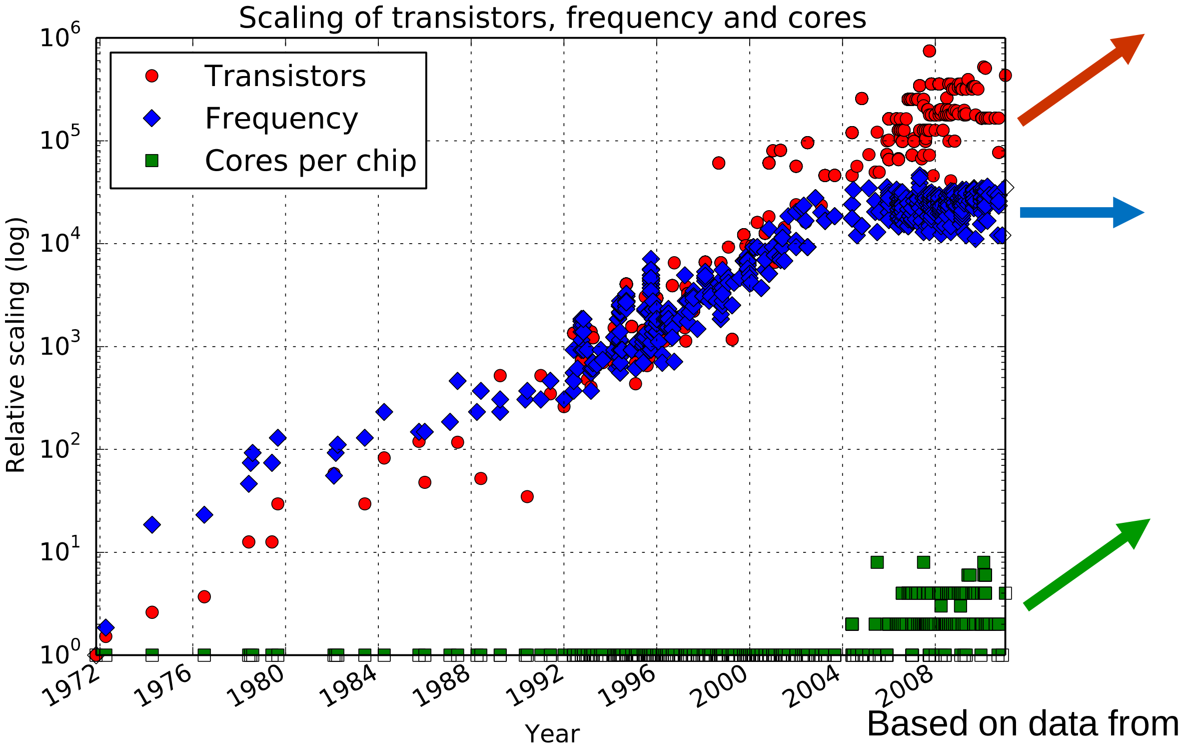
CHALMERS
UNIVERSITY OF TECHNOLOGY

Distributed Computing and Systems
Computer Science and Engineering Department

Lock-free Concurrent Data Structures

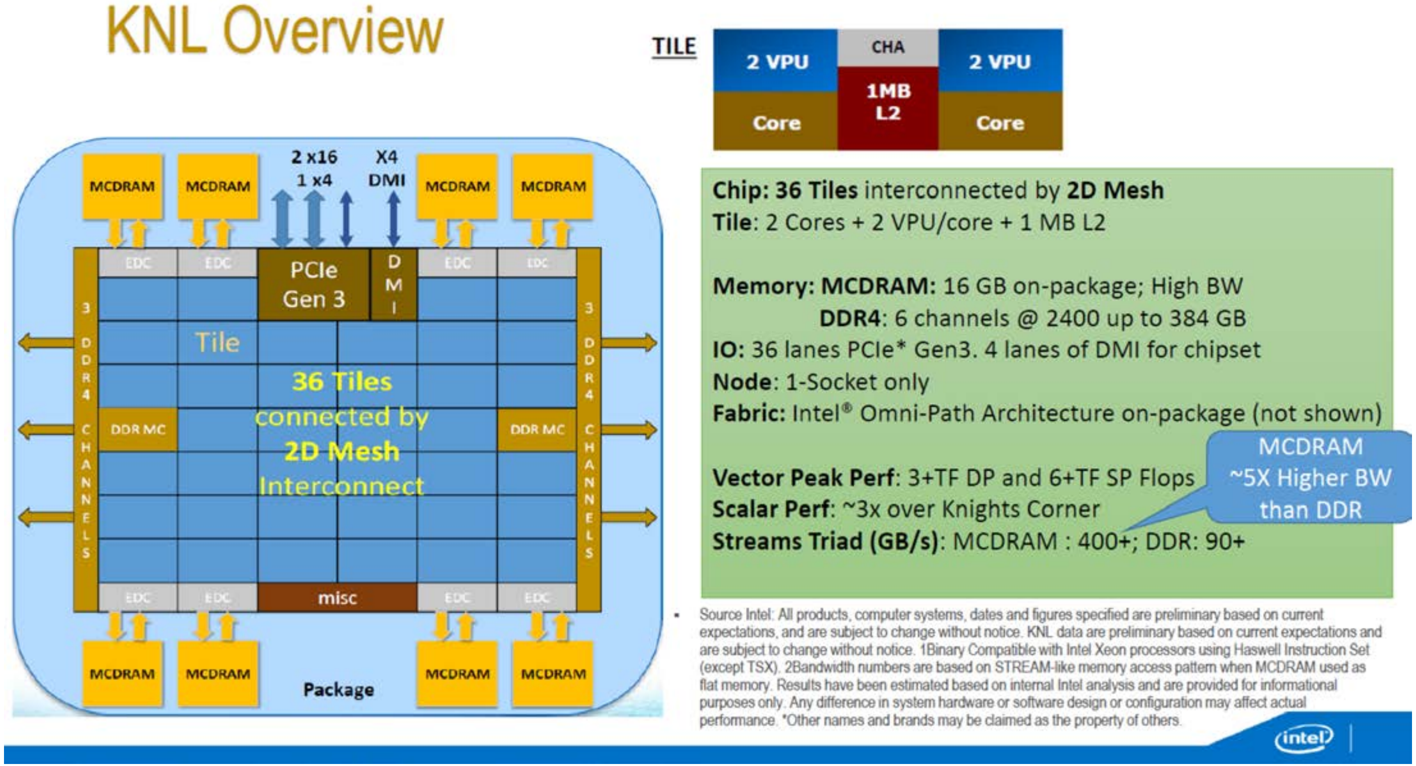
Philippas Tsigas
UPMARC Summer School 2018

Why Lock-free Concurrent Data Structures?



Based on data from the CPU DB
[cpudb.stanford.edu]

Multi-cores are here to stay!



KNL Overview

Chip: 36 Tiles interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384 GB
IO: 36 lanes PCIe* Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Intel® Omni-Path Architecture on-package (not shown)

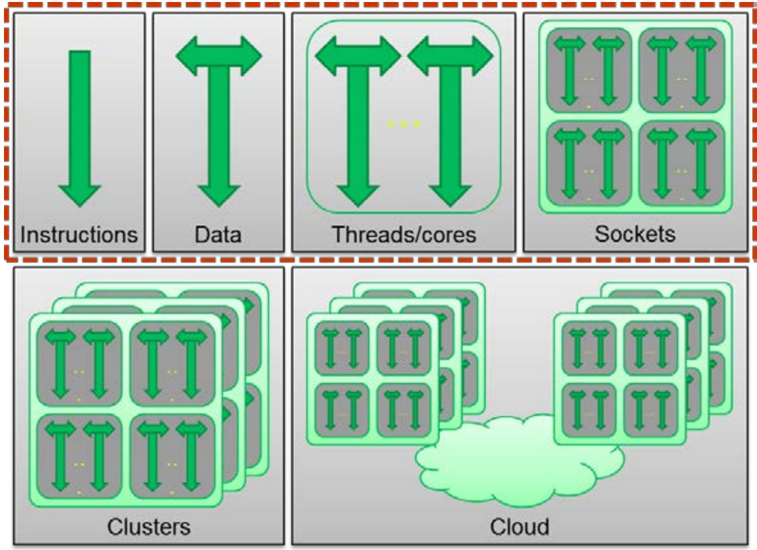
Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

MCDRAM ~5X Higher BW than DDR

- Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance. *Other names and brands may be claimed as the property of others.

Xeon Phi
7290F
72 cores

What kind of parallelism?



- Processes / threads
    - Each executes a sequence of instructions
    - Asynchronous
- Shared memory
    - Processes can read/write single memory words atomically
    - Synchronization primitives/instructions
        - Built into CPU and memory system
        - Atomic read-modify-write (i.e. a critical section of one instruction)
            - Compare-and-Swap
            - Load-Linked / Store-Conditional

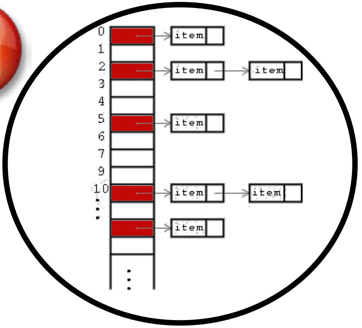
Parallel programming is rarely a recreational activity.

- Hard to make **correct** *and* **efficient**
  - We need to exploit parallelism
  - Need to identify *and* manage concurrency
- The human mind tends to be sequential
  - Concurrent specifications
  - Non-deterministic executions
- What about races? deadlocks? livelocks? starvation? fairness?
  - Need synchronization (correctness)…

  … but not too much (performance)

Why Lock-free Concurrent Data Structures?

- Locks are great to ensure correctness
  - Going back to sequential reasoning (coarse grained ones)



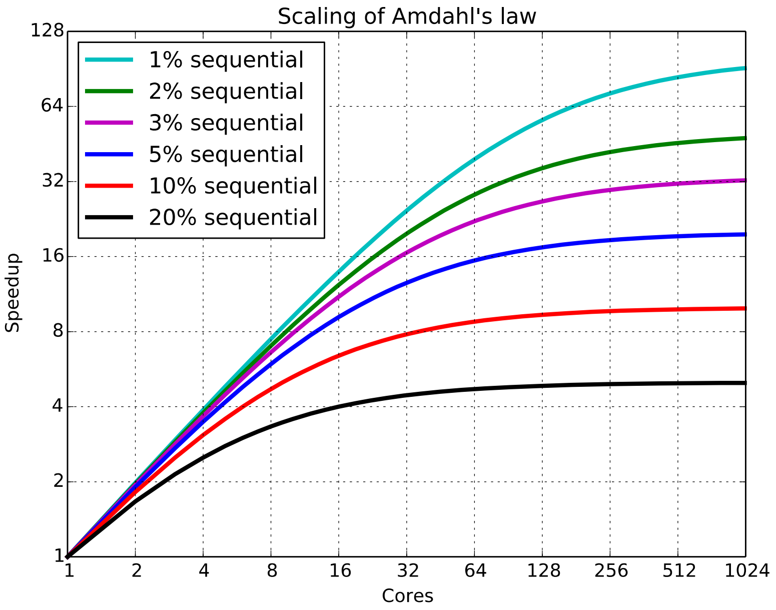
- Locks bad for performance (especially coarse grained ones)

1. Sequential computations use single core
   - More locks ⇒ less concurrency
2. Concurrent systems are “asynchronous”
   - Thread preempted while holding lock ⇒ no progress (any grained lock)
   - Deadlocks (any grained lock)

How important?

- Speedup is function of “parallel” (*p*) and “sequential” (*1-p*) fractions of program

$$Speedup = \frac{1}{1 - p + \frac{p}{n}}$$

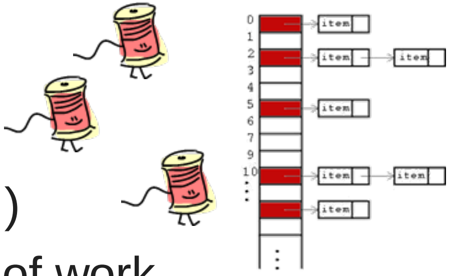


Why Lock-free Concurrent Data Structures?

Algorithms + Data Structures = Programs

[Wirth78]

Why Lock-free Concurrent Data Structures?



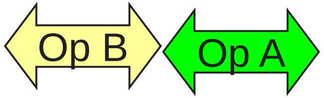
- Used directly by applications (e.g., in C/C++, Java, C#, …)
- Used in the language runtime system (e.g., management of work, implementations of message passing, …)
- Used in traditional operating systems (e.g., synchronization between top/bottom-half code)
- Used in Stream Processing (Engines, Operators, …)

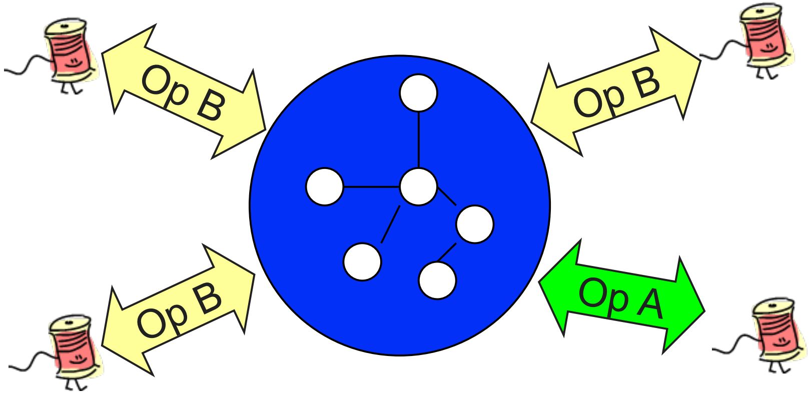
Data structures come closer to the eyes of application user:

…

Lock-free data structures

- Objects in shared memory
  - Supports some set of operations (ADT)
  - Supports concurrent access by many processes/threads
  - Cannot block operations





Liveness - Progress Guarantees

- **Non-blocking implementations**
  - **Wait-free implementation of a CDS** [Lamport, 1977]
    - Every operation finishes in a finite number of its own steps.
  - **Lock-free (≠ FREE of LOCKS) implementation of a CDS** [Lamport, 1977]
    - At least one operation in a set of concurrent operation always makes progress, finishes in a finite number of its own steps.
  - **Obstruction-free implementation** [Herlihy et. al. 2003]
    - Any operation that executes in isolation is guaranteed to make progress and finish in a finite number of its own steps.

- **Definitions can be defined also on operation level.**

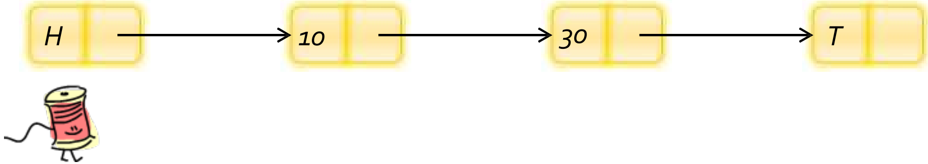
Liveness cnt.

- every garbage node is eventually collected

Let us add concurrent acesses: What behavior do we want?

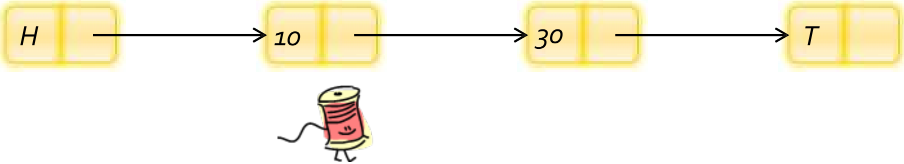
Searching a sorted list, sequencial case:

- *find(20):*



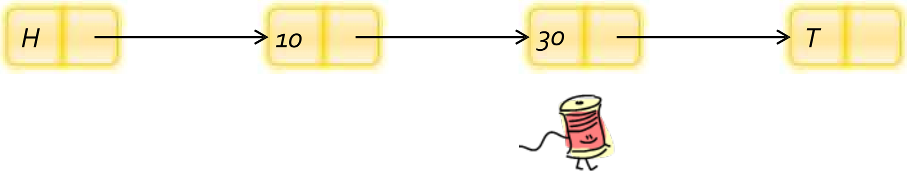
Searching a sorted list, sequencial case:

- *find(20):*



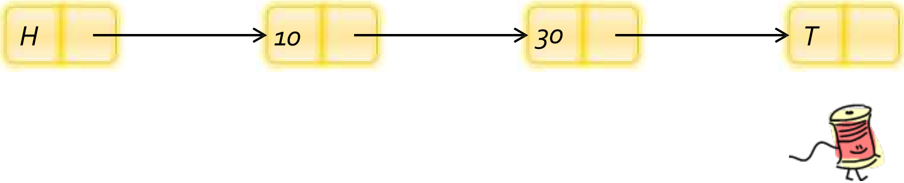
Searching a sorted list, sequencial case:

- *find(20):*



Searching a sorted list, sequencial case:

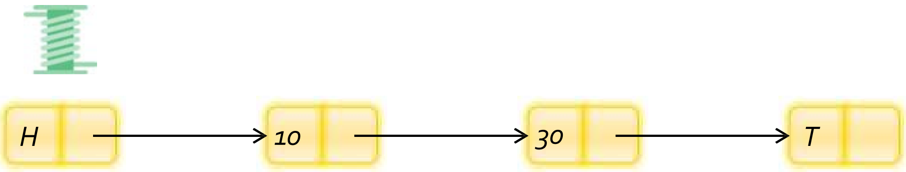
- *find(20):*



find(20) -> false

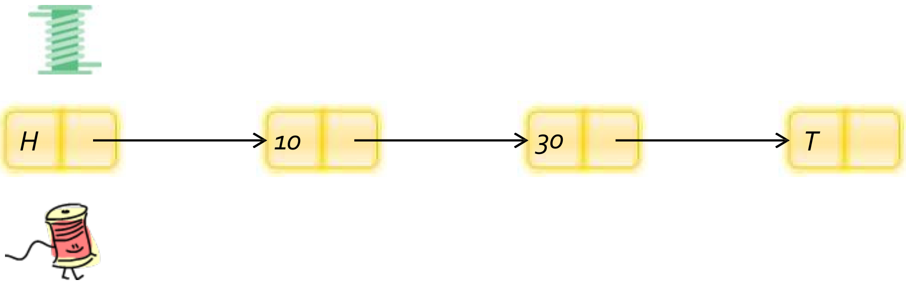
Inserting an item with CAS + Find

- *insert(20):*



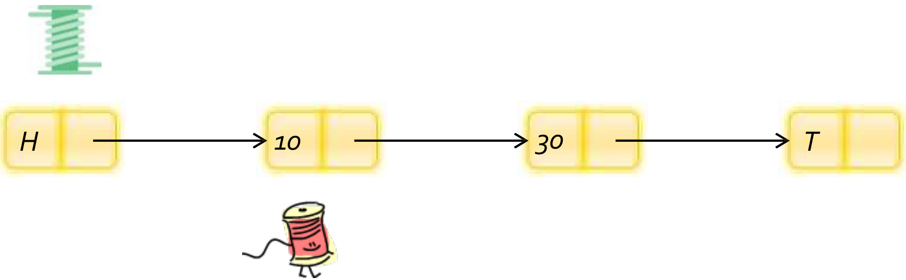
insert is invoked first

- *insert(20):*



- *find(20):*

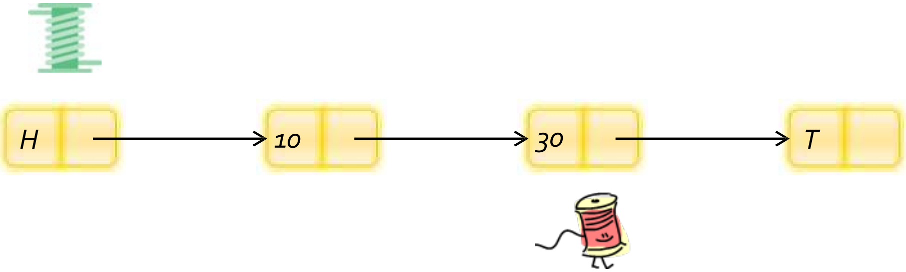
- *insert(20):*



- *find(20):*

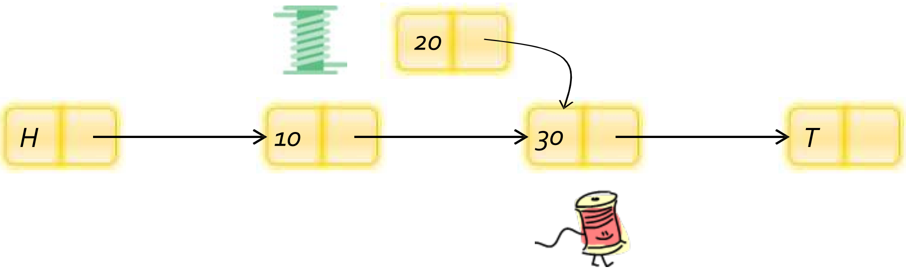
Concurrent case : Behavior

- *insert(20):*



- *find(20):*

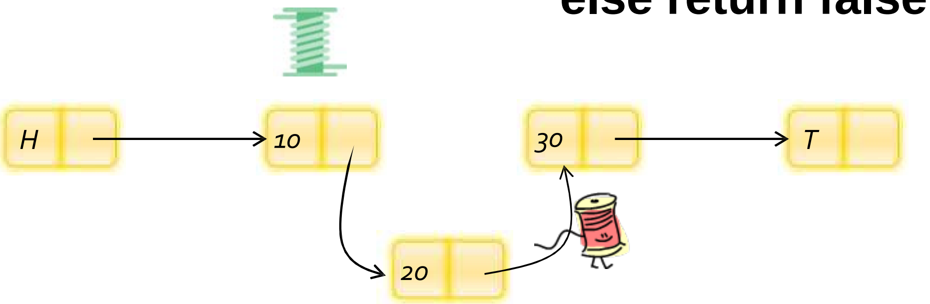
- *insert(20):*



- *find(20):*

```
CAS(p:pointer to word, old:word,
new:word):boolean
atomic do
    if *p = old then
        *p := new;
        return true;
    else return false;
```

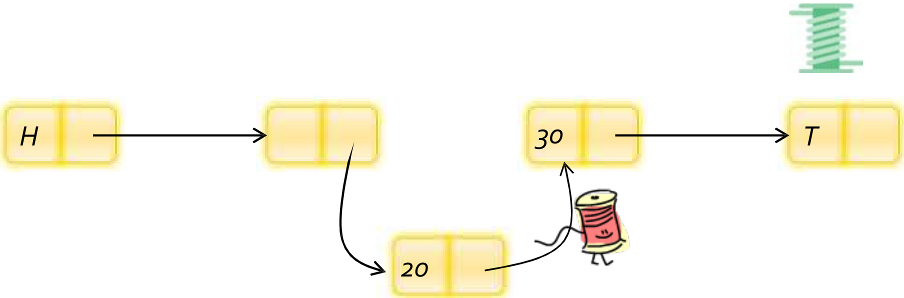
- *insert(20):*



insert(20) -> true

Insert returns first

- *insert(20):*

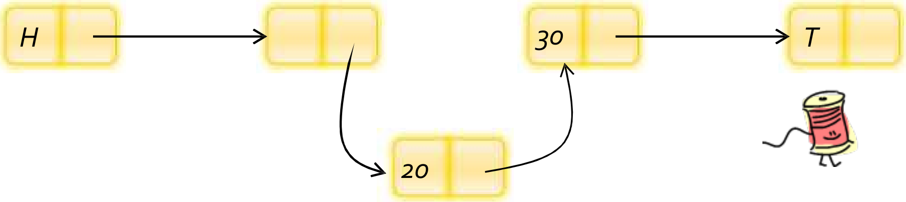


- *find(20):*


insert(20) -> true

find(20) returns after insert(20) with false: Is this a problem?

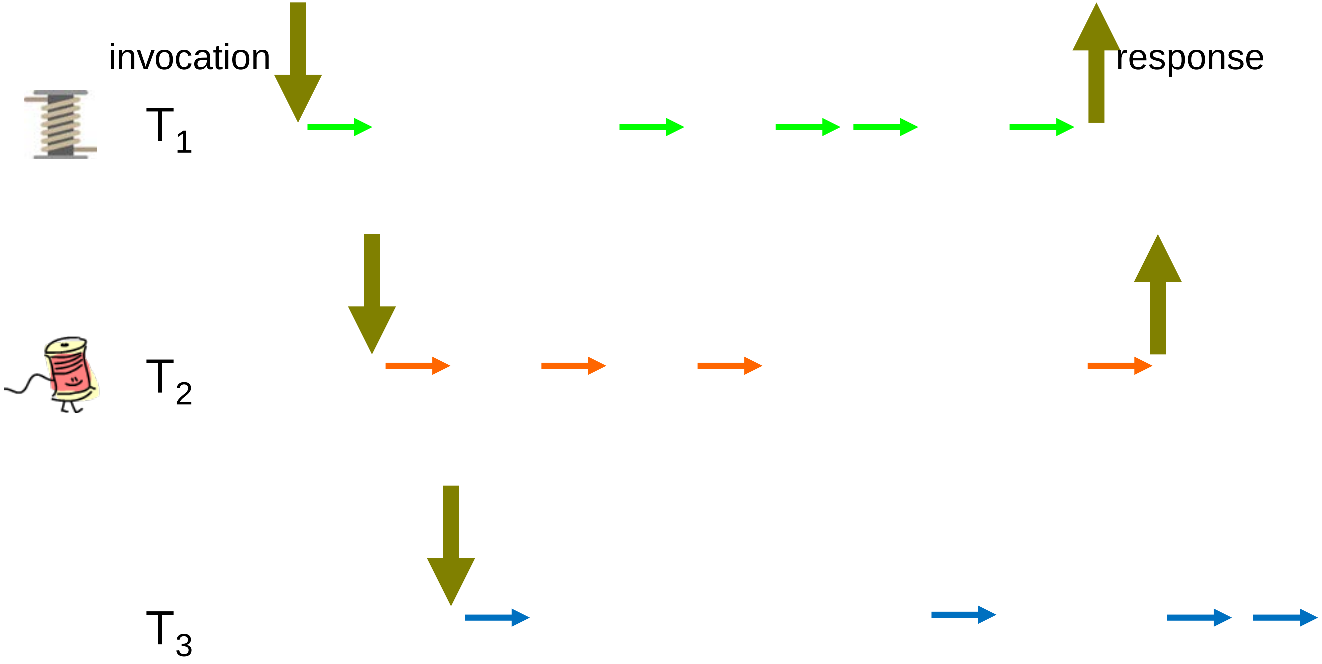
- *insert(20):*



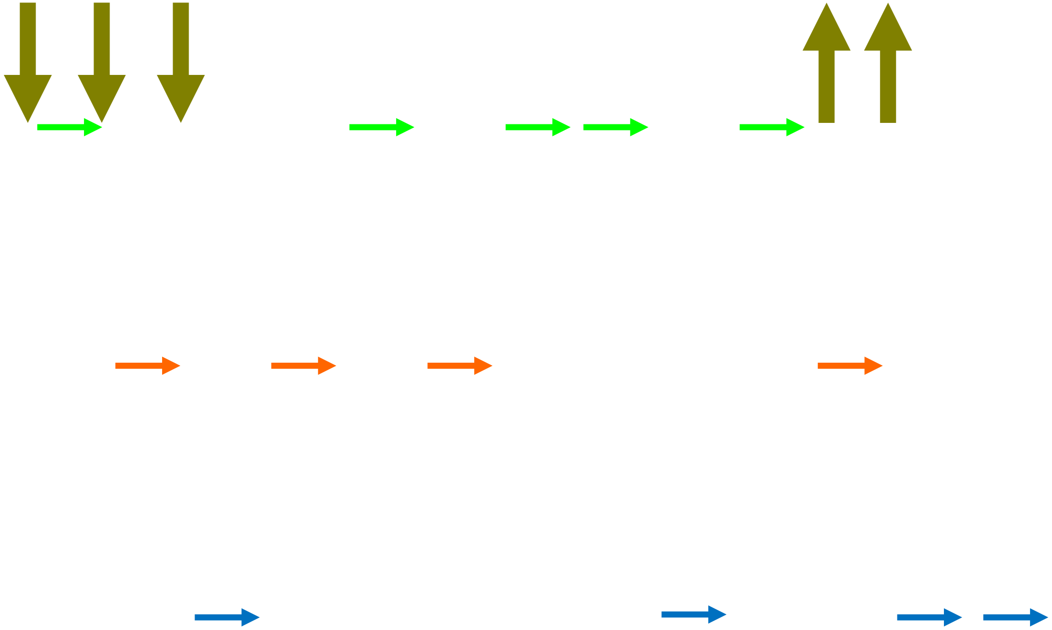
insert(20) -> true

find(20) -> false

Concurrent execution of operations



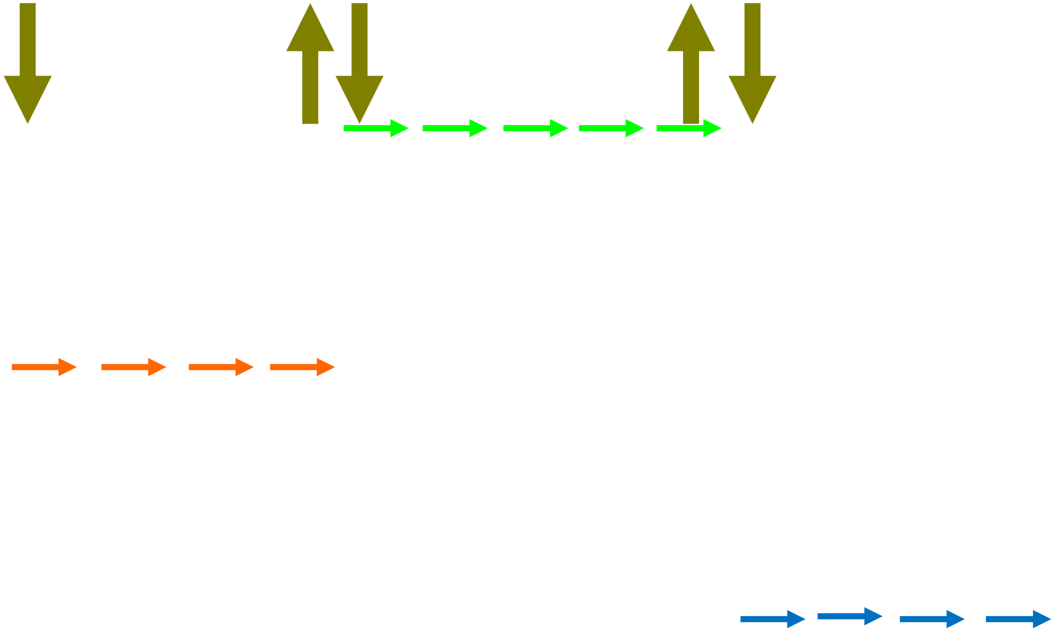
Interleaving operations



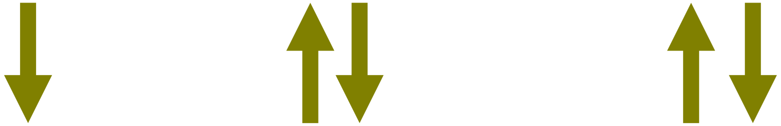
Concurrent execution

(External) behavior

Behavior that we observed with find&insert is equivalent to:



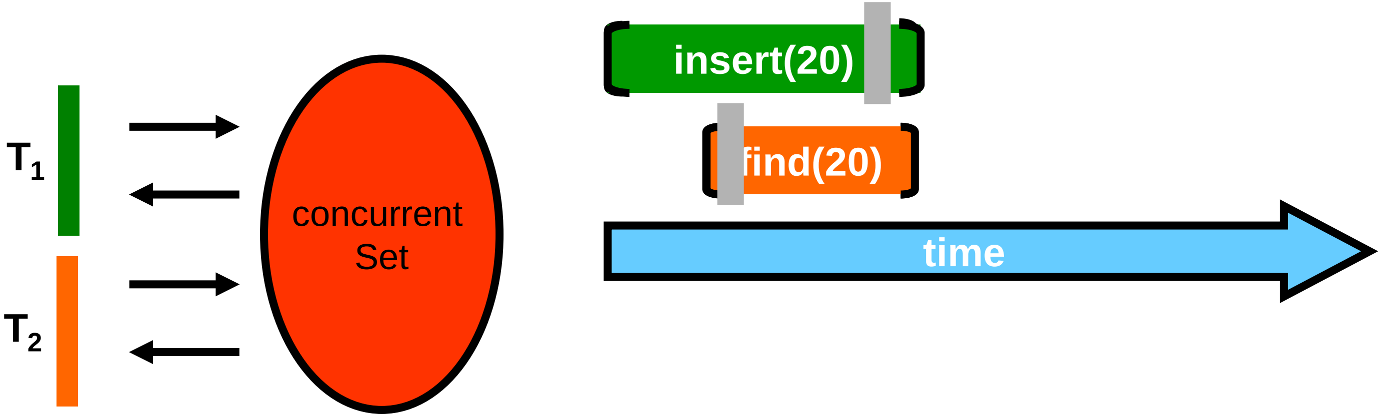
Sequential execution



- Sequential behavior: invocations & response alternate and match (on process & object)
- Sequential specification: All the legal sequential behaviors, satisfying the semantics of the ADT
  - E.g., for a stack: pop returns the last item pushed

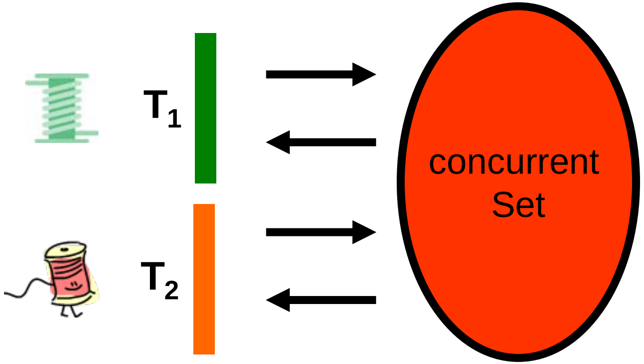
Safety: Linearizability

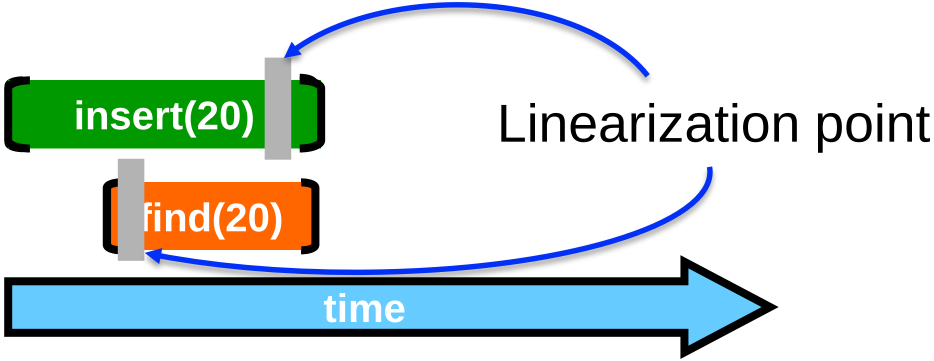
- **Sequential specification** defines legal sequential executions
- Concurrent operations allowed to be **interleaved**
- For every concurrent execution there is a sequential execution that
  - Contains the same operations
  - Is legal (obeys the sequential specification)
  - Preserves the real-time order of all operations



Safety: Linearizability

- For each operation there must be one single time instant during its duration where the operation appears to take effect.
- The observed effects should be consistent with a sequential execution of the operations in that order.





A recurring technique for finding linerization points

- For updates:
  - Perform an essential step of an operation by a single atomic instruction (E.g. CAS to insert an item into a list)
    - Not always the same instruction
    - Not always an instruction an instruction of the "update code"
- For reads:
  - Identify a point during the operation's execution when the result is valid
    - Not always a specific instruction
    - Not always the same instruction
    - Not always an instruction an instruction of the "read code"

Safety Cnt.

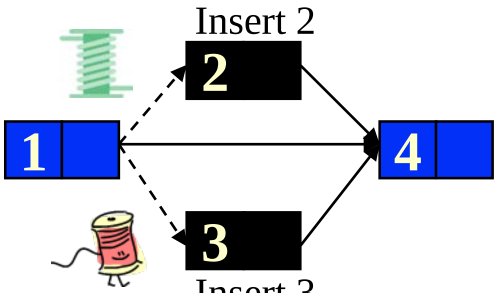
An accessible node is never freed.

Basic recurent algorithmic issues

Back to the algorithmic design of the List: Concurrent Inserts

- Example: Insert operation
  - which of 2 or 3 gets inserted?
- Compare-And-Swap
  atomic primitive takes care of this:

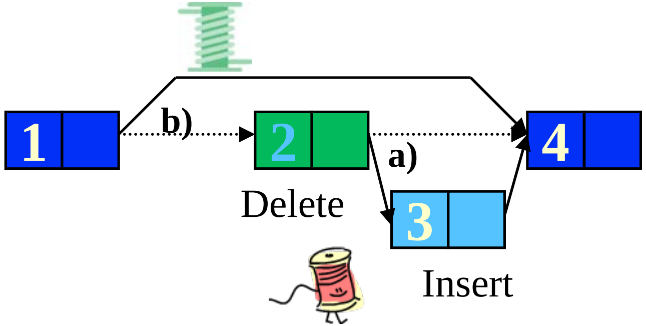
```
CAS(p:pointer to word, old:word, new:word):boolean
atomic do
  if *p = old then
          *p := new;
          return true;
  else return false;
```



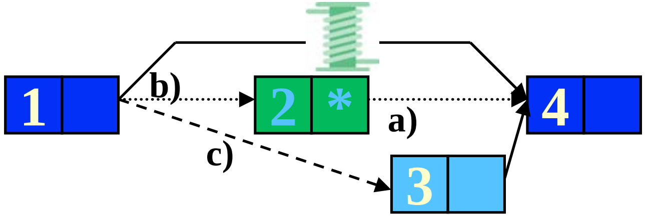
What about concurrent inserts and deletes

- Problem:

  - both nodes are deleted!

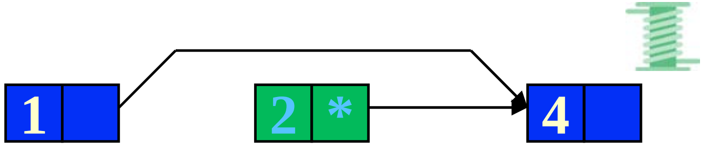


- Solution: Use bit 0 of pointer to mark deletion status and 2 CAS



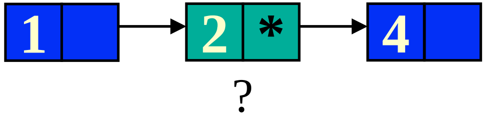
When is it safe to reclaim nodes: Explicit memory managment

- Is it safe to reclaim node 2?

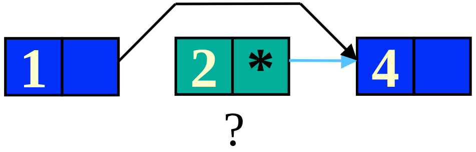


“Help me help you!” schemas

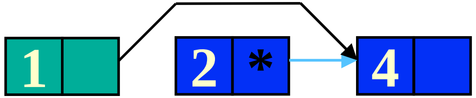
- Threads need to traverse safely



or

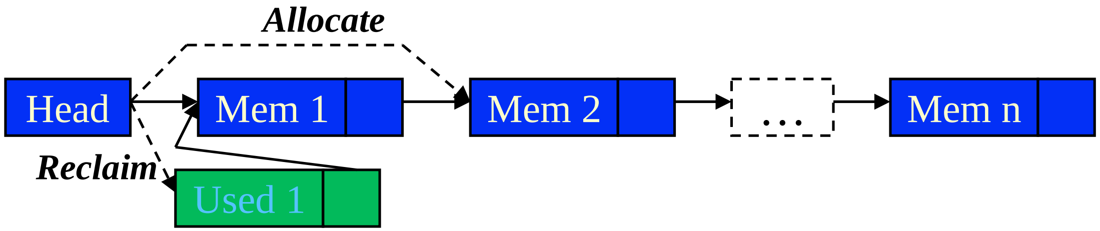


- Need to remove marked-to-be-deleted nodes while traversing – Help!
- Finds previous node, finish deletion and continues traversing from previous node



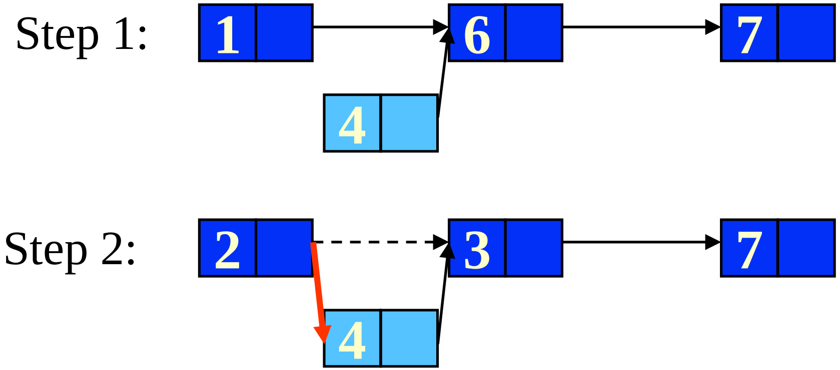
Dynamic memory management

- Problem: System memory allocation functionality is blocking!
- Solution (lock-free), IBM freelists:
  - Pre-allocate a number of nodes, link them into a dynamic stack structure, and allocate/reclaim using CAS



The ABA problem

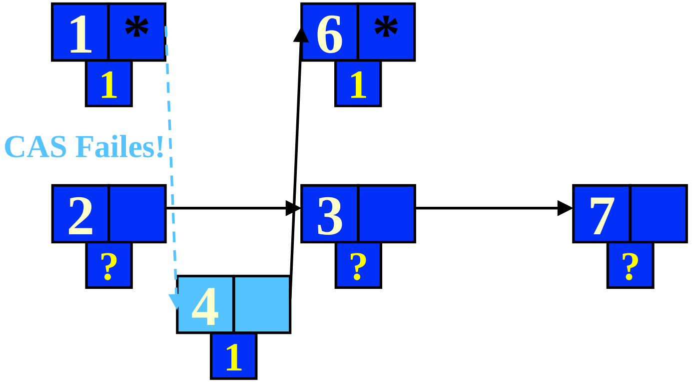
- Problem: Because of concurrency (pre-emption in particular), same pointer value does not always mean same node (i.e. CAS succeeds)!!!



Reference counting

- Solution: (Valois et al) Add reference counting to each node, in order to prevent nodes that are of interest to some thread to be reclaimed until all threads have left the node

New Step 2:



Some techniques for increasing performance I

- For pre-emptive systems, helping is necessary for efficiency and lock-freeness
- For highly concurrent systems, overlapping CAS operations (caused by helping and others) on the same node can cause heavy contention (Lecture tomorrow)
- Solutions: Manage contention by actively managing back-off and memory management (Aras et al, lecture tomorrow)

Some techniques for increasing performance II

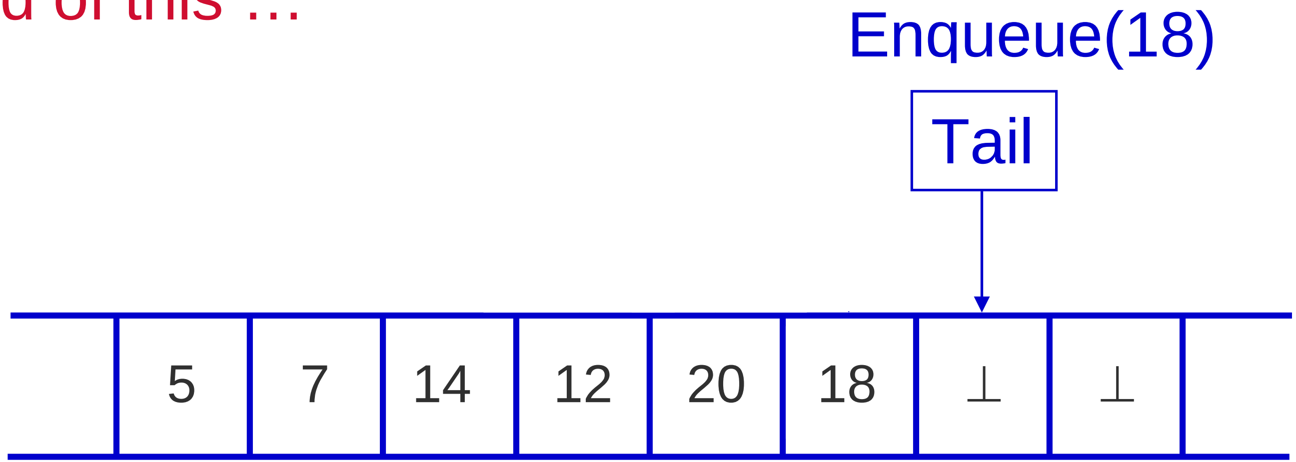
- CAS, FAA, SWAP are expensive and there performance degrades as contention increases (see CAS expansion tomorrow).

  [Lazy 1] Reduce the number of CAS's by allowing *shared pointers* to lag behind *real* pointers. (Tsigas, Zhang)

  [Lazy2] Reduce the number of read/update global shared variables by allowing *local pointers* to lag behind *shared pointers* that lag behind *real* pointers. (Gidenstam et al.)

- Mind the cache:
  - Array/block based designs

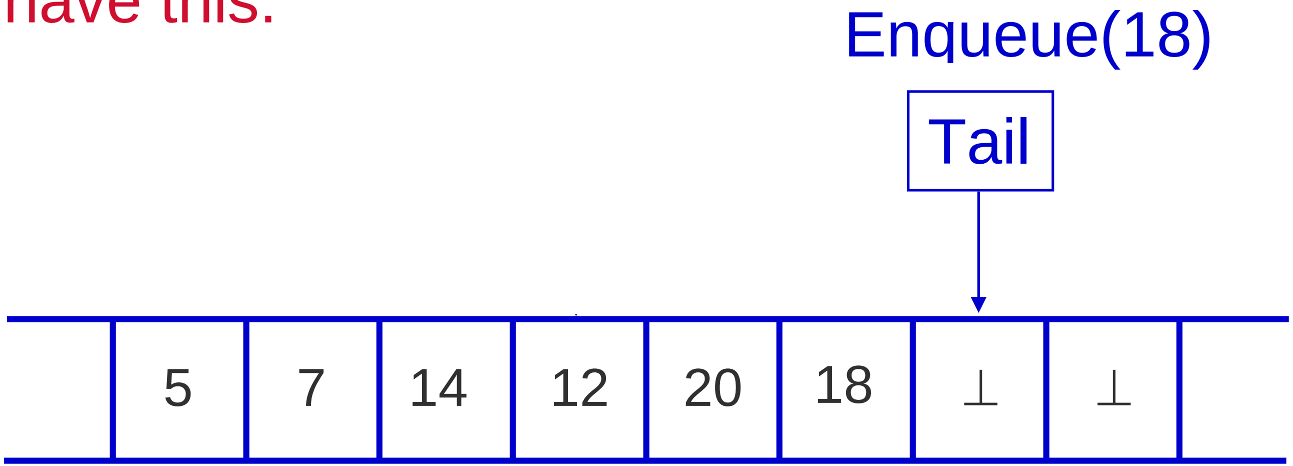
Instead of this …


Enqueue(18)

Tail

5 | 7 | 14 | 12 | 20 | 18 | ⊥ | ⊥

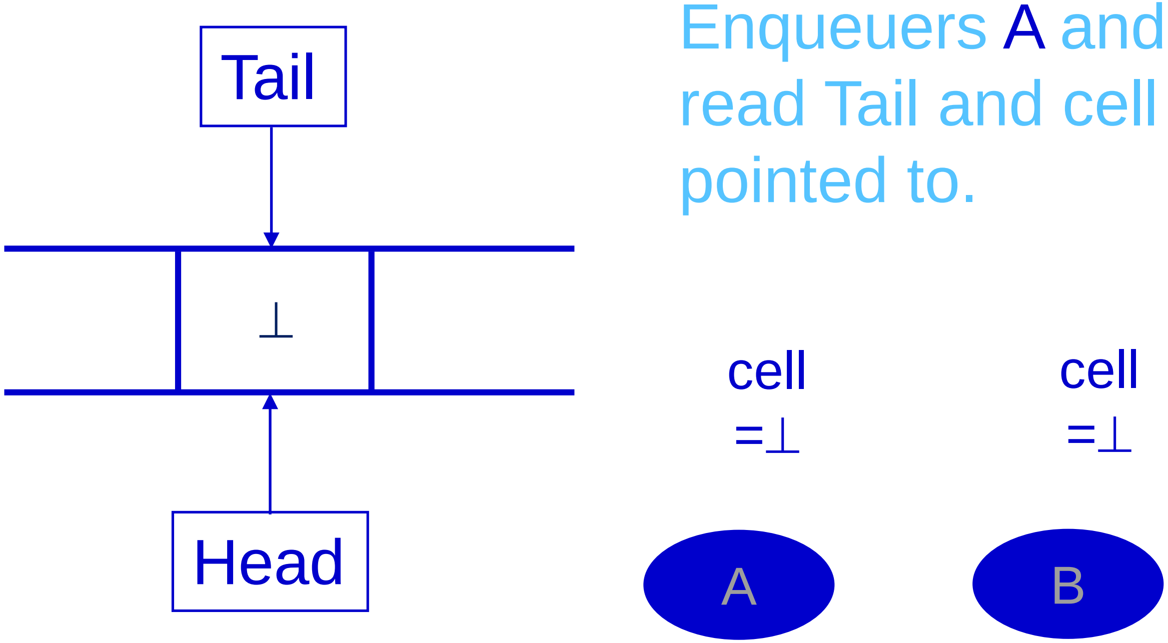
… we have this:



Every mth Enqueue updates Tail with CAS.

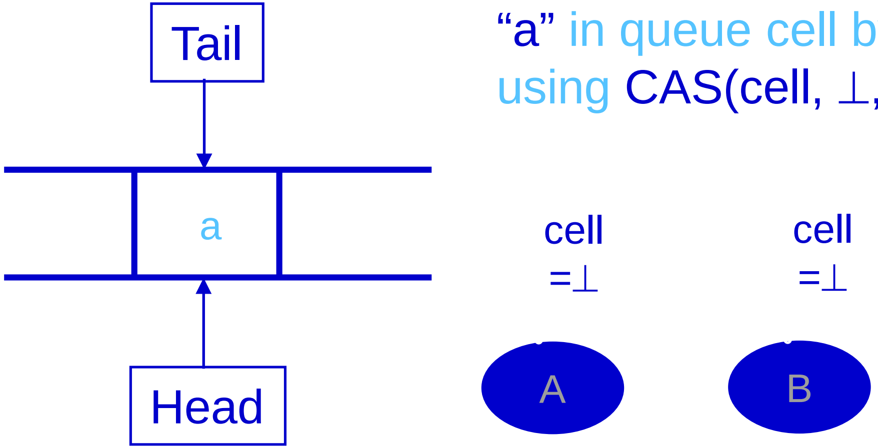
Assume m > 1.

Enqueuers A and B read Tail and cell pointed to.



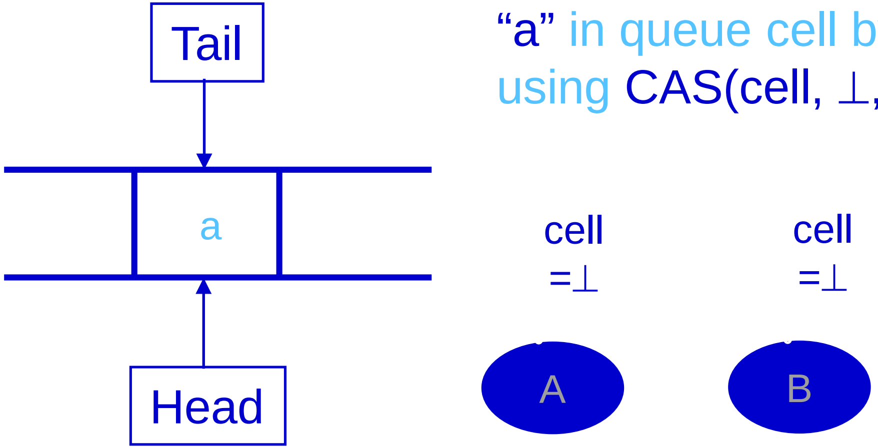
“(Almost) Empty”

Assume m > 1.



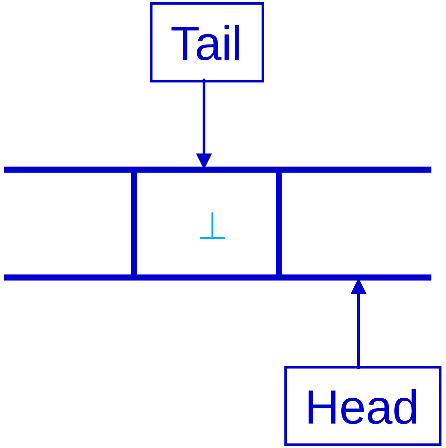
A stores its value “a” in queue cell by using CAS(cell, ⊥, a).

Assume m > 1.

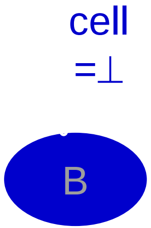


A stores its value "a" in queue cell by using CAS(cell, ⊥, a).

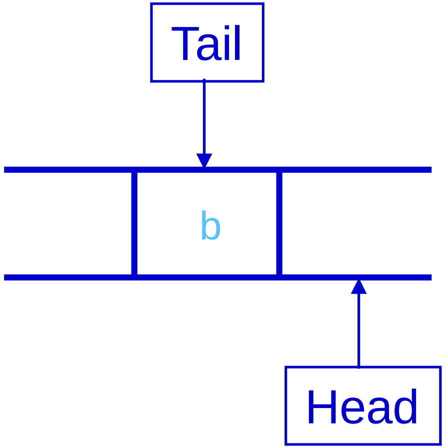
Assume m > 1.



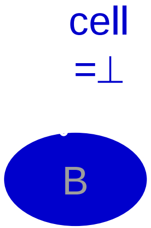
Dequeuer C dequeues “a” and updates Head.



Assume m > 1.

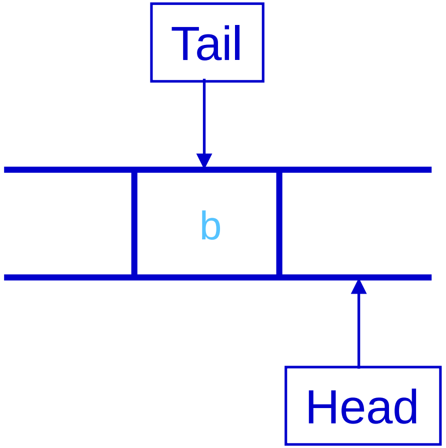


Enqueuer B finally updates queue cell by performing CAS(cell, ⊥, b).

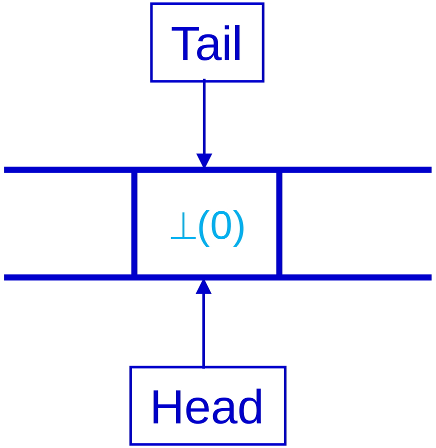


Assume m > 1.

“b” is lost!



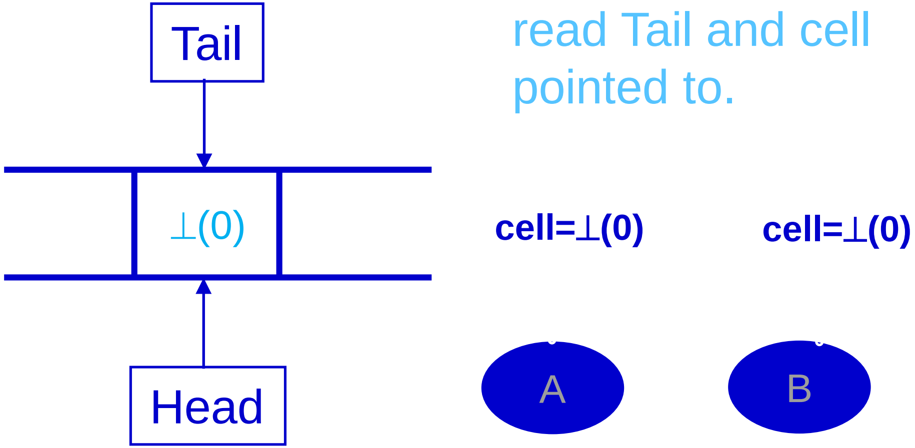
Assume m > 1.



By adding ***one bit*** to each word, this problem can be solved.

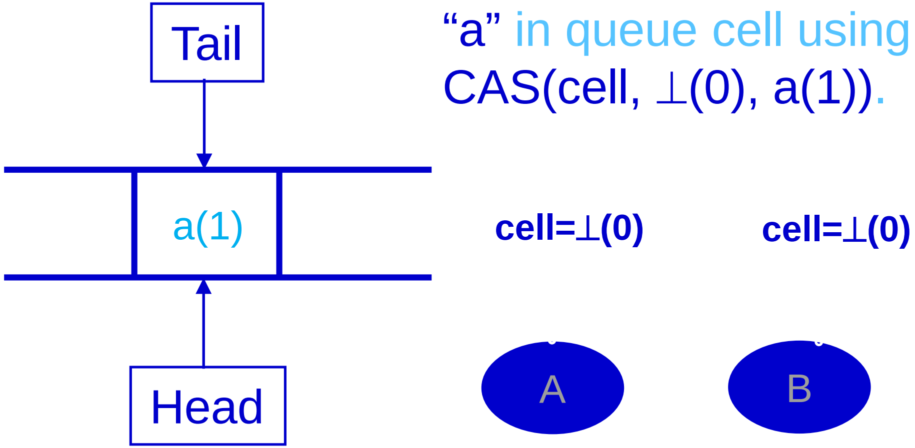
Assume m > 1.

Enqueuers A and B read Tail and cell pointed to.



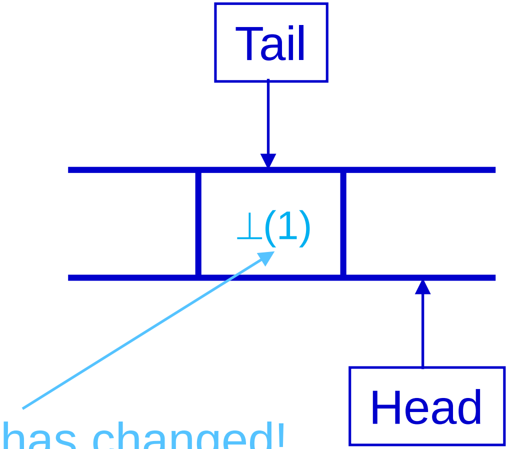
Assume m > 1.

A stores its value "a" in queue cell using CAS(cell, ⊥(0), a(1)).

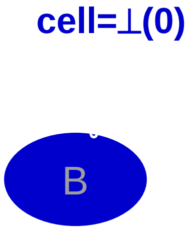


Assume m > 1.

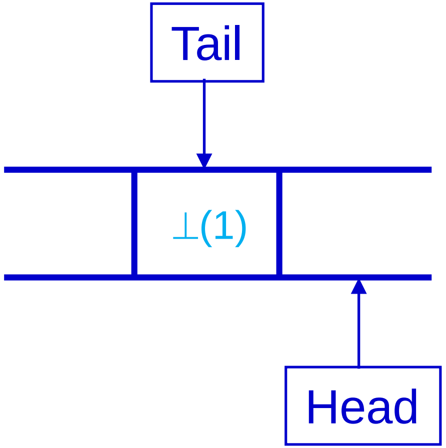
Dequeuer C dequeues “a” and updates Head.



Note: The bit has changed!



Assume m > 1.

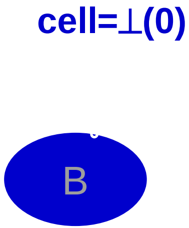

Tail

$\perp(1)$

Head

Enqueuer B *fails* to update queue cell because

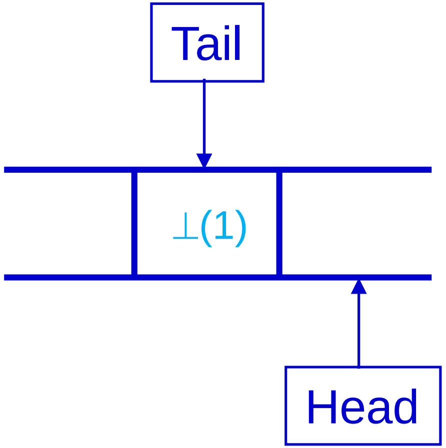
$\perp(0) \neq \perp(1)$!


cell=$\perp(0)$

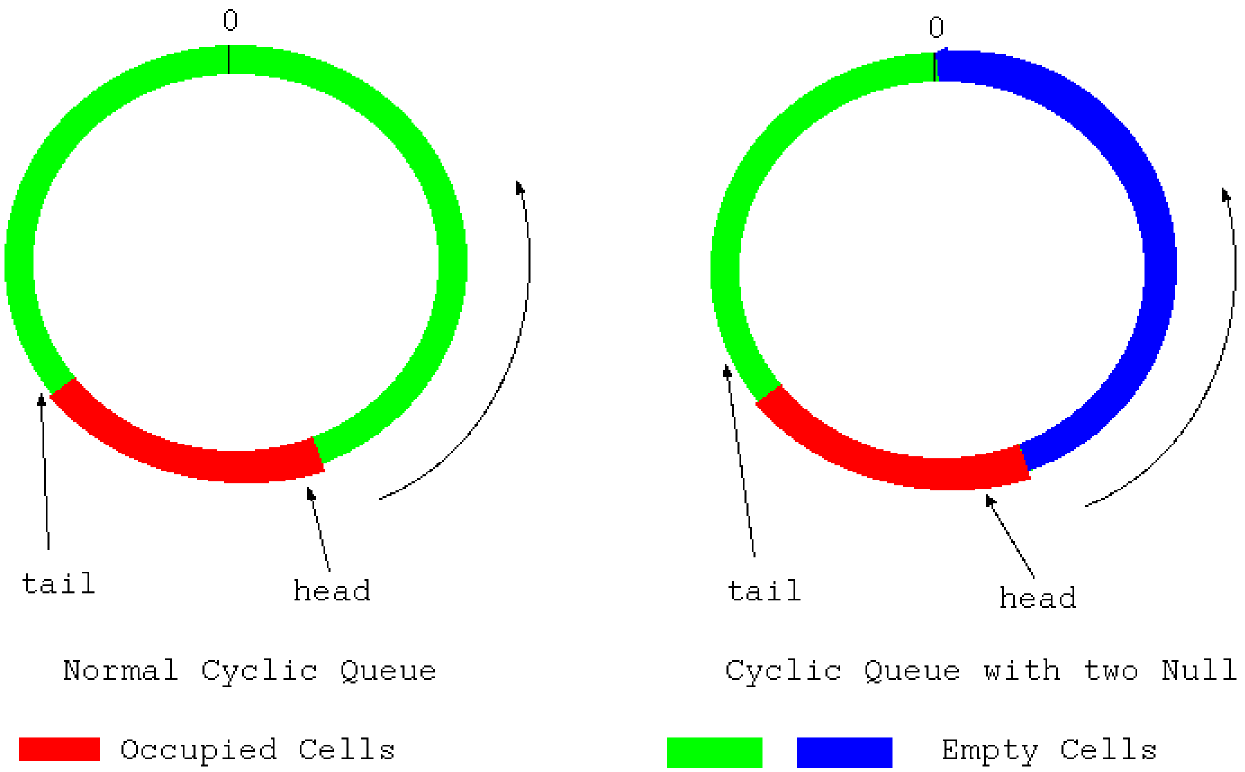
B

Assume m > 1.

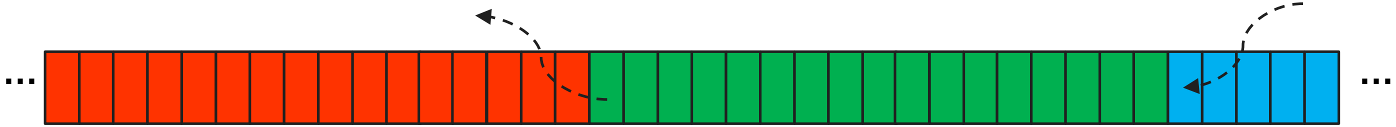
B must *retry* its operation.



Queue using cyclical array [Tsigas, Zhang]

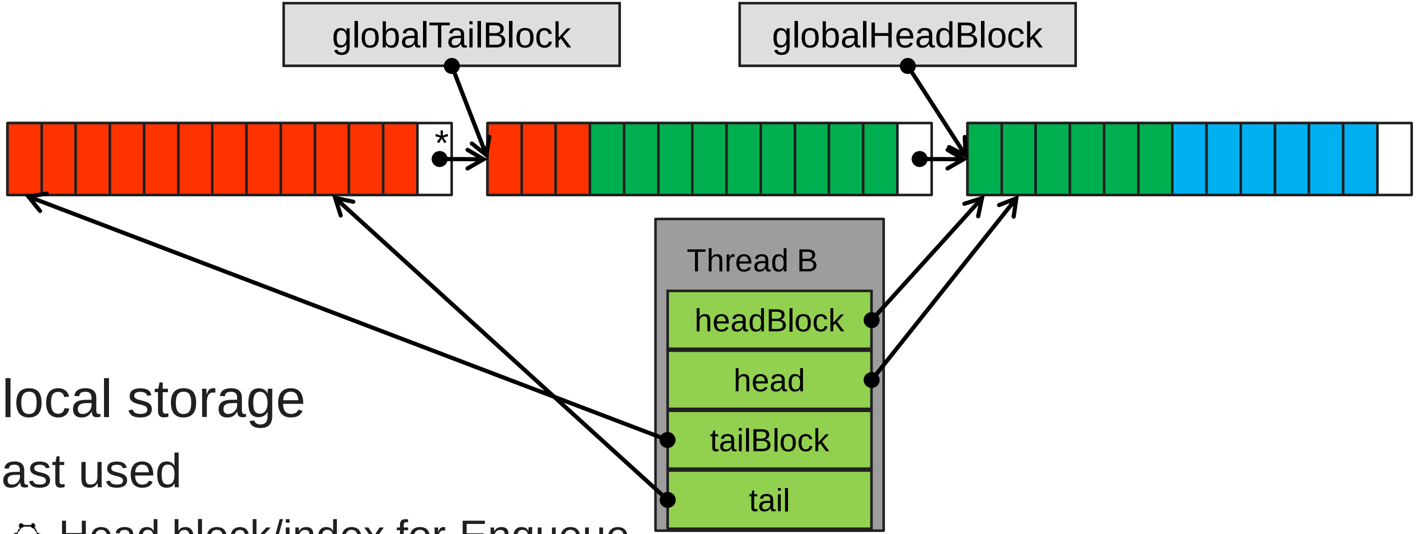


Making it dynamic (queue) [Gidenstam et al.]



- Basic idea:
  - Cut and unroll the circular array queue
  - Primary synchronization on the elements
    - Compare-And-Swap
      (NULL1 -> Value -> NULL2 avoids the ABA problem)
  - Head and tail both move to the right
    - Need an “infinite” array of elements

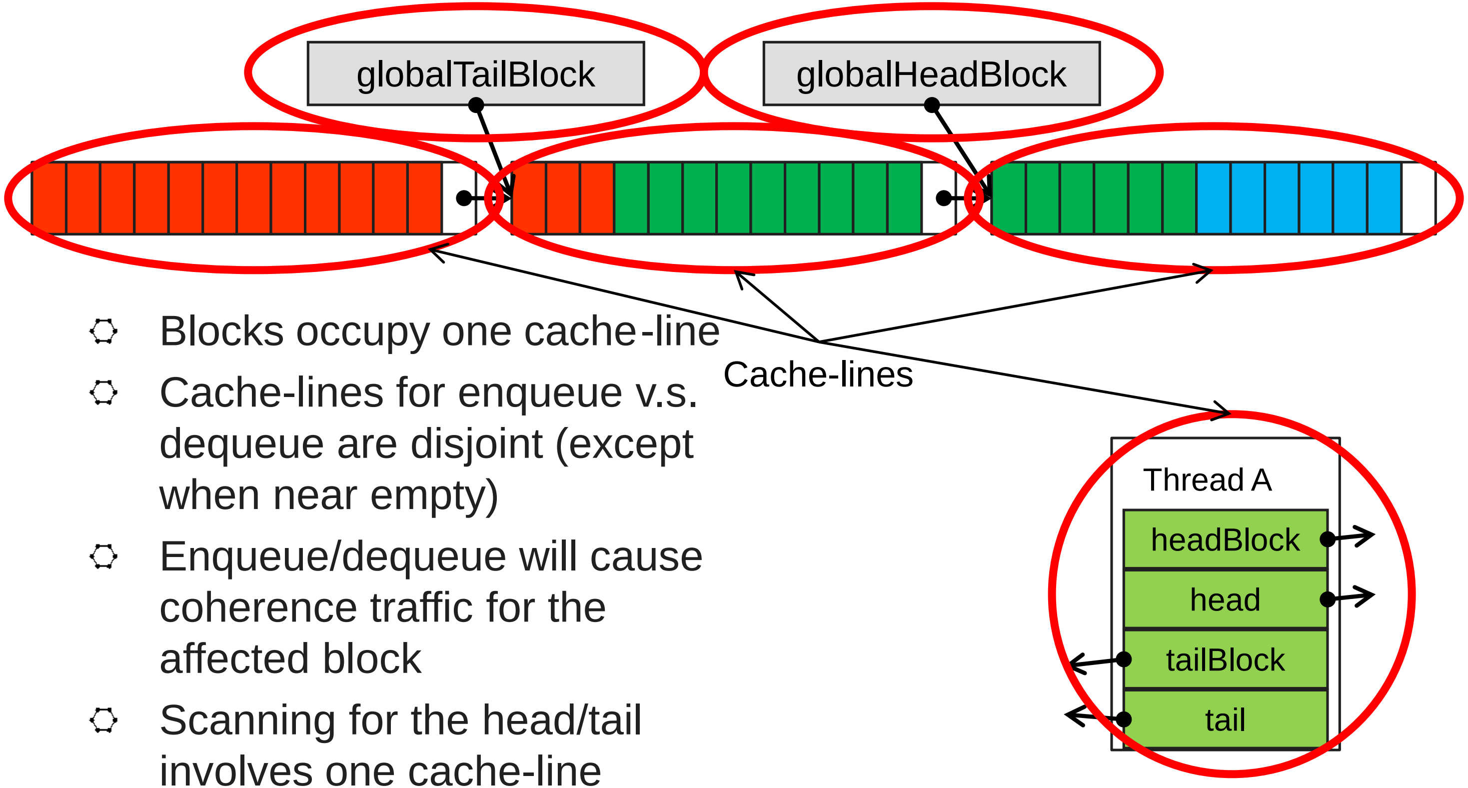
Lazy updating local, shared, data structure pointers



Thread local storage

- Last used
  - Head block/index for Enqueue
  - Tail block/index for Dequeue
- Reduces need to read/update global shared variables

Minding the cache



- Blocks occupy one cache-line
- Cache-lines for enqueue v.s. dequeue are disjoint (except when near empty)
- Enqueue/dequeue will cause coherence traffic for the affected block
- Scanning for the head/tail involves one cache-line

References

- Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst.12, 3 (July 1990), 463-492.
- Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In Proceedings of the 15th International Conference on Distributed Computing (DISC '01), LNCS Springer-Verlag, pp. 300-314.
- Anders Gidenstam, Håkan Sundell, and Philippas Tsigas. 2010. Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency. In Proceedings of the 14th international conference on Principles of distributed systems (OPODIS'10), LNCS Springer-Verlag, pp. 302-317.
- Philippas Tsigas and Yi Zhang. 2001. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '01). ACM, pp. 134-143.
- John David Valois. 1996. Lock-Free Data Structures. Ph.D. Dissertation. Rensselaer Polytechnic Institute, Troy, NY, USA. UMI Order No. GAX95-44082.

- D Cederman, A Gidenstam, P Ha, H Sundell, M Papatriantafilou, P Tsigas. 2017. Lock-free concurrent data structures. In Programming Multicore and Many-core Computing Systems. Willey.
- M. Herlihy, N. Shavit. The Art of Multiprocessor Programming, Morgan Kaufmann.

Thank you! Questions?