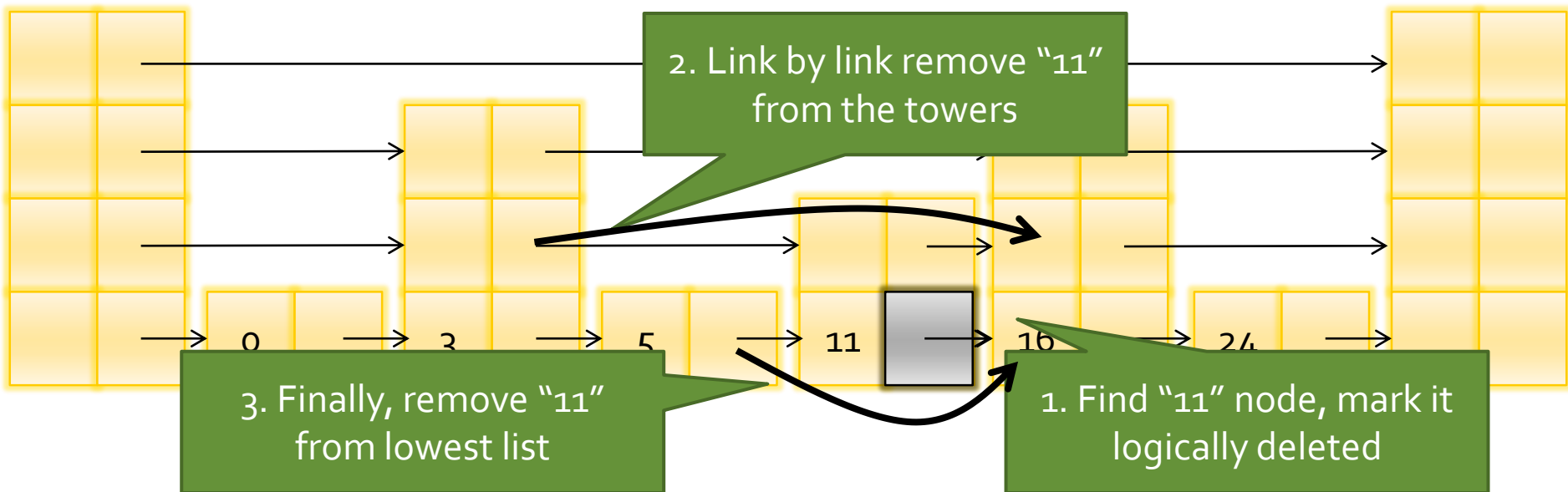
Each node is a "tower" of random size. High levels skip over lower levels



All items in a single list:  
this defines the set's  
contents

# Skip lists: Delete(11)

Principle: lowest list is the truth

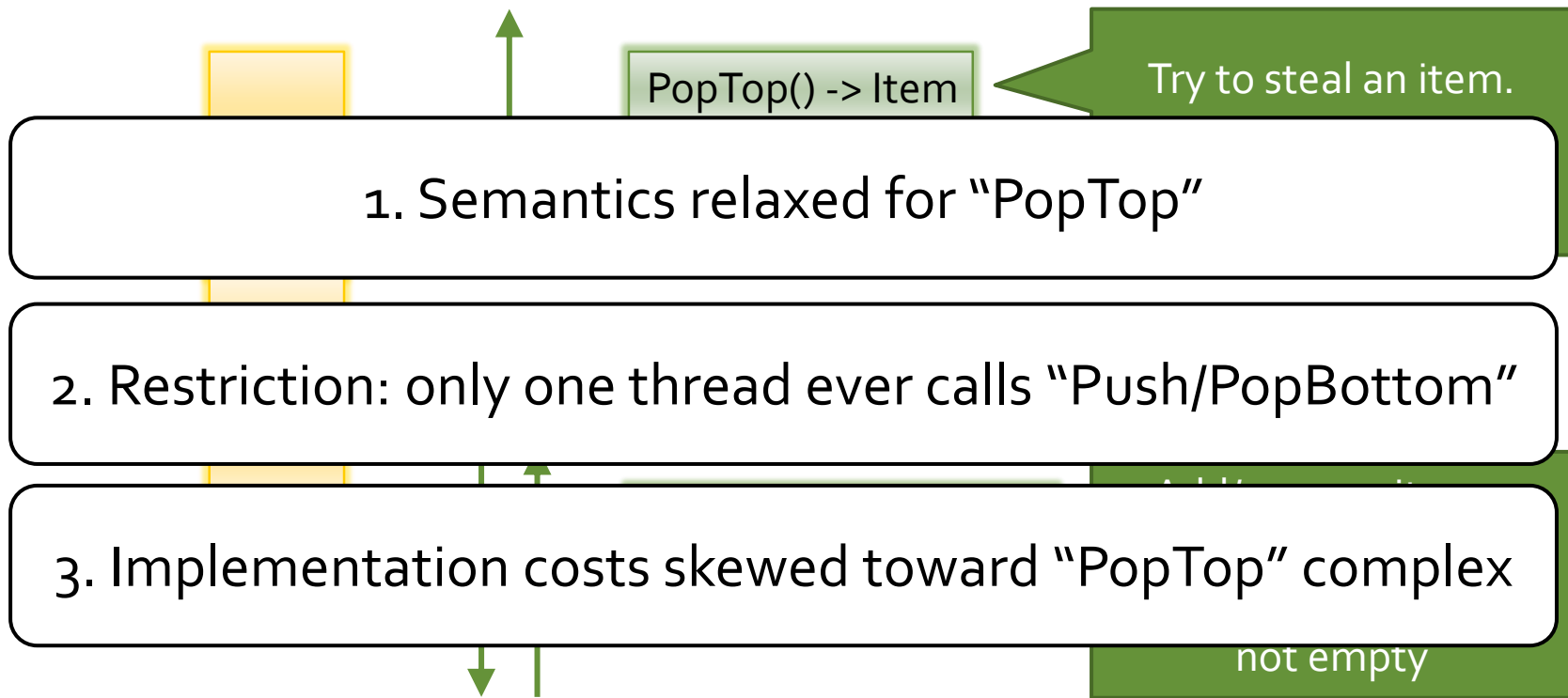


# Course overview: structure

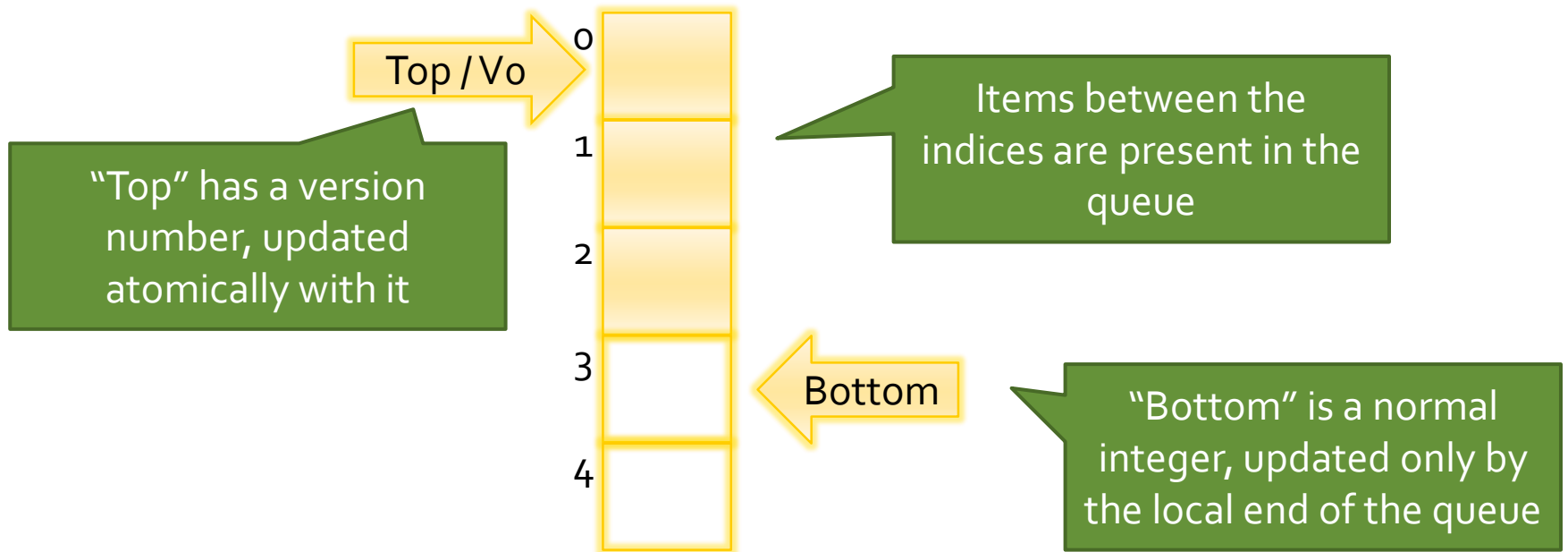
- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory



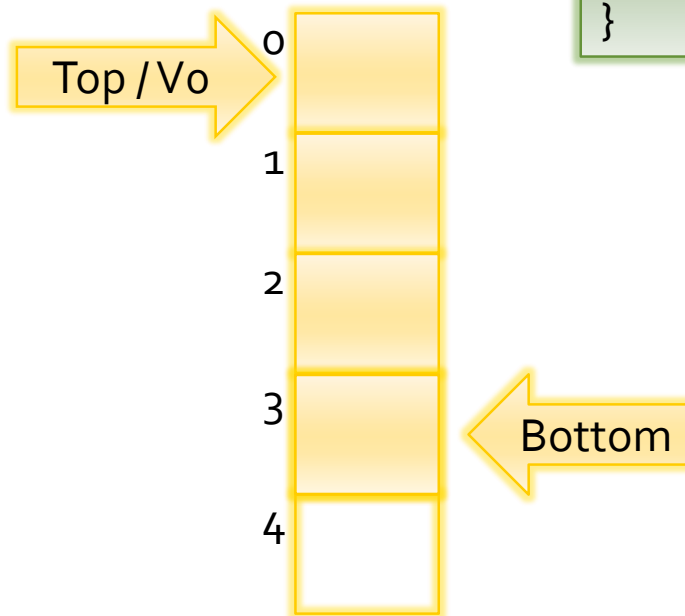
# Work stealing queues



# Bounded deque

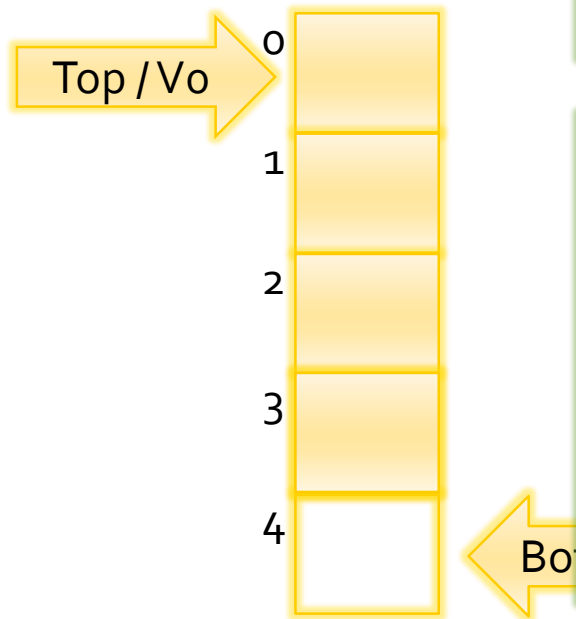


# Bounded deque



```
void pushBottom(Item i){  
    tasks[bottom] = i;  
    bottom++;  
}
```

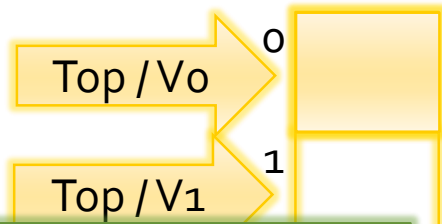
# Bounded deque



```
void pushBottom(Item i){  
    tasks[bottom] = i;  
    bottom++;  
}
```

```
Item popBottom() {  
    if (bottom == 0) return null;  
    bottom--;  
    result = tasks[bottom];  
    <tmp_top,tmp_v> = <top,version>;  
    if (bottom > tmp_top) return result;  
    ....  
    return null;  
}
```

# Bounded deque



```

Item popTop() {
    if (bottom <= top) return null;
    <tmp_top,tmp_v> = <top, version>;
    result = tasks[tmp_top];
    if (CAS( &<top,version>,
            <tmp_top, tmp_v>,
            <tmp_top+1, v+1>)) {
        return result;
    }
    return null;
}
    
```

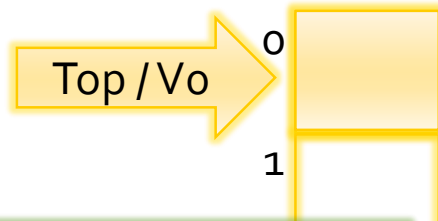
```

void pushBottom(Item i){
    tasks[bottom] = i;
    bottom++;
}
    
```

```

Item popBottom() {
    if (bottom == 0) return null;
    if (bottom==top) {
        bottom = 0;
        if (CAS( &<top,version>,
                <tmp_top,tmp_v>,
                <0,v+1>)) {
            return result;
        }
    }
    <top,version>=<0,v+1>
}
    
```

# Bounded deque



```

Item popTop() {
  if (bottom <= top) return null;
  <tmp_top,tmp_v> = <top, version>;
  result = tasks[tmp_top];
  if (CAS( &<top,version>,
          <tmp_top, tmp_v>,
          <tmp_top+1, v+1>)) {
    return result;
  }
  return null;
}

```

```

void pushBottom(Item i){
  tasks[bottom] = i;
  bottom++;
}

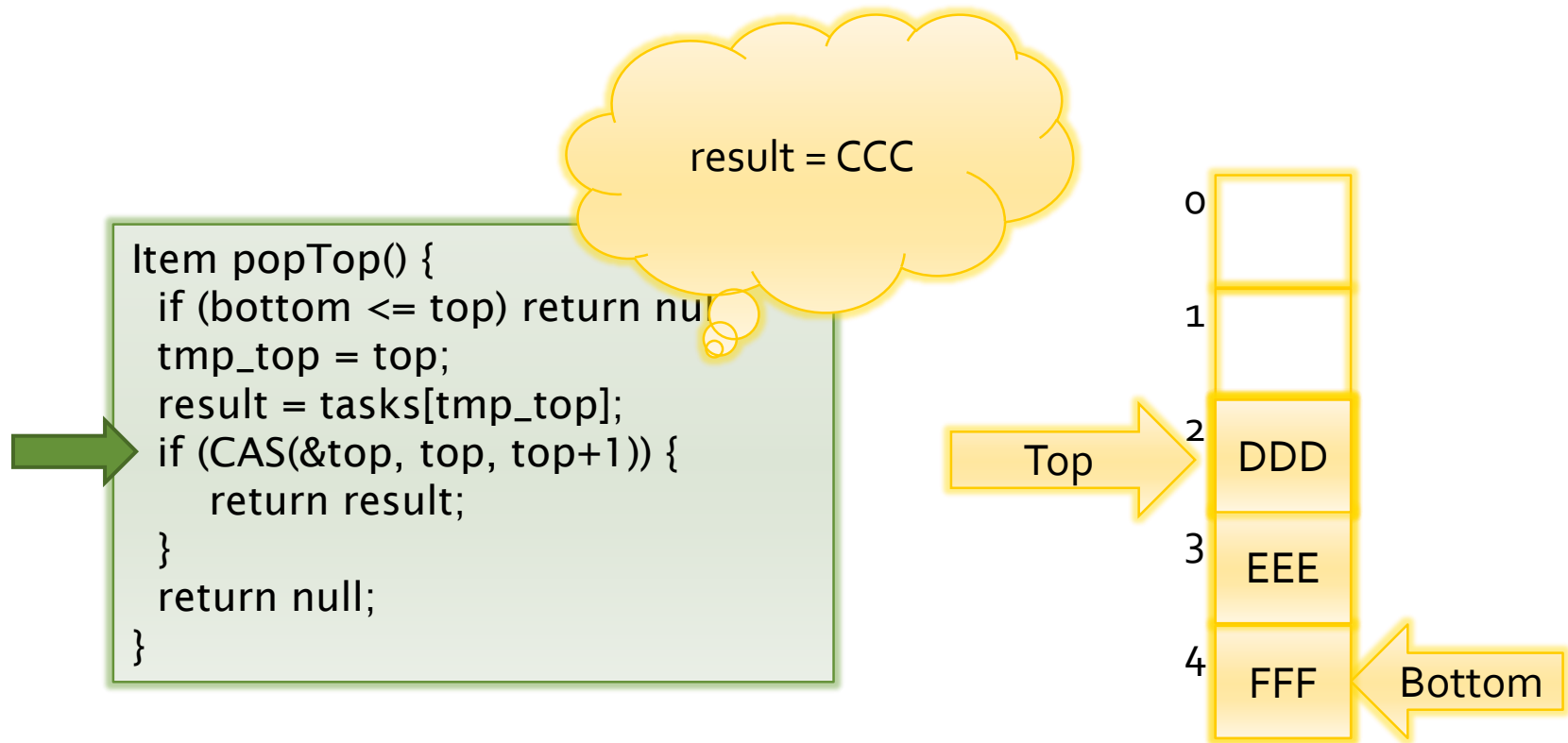
```

```

Item popBottom() {
  if (bottom == 0) return null;
  if (bottom == top) {
    bottom = 0;
    if (CAS( &<top,version>,
            <tmp_top,tmp_v>,
            <0,v+1>)) {
      return result;
    }
  }
  <top,version>=<0,v+1>
}

```

# ABA problems



# General techniques

- Local operations designed to avoid CAS
  - Traditionally slower, less so now
  - Costs of memory fences can be important (“Idempotent work stealing”, Michael et al)
- Local operations just use read and write
  - Only one accessor, check for interference
- Use CAS:
  - Resolve conflicts between stealers
  - Resolve local/stealer conflicts
  - Version number to ensure conflicts seen



# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

# Reducing contention

- Suppose you're implementing a shared counter with the following sequential spec:

```
void increment(int *counter) {  
    atomic {  
        (*counter) ++;  
    }  
}
```

```
void decrement(int *counter) {  
    atomic {  
        (*counter) --;  
    }  
}
```

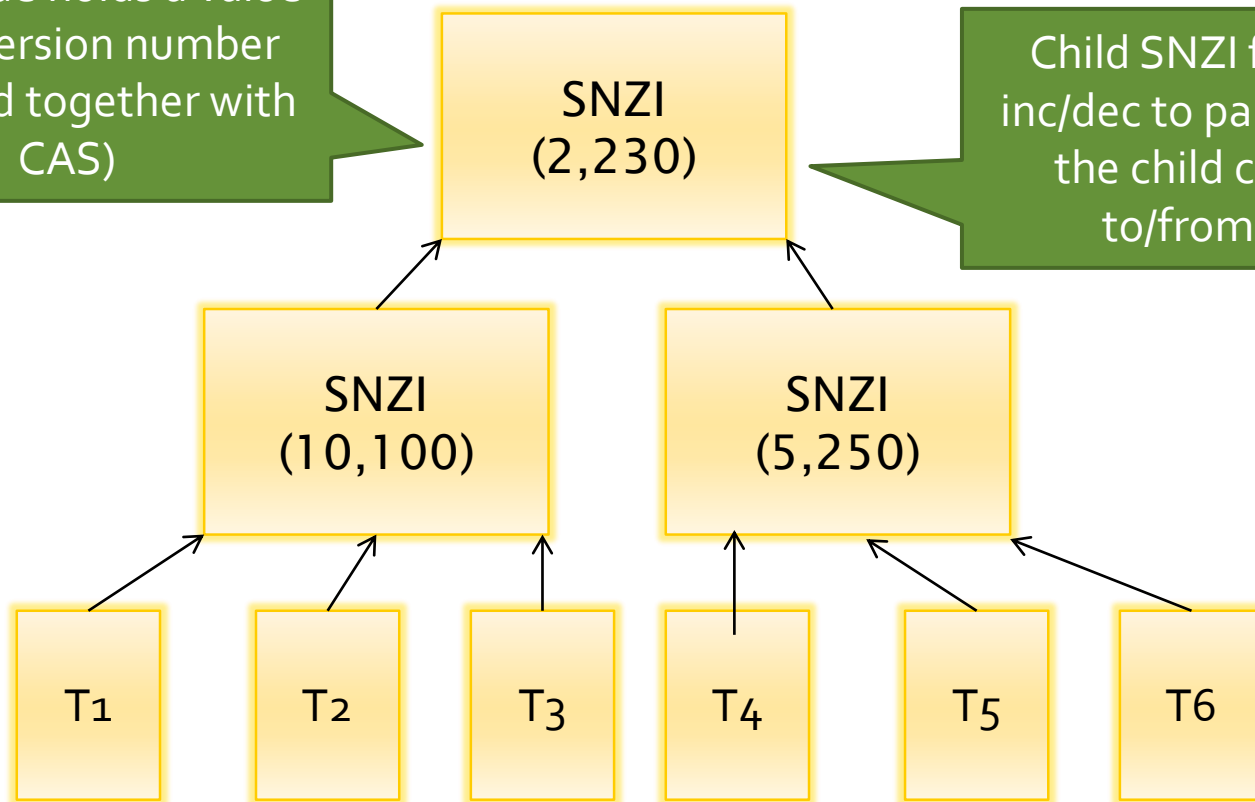
```
bool isZero(int *counter) {  
    atomic {  
        return (*counter) == 0;  
    }  
}
```

How well can this scale?

# SNZI trees

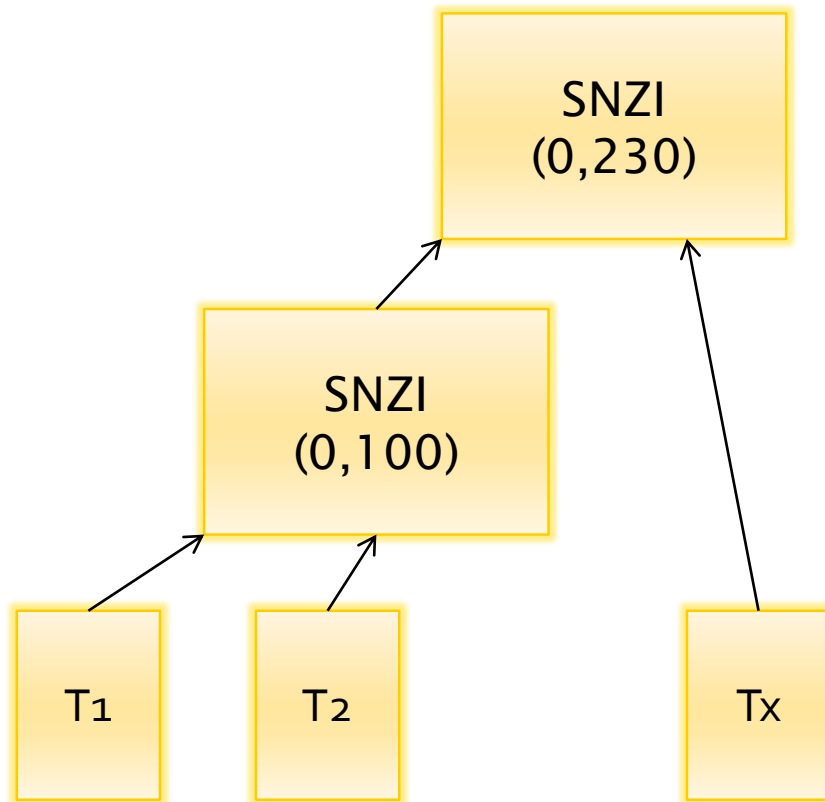
Each node holds a value and a version number (updated together with CAS)

Child SNZI forwards inc/dec to parent when the child changes to/from zero



SNZI: Scalable NonZero Indicators, Ellen et al

# SNZI trees, linearizability on 0->1 change

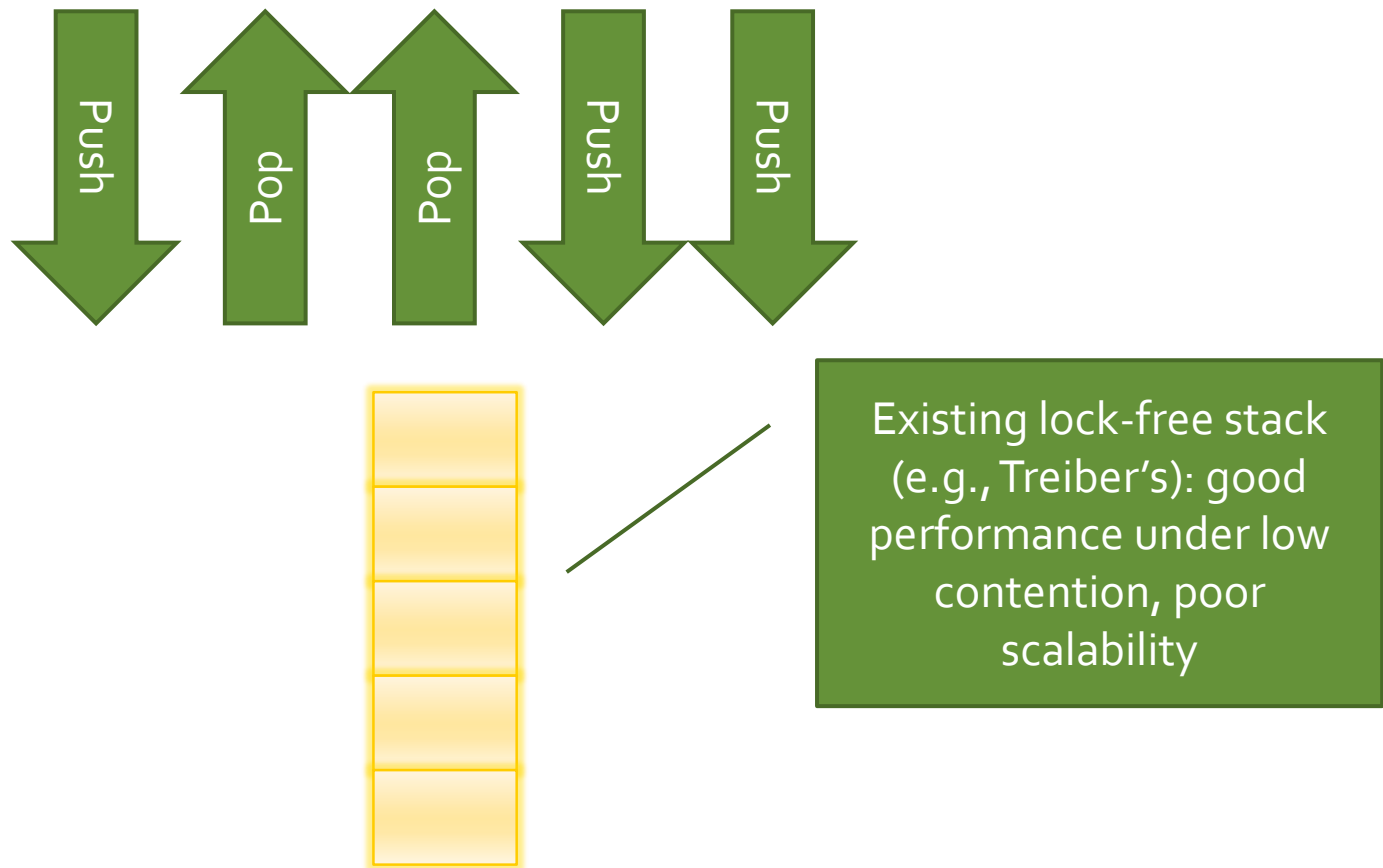


1. T<sub>1</sub> calls increment
2. T<sub>1</sub> increments child to 1
3. T<sub>2</sub> calls increment
4. T<sub>2</sub> increments child to 2
5. T<sub>2</sub> completes
6. Tx calls isZero
7. Tx sees 0 at parent
8. T<sub>1</sub> calls increment on parent
9. T<sub>1</sub> completes

# SNZI trees

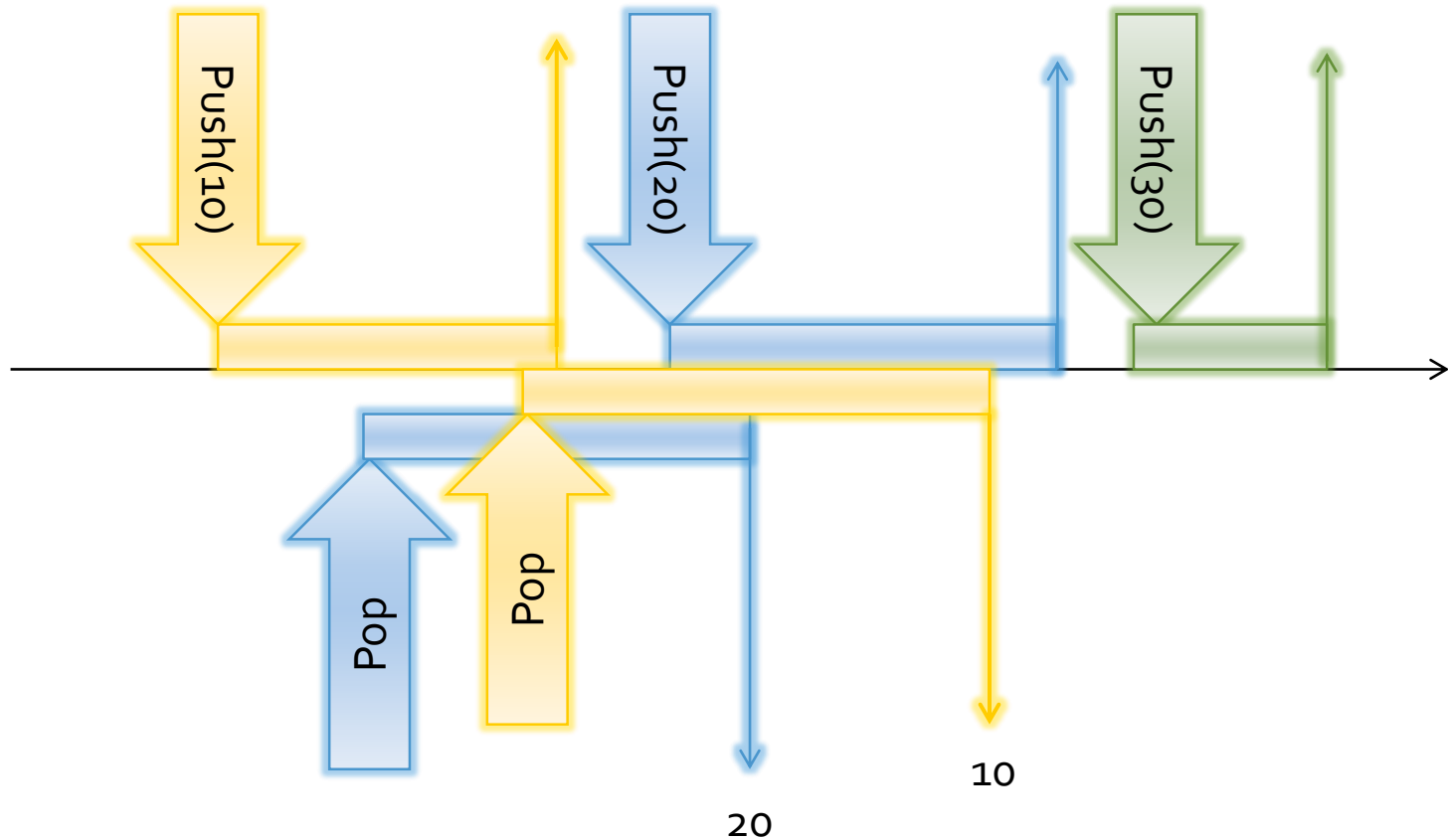
```
void increment(snzi *s) {
    bool done=false;
    int undo=0;
    while(!done) {
        <val,ver> = read(s->state);
        if (val >= 1 && CAS(s->state, <val,ver>, <val+1,ver>)) { done = true; }
        if (val == 0 && CAS(s->state, <val,ver>, <1/2, ver+1>)) {
            done = true; val=1/2; ver=ver+1
        }
        if (val == 1/2) {
            increment(s->parent);
            if (!CAS(s->state, <val, ver>, <1, ver>)) { undo ++; }
        }
    }
    while (undo > 0) {
        decrement(s->parent);
    }
}
```

# Reducing contention: stack

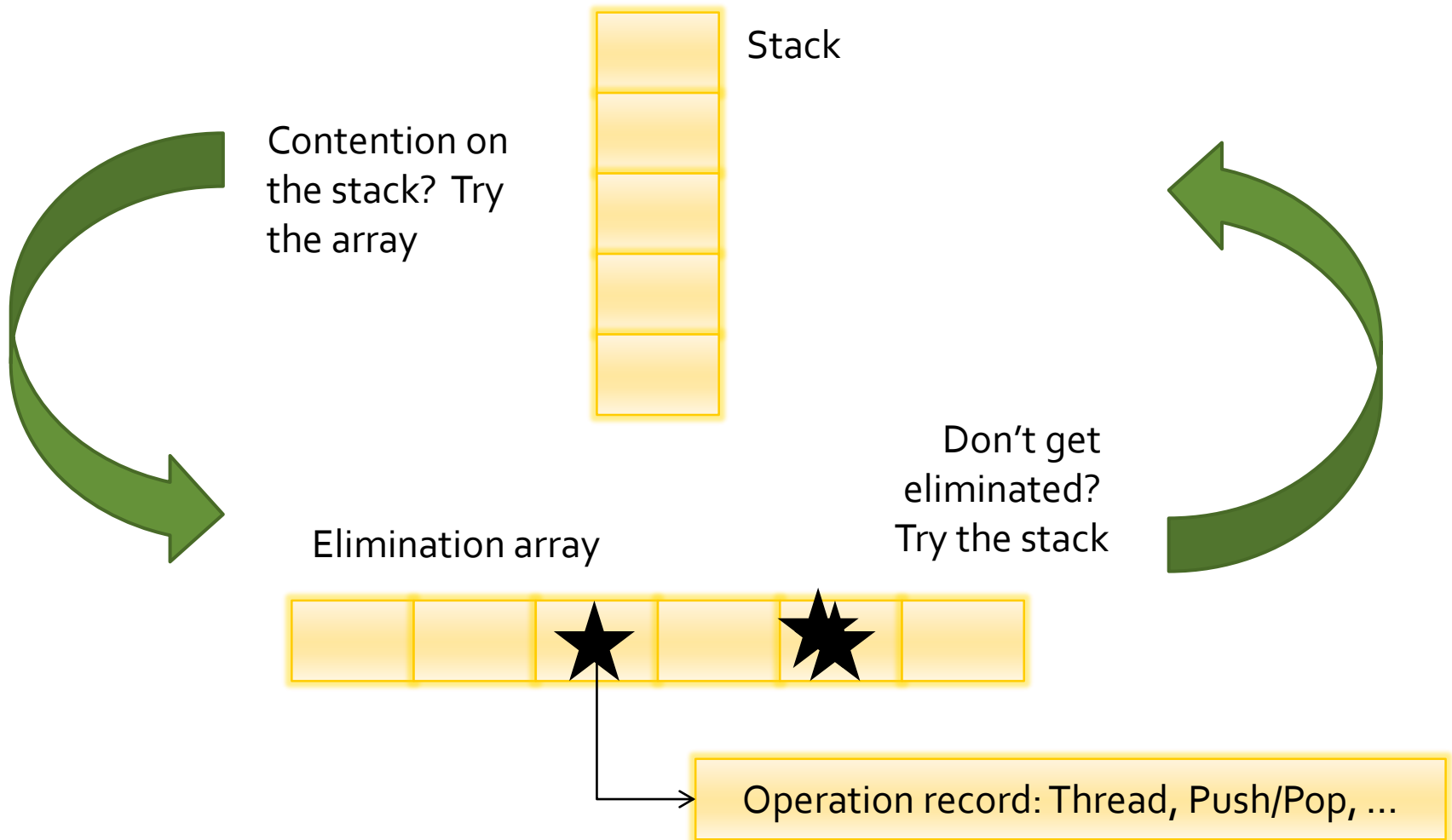


A scalable lock-free stack algorithm, Hendler et al

# Pairing up operations



# Back-off elimination array





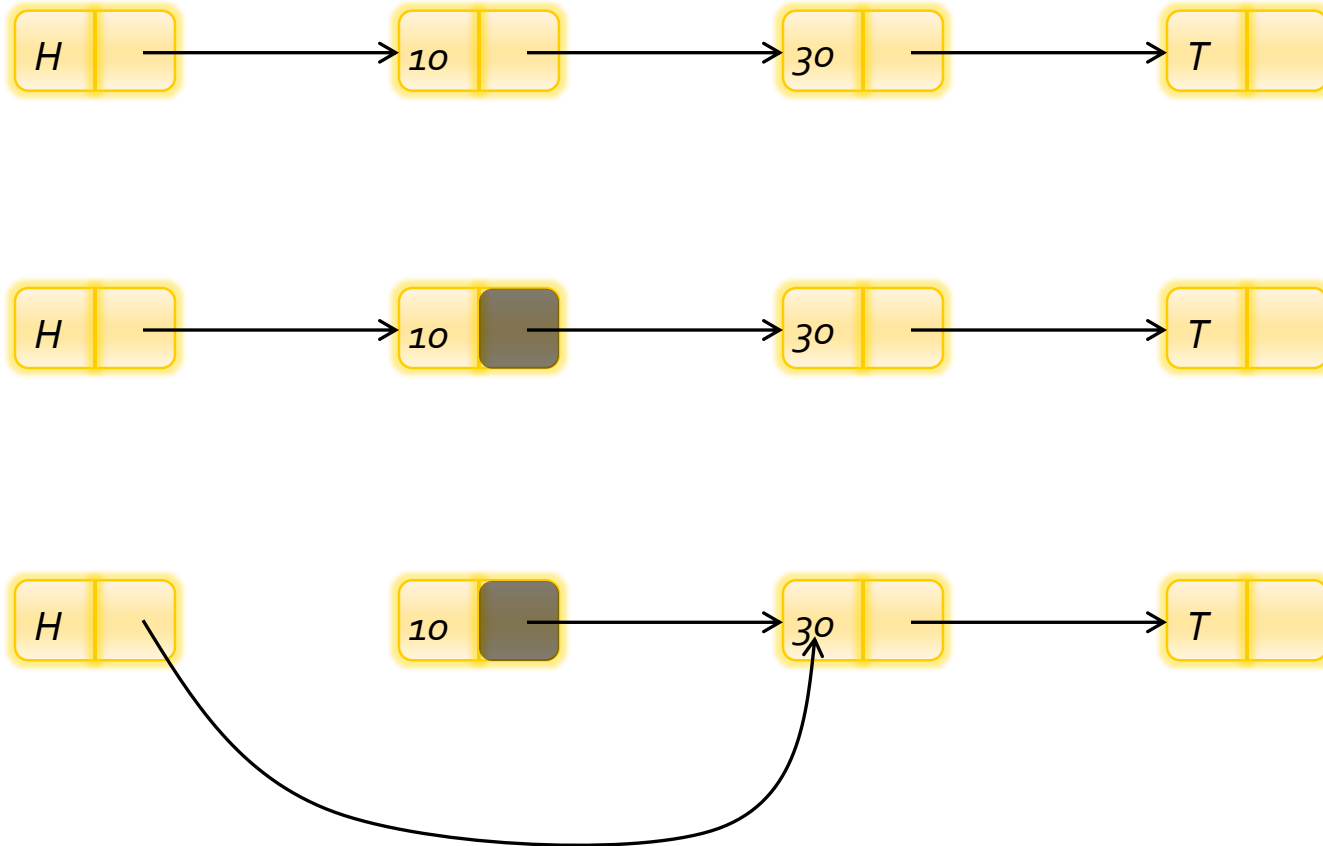
# Course overview: structure

- Building locks
- Lock-free programming
  - What's wrong with locks?
  - Lists without locks, linearizability
  - Lock-free progress
  - Hashtables
  - Skiplists
  - Queues
  - Reducing contention
  - Memory management
- Transactional memory

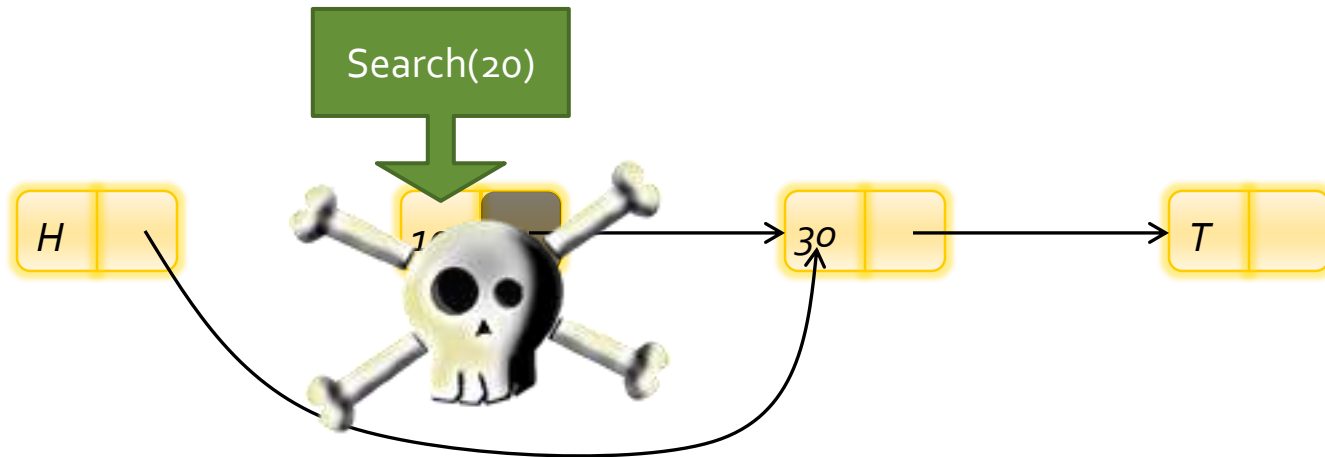
# Lock-free data structures in C

- Pseudo-code, Java, C#:
  - Explicit memory allocation
  - Deallocation by GC
- C/C++:
  - Explicit memory allocation & deallocation
- When is it safe to deallocate a piece of memory?

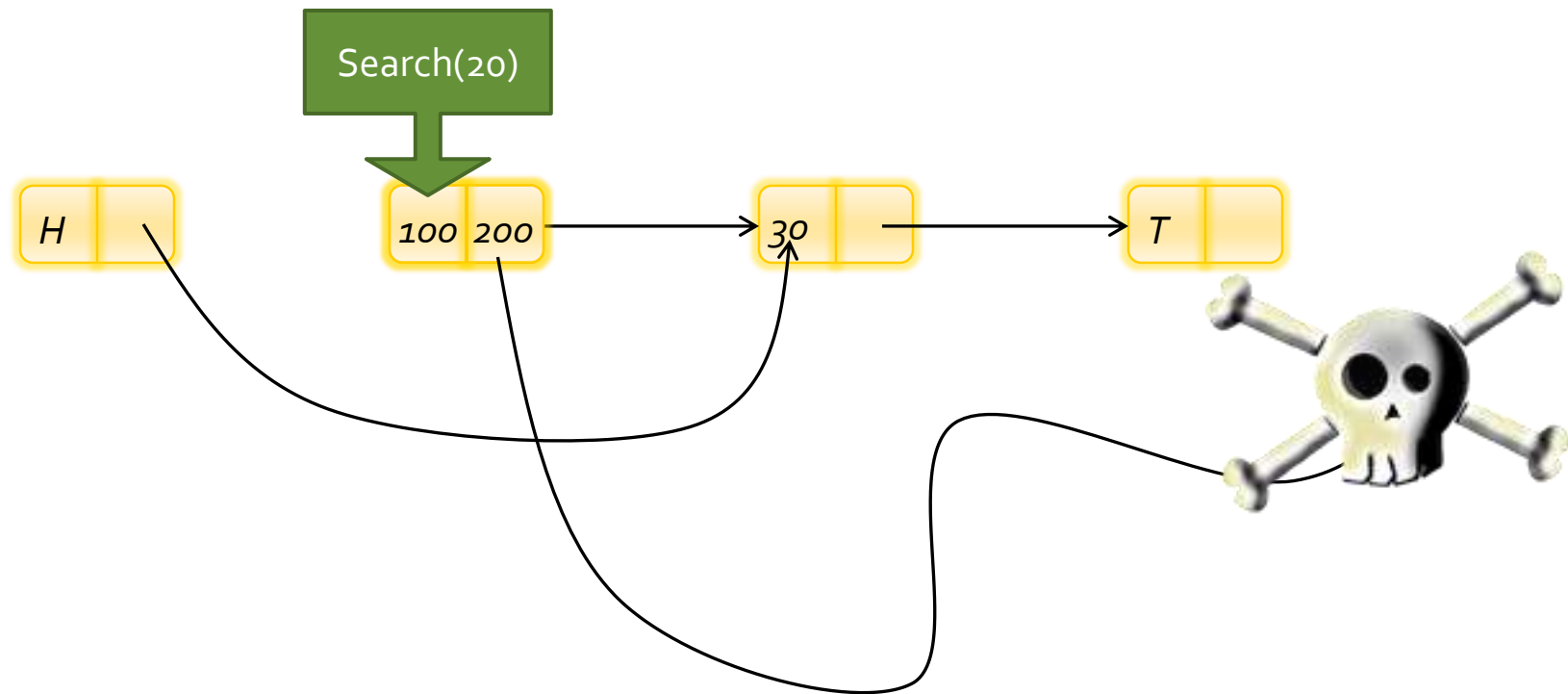
# Deletion revisited: Delete(10)



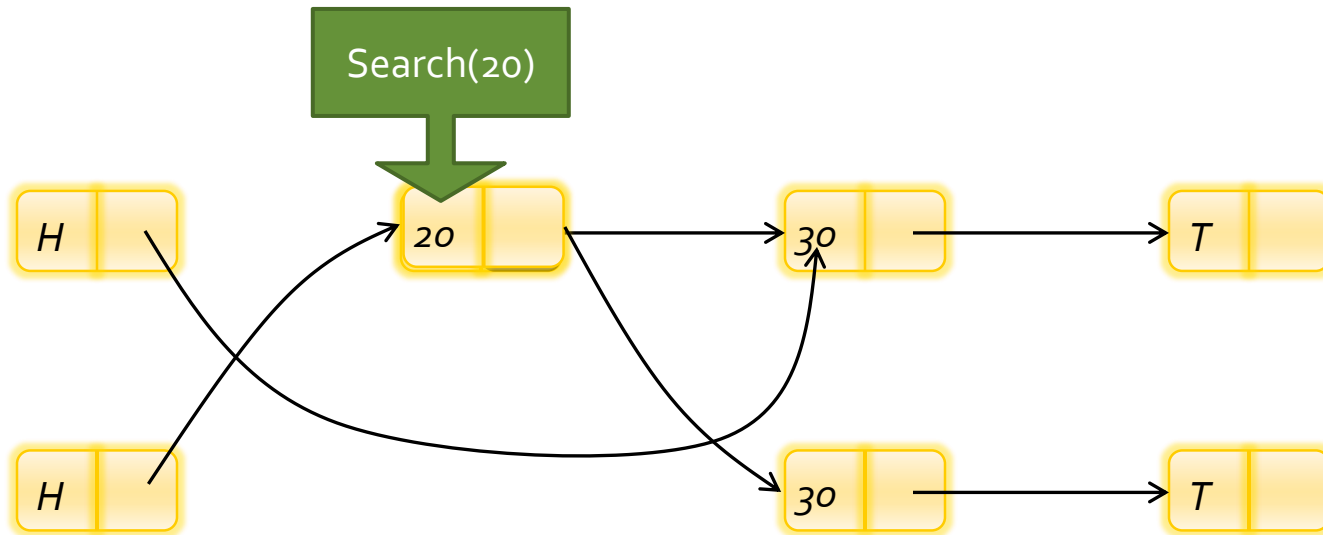
# De-allocate to the OS?



# Re-use as something else?

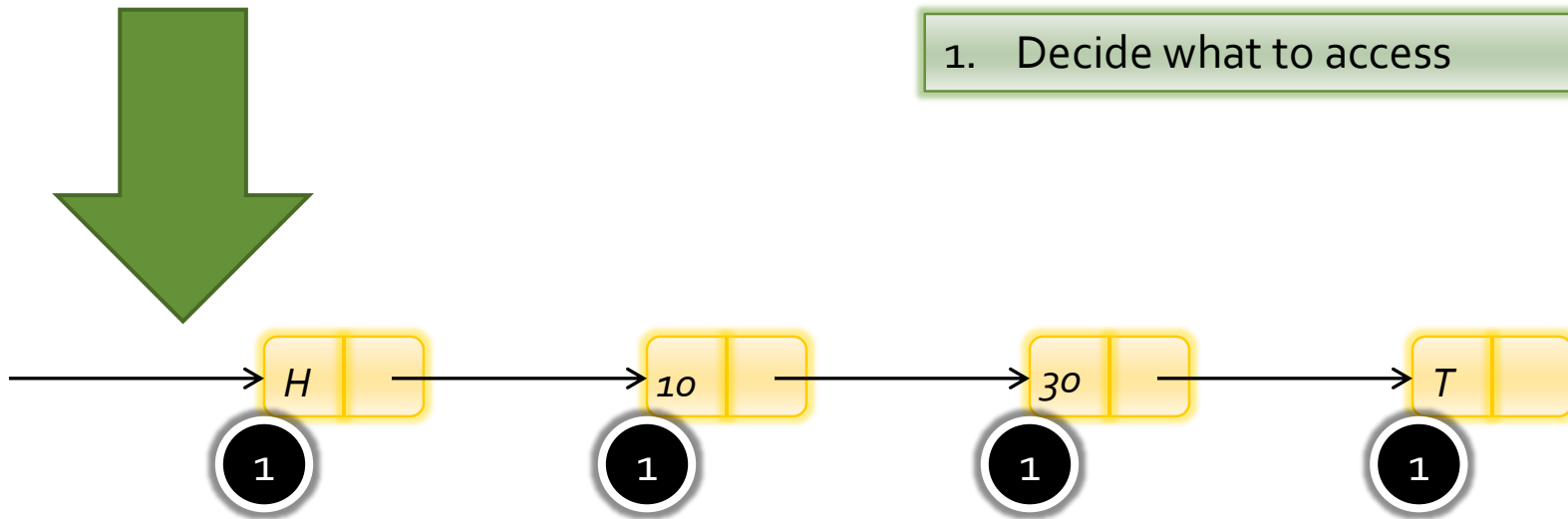


# Re-use as a list node?



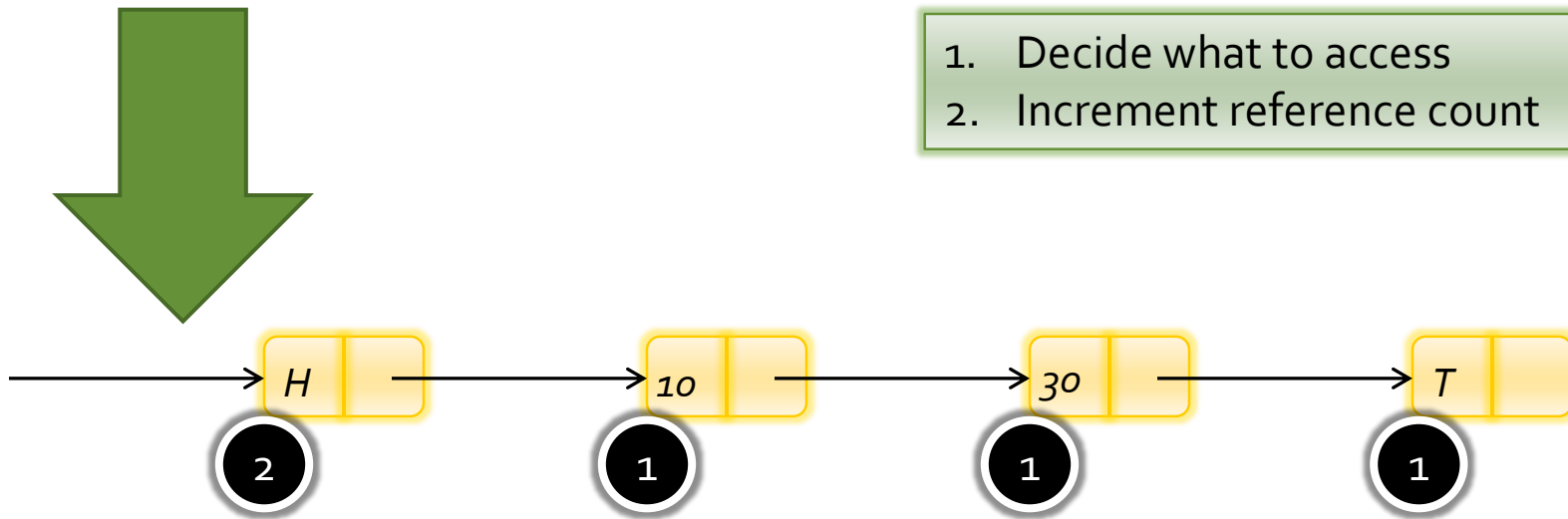
# Reference counting

1. Decide what to access



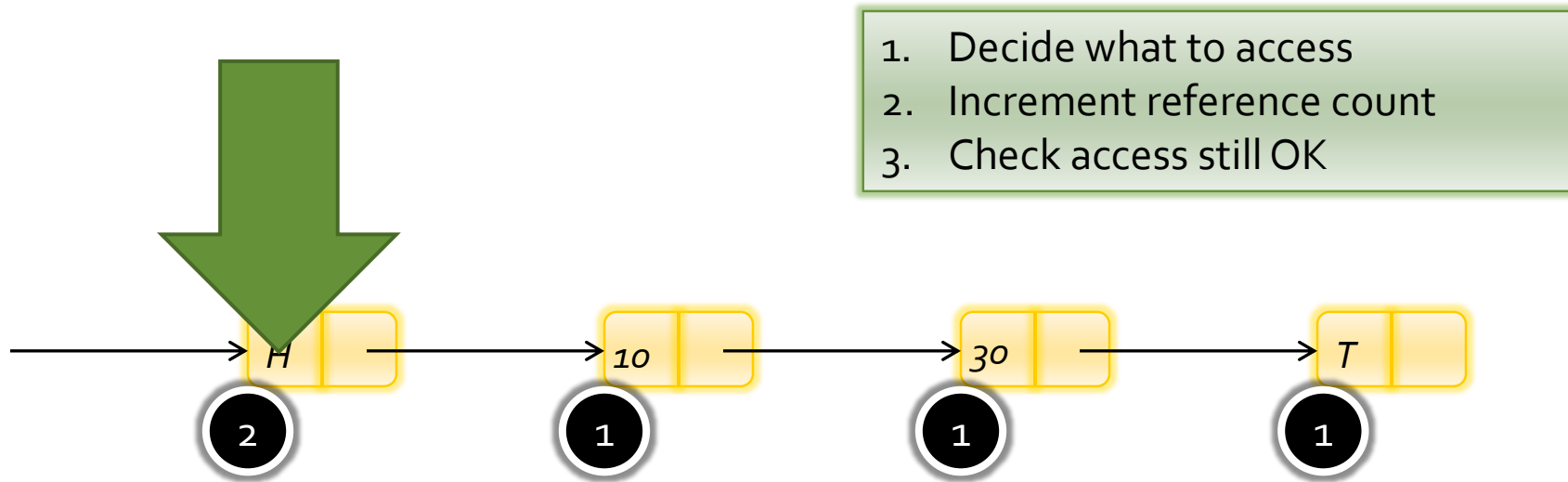
# Reference counting

1. Decide what to access
2. Increment reference count

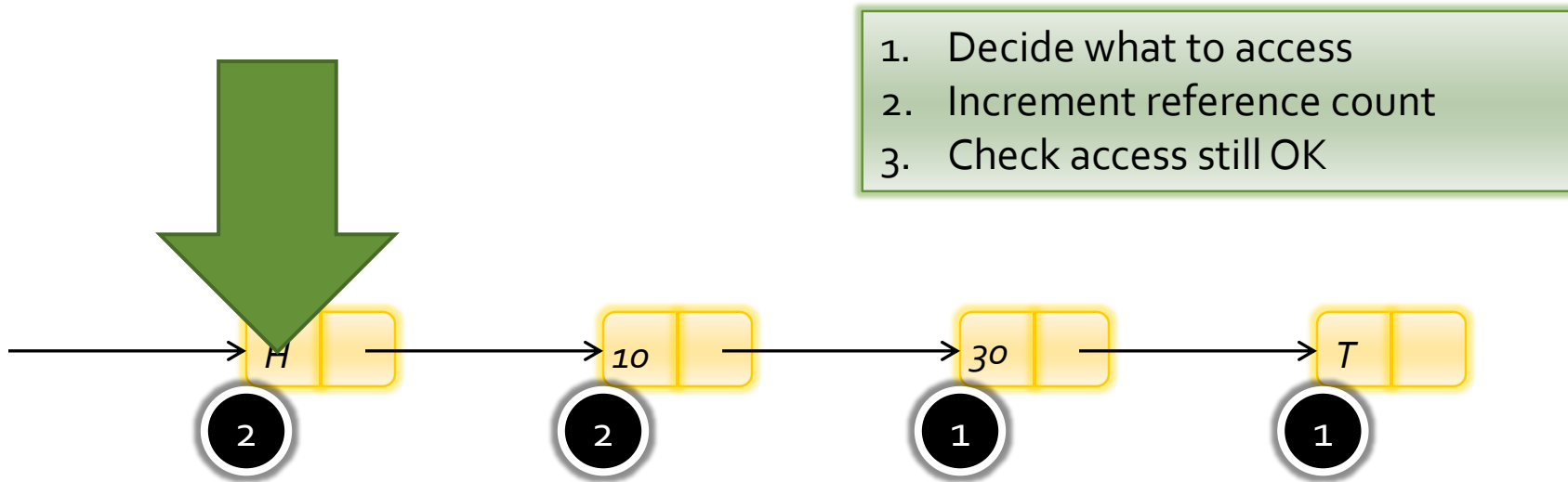




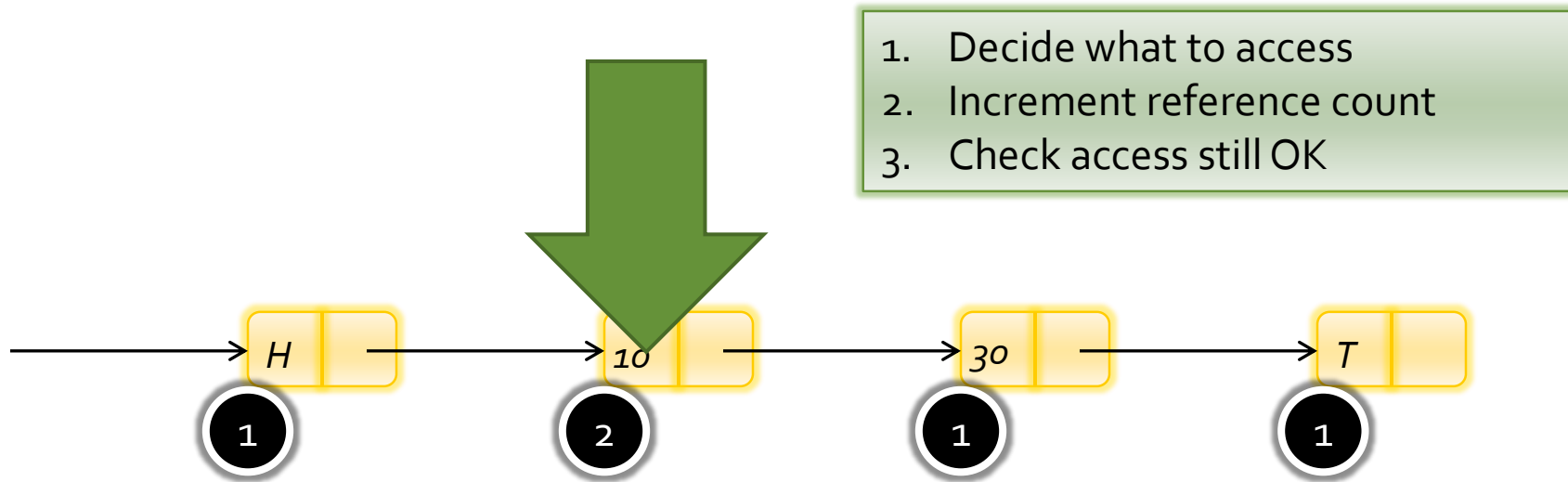
# Reference counting



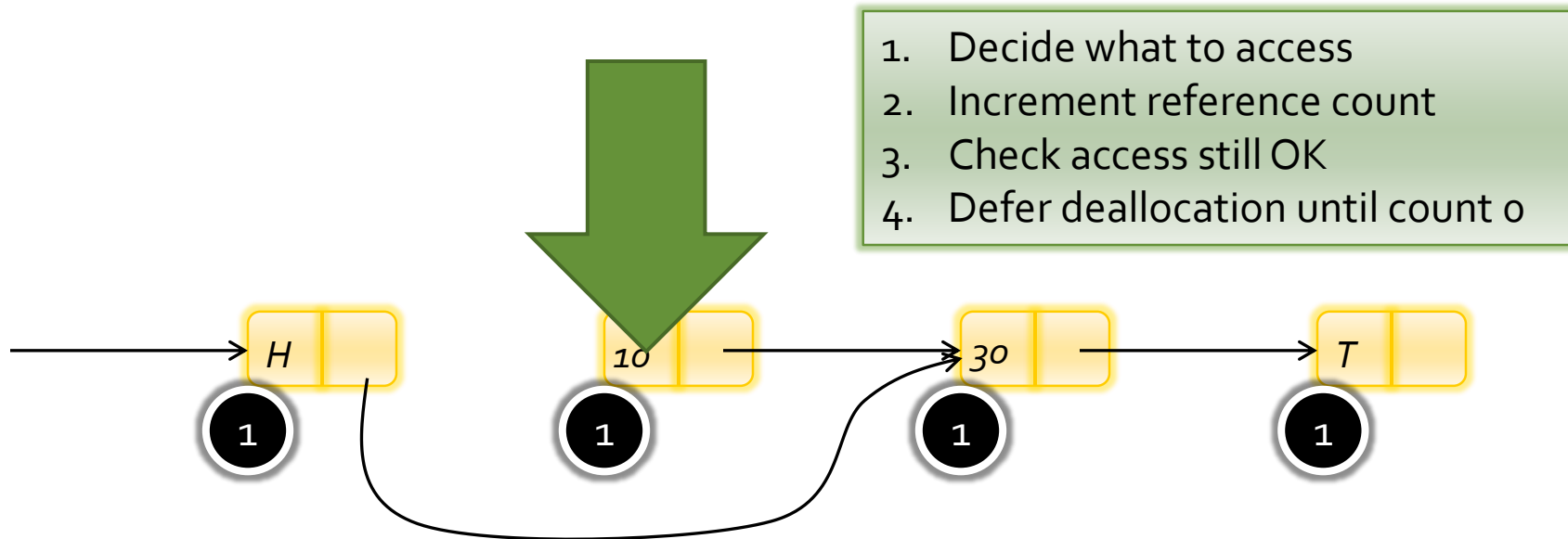
# Reference counting



# Reference counting

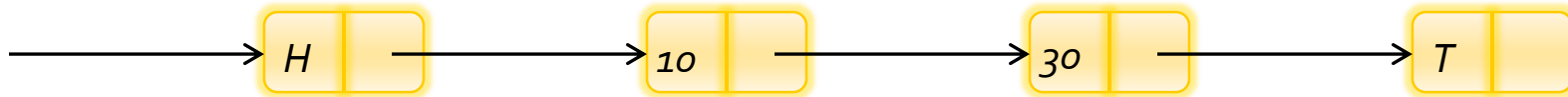


# Reference counting



# Epoch mechanisms

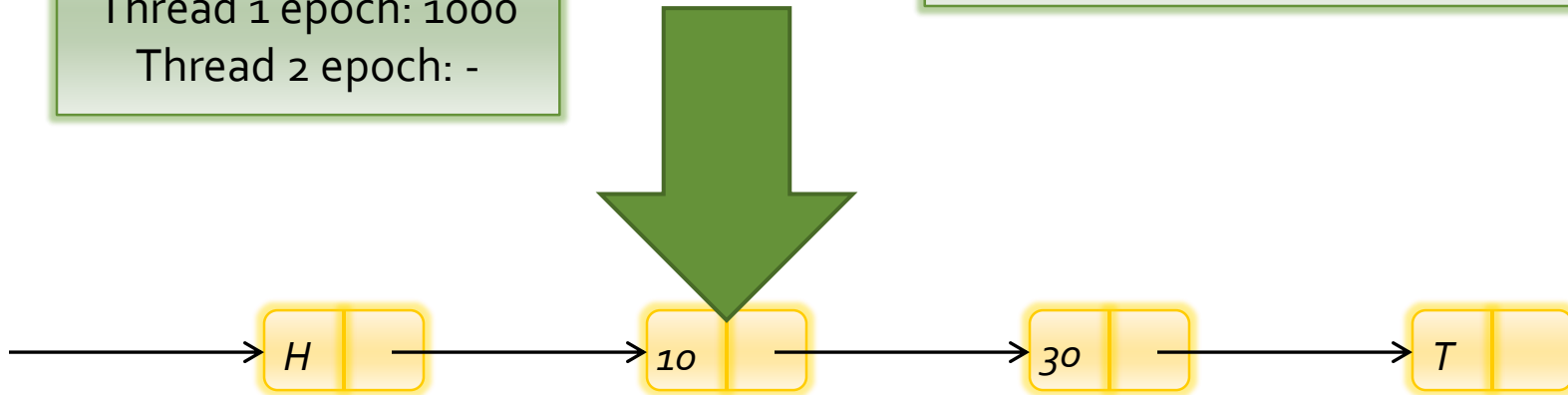
Global epoch: 1000  
Thread 1 epoch: -  
Thread 2 epoch: -



# Epoch mechanisms

Global epoch: 1000  
Thread 1 epoch: 1000  
Thread 2 epoch: -

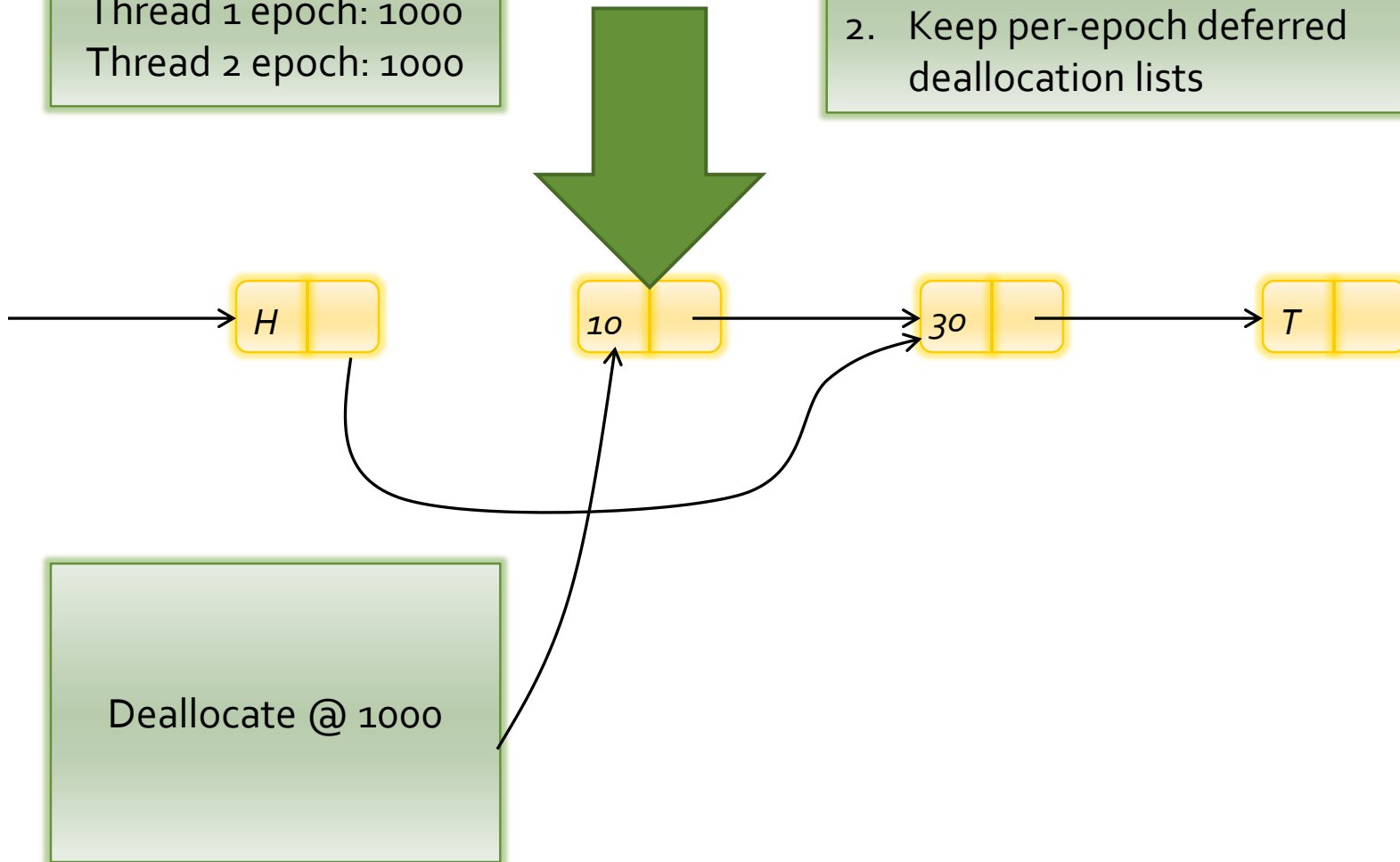
1. Record global epoch at start of operation



# Epoch mechanisms

Global epoch: 1000  
Thread 1 epoch: 1000  
Thread 2 epoch: 1000

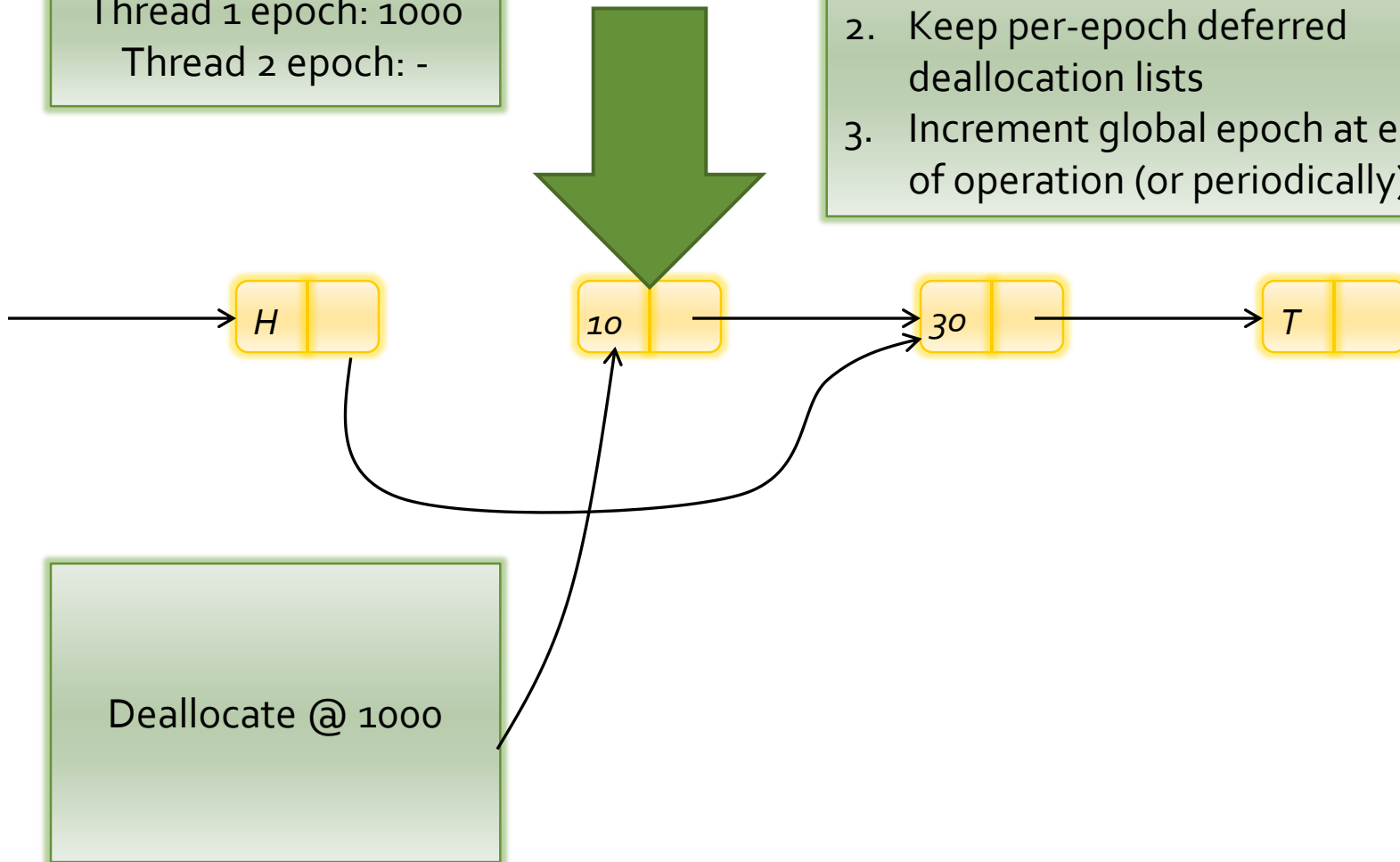
1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists



# Epoch mechanisms

Global epoch: 1001  
Thread 1 epoch: 1000  
Thread 2 epoch: -

1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)

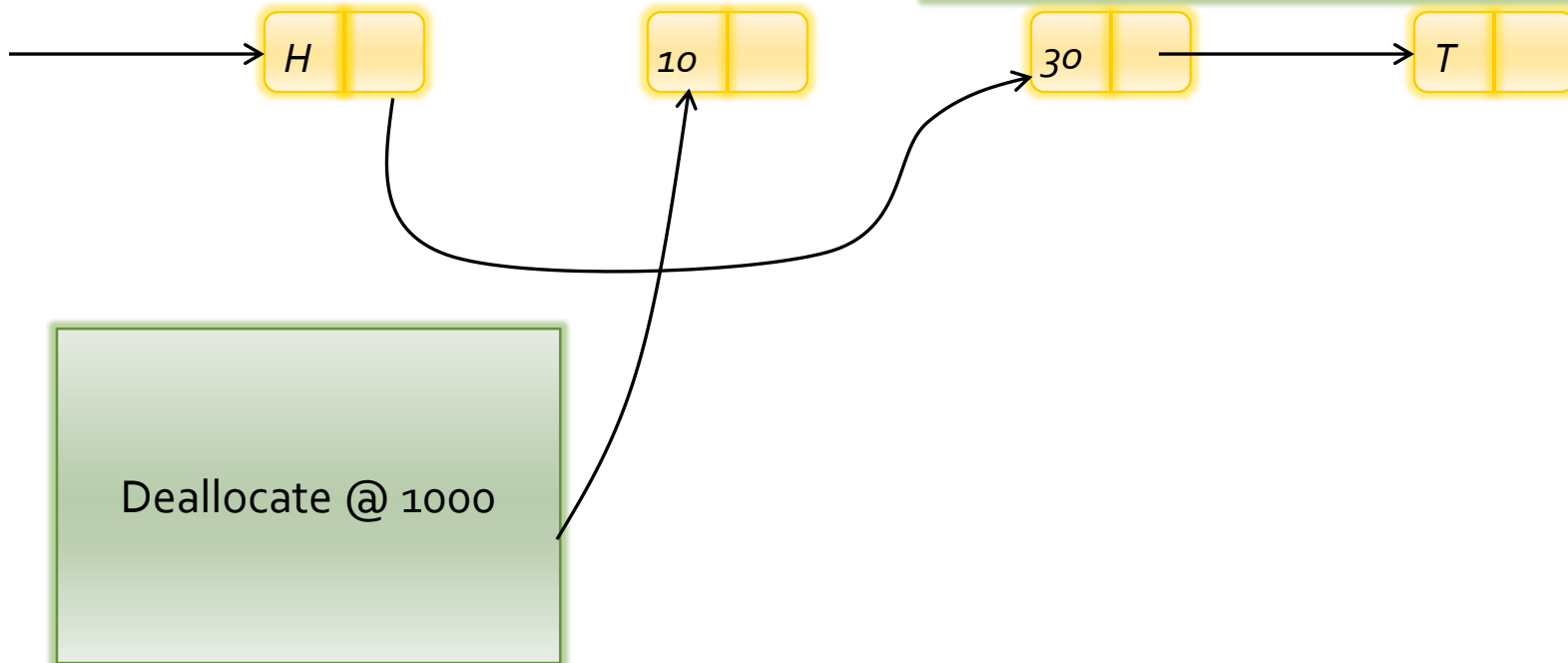




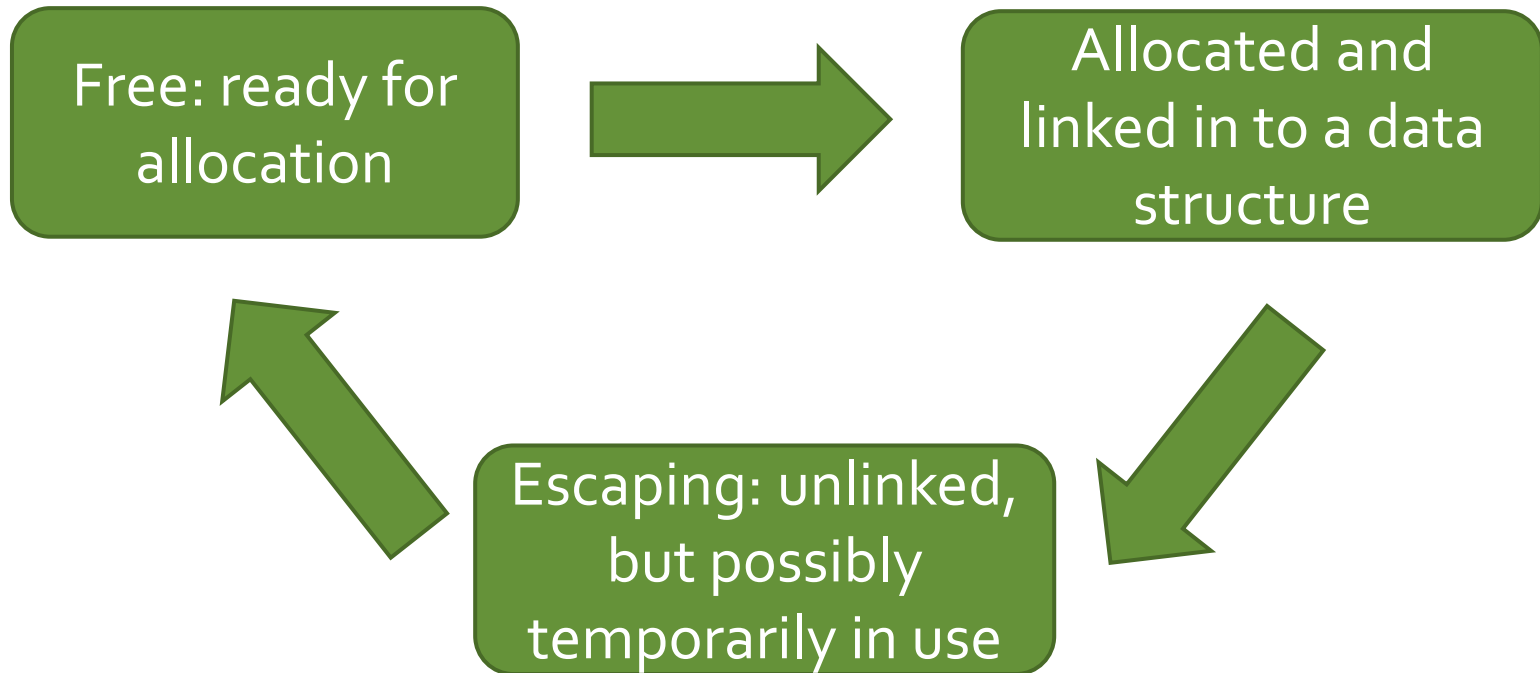
# Epoch mechanisms

Global epoch: 1002  
Thread 1 epoch: -  
Thread 2 epoch: -

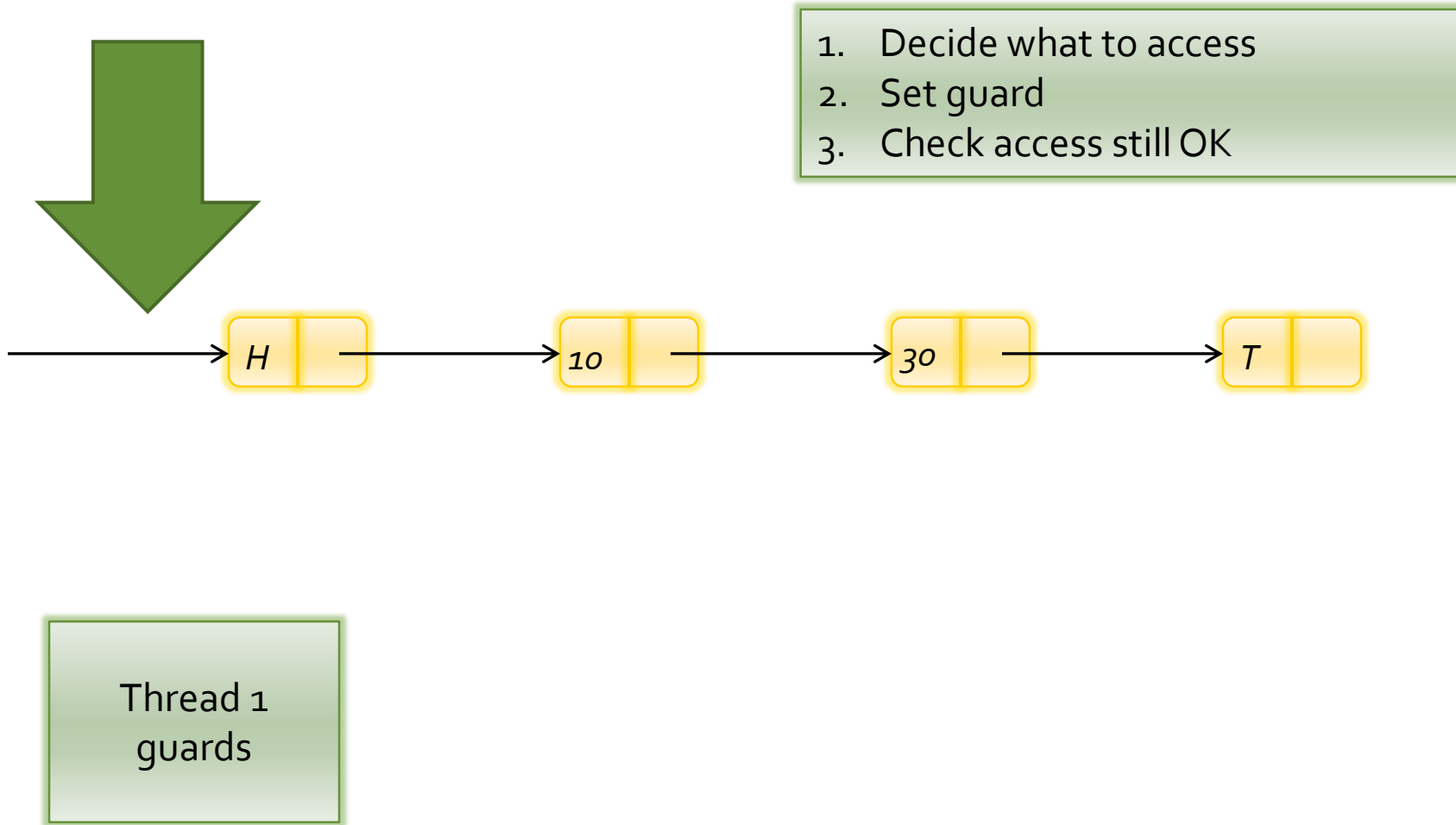
1. Record global epoch at start of operation
2. Keep per-epoch deferred deallocation lists
3. Increment global epoch at end of operation (or periodically)
4. Free when everyone past epoch



# The “repeat offender problem”

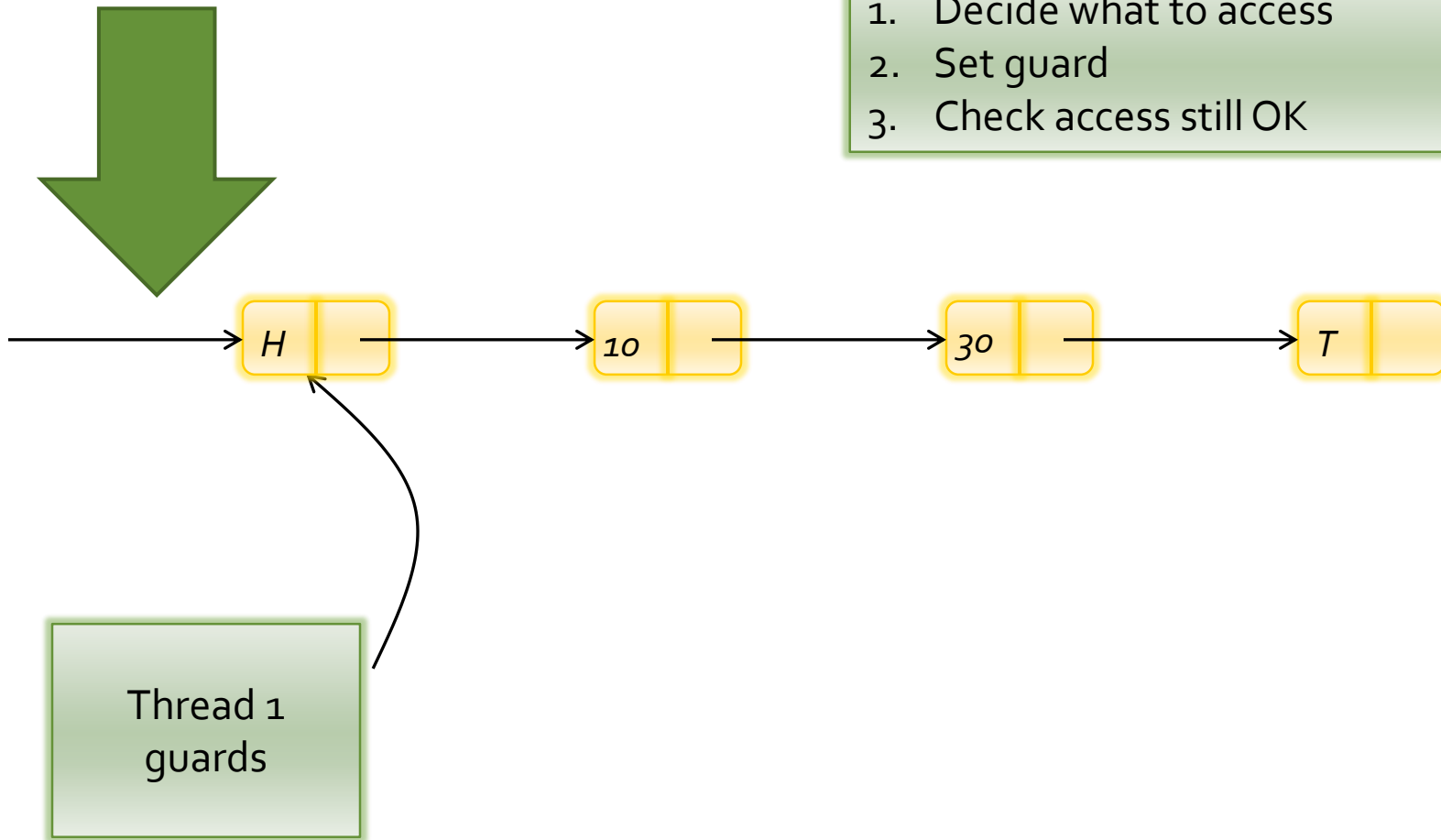


# Re-use via ROP

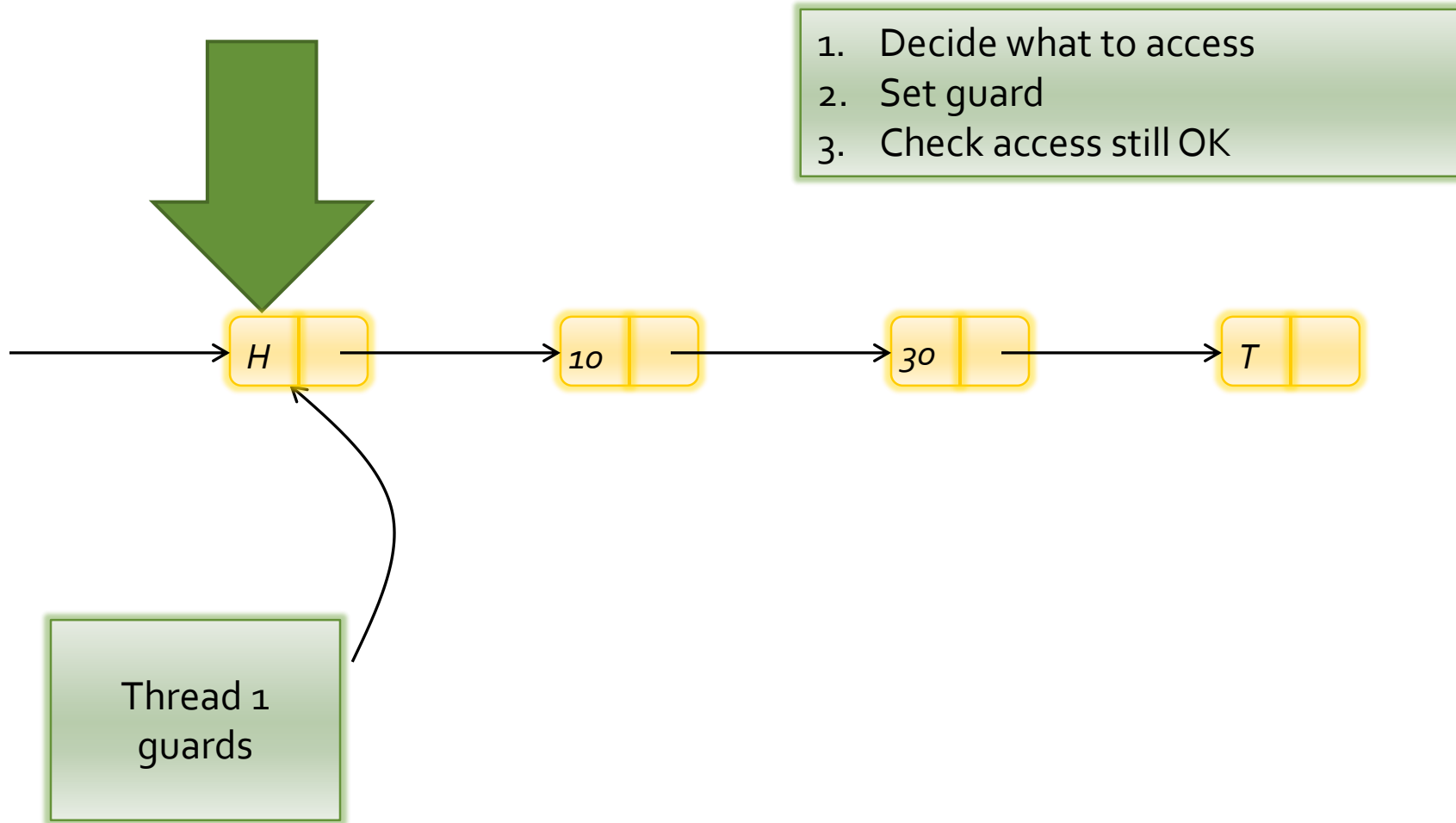


# Re-use via ROP

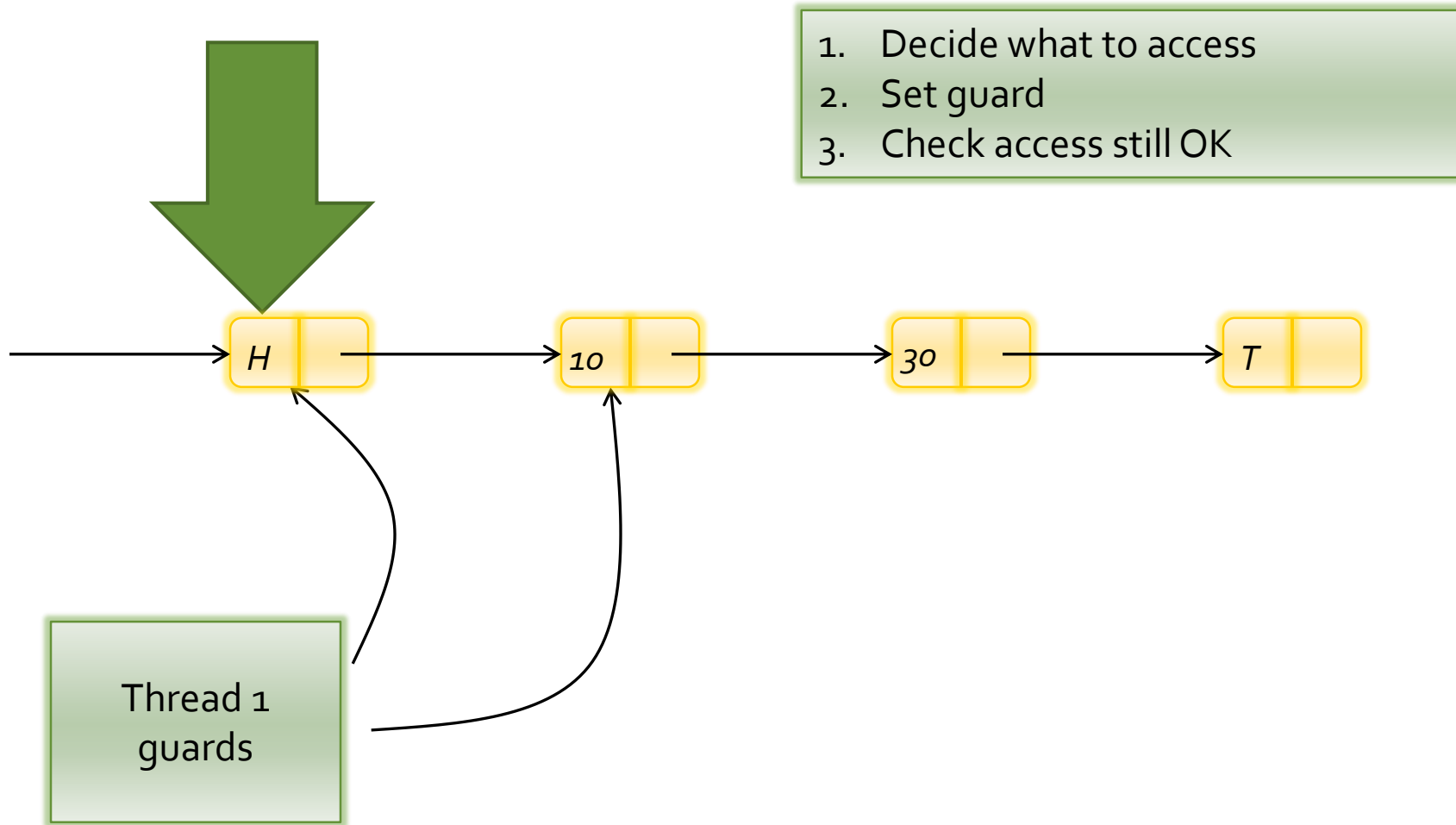
1. Decide what to access
2. Set guard
3. Check access still OK



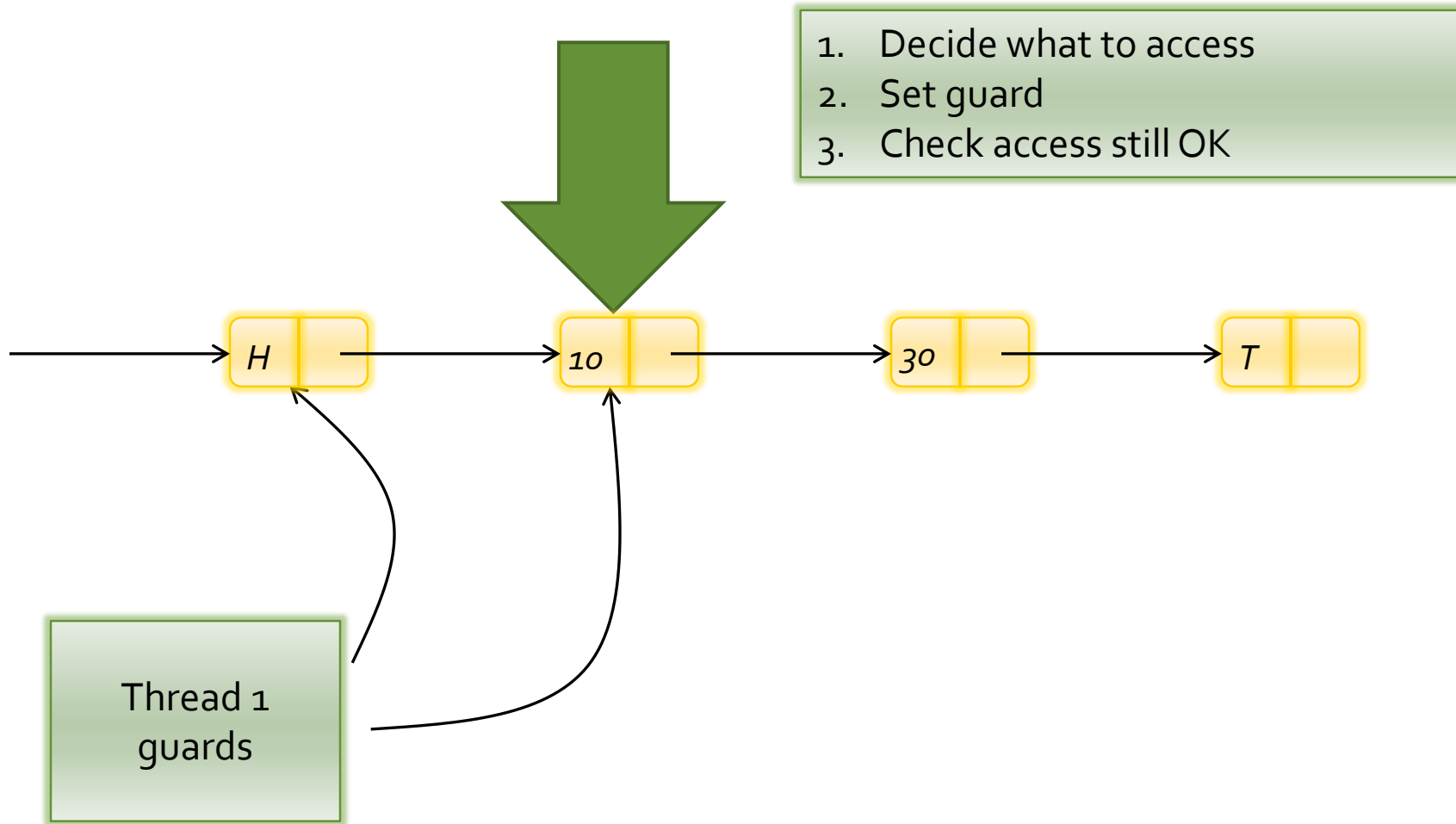
# Re-use via ROP



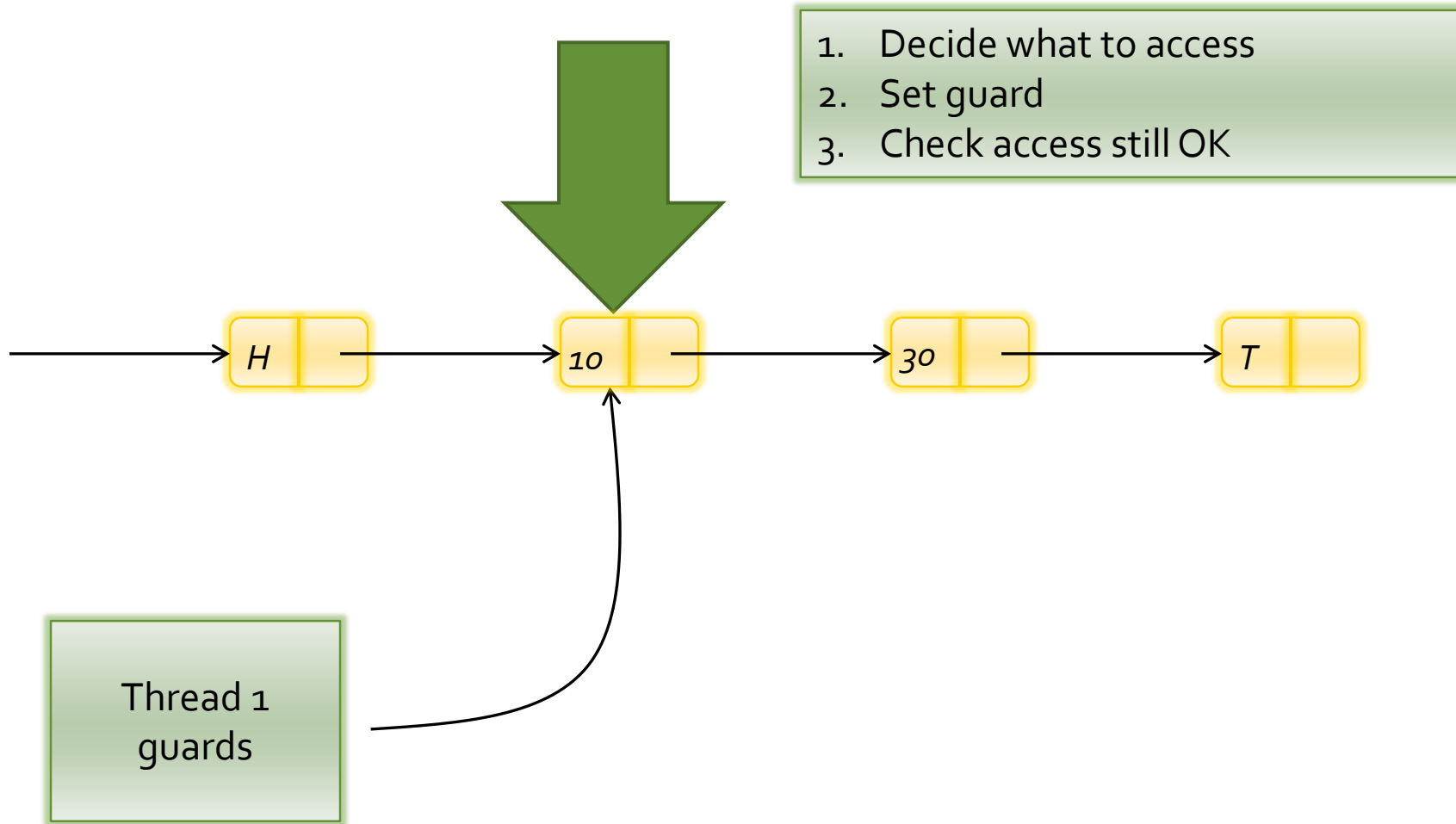
# Re-use via ROP



# Re-use via ROP



# Re-use via ROP

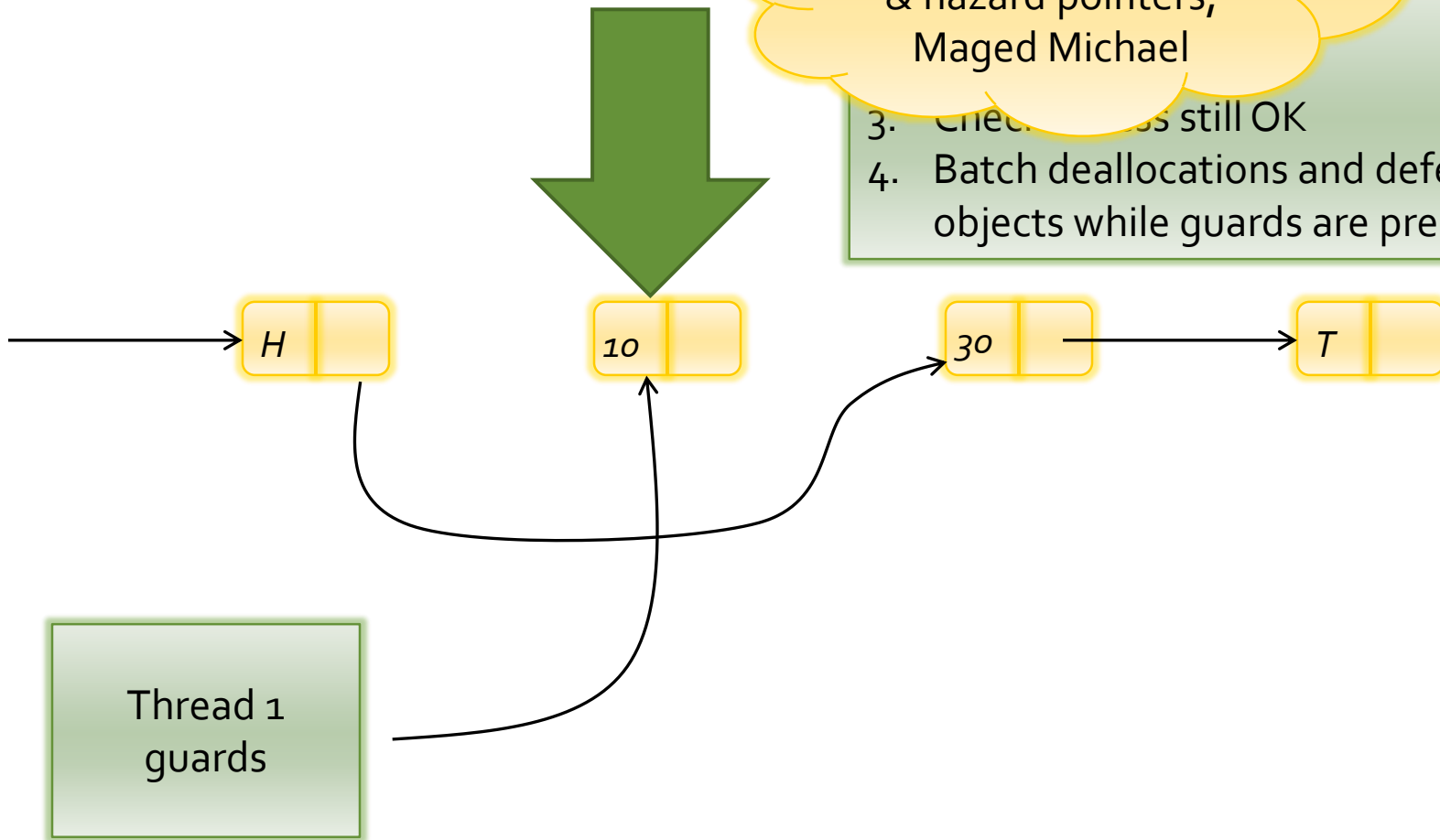




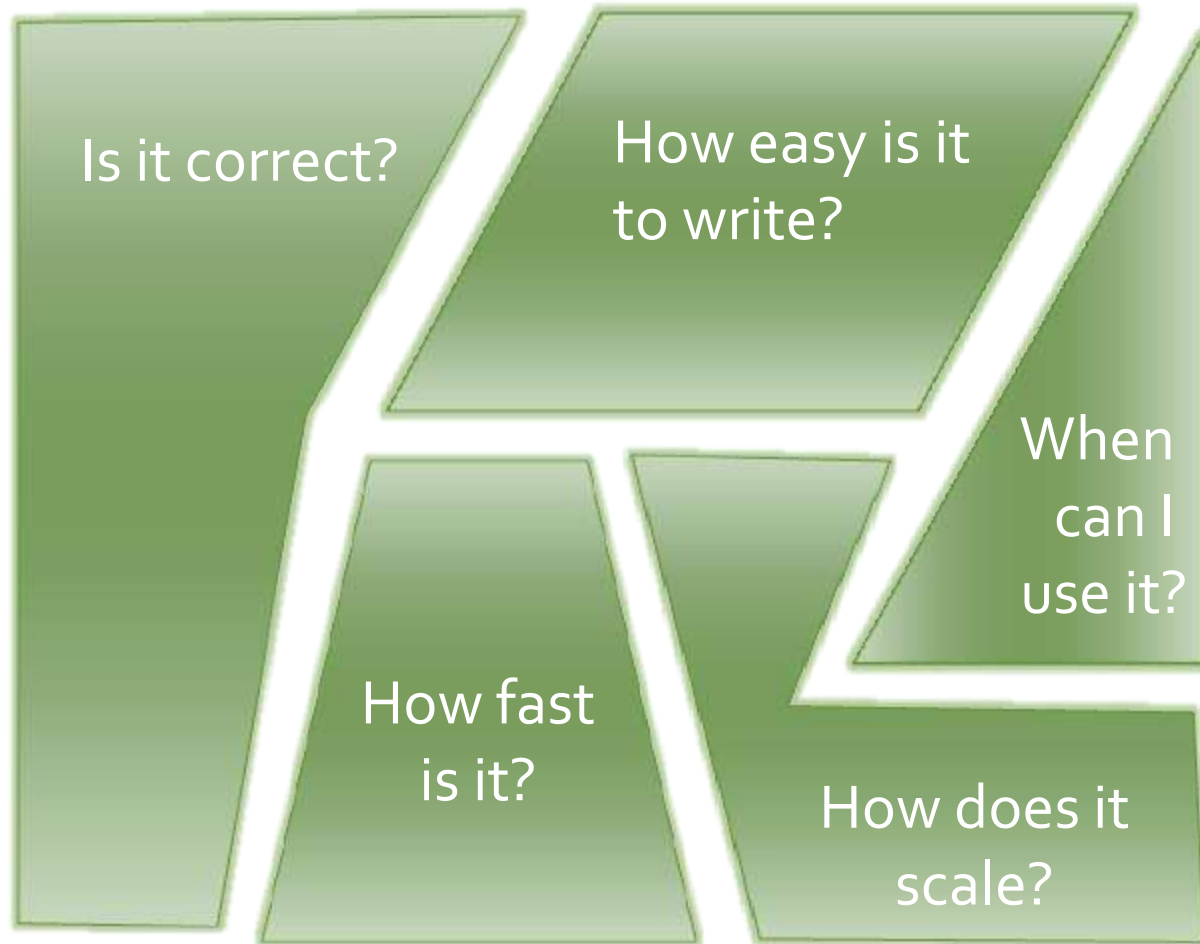
# Re-use via ROP

See also: "Safe memory reclamation" & hazard pointers, Maged Michael

- 3. Check guards still OK
- 4. Batch deallocations and defer on objects while guards are present



# What do we care about?

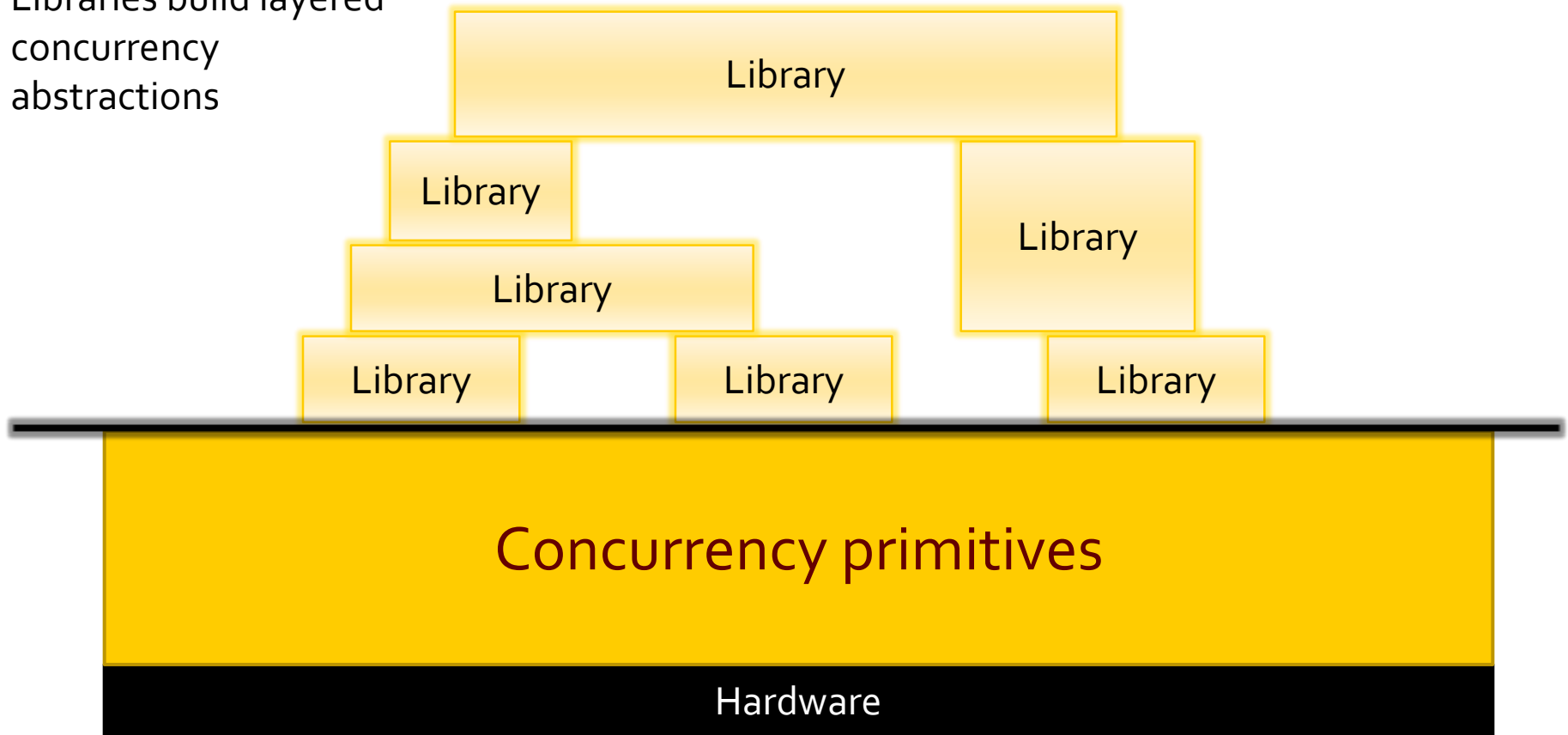


# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory
  - TM and composability
  - STM internals
  - Integration into a language runtime system
  - Sandboxing & strong isolation
  - Current performance and my perspective on TM

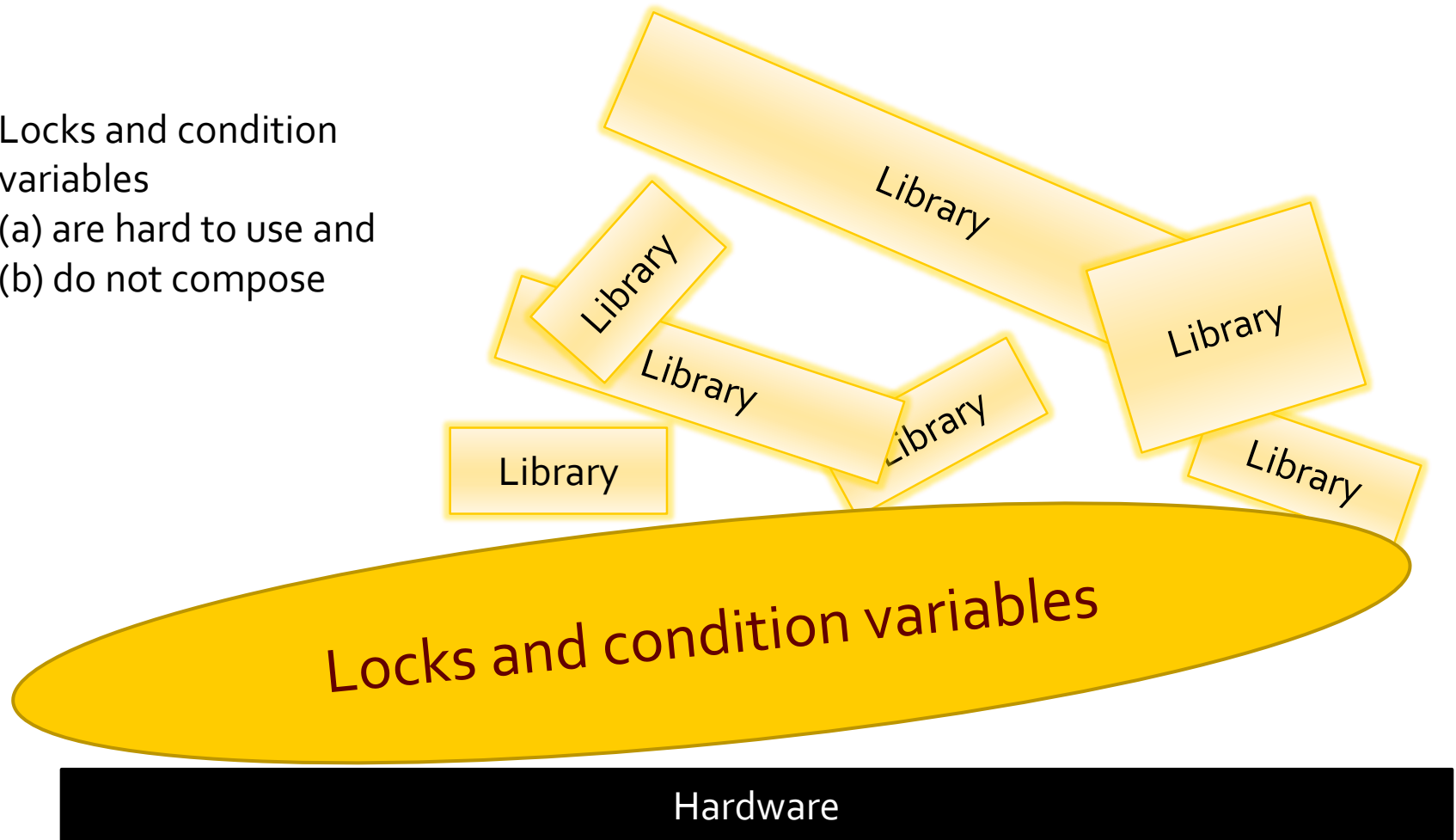
# What we want

Libraries build layered  
concurrency  
abstractions

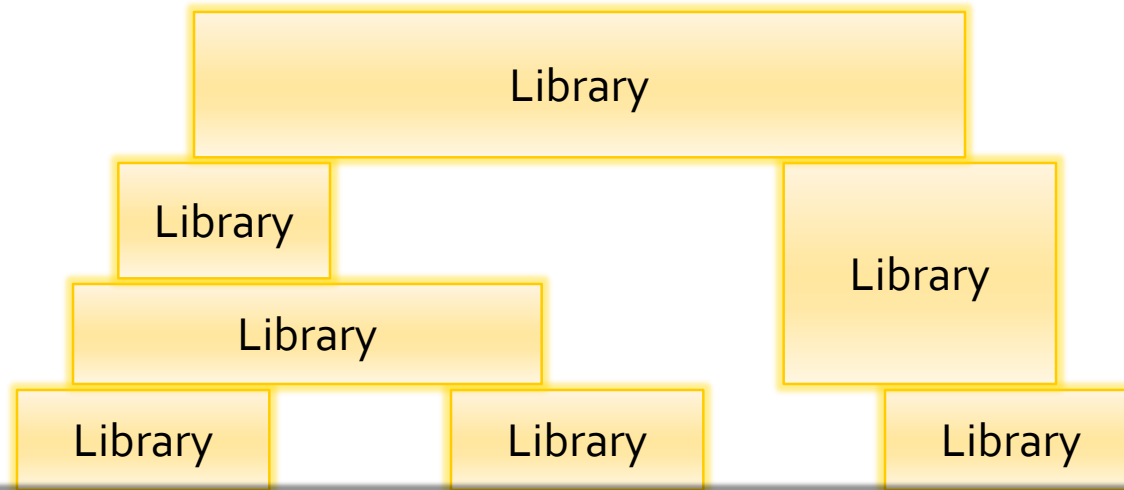


# What we have

Locks and condition variables  
(a) are hard to use and  
(b) do not compose



# Atomic blocks



Atomic blocks built over transactional memory.  
In Haskell: 3 primitives: atomic, retry, orElse

Hardware

# Atomic memory transactions

```
Item PopLeft() {  
    atomic { ... sequential code ... }  
}
```

Like database transactions

- To a first approximation, just write the sequential code, and wrap **atomic** around it
- All-or-nothing semantics: **Atomic** commit
- Atomic block executes in **Isolation**
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the **PopLeft** code)

Acid

# Atomic blocks compose (locks do not)

```
void GetTwo() {  
    atomic {  
        i1 = PopLeft();  
        i2 = PopLeft();  
    }  
    DoSomething( i1, i2 );  
}
```

- Guarantees to get two consecutive items
- PopLeft() is unchanged
- Cannot be achieved with locks (except by breaking the PopLeft abstraction)

Composition  
is THE way we  
build big  
programs  
that work



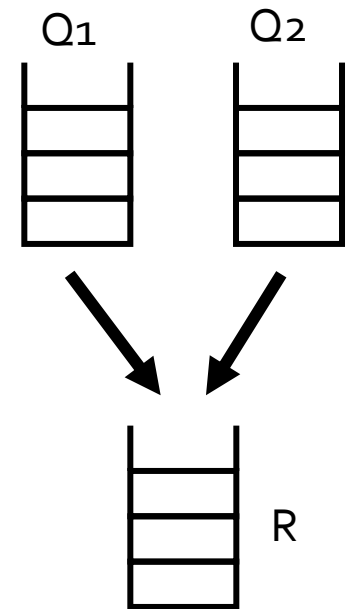
# Blocking: how does PopLeft wait for data?

```
Item PopLeft() {
    atomic {
        if (leftSentinel.right==rightSentinel) {
            retry;
        } else { ...remove item from queue... }
    }
}
```

- **retry** means “abandon execution of the atomic block and re-run it (when there is a chance it’ll complete)”
- No lost wake-ups
- No consequential change to GetTwo(), *even though GetTwo must wait for there to be **two** items in the queue*

# Choice: waiting for either of two queues

```
void GetEither() {  
    atomic {  
  
        do { i = Q1.Get(); }  
        or else { i = Q2.Get(); }  
  
        R.Put( i );  
    }  
}
```



- **do** {...this...} **or else** {...that...} tries to run “this”
- If “this” retries, it runs “that” instead
- If both retry, the do-block retries. GetEither() will thereby wait for there to be an item in *either* queue

# Programming with atomic blocks

With locks, you think about:

- Which lock protects which data? What data can be mutated when by other threads? Which condition variables must be notified when?
- None of this is explicit in the source code

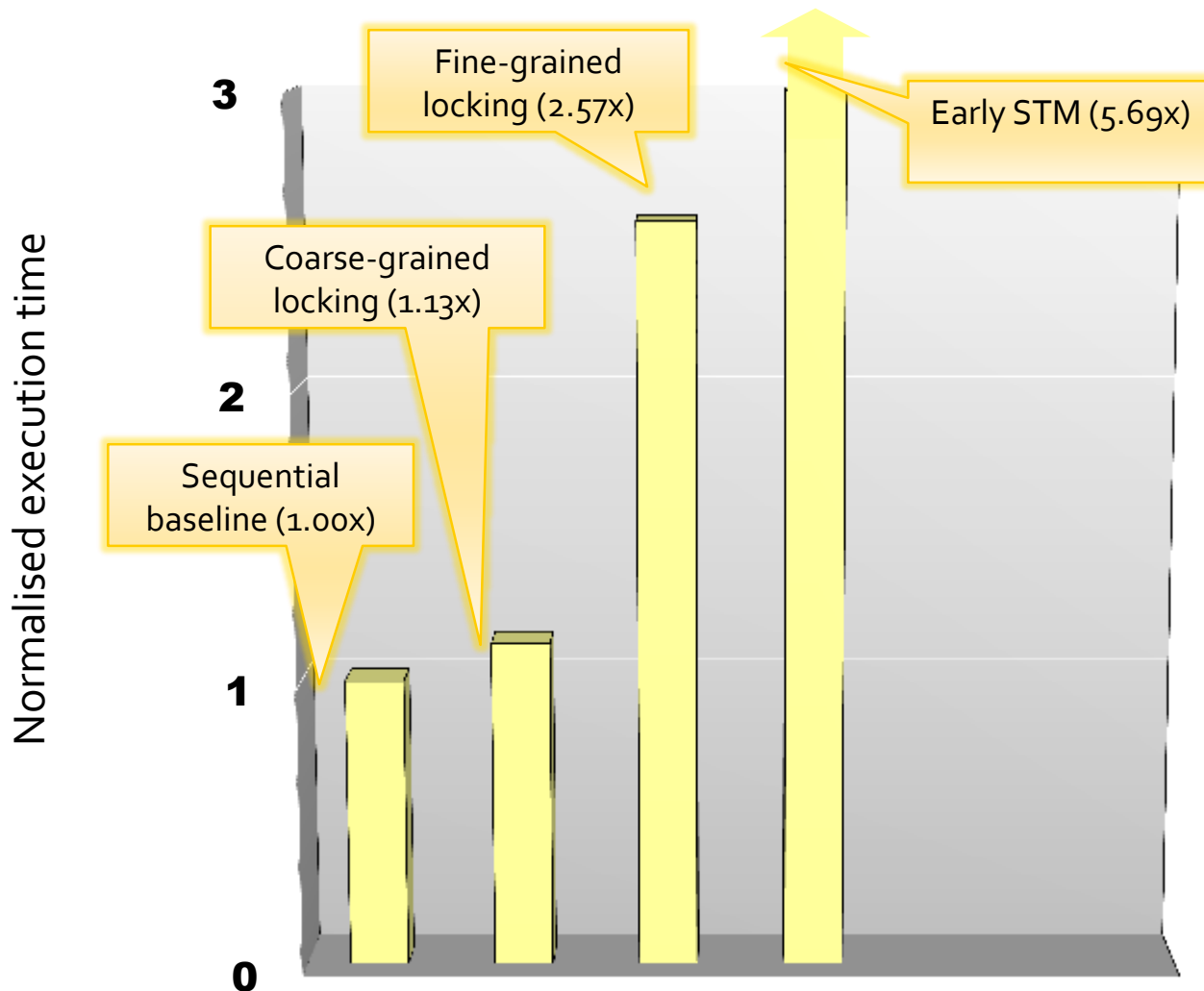
With atomic blocks you think about

- What are the **invariants** (e.g. the tree is balanced)?
- Each atomic block maintains the invariants
- **Purely sequential reasoning** within a block, which is dramatically easier
- Much easier setting for static analysis tools

# Summary so far

- Atomic blocks (atomic, retry, orElse) are a real step forward
- It's like using a high-level language instead of assembly code: whole classes of low-level errors are eliminated.
- Not a silver bullet:
  - you can still write buggy programs;
  - concurrent programs are still harder to write than sequential ones;
  - just aimed at shared memory.
- But the improvement is very substantial

# STM 5 years ago

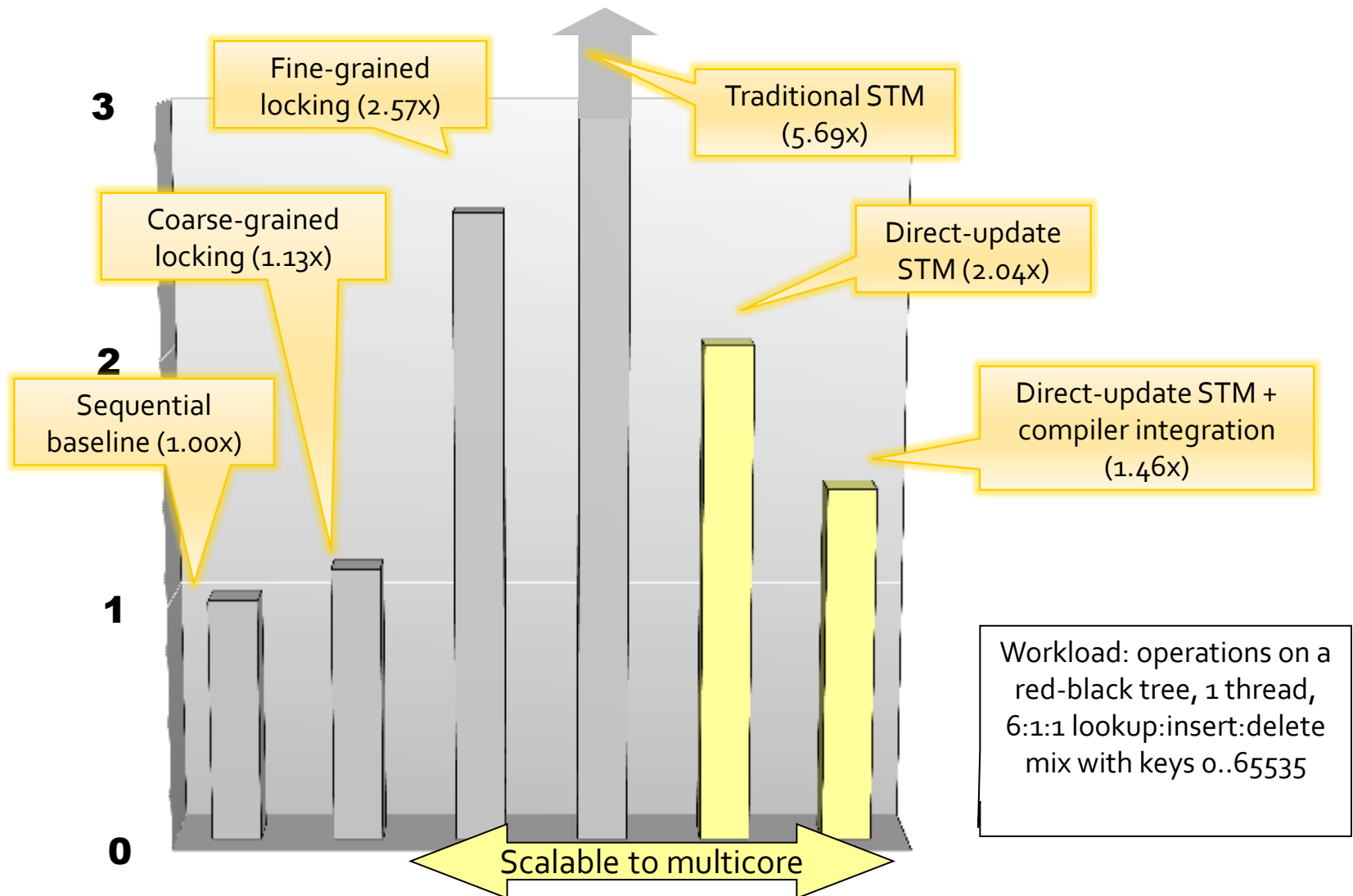


Workload: operations on a red-black tree, 1 thread, 6:1:1 lookup:insert:delete mix with keys 0..65535

# Implementation techniques

- Direct-update STM
  - Allow transactions to make updates in place in the heap
  - Avoids reads needing to search the log to see earlier writes that the transaction has made
  - Makes successful commit operations faster at the cost of extra work on contention or when a transaction aborts
- Compiler integration
  - Decompose the transactional memory operations into primitives
  - Expose the primitives to compiler optimization (e.g. to hoist concurrency control operations out of a loop)
- Runtime system integration
  - Integration with the garbage collector or runtime system components to scale to atomic blocks containing 100M memory accesses
  - Memory management system used to detect conflicts between transactional and non-transactional accesses

# Results: concurrency control overhead



# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory
  - Atomic transactions and composability
  - STM internals
  - Integration into a language runtime system
  - Sandboxing & strong isolation
  - Current performance and my perspective on TM



# Atomic blocks

```
Class O {
  Class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void pushLeft(int item) {
      do {
        tx = TxStart();
        QElem e = new QElem(item);
        TxWrite(tx, &e.right, TxRead(tx, &this.leftSentinel.right));
        TxWrite(tx, &e.left, this.leftSentinel);
        TxWrite(tx, &TxRead(tx, &this.leftSentinel.right).left, e);
        TxWrite(tx, &this.leftSentinel.right, e);
      } while (!TxCommit());
    }
  }
  ...
}
```

# Bartok-STM

- Use per-object meta-data (“TMWs”)
- Each TMW is either:
  - Locked, holding a pointer to the transaction that has the object open for update
  - Available, holding a version number indicating how many times the object has been locked
- Writers eagerly lock TMWs to gain access to the object, using eager version management
  - Maintain an undo log in case of roll-back
- Readers log the version numbers they see and perform lazy conflict detection at commit time

# Example: uncontended swap

a:

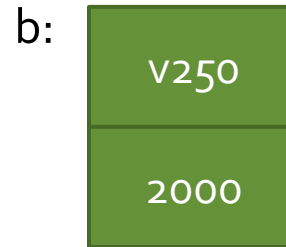


b:

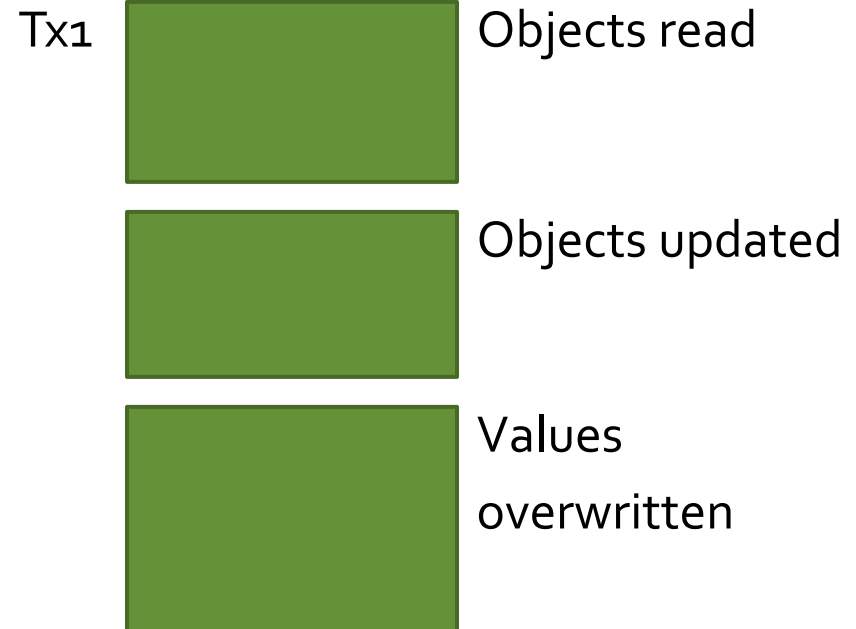


```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```

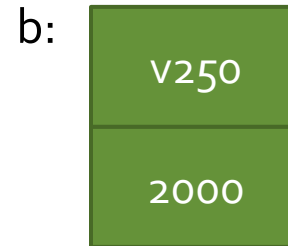
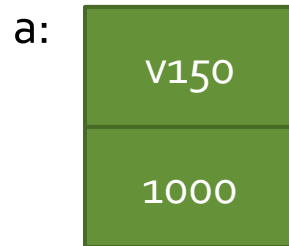
# Example: uncontended swap



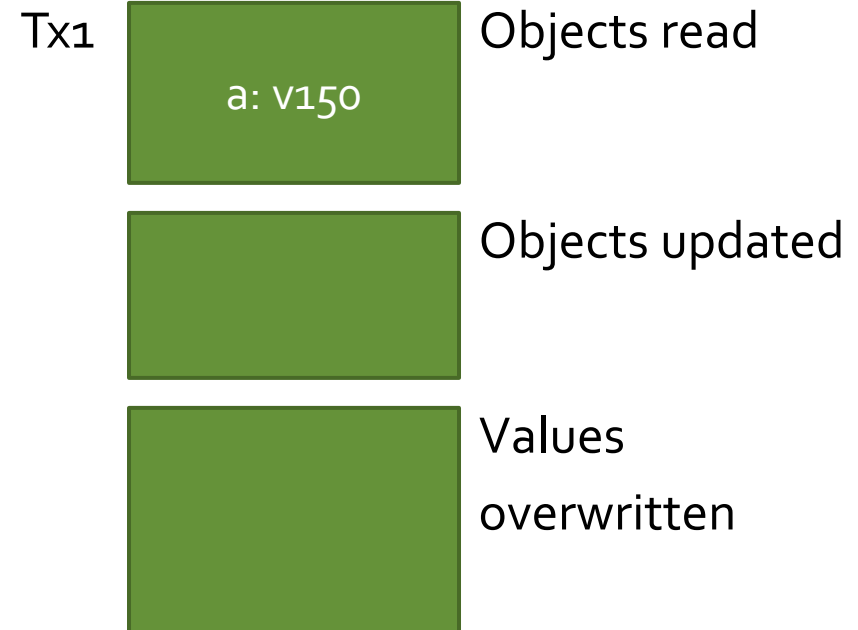
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



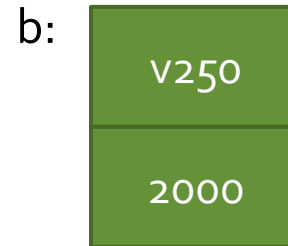
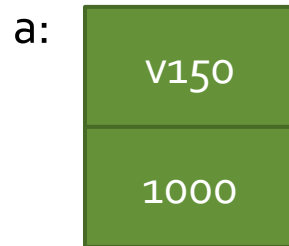
# Example: uncontended swap



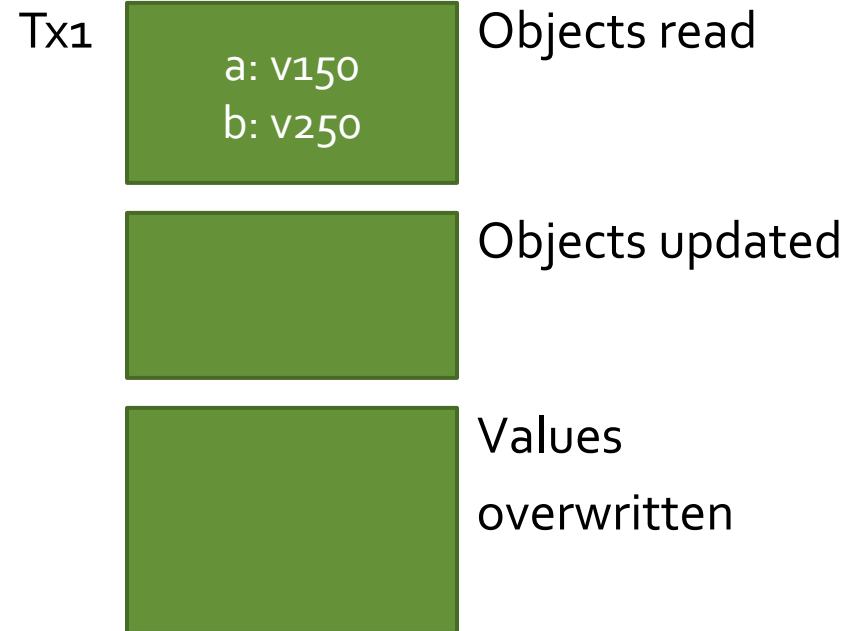
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



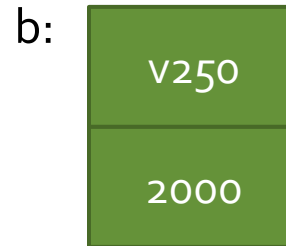
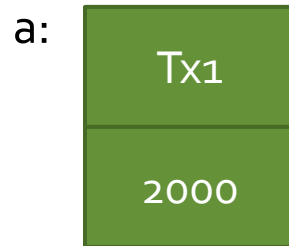
# Example: uncontended swap



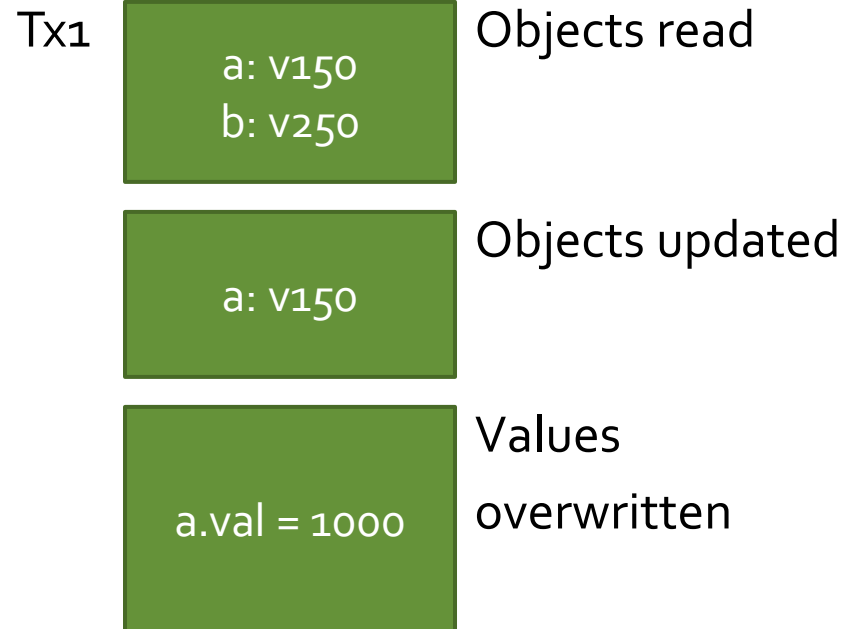
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



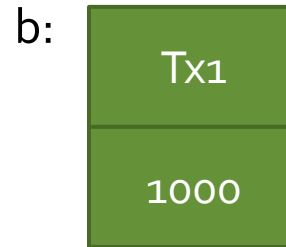
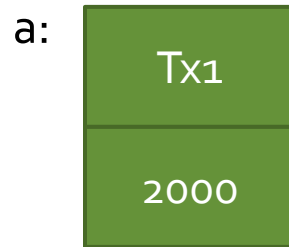
# Example: uncontended swap



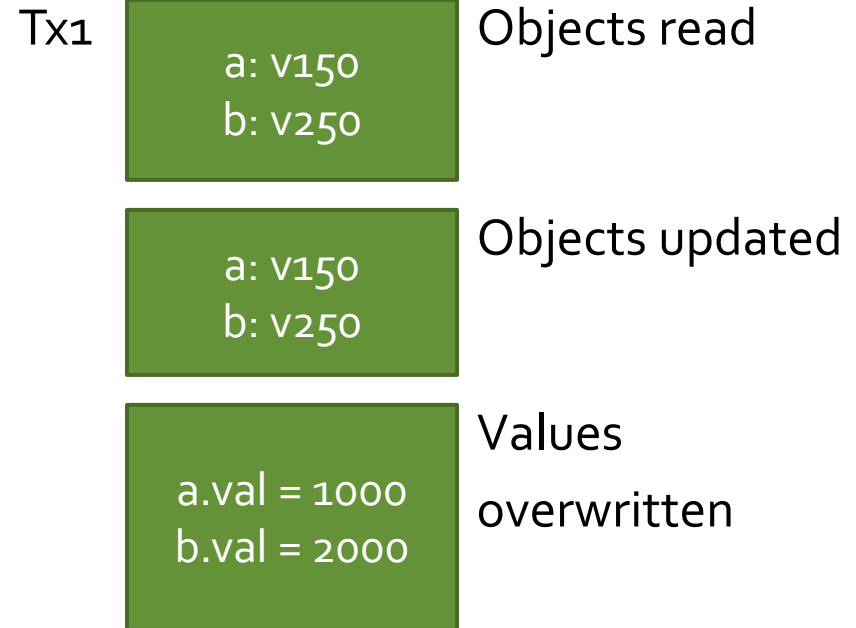
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



# Example: uncontended swap



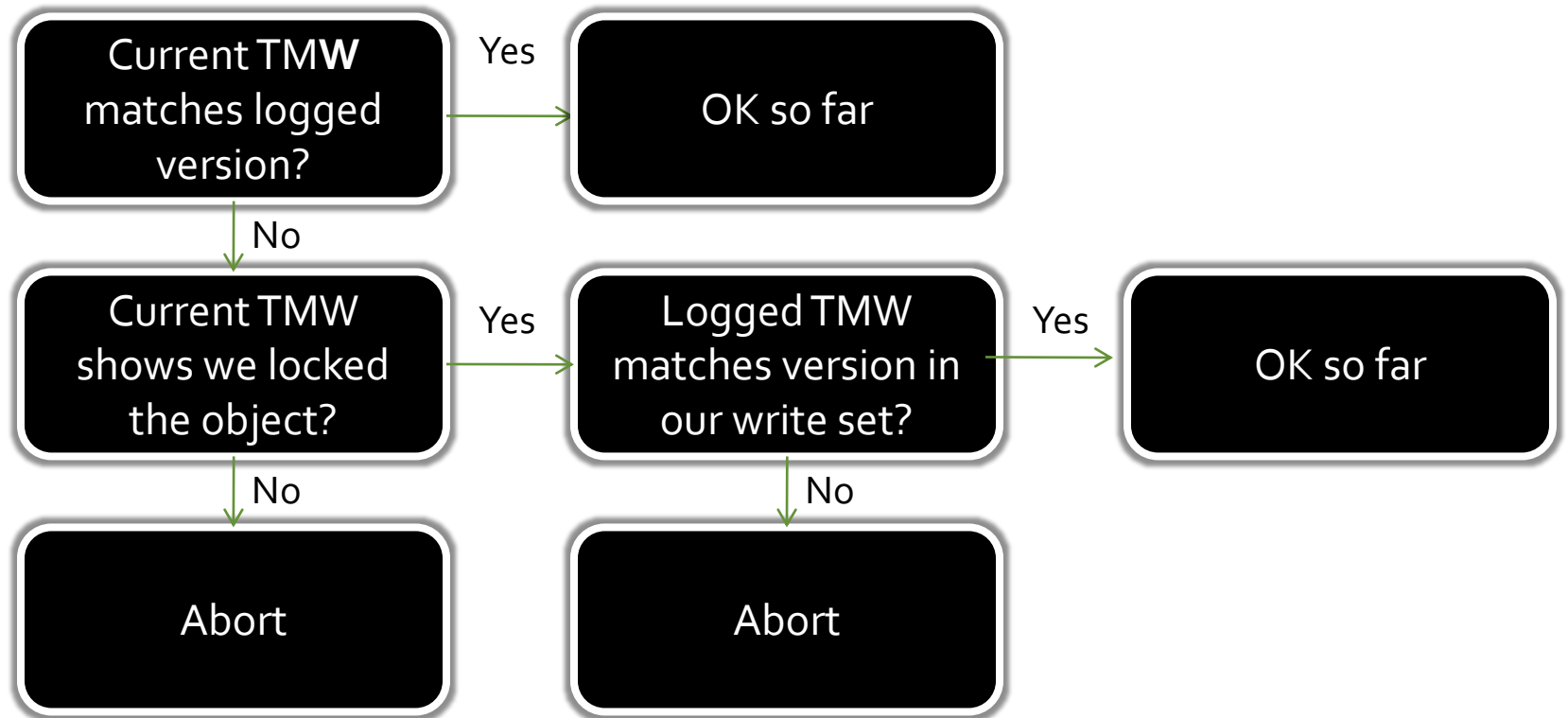
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



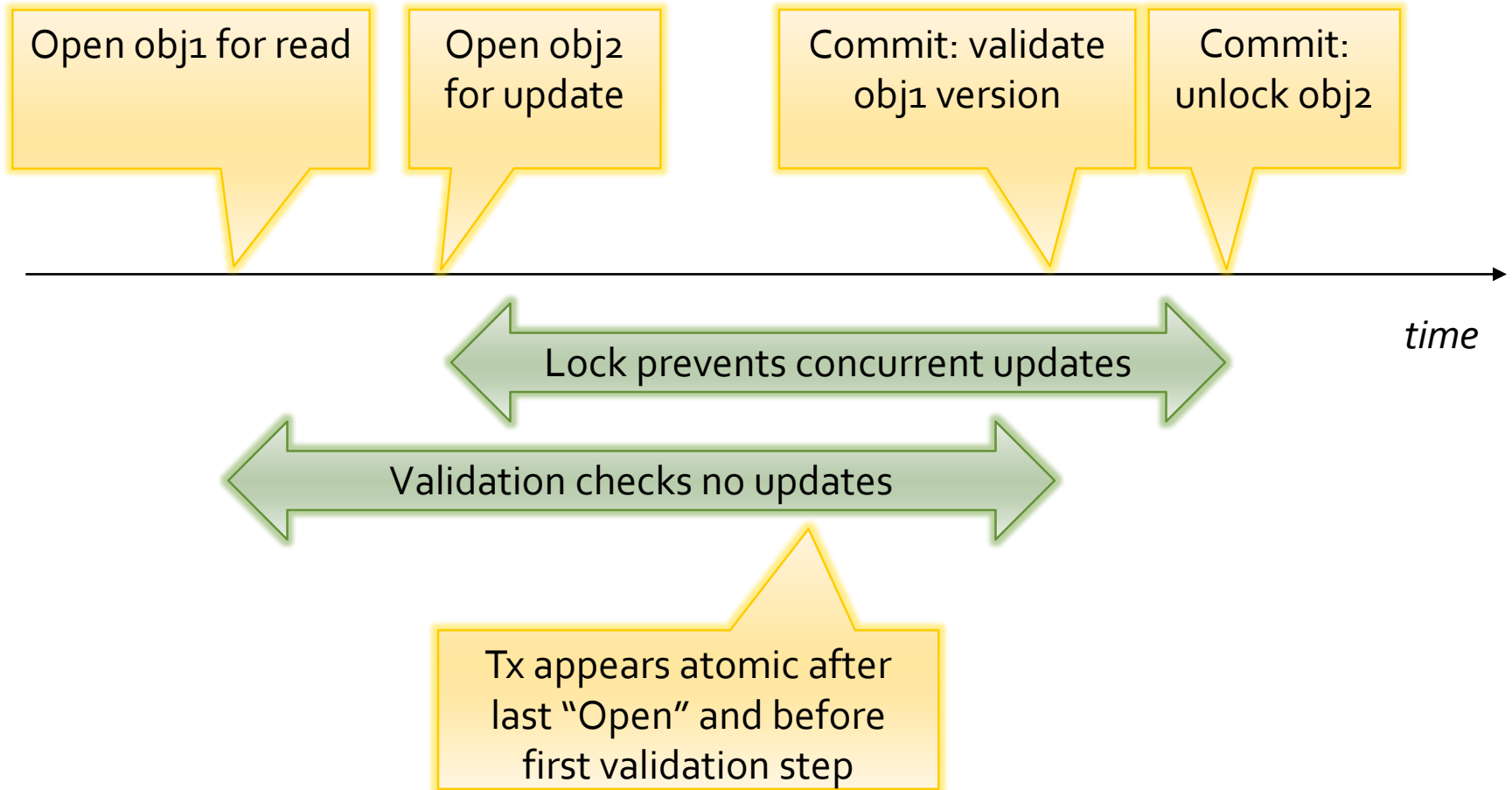


# Commit in Bartok-STM

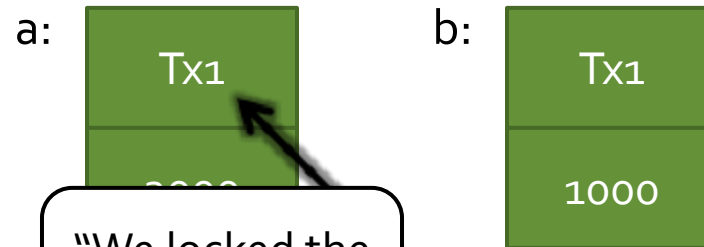
Iterate over  
the read set:



# Correctness sketch

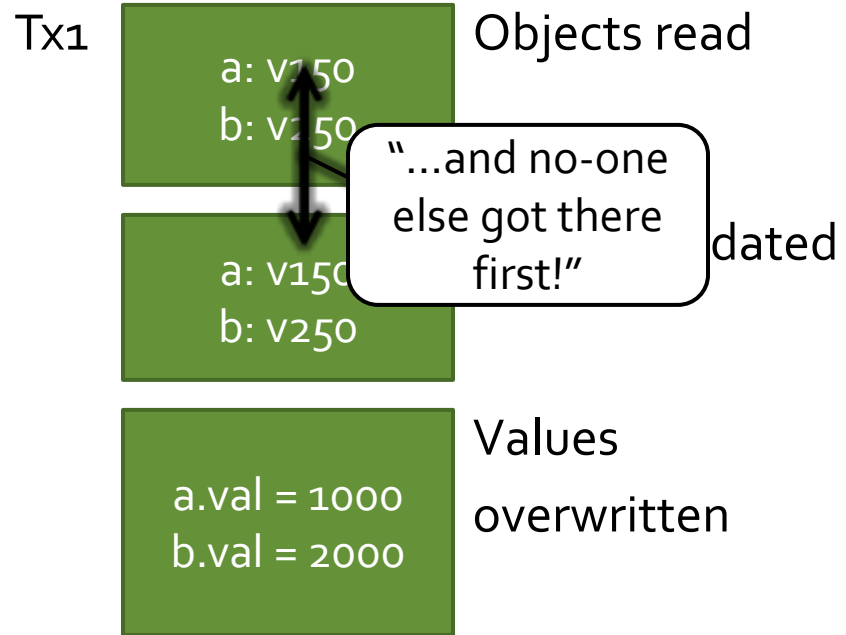


# Example: uncontended swap

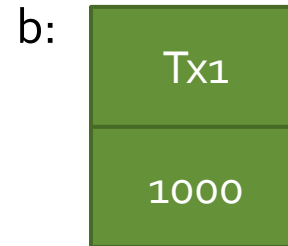
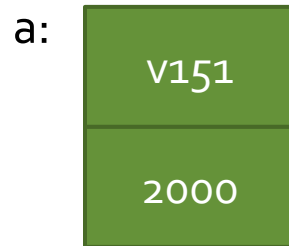


"We locked the object..."

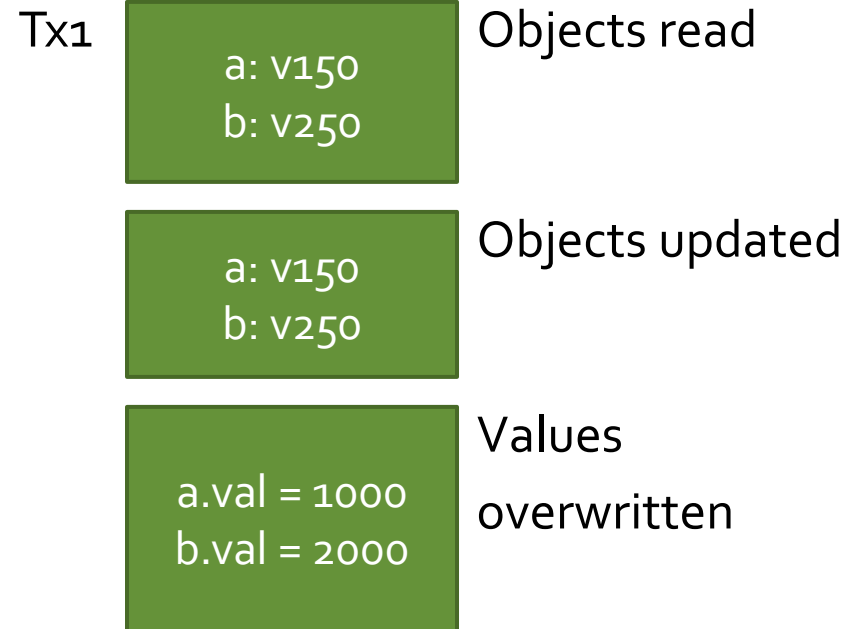
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



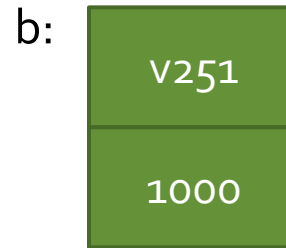
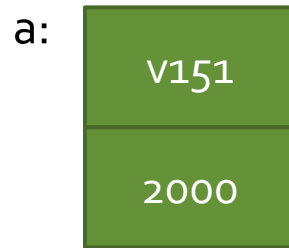
# Example: uncontended swap



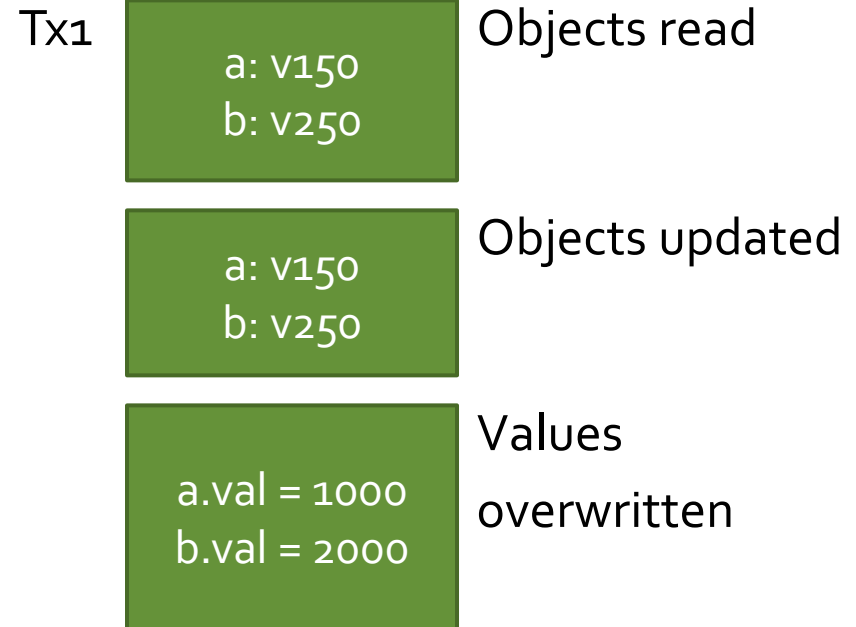
```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



# Example: uncontended swap



```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



# Example: uncontended swap

a:



b:



```
void Swap(int *a, int *b) {  
  do {  
    tx = TxStart();  
    va = TxRead(tx, &a);  
    vb = TxRead(tx, &b);  
    TxWrite(tx, &a, vb);  
    TxWrite(tx, &b, va);  
  } while (!TxCommit());  
}
```



# Tx-tx interaction in Bartok-STM

- Read-read: no problem, both readers see the same version number and verify it at commit time
- Read-write: reader sees that the writer has the object locked. Reader always defers to writer
- Write-write: competition for lock serializes writers (drop locks, then spin to avoid deadlock)

# Taxonomy: consistency during tx

- Gold standard:
  - During execution a transaction runs against a consistent view of memory
  - Won't be "tricked" into looping, etc.
  - "Opacity"
- What are the advantages / disadvantages when compared with an implementation giving weaker guarantees?



# Taxonomy: lazy/eager versioning

- We need some way to manage the tentative updates that a transaction is making
  - Where are they stored?
  - How does the implementation find them (so a transaction's read sees an earlier write)?
- Lazy versioning: only make “real” updates when a transaction commits
- Eager versioning: make updates as a transaction runs, roll them back on abort
- What are the advantages, disadvantages?

# Taxonomy: lazy/eager conflict detection

- We need to detect when two transactions conflict with one another
- Lazy conflict detection: detect conflicts at commit time
- Eager conflict detection: detect conflicts as transactions run
- Again, what are the advantages, disadvantages?

# Taxonomy: word/object based

- What granularity are conflicts detected at?
- Object-based:
  - Access to programmer-defined structures (e.g. objects)
- Word-based:
  - Access to words (or sets of words, e.g. cache lines)
  - Possibly after mapping under a hash function
- What are the advantages and disadvantages of these approaches?

# Bartok-STM

- Designed to work well on low-contention workloads
  - Eager version management to reduce commit costs
  - Eager locking to support eager version management
- Primitives do not guarantee that transactions see a consistent view of the heap while running
  - Can be sandboxed in managed code...
  - ...harder in native code

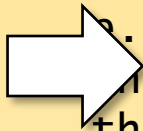
# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory
  - TM and composability
  - STM internals
  - Integration into a language runtime system
  - Sandboxing & strong isolation
  - Current performance and my perspective on TM

# Atomic blocks

```
class Q {
    QElem l;
    QElem r;

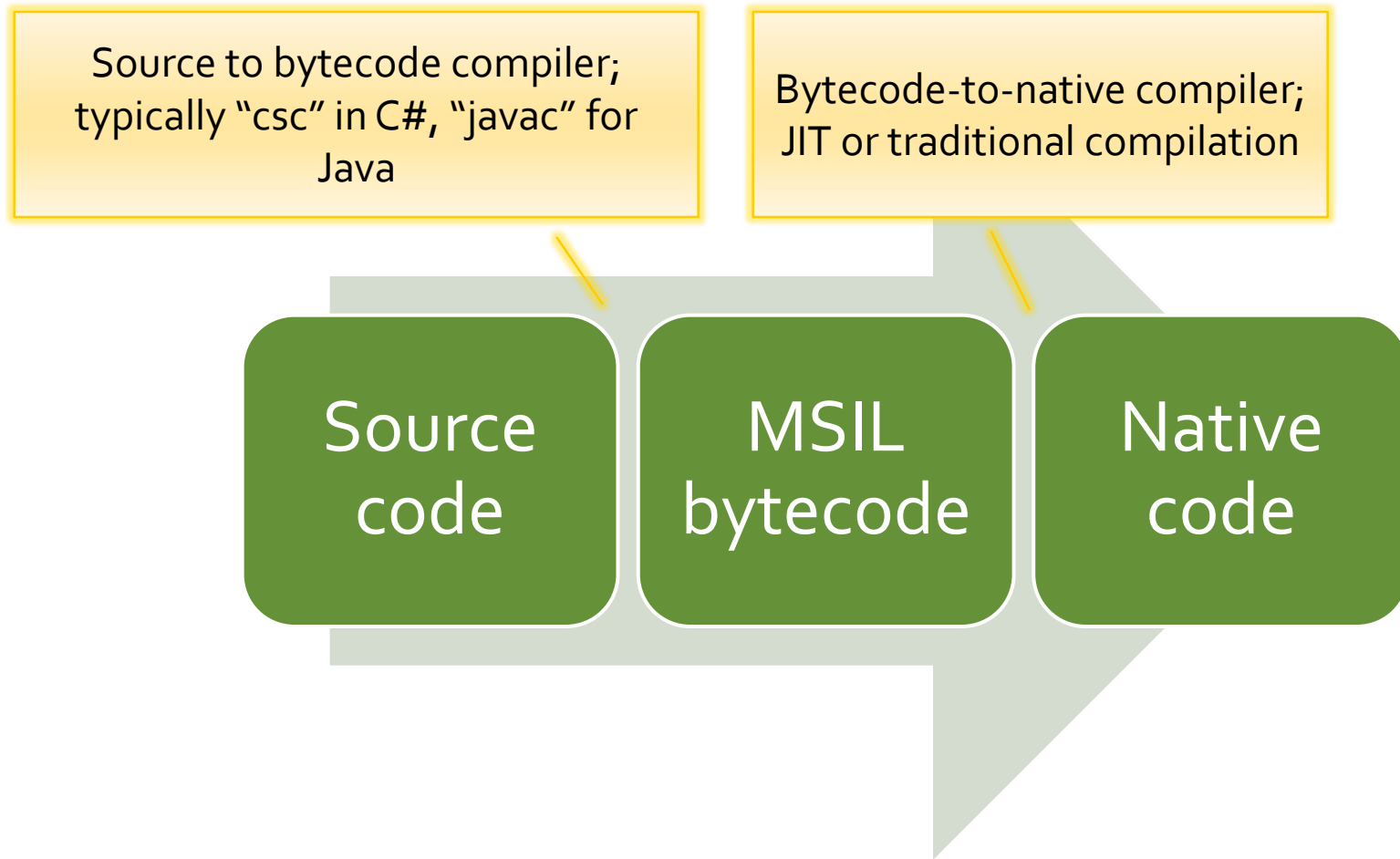
    void pushLeft(int item) {
        atomic {
            QElem e = new QElem(item);
            TxWrite(&e.right, TxRead(&this.leftSentinel.right));
            TxWrite(&e.left, this.leftSentinel);
            TxWrite(&TxRead(&this.leftSentinel.right).left, e);
            TxWrite(&this.leftSentinel.right, e);
        } while (!TxCommit());
    }
    ...
}
```



```
class Q {
    QElem leftSentinel;
    QElem rightSentinel;

    void pushLeft(int item) {
        do {
            TxStart();
            QElem e = new QElem(item);
            TxWrite(&e.right, TxRead(&this.leftSentinel.right));
            TxWrite(&e.left, this.leftSentinel);
            TxWrite(&TxRead(&this.leftSentinel.right).left, e);
            TxWrite(&this.leftSentinel.right, e);
        } while (!TxCommit());
    }
    ...
}
```

# Compilation



# Why divide things this way?

- Little information loss from source code to bytecode
- Source-to-bytecode works a file at a time, bytecode-to-native can see the whole program (or, at least, see all of the parts needed so far in execution)
- Lower level transformations possible at bytecode-to-native
- Integration between the STM and other parts of the runtime system

```
void Swap(Pair p) {  
    try {  
        va = p.a;  
        vb = p.b;  
        p.a = vb;  
        p.b = va;  
    } catch (AtomicException) {  
    }  
}
```



# Boilerplate around transactions

```
void Swap(Pair p) {  
  do {  
    done = true;  
    try {  
      try {  
        tx = StartTx();  
        va = p.a;  
        vb = p.b;  
        p.a = vb;  
        p.b = va;  
      } finally {  
        CommitTx();  
      }  
    } catch (TxInvalid) {  
      done = false;  
    }  
  } while (!done);  
}
```

Keep running the atomic block in a fresh tx each time

Commit (on normal or exn exit)

Commit fails by raising a TxInvalid exception; re-execute

(I'm using source code examples for clarity; in reality this would be in the compiler's internal intermediate code)

# Naïve expansion of data accesses

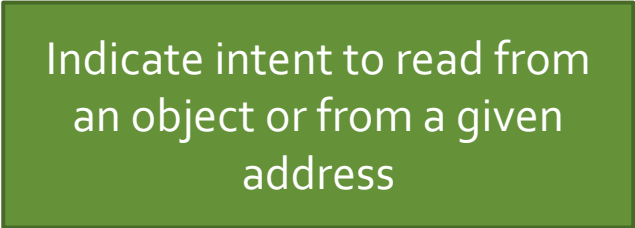
```
void Swap(Pair p) {
    do {
        done = true;
        try {
            try {
                tx = StartTx();
                TxWrite(tx, &va, TxRead(tx, &p.a));
                TxWrite(tx, &vb, TxRead(tx, &p.b));
                TxWrite(tx, &p.a, TxRead(tx, &vb));
                TxWrite(tx, &p.b, TxRead(tx, &va));
            } finally {
                CommitTx();
            }
        } catch (TxInvalid) {
            done = false;
        }
    } while (!done);
}
```

# What are the problems here?

- Using the STM for thread-private local variables
- Repeatedly mapping from addresses to concurrency control info
- Duplicating concurrency control work if it's implemented at a per-object granularity

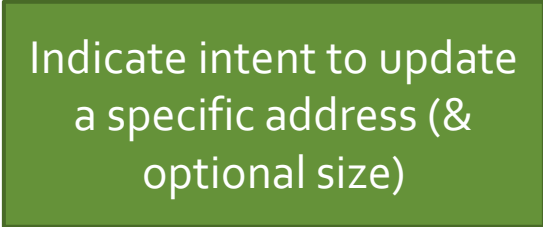
# Decomposed STM primitive API

- `OpenForRead(tx, obj)`
- `OpenForRead(tx, addr)`
- `OpenForUpdate(tx, obj)`
- `OpenForUpdate(tx, addr)`



Indicate intent to read from an object or from a given address

- `LogForUndo(tx, addr)`



Indicate intent to update a specific address (& optional size)

# Using the decomposed API

```
x = p.a;
```



```
OpenForRead(tx, p);  
x = p.a;
```

```
p.b = y;
```



```
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.b);  
p.b = y;
```

# Implementation using decomposed API

```
...  
OpenForUpdate(tx, p);  
OpenForRead(tx, p);  
va = p.a;  
OpenForRead(tx, p);  
vb = p.b;  
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.a);  
p.a = vb;  
OpenForUpdate(tx, p);  
LogForUndo(tx, &p.b);  
p.b = va;  
...
```

Always need update  
access: get it first

Second OpenForRead  
made unnecessary by first

Second OpenForUpdate  
made unnecessary by first

# Improved expansion of data accesses

```
void Swap(Pair p) {
  do {
    done = true;
    try {
      try {
        tx = StartTx();
        OpenForUpdate(tx, p);
        va = p.a;
        vb = p.b;
        LogForUndo(tx, &p.a);
        p.a = vb;
        LogForUndo(tx, &p.b);
        p.b = va;
      } finally {
        CommitTx();
      }
    } catch (TxInvalid) {
      done = false;
    }
  } while (!done);
}
```

# Are we done?

- Local variables
- By-ref parameters
- Method calls
  
- Keeping optimizations safe
- GC integration
- Finalizers
- Condition synchronization



# Keeping optimizations safe

Original (contrived) source code

```
void clear_tx(Pair p) {  
    for (int i = 0; i < 10; i++) {  
        p.a = 10;  
        p.b = i;  
    }  
}
```

# Keeping optimizations safe

Expanded with decomposed API operations

```
void Clear_tx(Pair p) {  
    for (int i = 0; i < 10; i++) {  
        OpenForUpdate(tx, p);  
        LogForUndo(tx, &p.a);  
        p.a = 10;  
        LogForUndo(tx, &p.b);  
        p.b = i;  
    }  
}
```

# Keeping optimizations safe

## Hoisting loop-invariant code

```
void Clear_tx(Pair p) {  
    p.a = 10;  
    for (int i = 0; i < 10; i ++) {  
        OpenForUpdate(tx, p);  
        LogForUndo(tx, &p.a);  
        LogForUndo(tx, &p.b);  
        p.b = i;  
    }  
}
```

# Keeping optimizations safe

Introduce dependencies

```
void Clear_tx(Pair p) {  
  for (int i = 0; i < 10; i ++) {  
    tmp1 = OpenForUpdate(tx, p);  
    tmp2 = LogForUndo(tx, &p.a) <tmp1>;  
    p.a = 10 <tmp2>;  
    tmp3 = LogForUndo(tx, &p.b) <tmp1>;  
    p.b = i <tmp3>;  
  }  
}
```

# Keeping optimizations safe

Transformations must respect dependencies

```
void Clear_tx(Pair p) {
    tmp1 = OpenForUpdate(tx, p);
    tmp2 = LogForUndo(tx, &p.a) <tmp1>;
    tmp3 = LogForUndo(tx, &p.a) <tmp1>;
    p.a = 10 <tmp2>;
    for (int i = 0; i < 10; i ++) {
        p.b = i <tmp3>;
    }
}
```

# GC integration

## Another contrived program

```
void Temp() {  
    Pair result;  
    atomic {  
        for (int i = 0; i < 100000; i ++) {  
            result = new Pair();  
            result.a = i;  
        }  
    }  
    return result;  
}
```

Lots of temporary objects are allocated as the atomic block runs

# GC integration

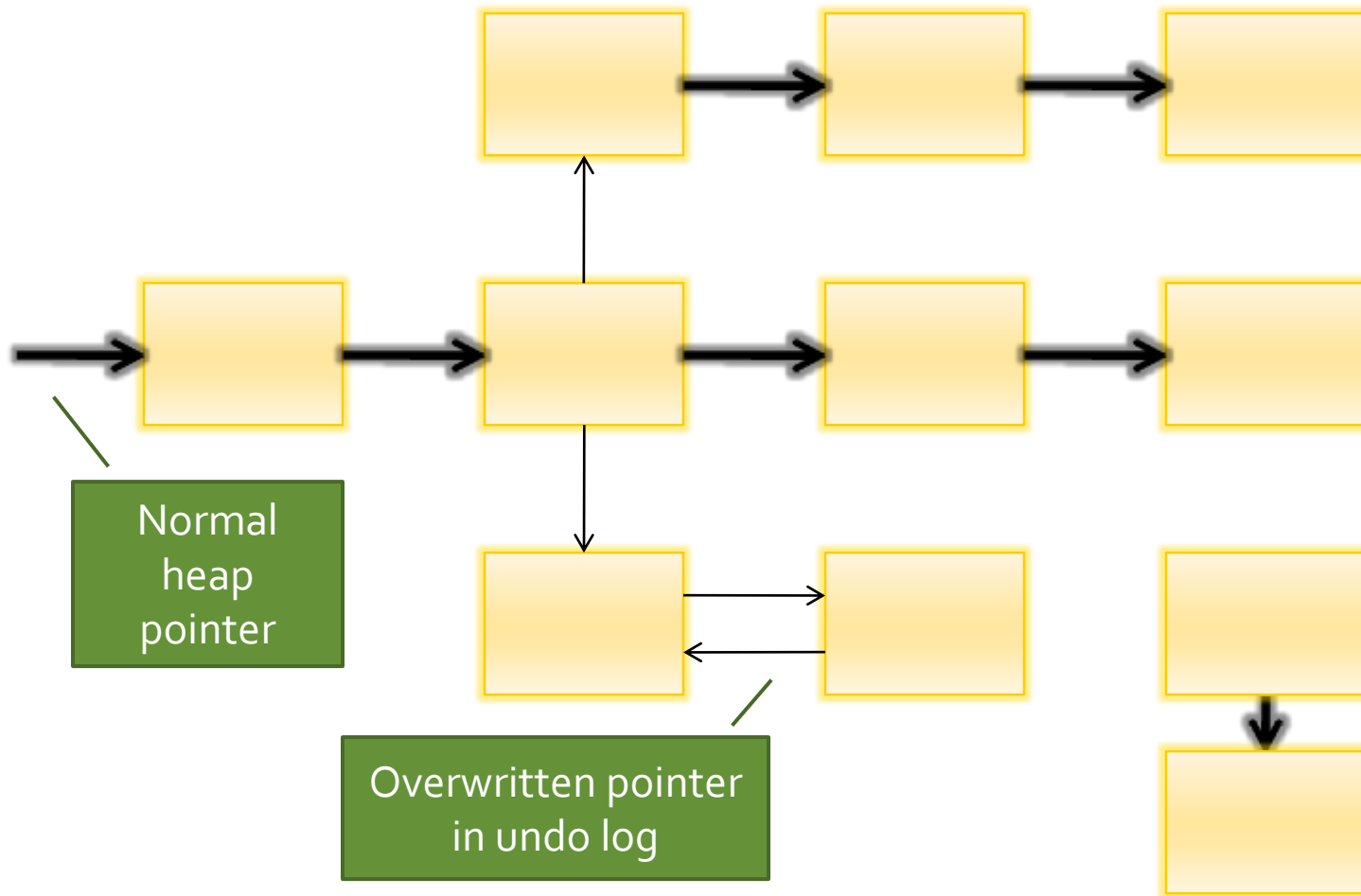
- Abort all running tx on GC?
  - Not ideal: long running tx will not be able to commit
  - Is there a precedent for language features with this kind of perf?
- Treat all the references from the logs as roots?
  - Not ideal: we'd keep all those temporaries

# GC integration

- Principle:
  - Consider the possible heaps based on whether tx commit or abort
  - Retain an object if it is alive in any of these cases (ideally “iff”)
- Do we need to consider  $2^n$  possibilities with  $n$  running tx?
  - No: validate all the tx first so we know they are not conflicting
  - Consider the world if they all commit, consider the world if they all roll back



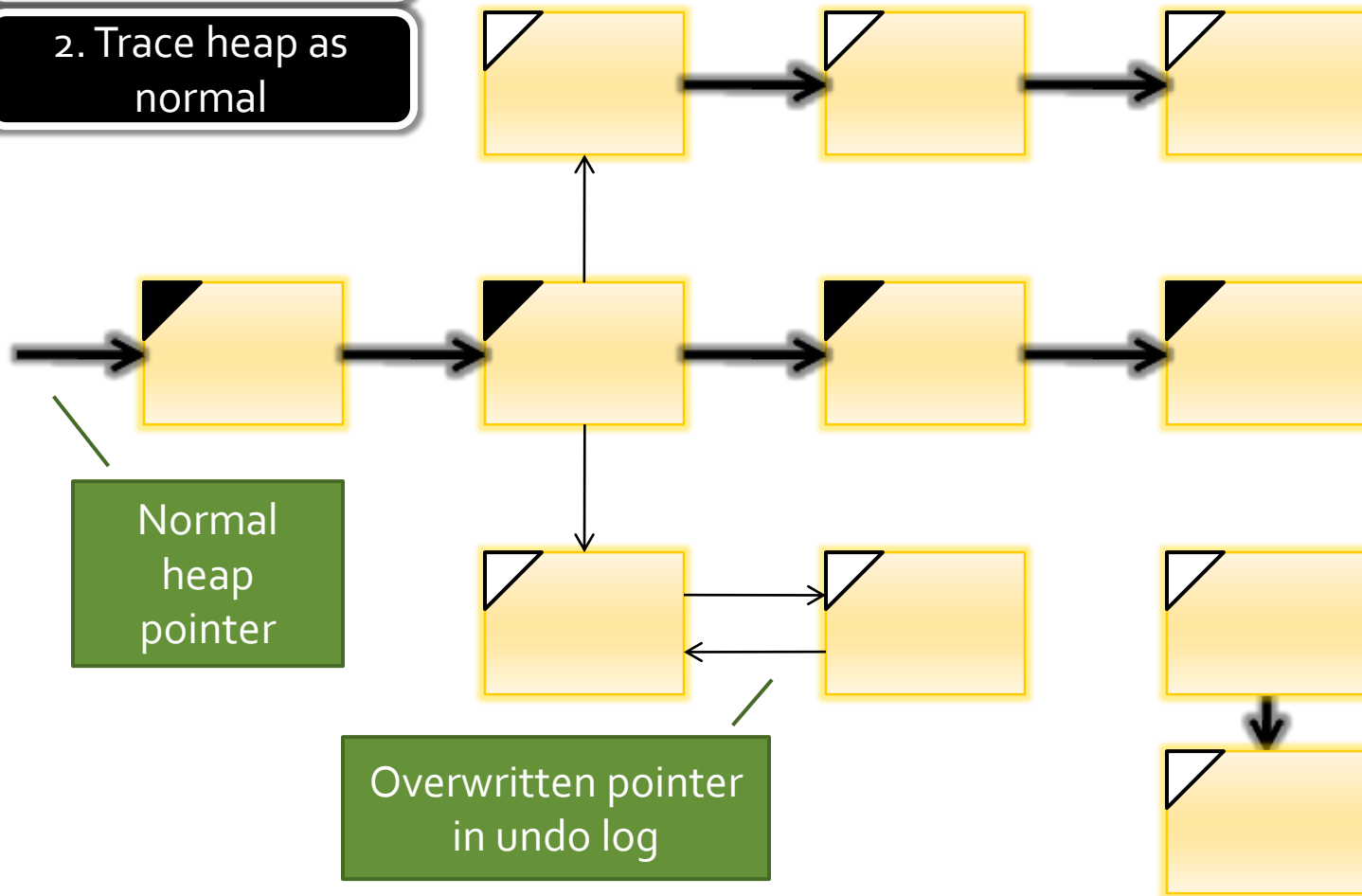
# Example heap



# Conservative algorithm

1. Validate tx

2. Trace heap as normal

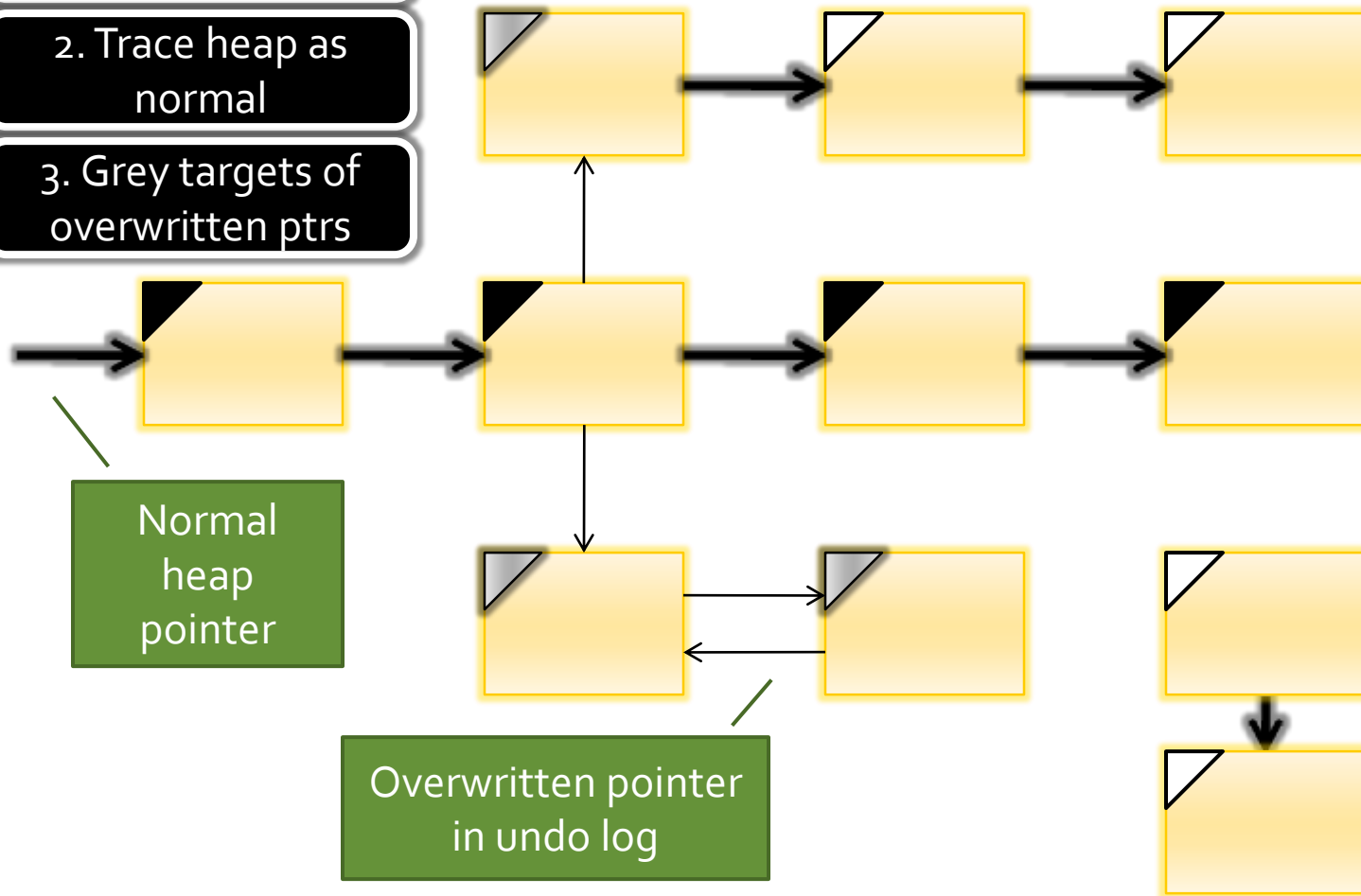


# Conservative algorithm

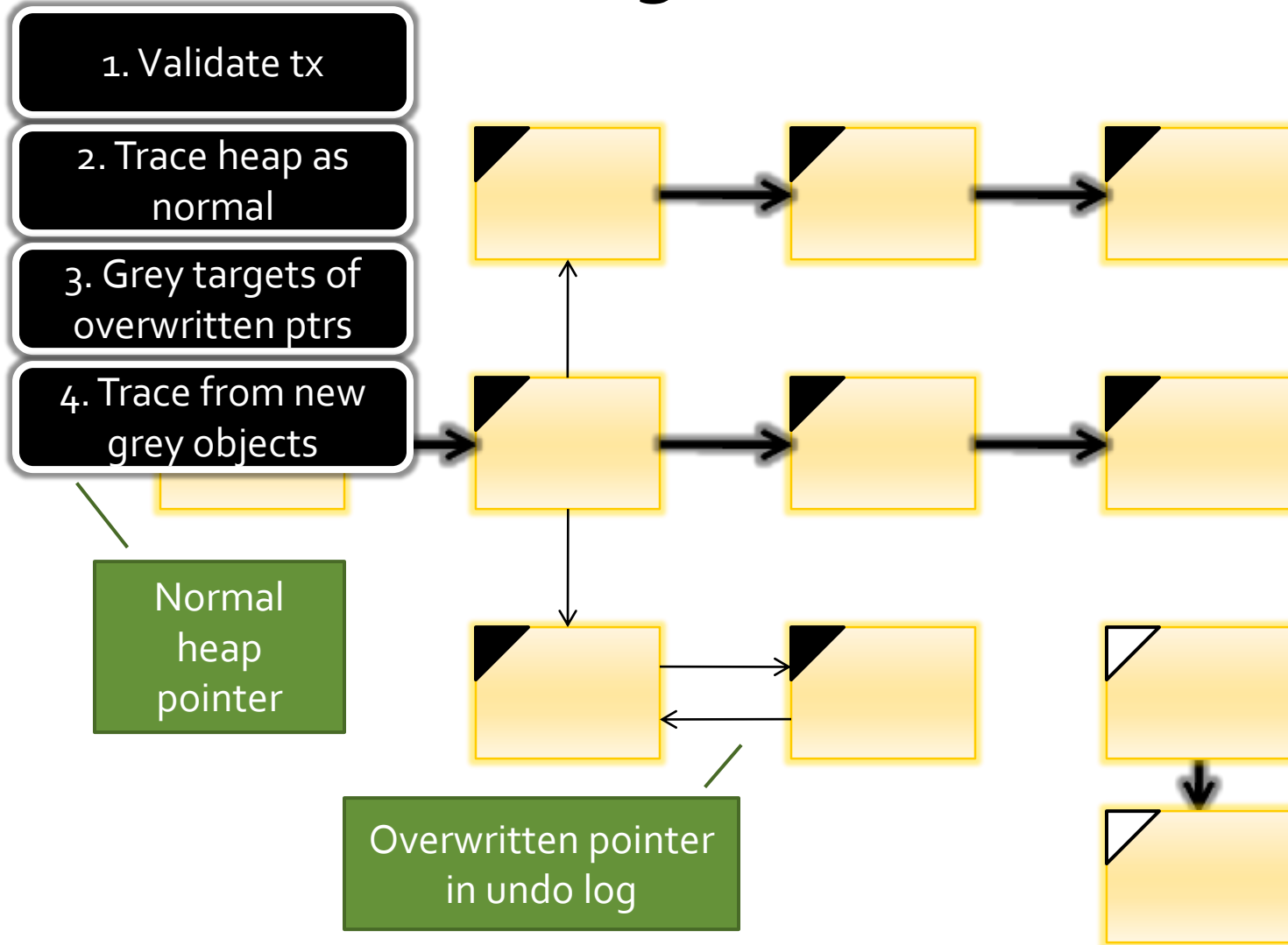
1. Validate tx

2. Trace heap as normal

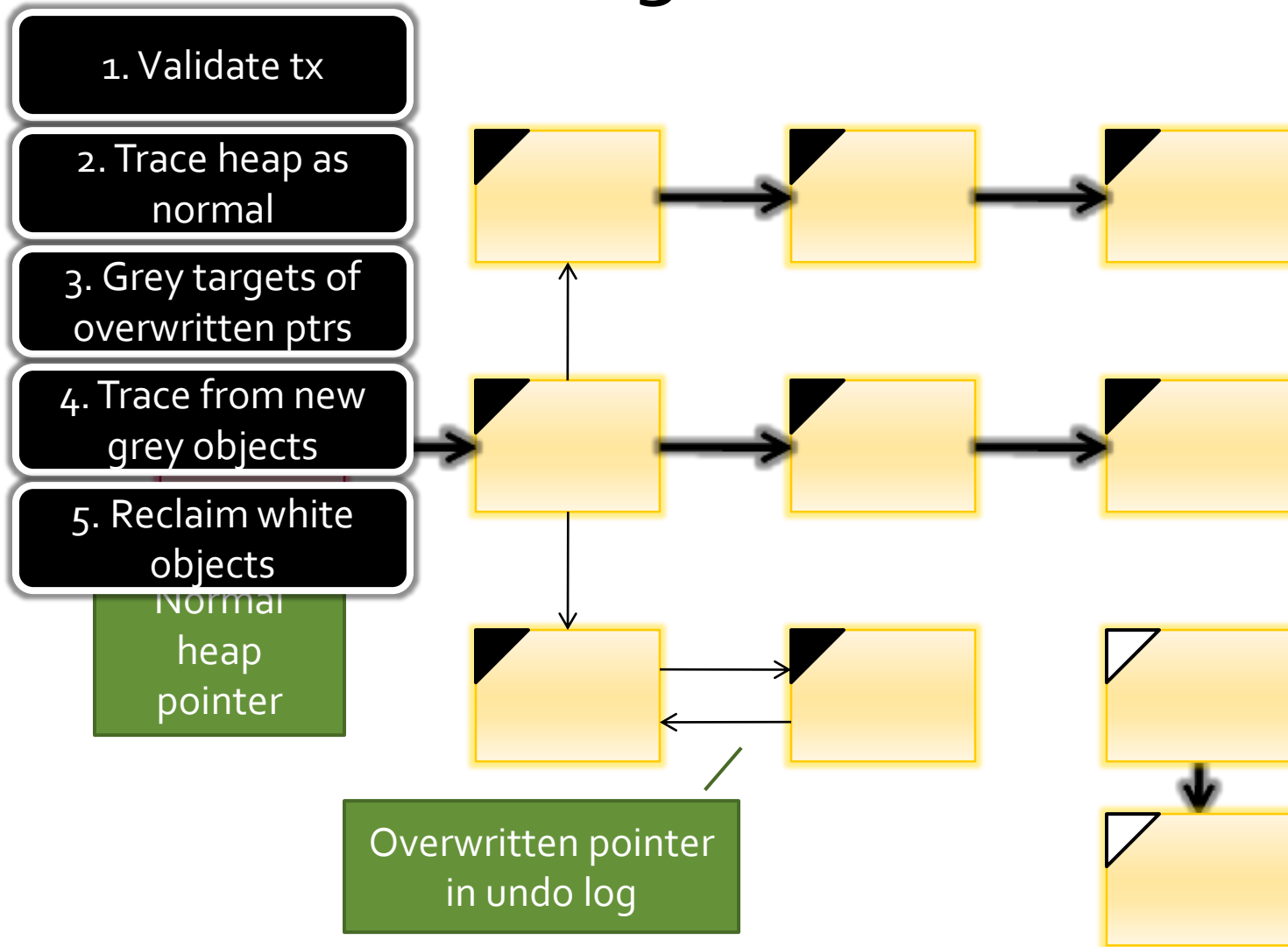
3. Grey targets of overwritten ptrs



# Conservative algorithm



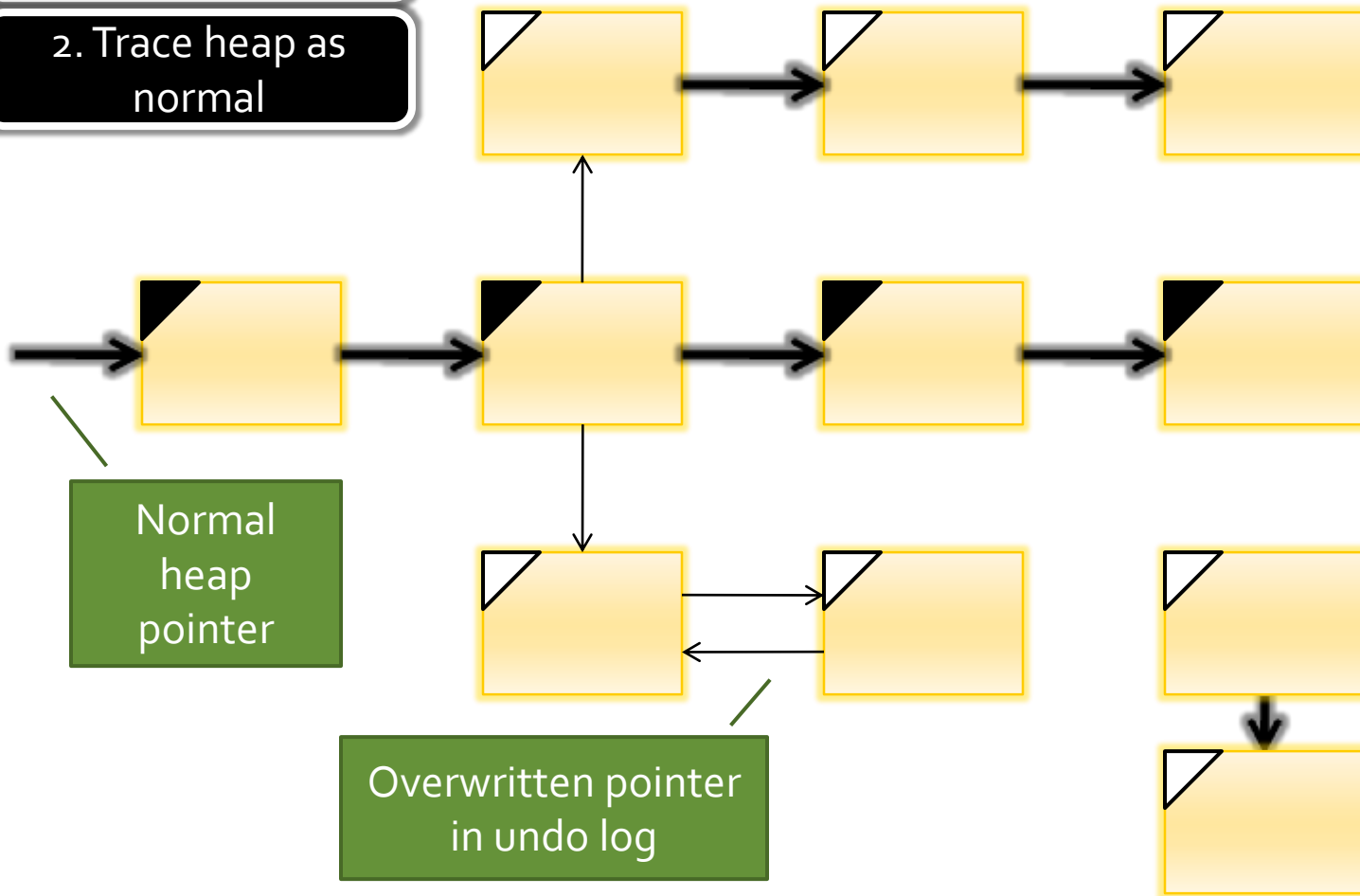
# Conservative algorithm



# Precise algorithm

1. Validate tx

2. Trace heap as normal

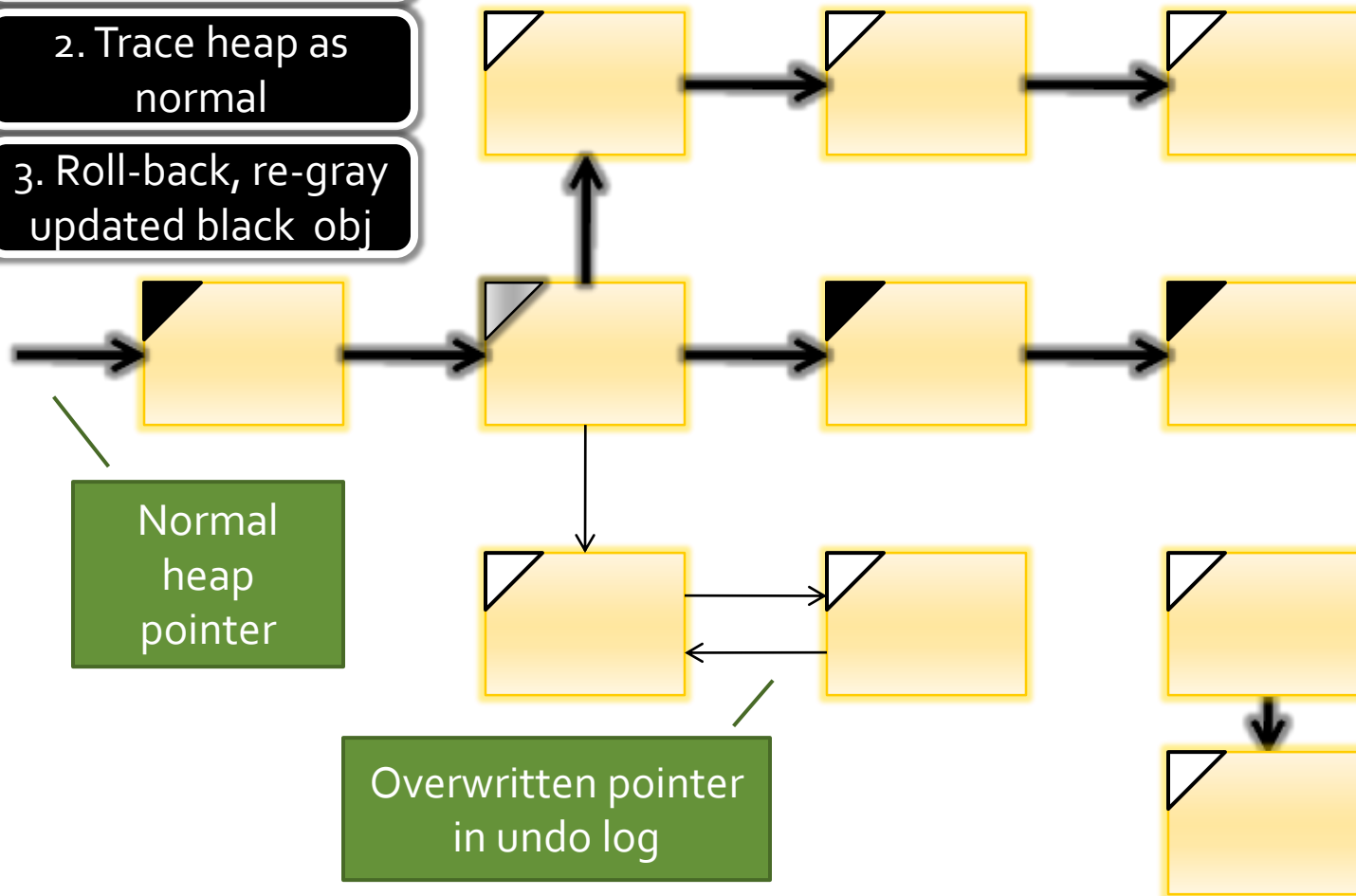


# Precise algorithm

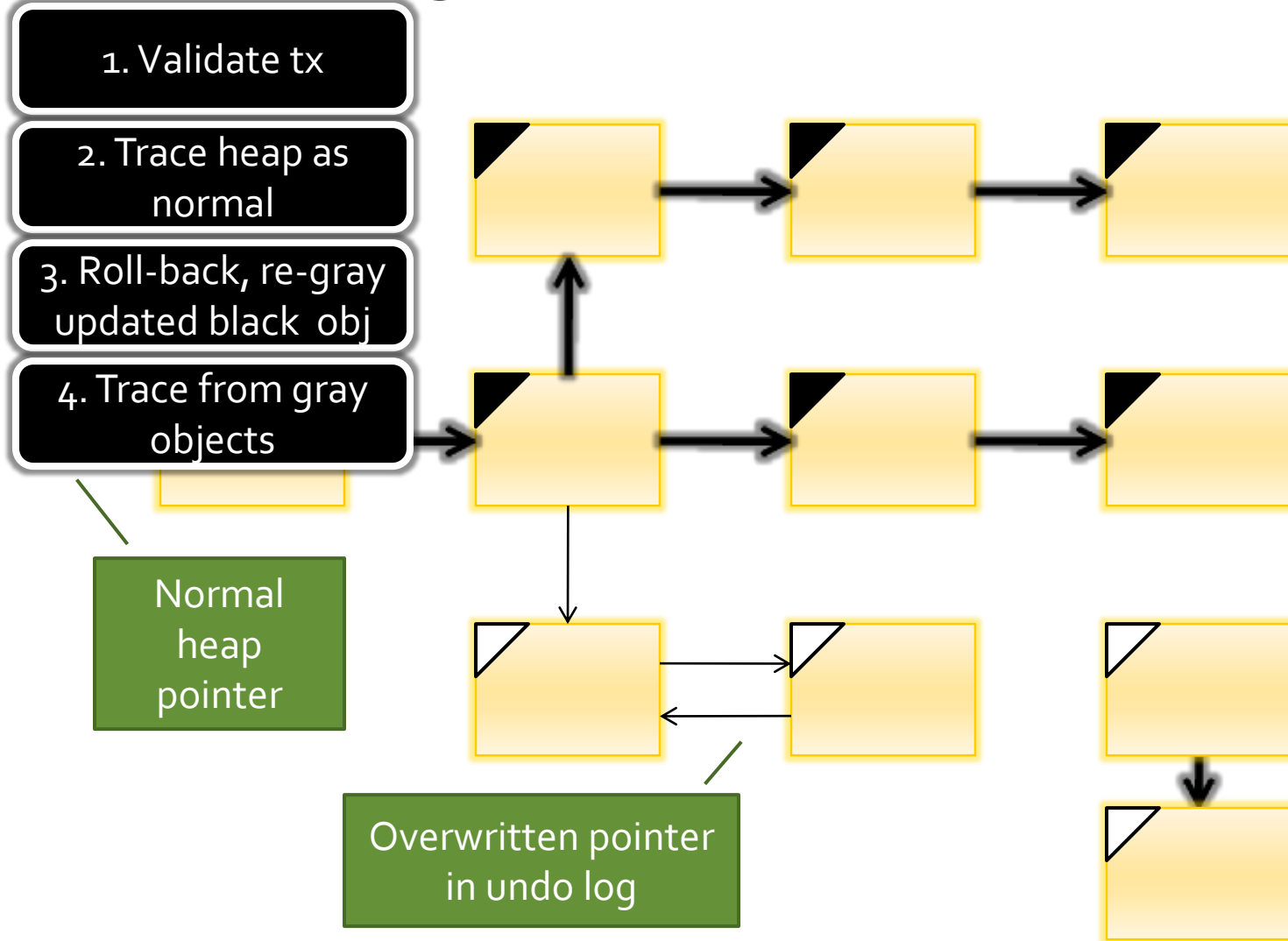
1. Validate tx

2. Trace heap as normal

3. Roll-back, re-gray updated black obj



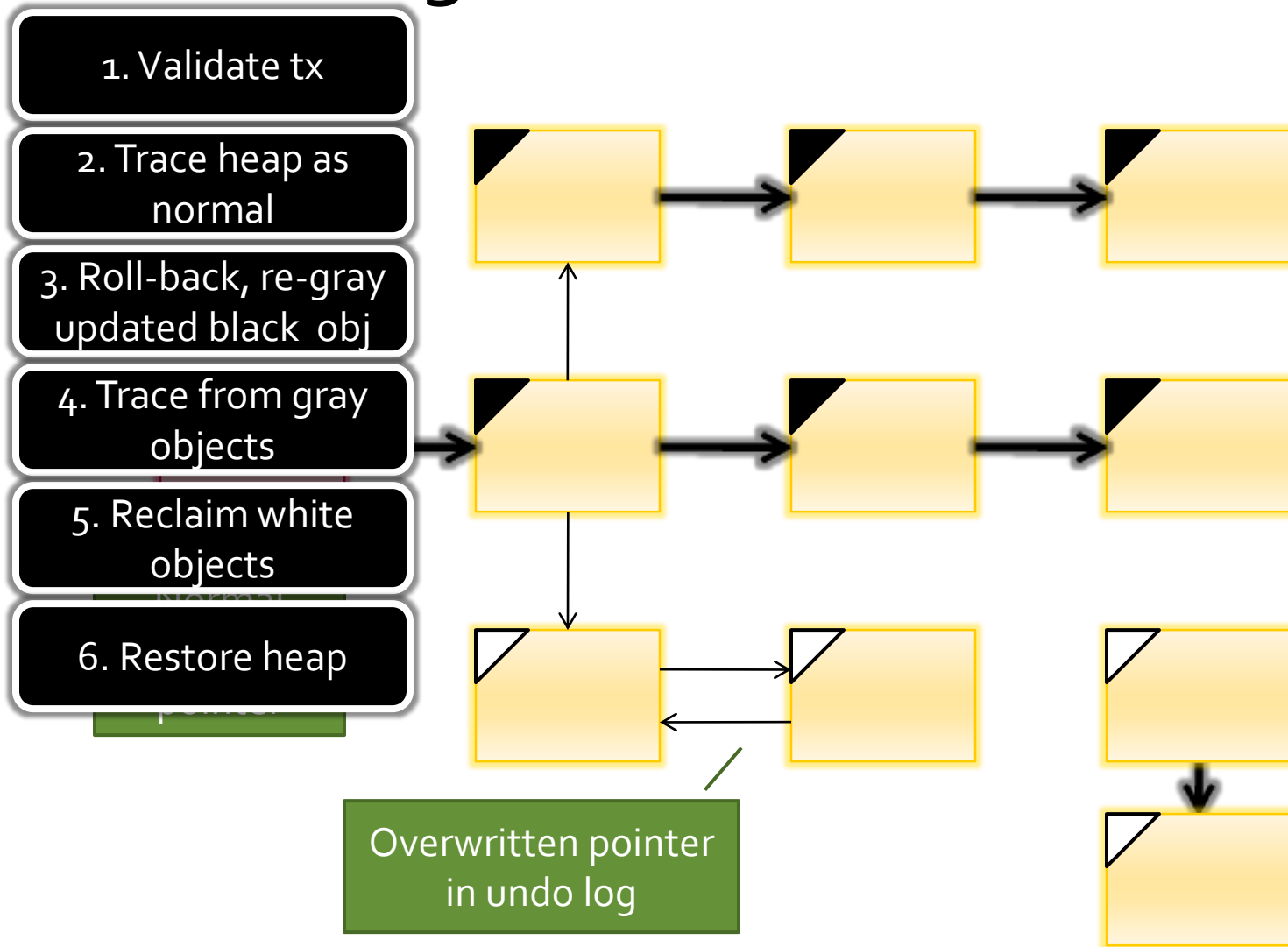
# Precise algorithm







# Precise algorithm



# Finalizers

```
Pair p;  
atomic {  
    p = new Pair();  
}
```

Suppose this block is attempted twice

How many times is this printed? (Or is this program wrong?)

```
Class Pair {  
    void Finalize() {  
        Console.Out.WriteLine("Hello world\n");  
    }  
}
```

# Finalizers

- Remember the intended semantics:
  - Exactly once execution
- Transactionally-allocated objects are only eligible for finalization when the tx commits
- Tentative allocation, non-finalization, and (re-)execution remains entirely transparent

# Condition synchronization

```
atomic {  
  buffer.data = 42;  
  buffer.full = true;  
}
```

This atomic block is  
only ready to run when  
buffer.full is true

```
atomic {  
  if (!buffer.full) {  
    retry;  
  }  
  result = buffer.data;  
  buffer.full = false;  
}
```

- Semantically: in STM-Haskell we required the scheduler to only run atomic blocks when they succeed without calling "retry"

# Primitive for synchronization

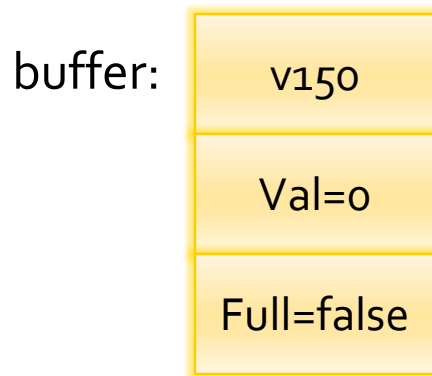
- `void WaitTX(tx)`
  - Semantically equivalent to `AbortTx`
  - Implementation may assume caller will immediately re-execute a (deterministic) tx
  - Implementation may introduce a delay to avoid unnecessary spinning
- Intuition:
  - No point re-executing the consumer until the producer has run

# Compiling "retry" to "WaitTx"

```
atomic {  
  if (!buffer.full)  
    retry;  
}  
result = buffer.  
buffer.full =  
}
```

```
void consume(Buffer b) {  
  do {  
    done = true;  
    try {  
      try {  
        tx = StartTx();  
        OpenForRead(tx, b);  
        if (!b.full) {  
          waitTx();  
        }  
        OpenForUpdate(tx, b);  
        result = b.data;  
        LogForUndo(tx, &b.full);  
        b.full = false;  
      } finally {  
        CommitTx();  
      }  
    } catch (TxInvalid) {  
      done = false;  
    }  
  } while (!done);  
}
```

# Implementing WaitTx

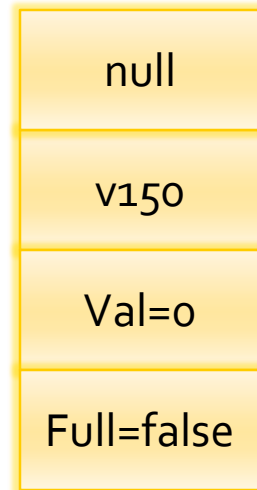




# Implementing WaitTx

1. Extend object header with list of waiters

buffer:

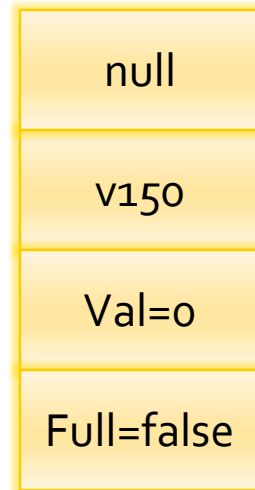


# Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

buffer:

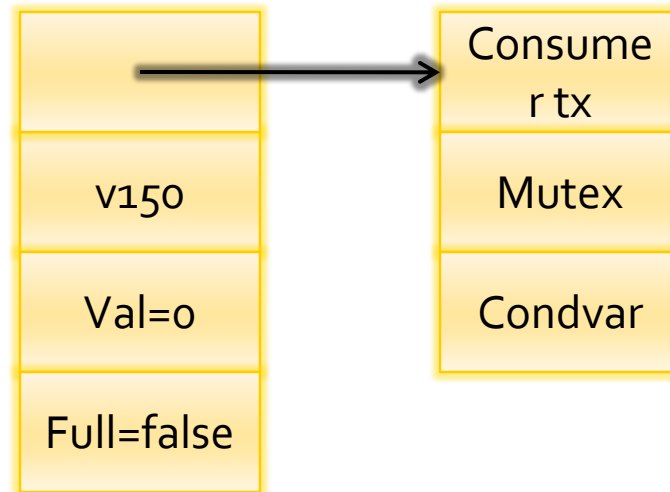


# Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set



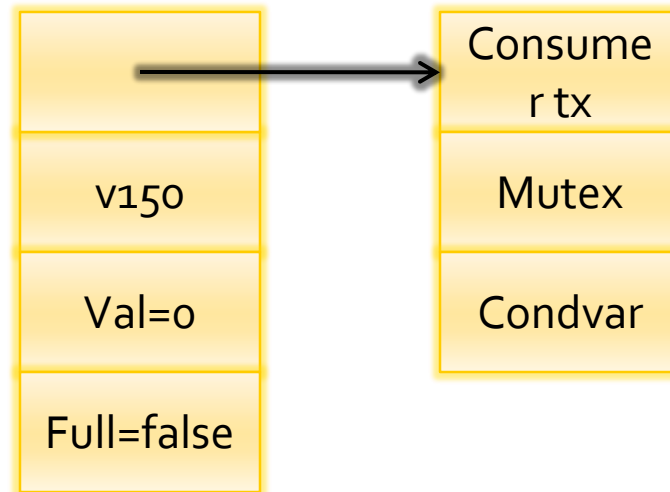
# Implementing WaitTx

1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks



# Implementing WaitTx

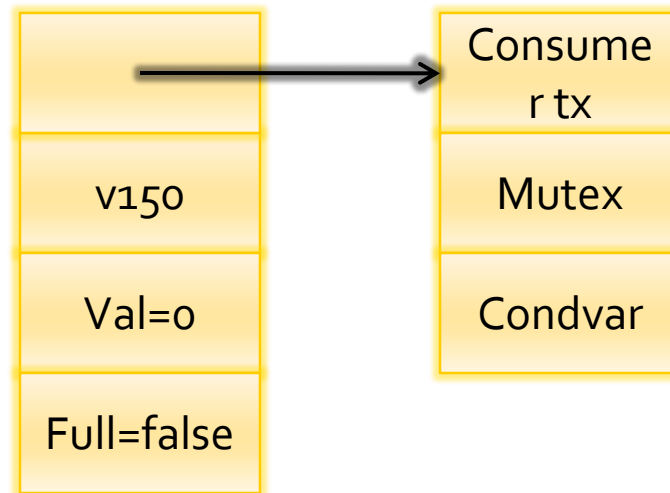
1. Extend object header with list of waiters

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

5. CommitTx wakes waiters on objects in its write set



# Implementing WaitTx

1. Extend object header with list of waiters

Use "thin locks" style tricks to avoid fixed header word allocation

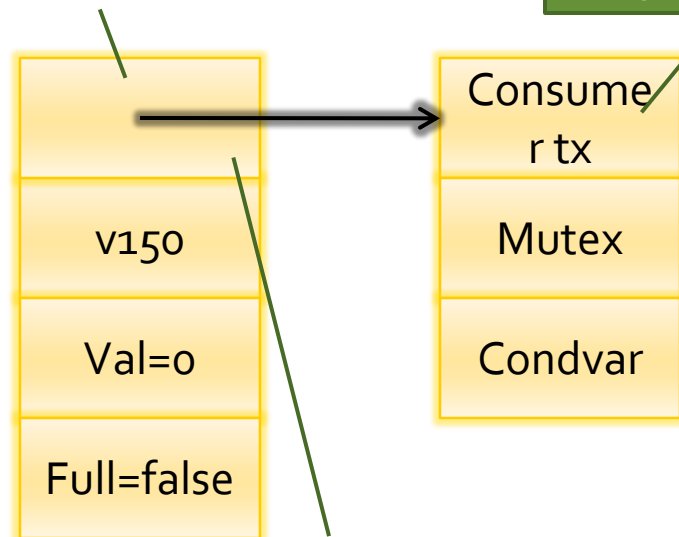
NB: many-to-many relationship, so probably use separate doubly linked list

2. Extend tx records with a mutex & condvar pair

3. WaitTx links the consumer to the lists in its read set

4. WaitTx validates, locks the mutex, updates its status, blocks

5. CommitTx wakes waiters on objects in its write set



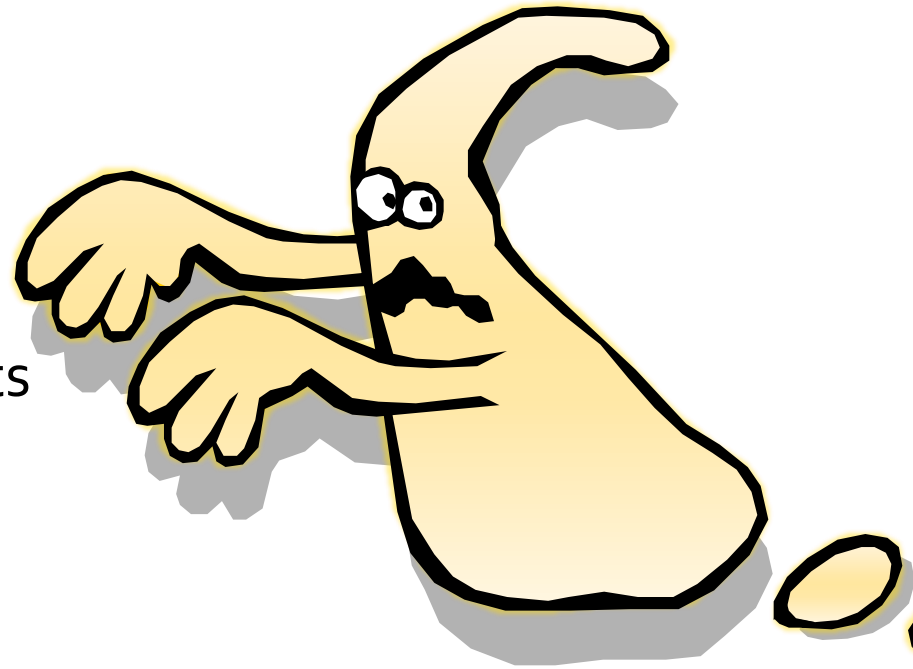
Use latch in the header for concurrency control on the list

# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory
  - TM and composability
  - STM internals
  - Integration into a language runtime system
  - Sandboxing & strong isolation
  - Current performance and my perspective on TM

# Sandboxing zombie transactions

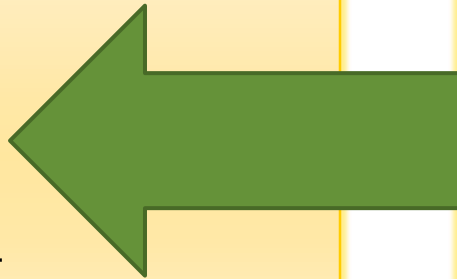
- Those that have become invalid but don't yet know it
- May access memory
- May raise exceptions
- May attempt system calls etc
- General principle – validate before revealing any tx's effects outside the STM world





# Looping / slow zombies

```
void Method1(Pair p) {  
  atomic {  
    ta = p.a;  
  
    tb = p.b;  
    if (ta != tb) {  
      while (true) {  
    } } } }  
}
```



```
void Method2(Pair p) {  
  atomic {  
    p.a = 100;  
    p.b = 100;  
  } }  
}
```

- Method2 runs between Method1's memory accesses
- The transaction running Method1 becomes a zombie... but never attempts to commit

# Looping / slow zombies

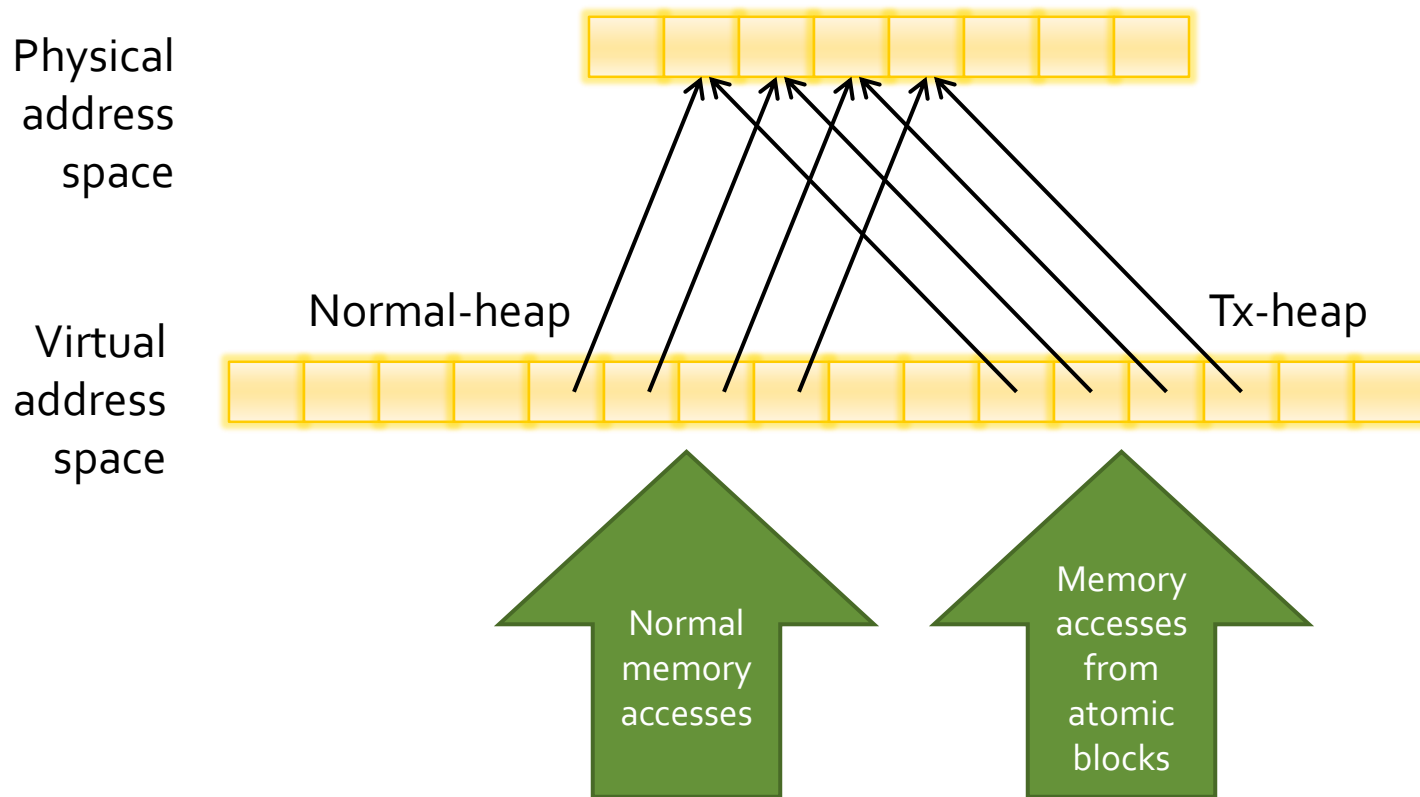
- Add new API function "ValidationTick"
- ValidationTick guarantees: it will eventually detect if its calling transaction is invalid
- Call it in any loop not otherwise calling a TM API function
- Optimize ValidationTick so it only does "real" validation occasionally
- (Could also optimize the placement of ValidationTick calls)

```
OpenForRead(p);  
ta = p.a;  
tb = p.b;  
if (ta != tb) {  
    while (true) {  
        validationTick();  
    }  
}
```

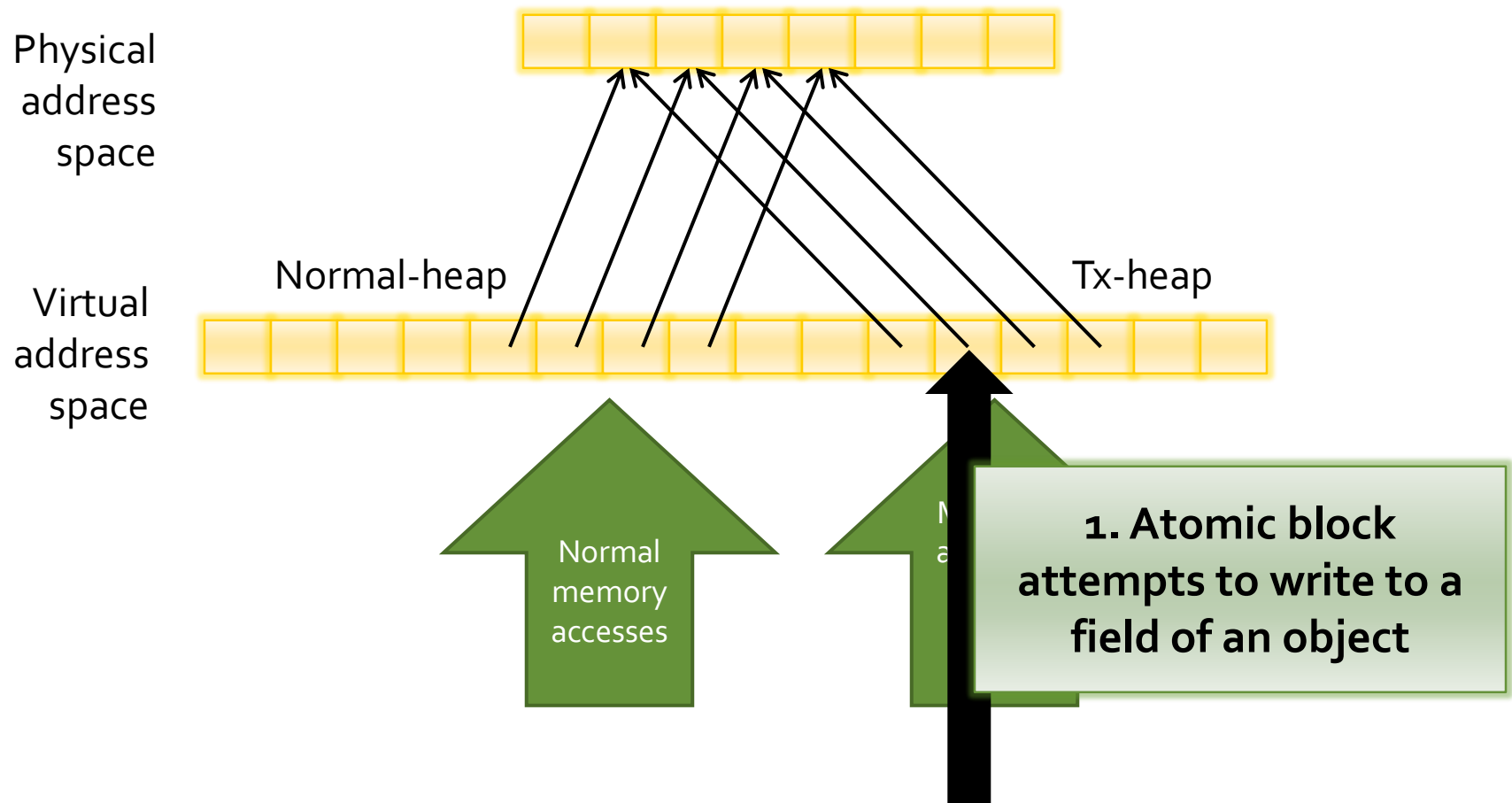
# Strong isolation

- Add a mechanism to detect conflicts between tx and normal accesses
- We would like:
  - No overhead on direct accesses
  - Predictable performance
  - Little overhead over weak atomicity

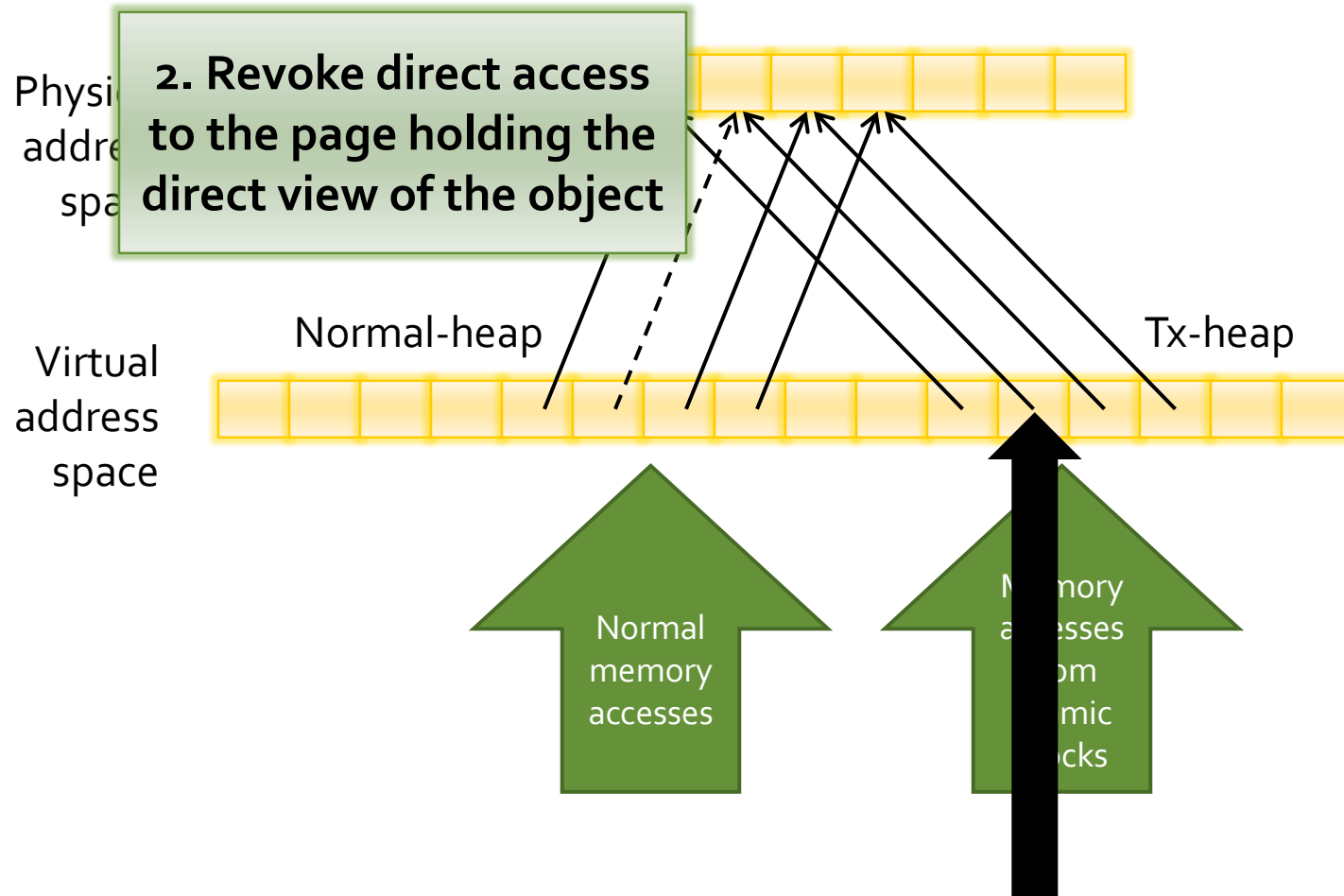
# Strong isolation: implementation



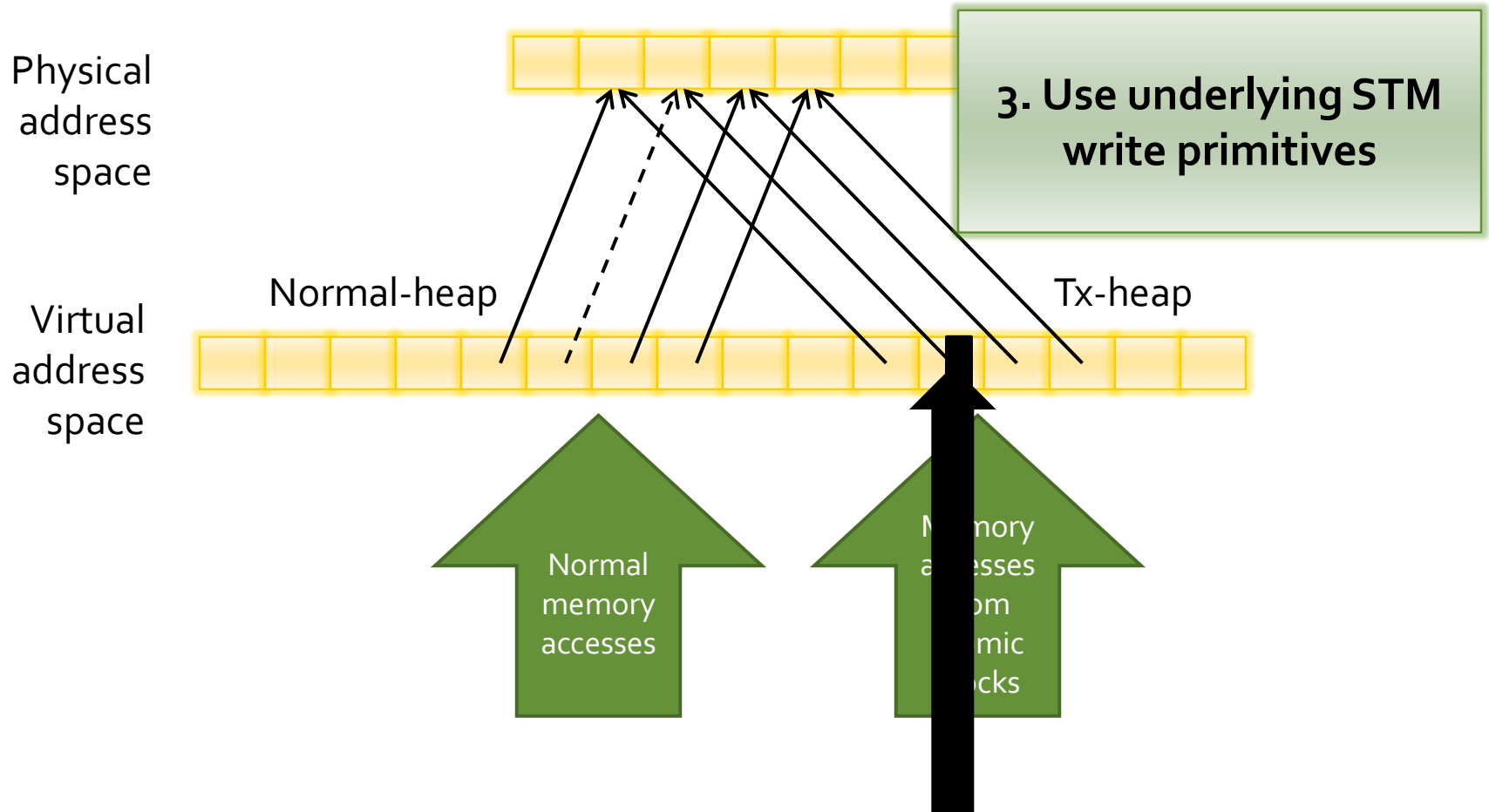
# Writes from atomic blocks



# Writes from atomic blocks

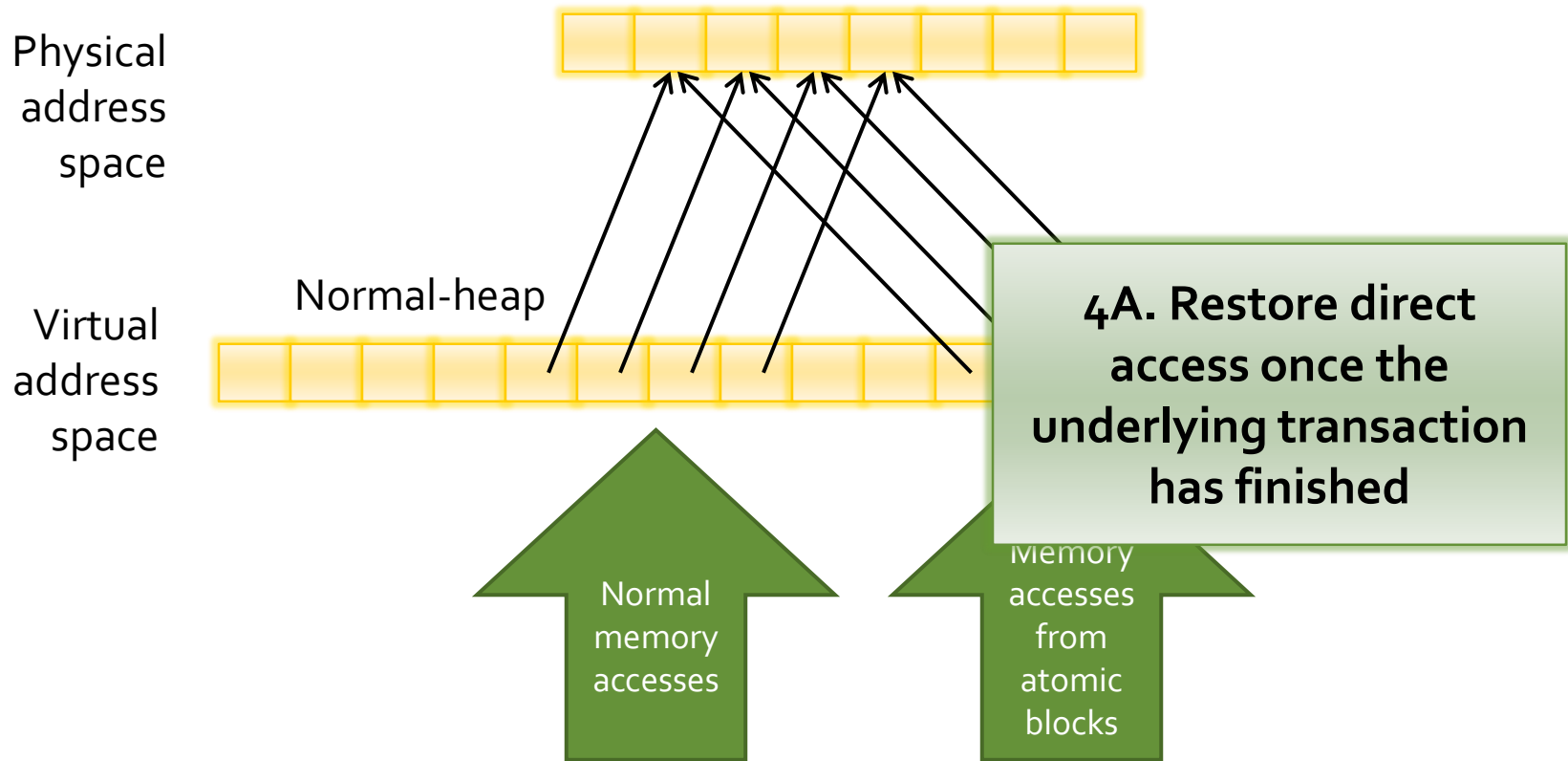


# Writes from atomic blocks



# Case 1

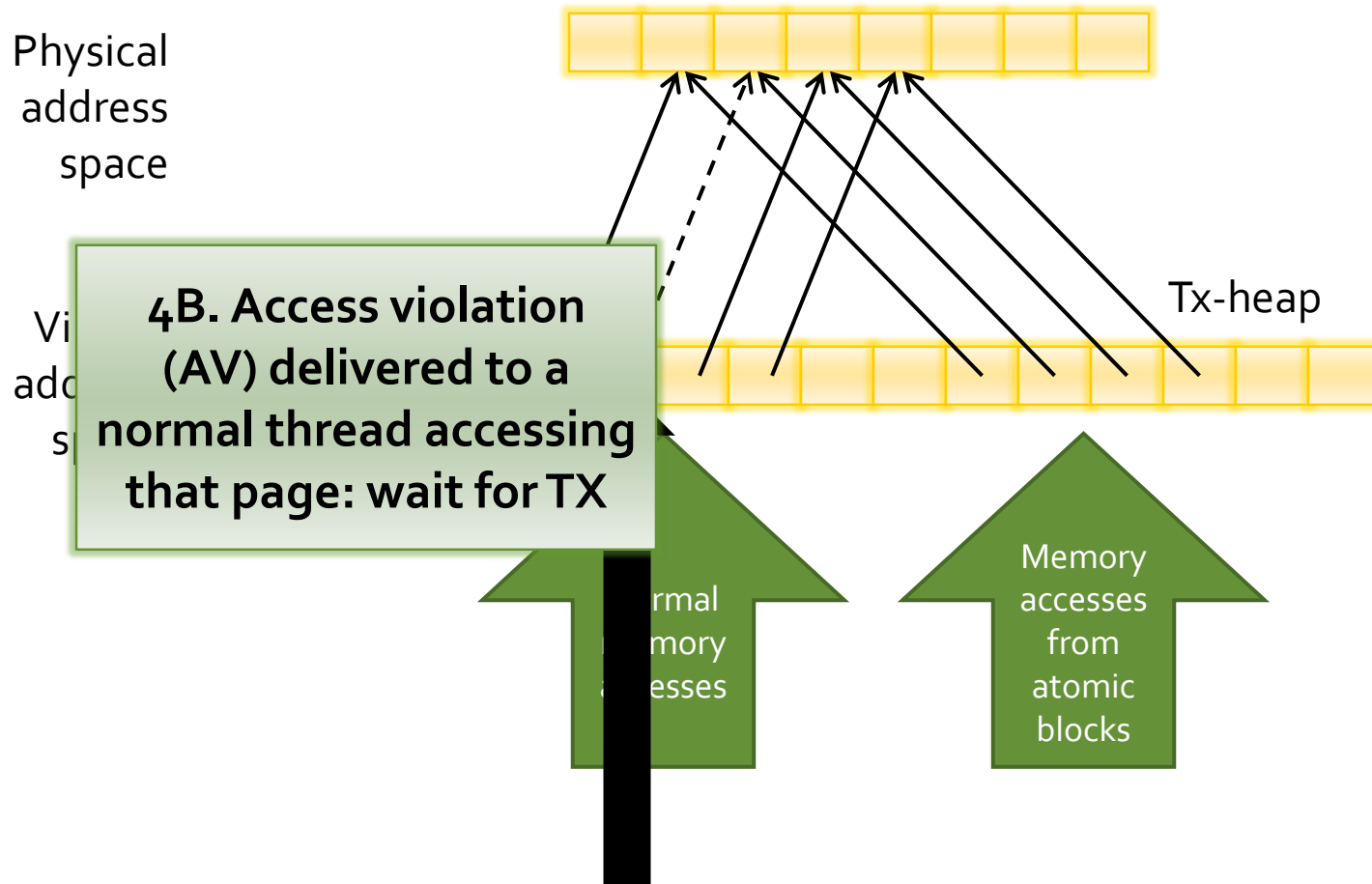
## Accesses from atomic blocks



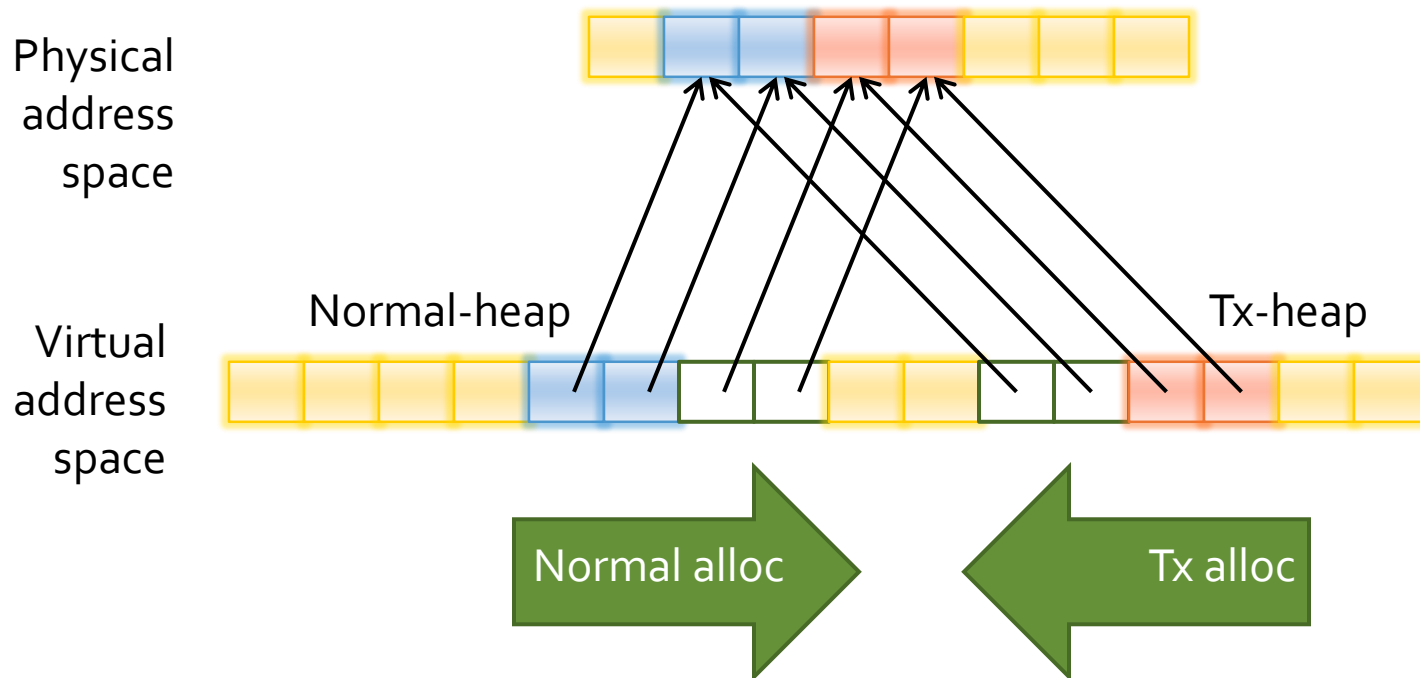


# Case 2

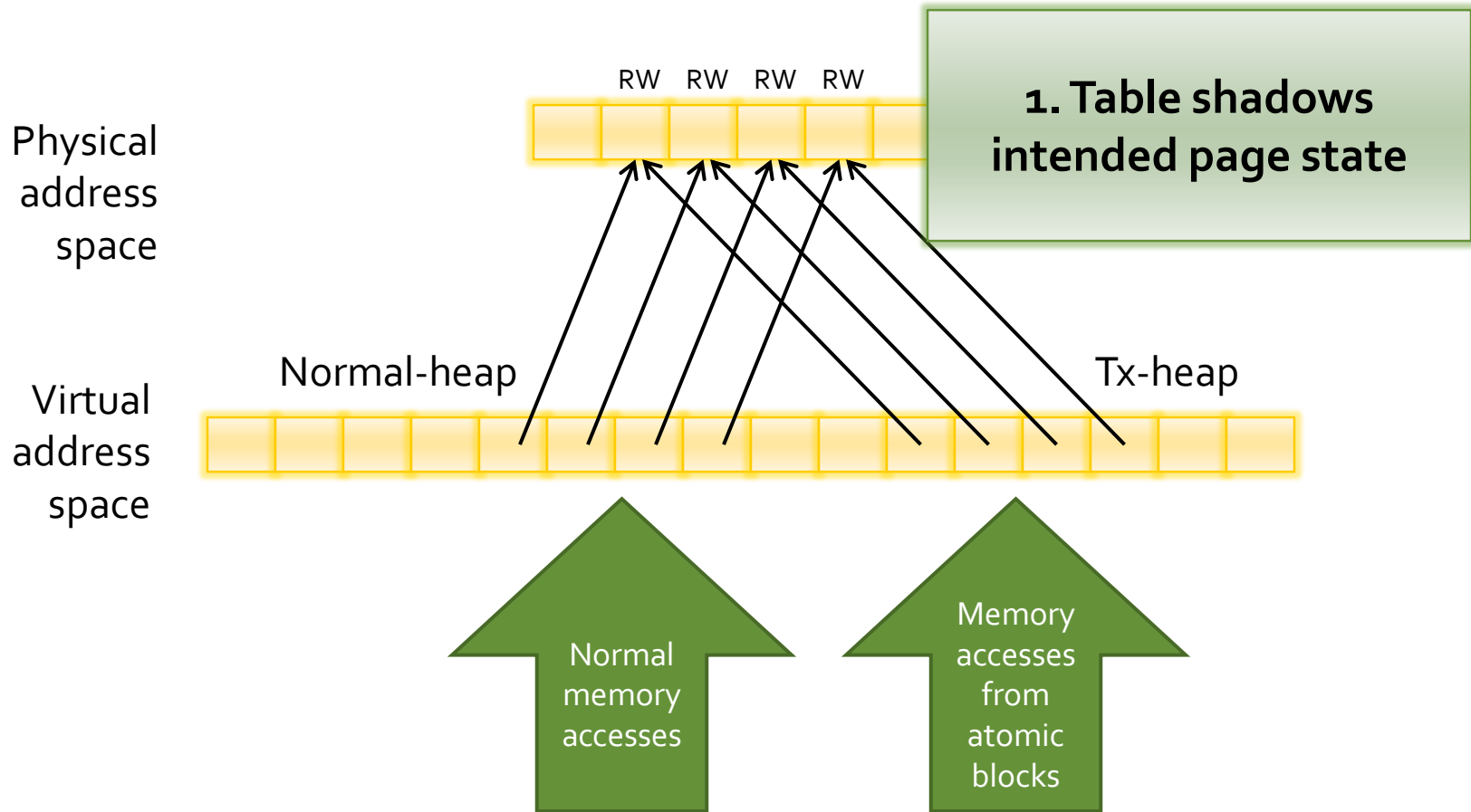
## Conflicting normal access



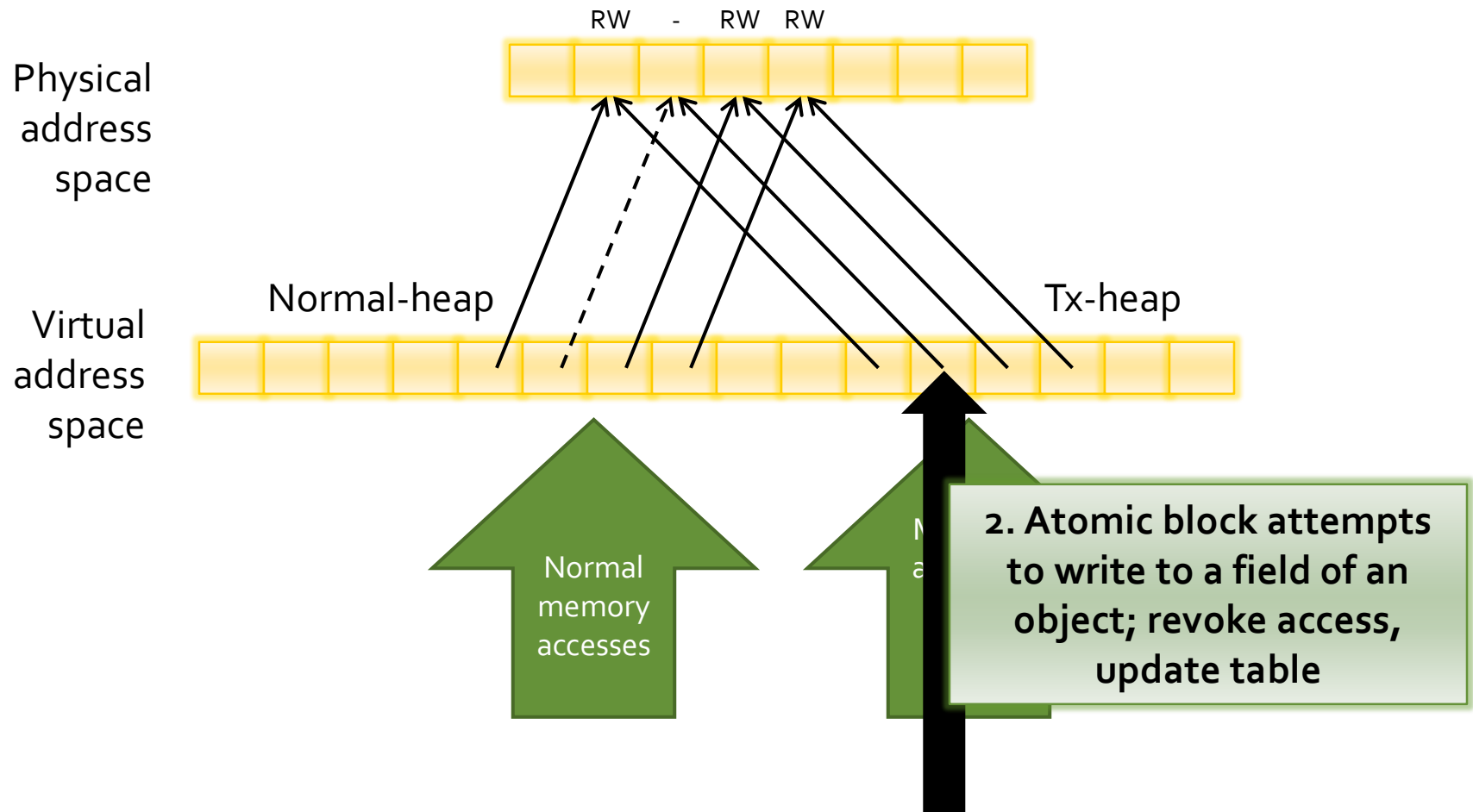
# Separate tx / non-tx allocations



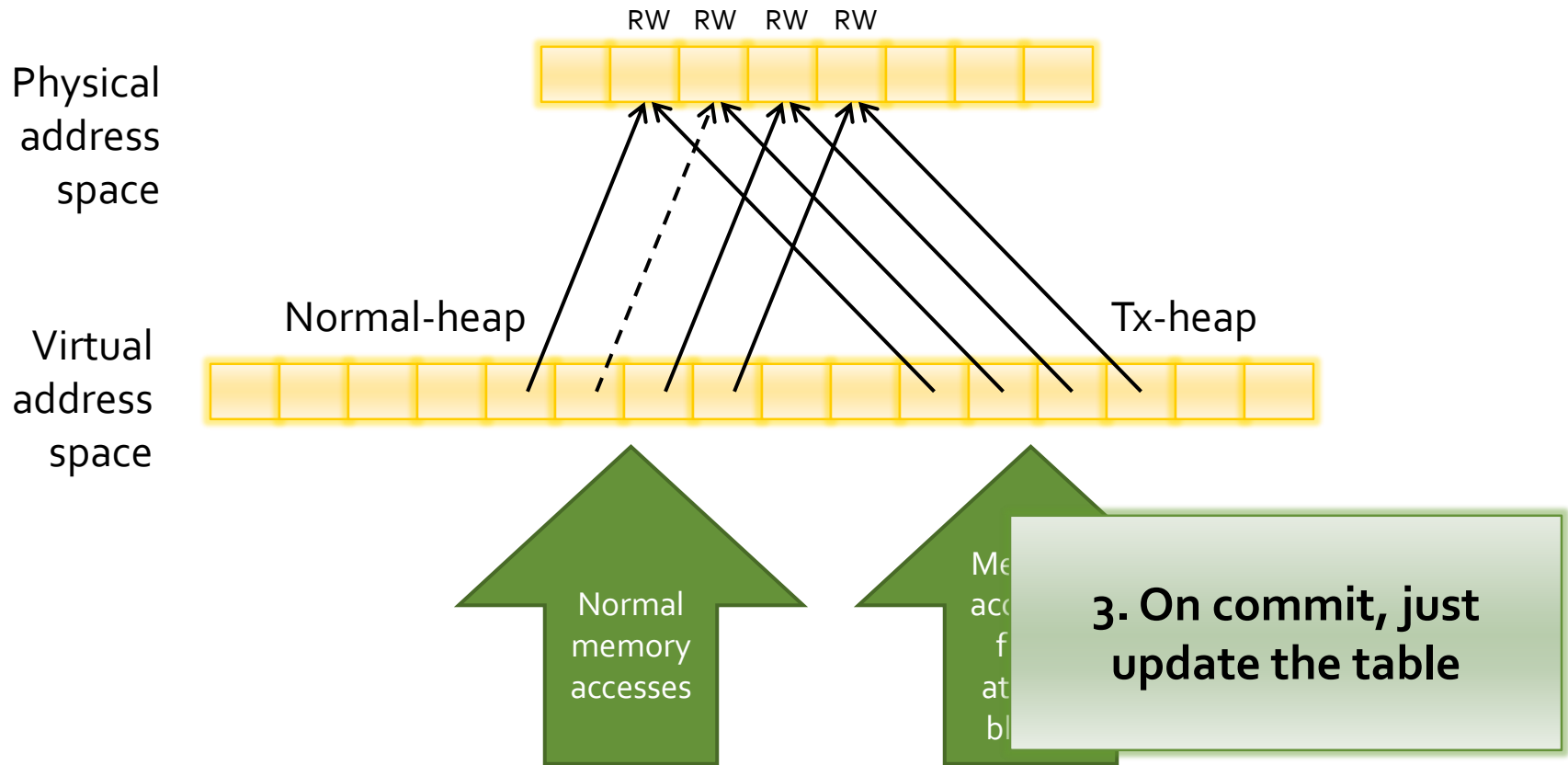
# Make page protections lazily



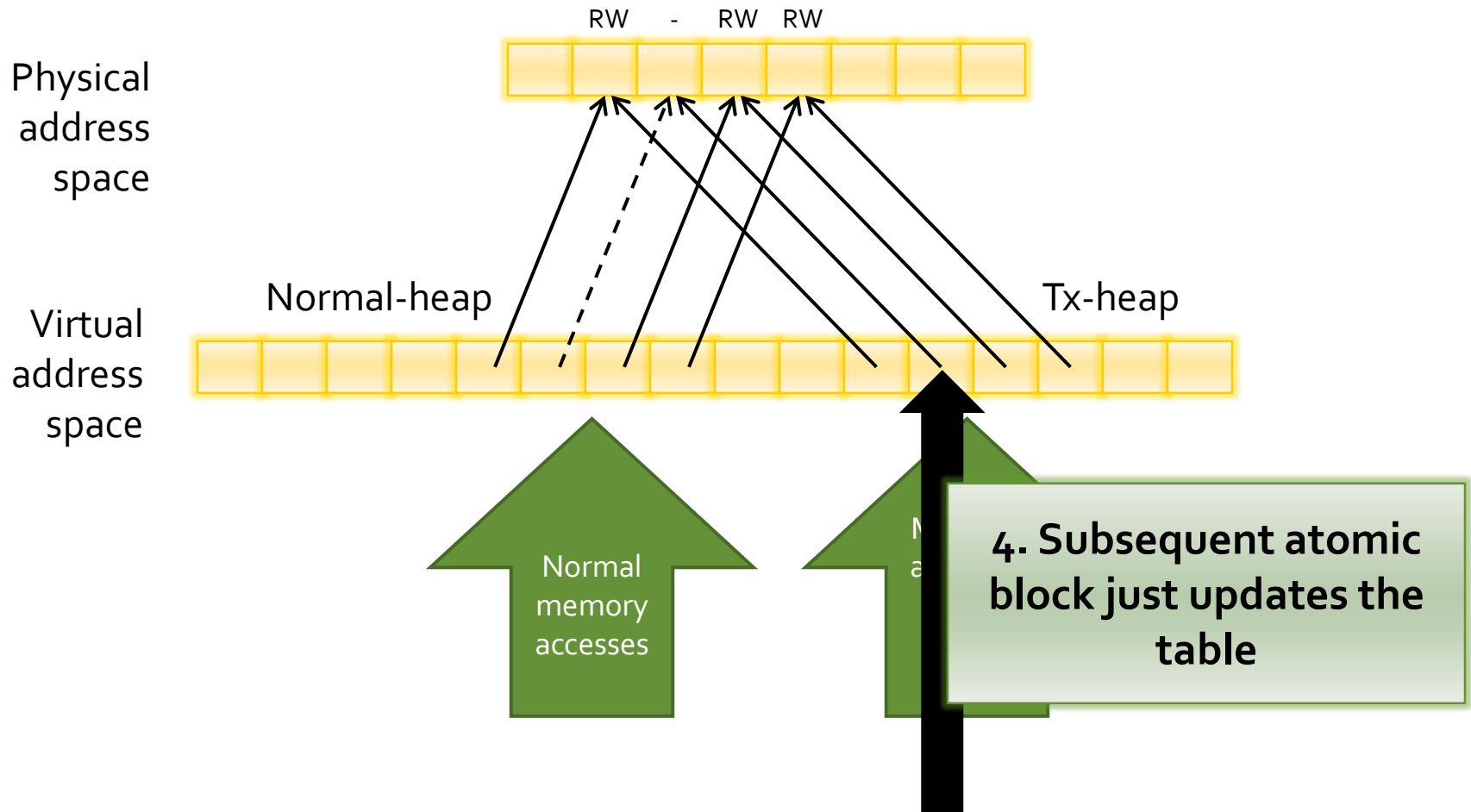
# Make page protections lazily



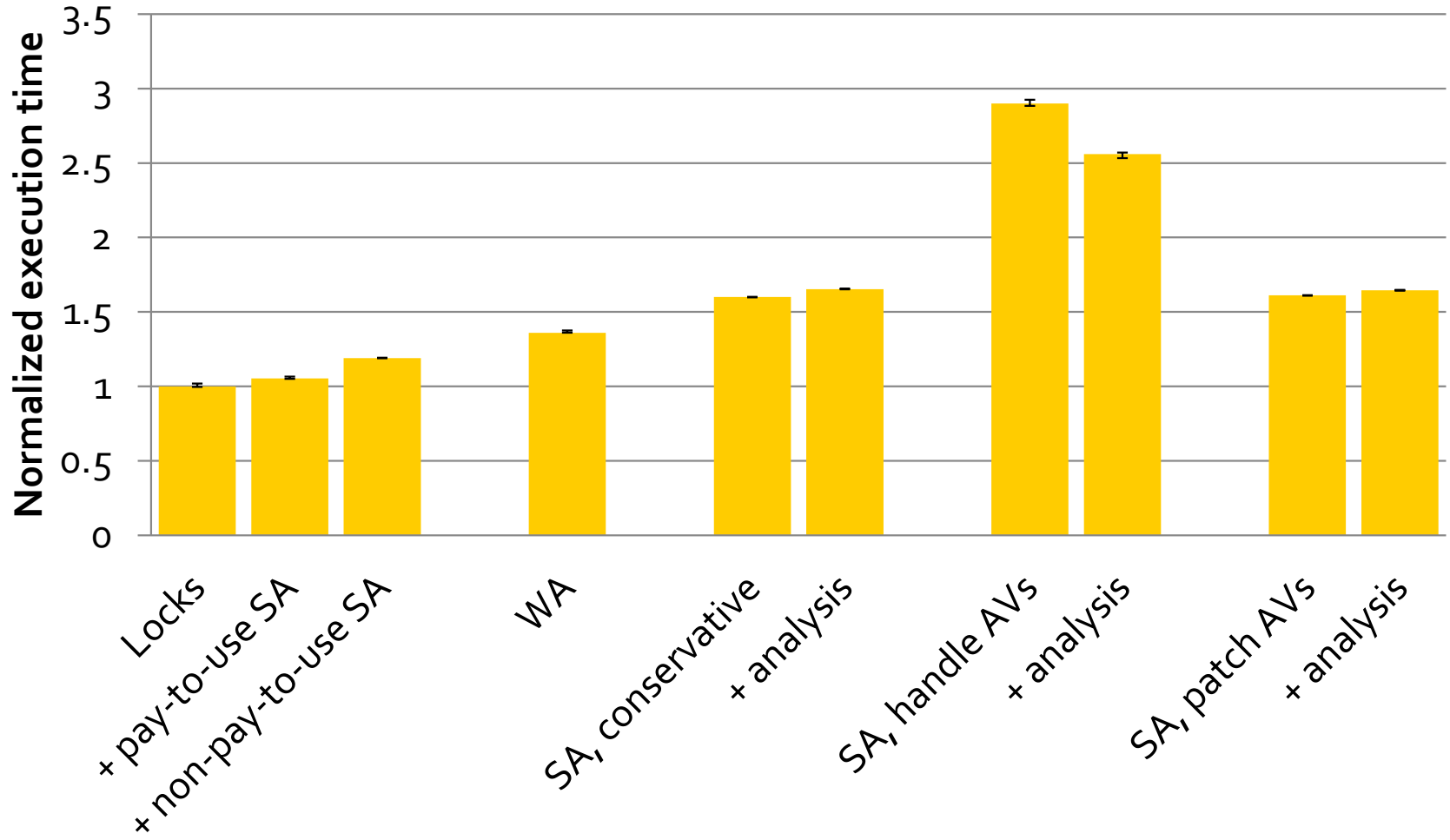
# Make page protections lazily



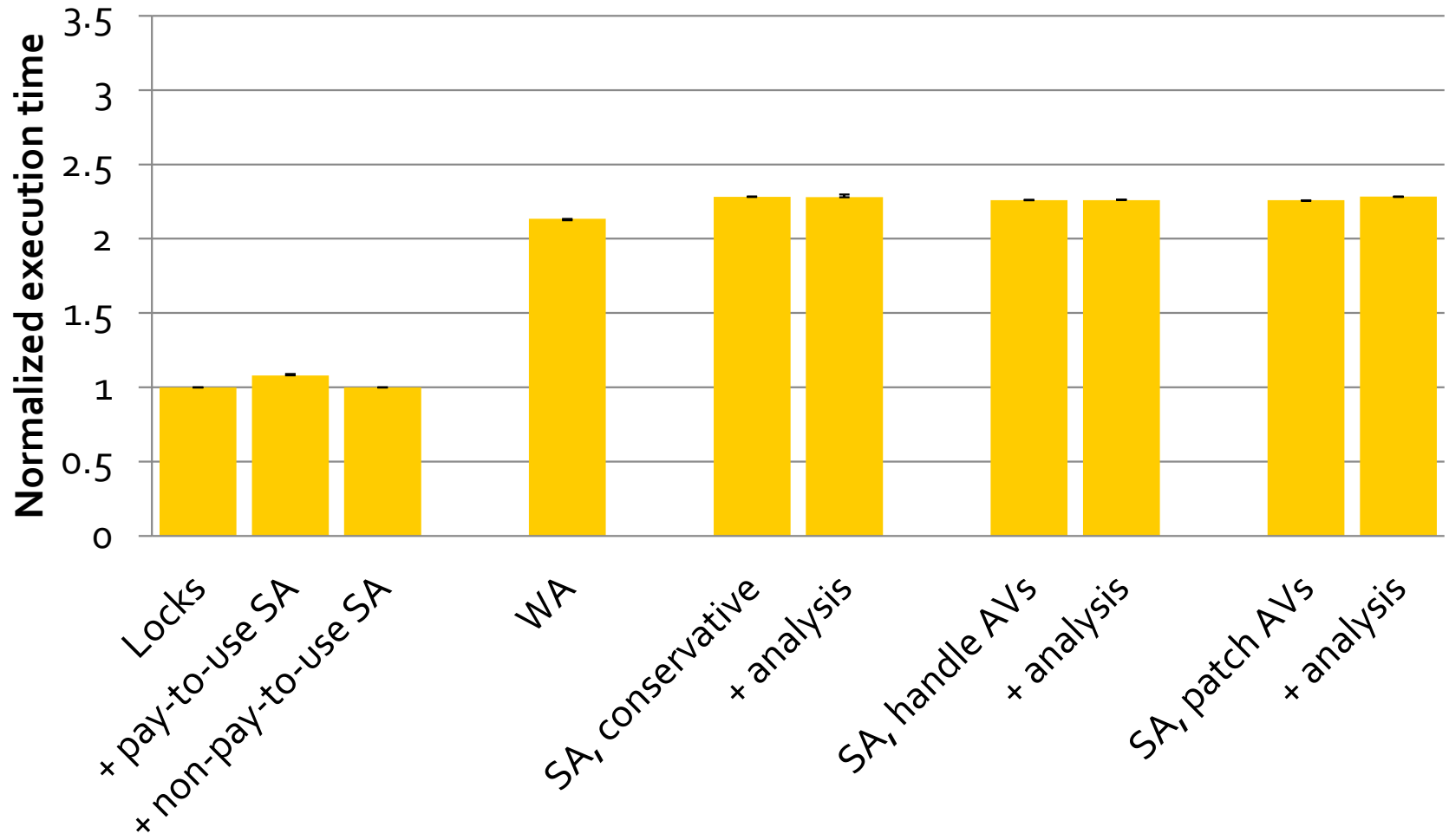
# Make page protections lazily



# Genome



# Labyrinth





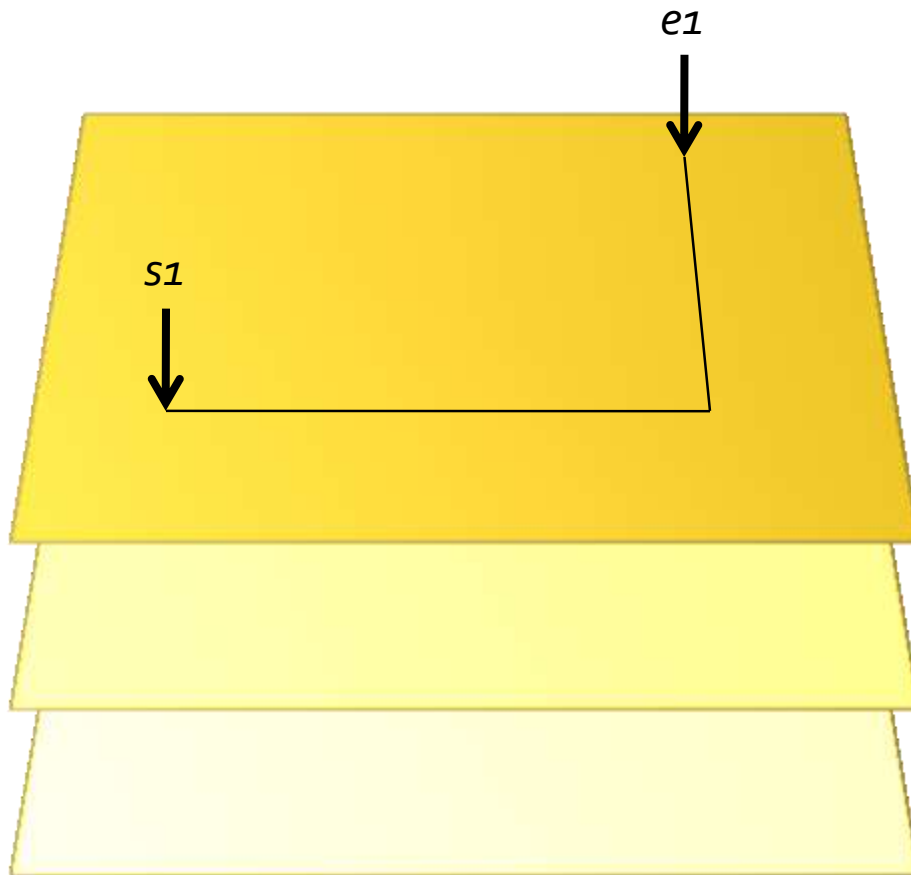
# Course overview: structure

- Building locks
- Lock-free programming
- Transactional memory
  - TM and composability
  - STM internals
  - Integration into a language runtime system
  - Sandboxing & strong isolation
  - Combining TM with libraries, locking, and IO
  - Current performance and my perspective on TM

# Performance figures depend on...

- **Workload** : What do the atomic blocks do? How long is spent inside them?
- **Baseline implementation**: Mature existing compiler, or prototype?
- **Intended semantics**: Support static separation? Violation freedom (TDRF)?
- **STM implementation**: In-place updates, deferred updates, eager/lazy conflict detection, visible/invisible readers?
- **STM-specific optimizations**: e.g. to remove or downgrade redundant TM operations
- **Integration**: e.g. dynamically between the GC and the STM, or inlining of STM functions during compilation
- **Implementation effort**: low-level perf tweaks, tuning, etc.
- **Hardware**: e.g. performance of CAS and memory system

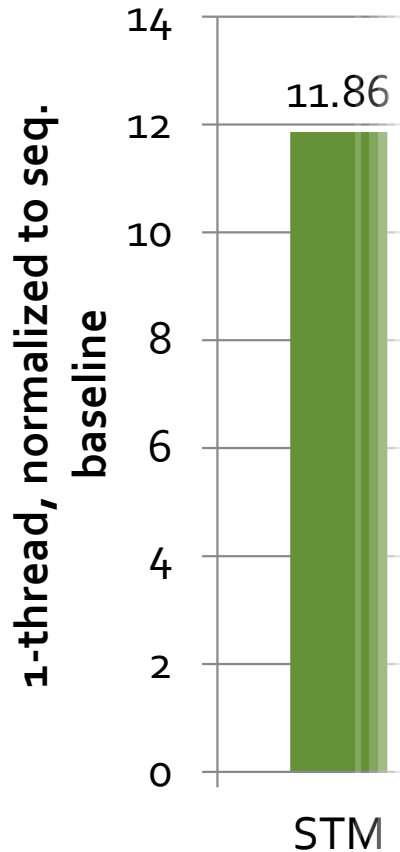
# Labyrinth



- STAMP v0.9.10
- 256x256x3 grid
- Routing 256 paths
- Almost all execution inside atomic blocks
- Atomic blocks can attempt 100K+ updates
- C# version derived from original C
- Compiled using Bartok, whole program mode, C# -> x86 (~80% perf of original C with VS2008)
- Overhead results with Core2 Duo running Windows Vista

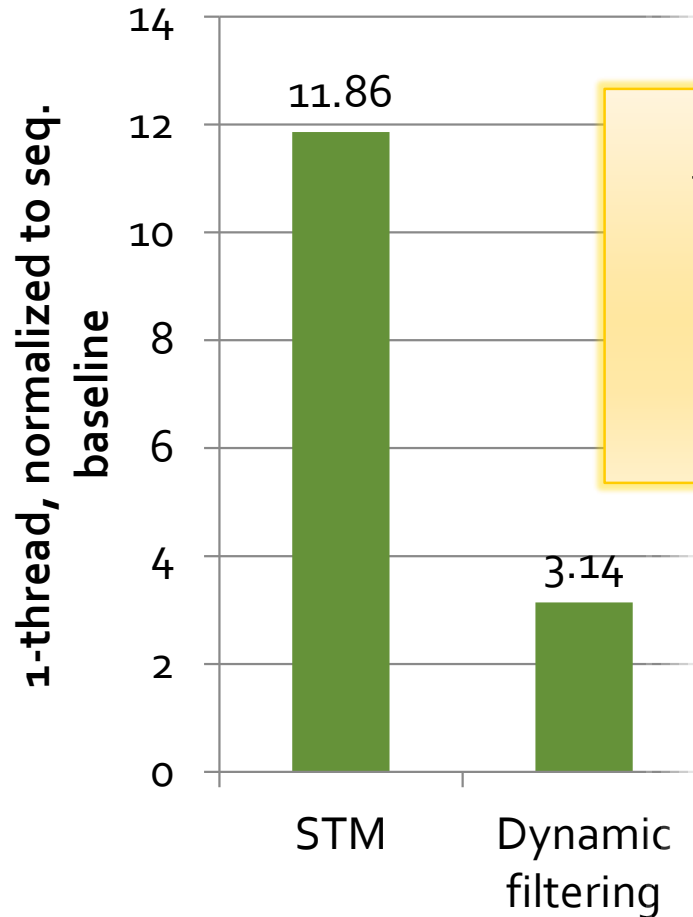
“STAMP: Stanford Transactional Applications for Multi-Processing”  
Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, IISWC 2008

# Sequential overhead



STM implementation supporting static separation  
In-place updates  
Lazy conflict detection  
Per-object STM metadata  
Addition of read/write barriers before accesses  
Read: log per-object metadata word  
Update: CAS on per-object metadata word  
Update: log value being overwritten

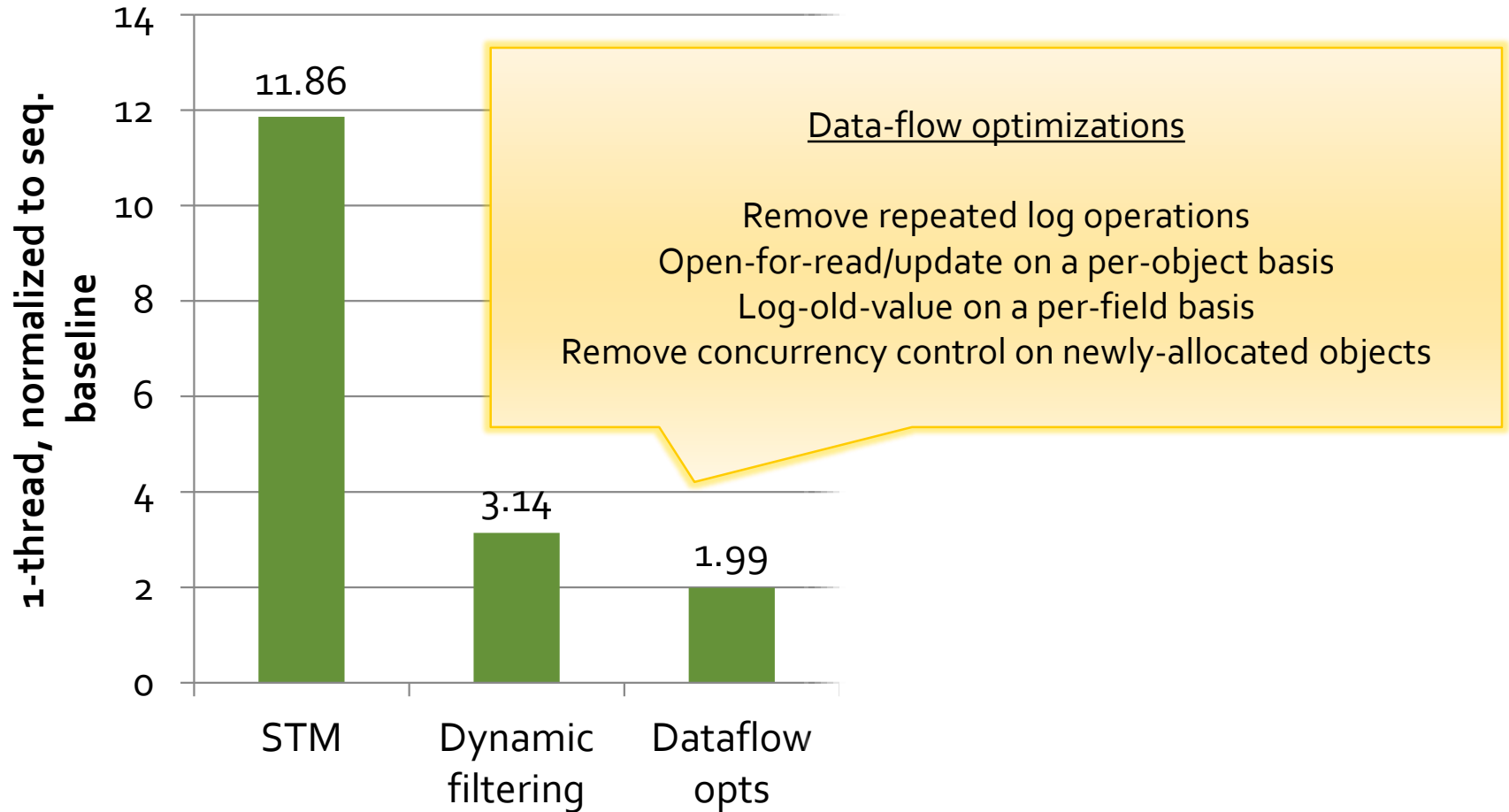
# Sequential overhead



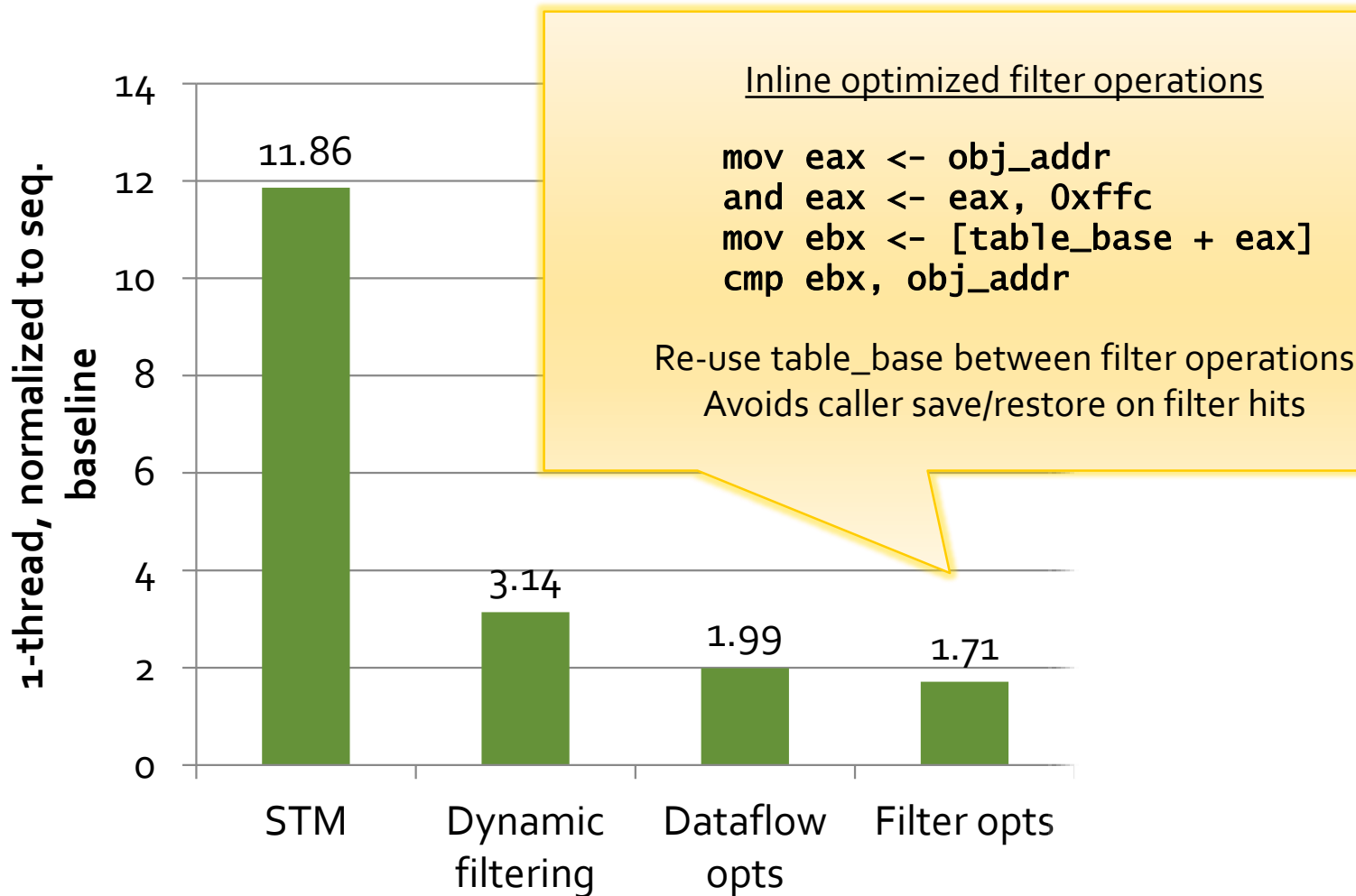
Dynamic filtering to remove redundant logging

Log size grows with #locations accessed  
Consequential reduction in validation time  
1<sup>st</sup> level: per-thread hashtable (1024 entries)  
2<sup>nd</sup> level: per-object bitmap of updated fields

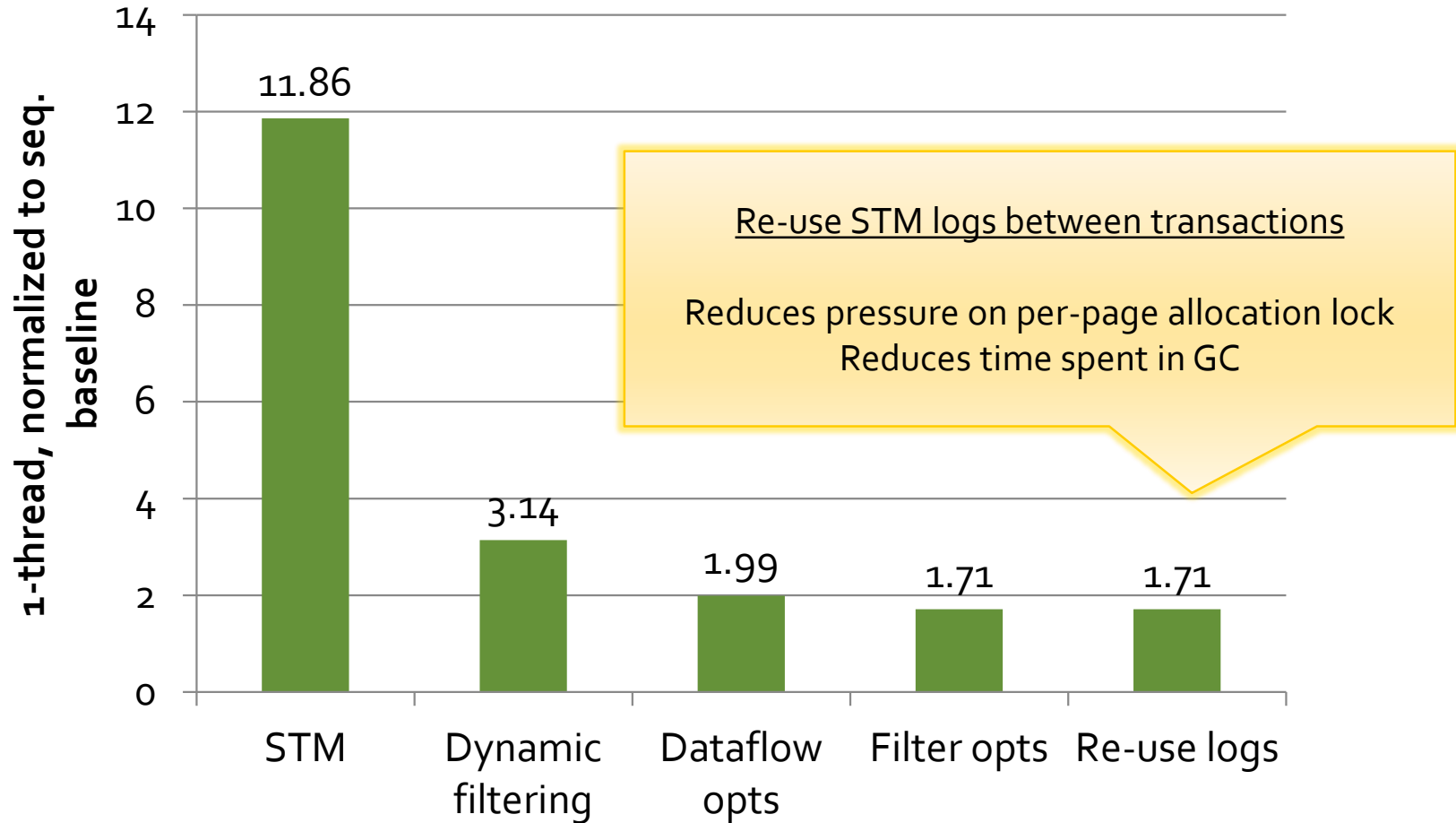
# Sequential overhead



# Sequential overhead

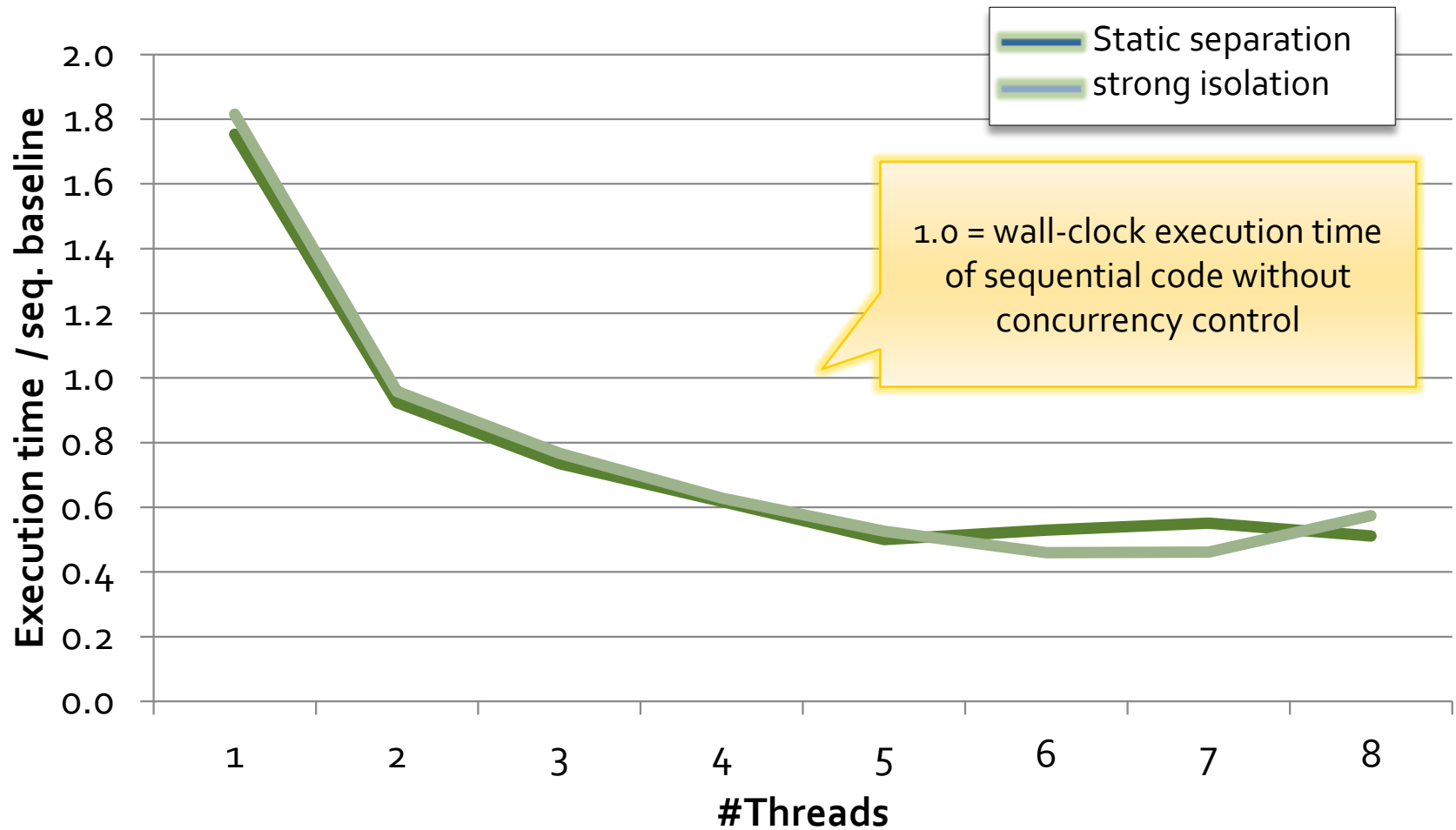


# Sequential overhead

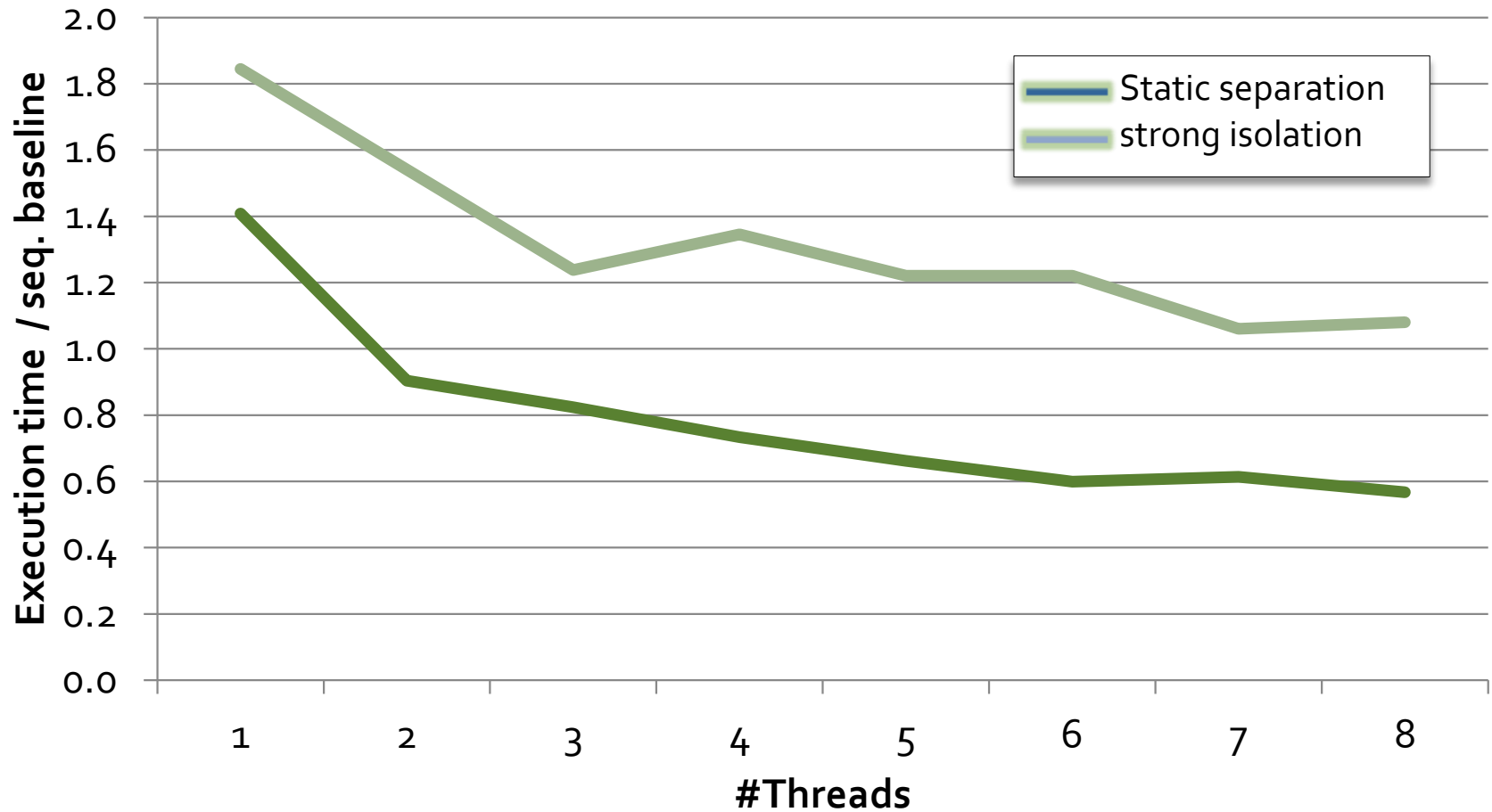




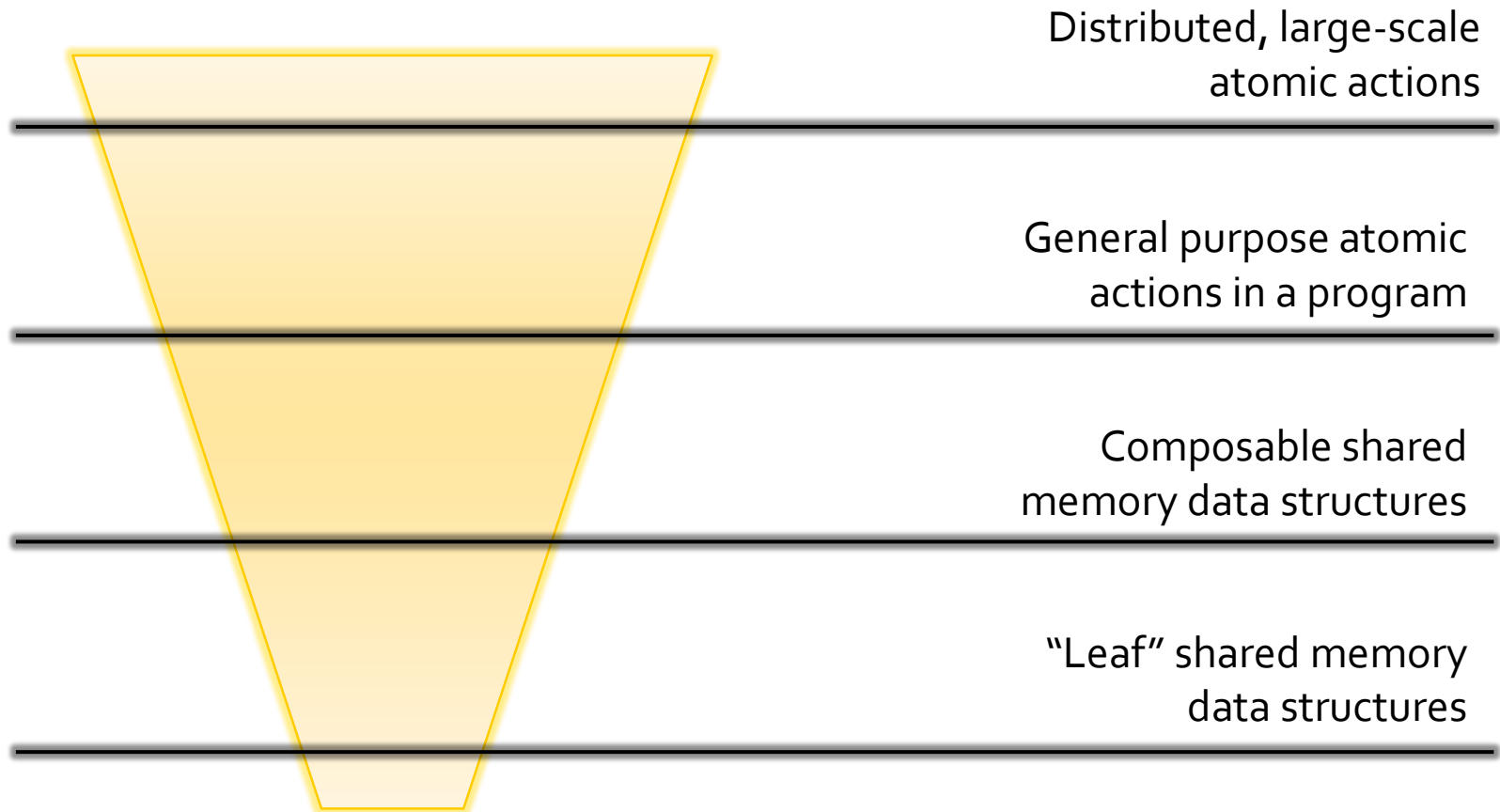
# Scaling – Labyrinth



# Scaling – Genome



# Granularity



# Programming abstraction

## Lock elision

The program's semantics is defined using locks. TM is used as an implementation mechanism.

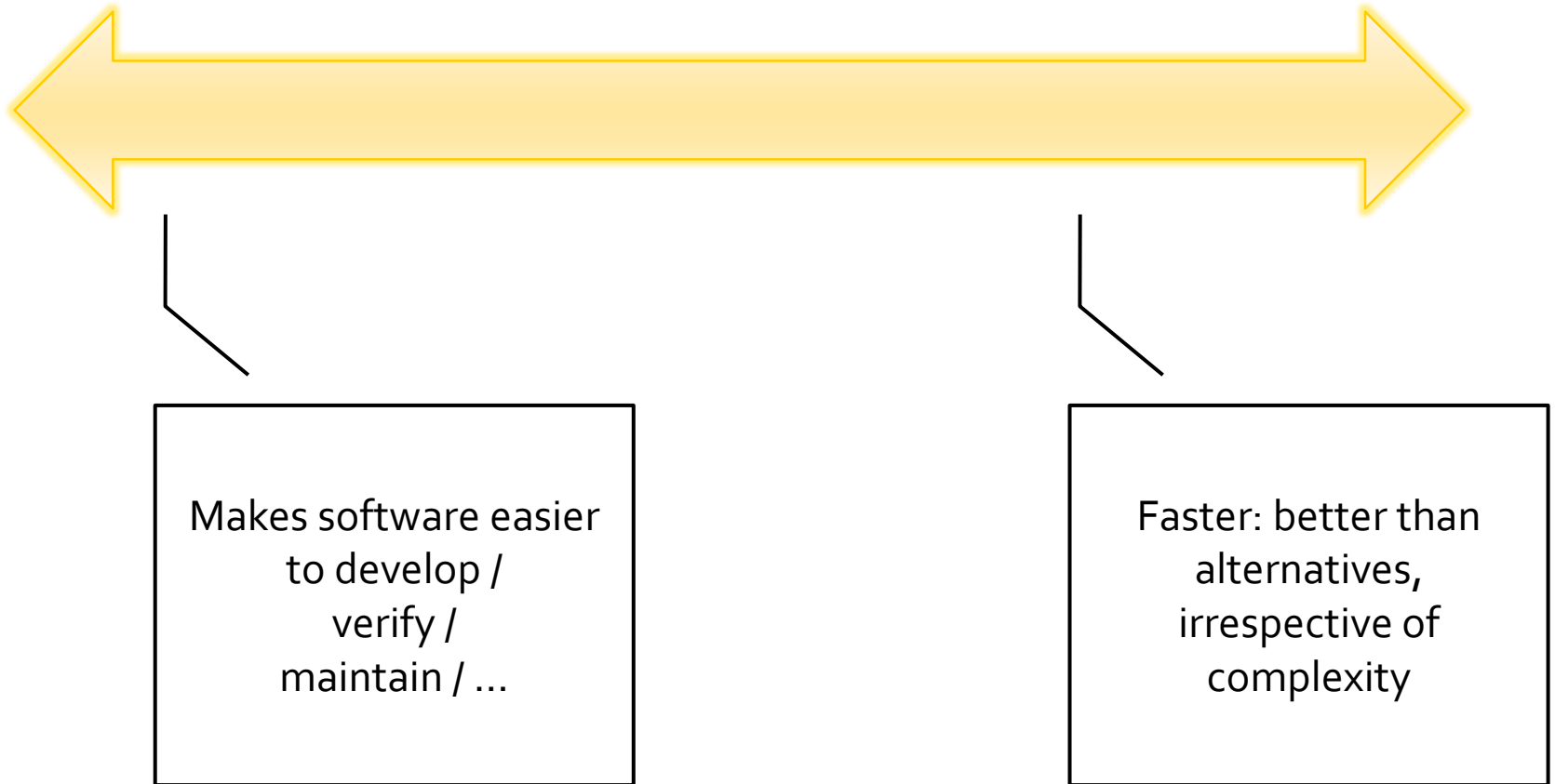
## Speculation

Semantics defined by speculative execution, commit, etc. (either implicitly, or explicitly)

## Atomic

Semantics defined by atomic execution (e.g. "atomic { X }"). Speculation, if used, is abstracted by the implementation.

# Purpose



# Design points that I like

HW DCAS / 3-CAS / ...

Granularity: leaf data structures

Abstraction: atomic multi-word CAS

Purpose: faster

HTM with limited guarantees (~ASF)

Granularity: leaf data structures

Abstraction: short transactions

Purpose: faster

Static separation (e.g., STM-Haskell)

Granularity: composable data structures

Abstraction: atomic actions

Purpose: easier, decent perf

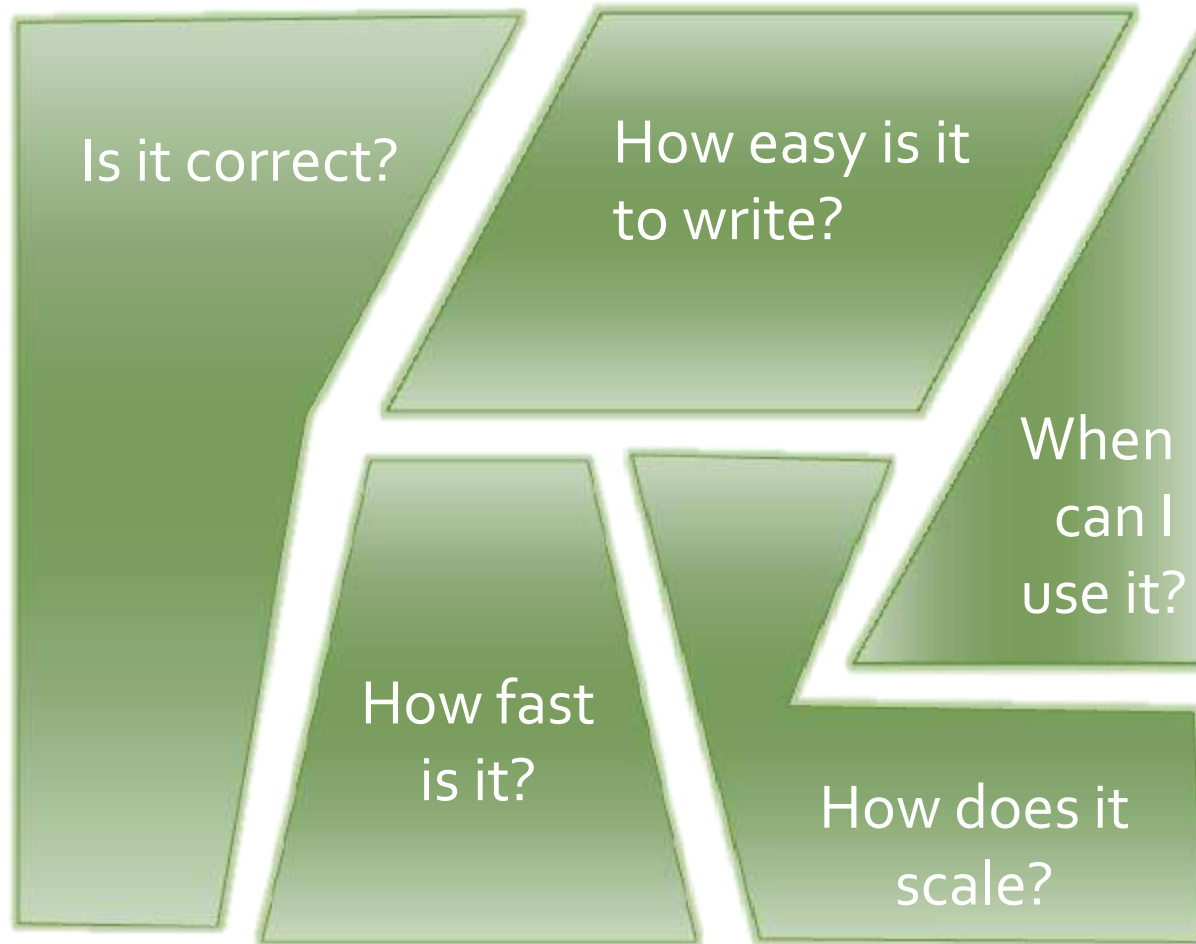
# Design points I am sceptical about

Speculative lock elision on general-purpose s/w

“atomic” blocks over normal data in a high-level language  
(C#/Java)

(prove me wrong, I would like either of these to work!)

# What do we care about?





[www.research.microsoft.com](http://www.research.microsoft.com)

©2011 Microsoft Corporation. All rights reserved.

This material is provided for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY. Microsoft is a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries.