Memory Management for Lock-Free Concurrent Data-structures

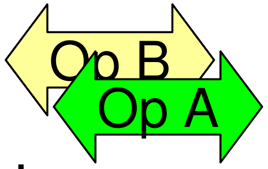
Anders Gidenstam
PostDoc,
AG1, Max-Planck-Institut für Informatik

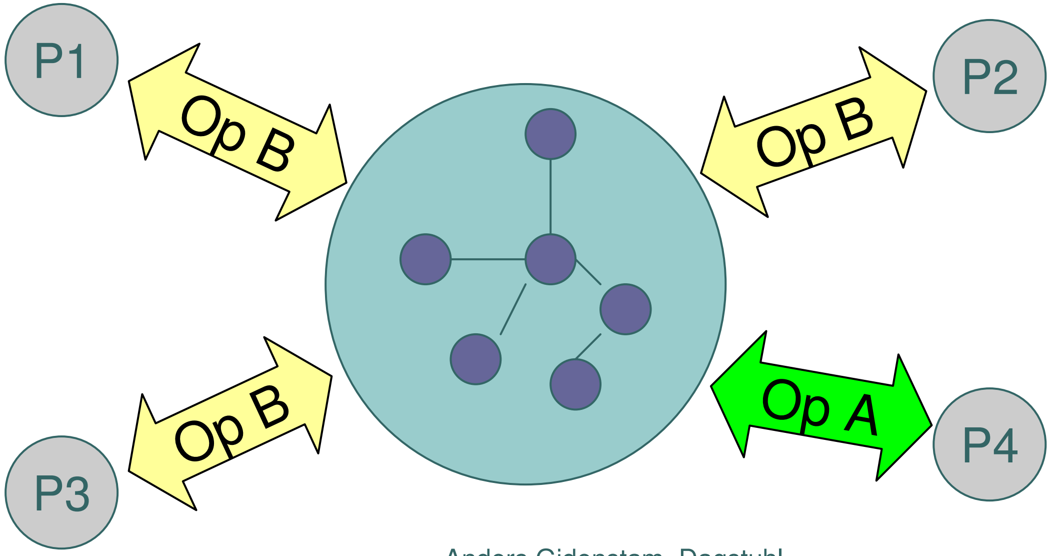
Outline

- Introduction to non-blocking algorithms
  - System model
  - Correctness criteria
  - Progress guarantees
  - Motivating example
- The lock-free memory reclamation problem
  - Solutions
  - LFMR [Gidenstam, Papatriantafilou, Sundell & Tsigas. I-SPAN 2005]
    - Idea
    - Properties
- What's out there: Some lock-free data-structures
- Current work

Concurrent Shared Memory Data-structures

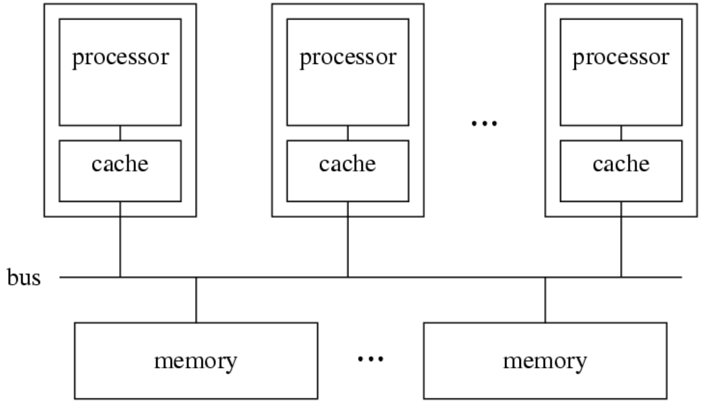
- Object in shared memory
  - Supports some set of operations (ADT)
  - Concurrent access by many processes/threads
  - Useful to e.g.
    - Exchange data between threads
    - Coordinate thread activities





System model

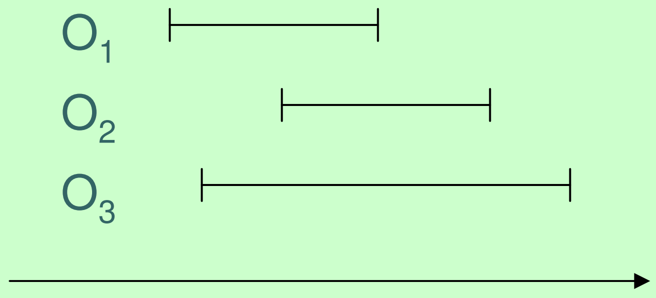
- Processes / threads
  - Asynchronous
  - Each executes a sequence of instructions



- Shared memory
  - Processes can read/write single memory words atomically
  - Hardware synchronization primitives/instructions
    - Compare-and-Swap(address, old, new)
      - Atomic read-modify-write (i.e. a critical section of one instruction)
    - Load-Linked(address) / Store-Conditional(address, new)

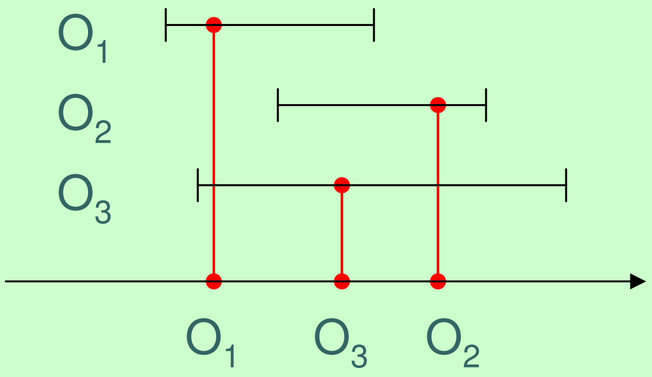
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation 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.



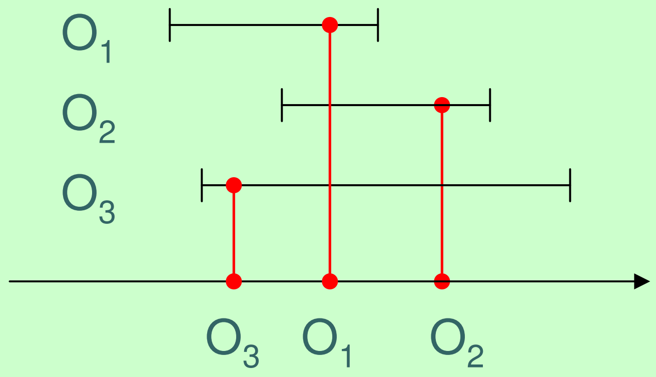
Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation 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.



Correctness of a concurrent object

- Desired semantics of a shared data object
  - Linearizability [Herlihy & Wing, 1990]
    - For each operation invocation 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.

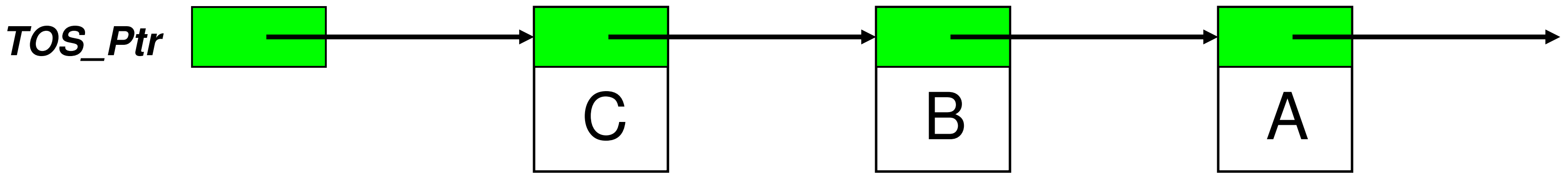


Progress Guarantees

- **Traditional solution: Mutual exclusion**
  - Semaphores, mutexes, spin-locks, disabling interrupts
    - Protects critical sections
  - Drawbacks: blocking, deadlock, priority-inversion, limits parallelism
- **Non-blocking synchronization**
  - Wait-free synchronization [Lamport, 1977]
    - Every operation finishes in a finite number of its own steps.
  - **Lock-free synchronization** [Lamport, 1977]
    - At least one operation in a set of concurrent operation always makes progress.
  - Obstruction-free synchronization [Herlihy et. al. 2003]
    - Any operation that eventually executes in isolation is guaranteed to make progress.

Example: Lock-free Stack

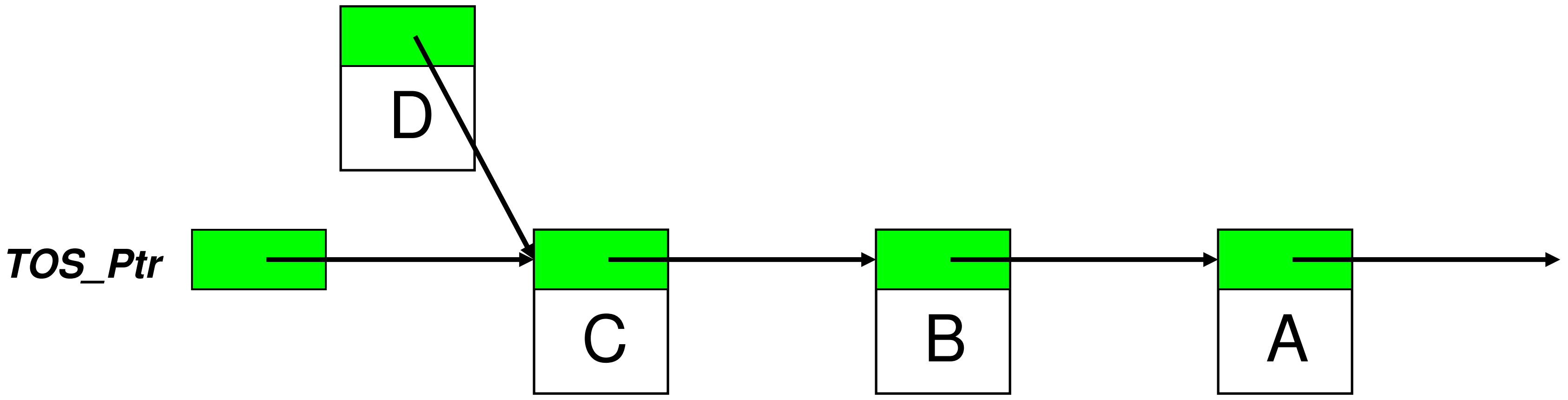
- Operations
  - Push(item)
  - Pop: item
- Linked list based algorithm [IBM 1983]



Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item
- Linked list based algorithm [IBM 1983]

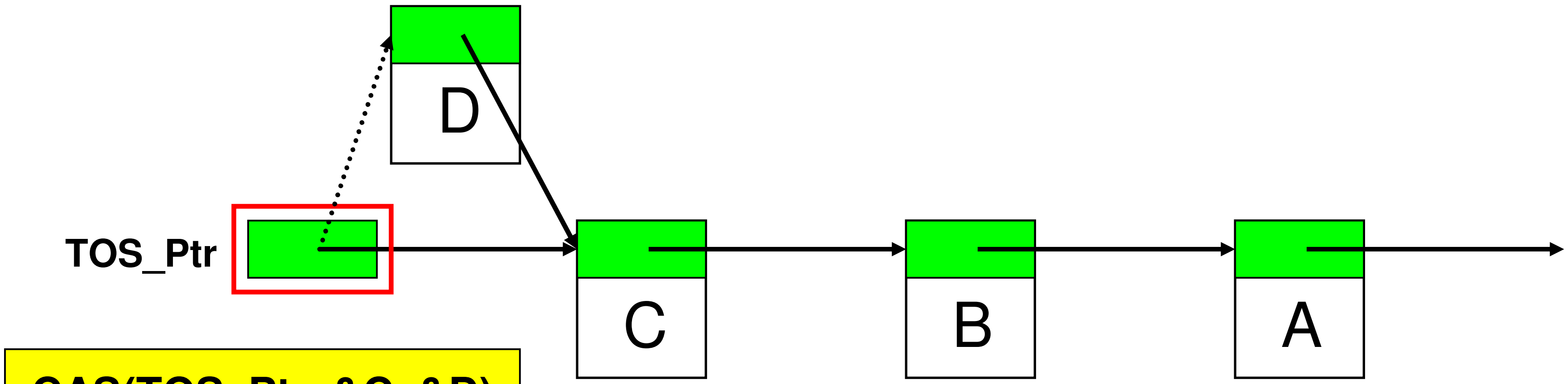
1. Read *TOS_Ptr*
2. Prepare the new node



Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item
- Linked list based algorithm [IBM 1983]

1. Read *TOS_Ptr*
2. Prepare the new node
3. Try to update *TOS_Ptr* with Compare&Swap

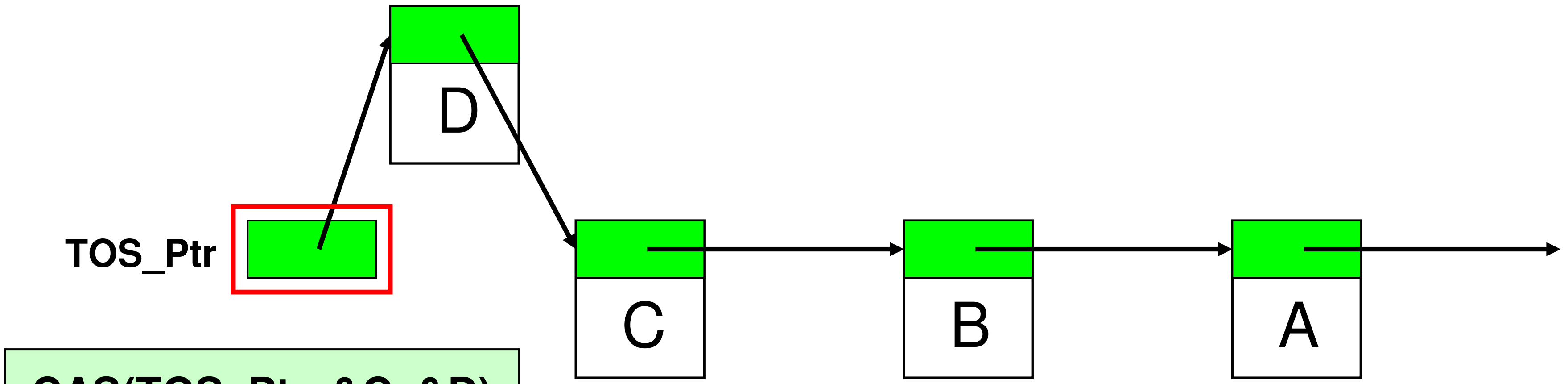


CAS(TOS_Ptr, &C, &D)

Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item
- Linked list based algorithm [IBM 1983]

1. Read *TOS_Ptr*
2. Prepare the new node
3. Try to update *TOS_Ptr* with Compare&Swap
4. If successful then done
   else retry from 1.



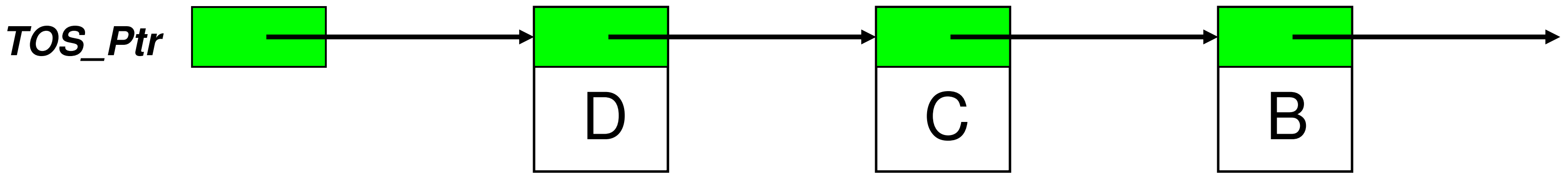
CAS(TOS_Ptr, &C, &D)

Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item

1. Read *TOS_Ptr*
2. Read *TOS_Ptr->next*

- Linked list based algorithm [IBM 1983]

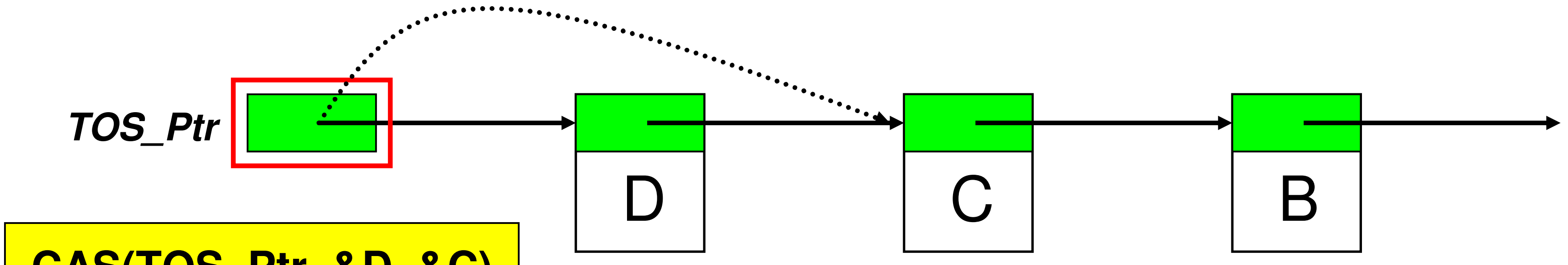


Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item

1. Read *TOS_Ptr*
2. Read *TOS_Ptr->next*
3. Try to update *TOS_Ptr* with Compare&Swap

- Linked list based algorithm [IBM 1983]

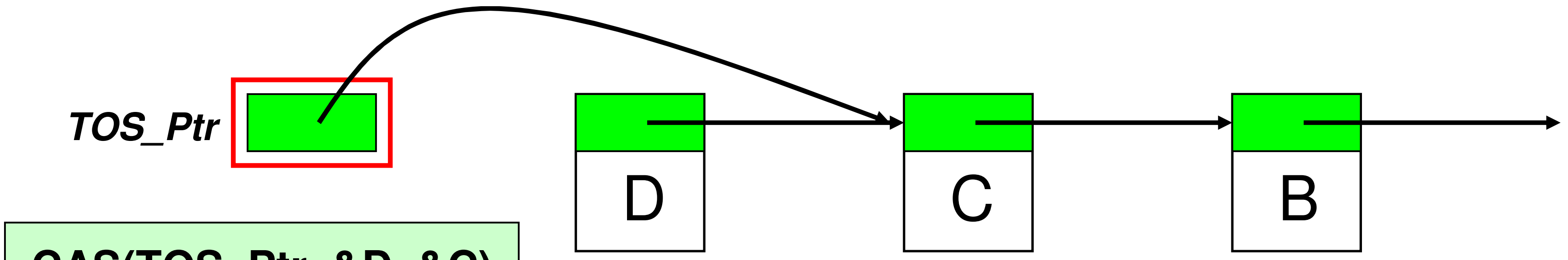


CAS(TOS_Ptr, &D, &C)

Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item
- Linked list based algorithm [IBM 1983]

1. Read *TOS_Ptr*
2. Read *TOS_Ptr->next*
3. Try to update *TOS_Ptr* with Compare&Swap
4. If successful then delete the node and return the item else retry from 1.



CAS(TOS_Ptr, &D, &C)

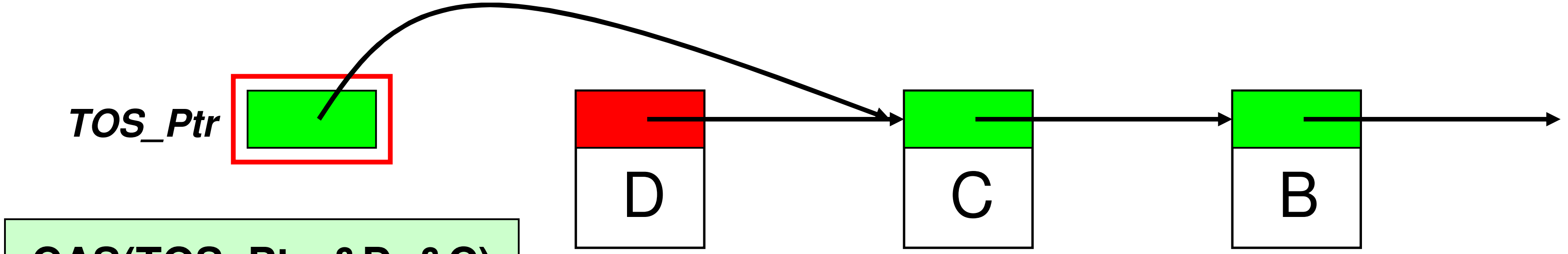
Example: Lock-free Stack

- Operations
  - Push(item)
  - Pop: item

1. Read *TOS_Ptr*
2. Read *TOS_Ptr->next*
3. Try to update *TOS_Ptr* with Compare&Swap
4. If successful then delete the node and return the item else retry from 1.

- Linked list based algorithm [IBM 1983]

But is it safe to delete node D now?

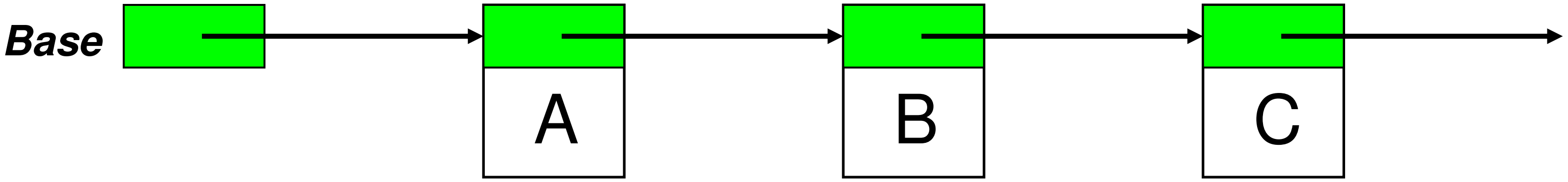


CAS(TOS_Ptr, &D, &C)

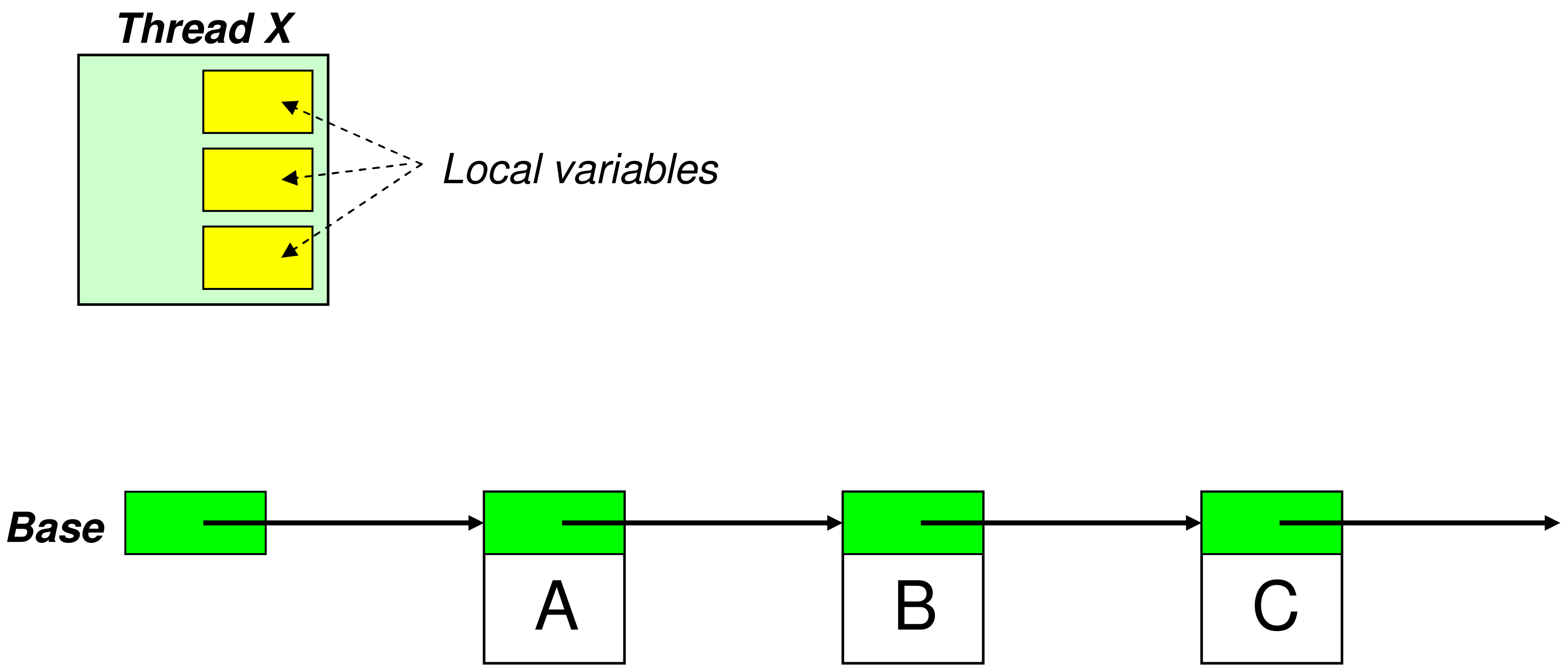
The Lock-Free Memory Reclamation Problem

- Concurrent shared data structures with
  - Dynamic use of shared memory
  - Concurrent and overlapping operations by threads or processes

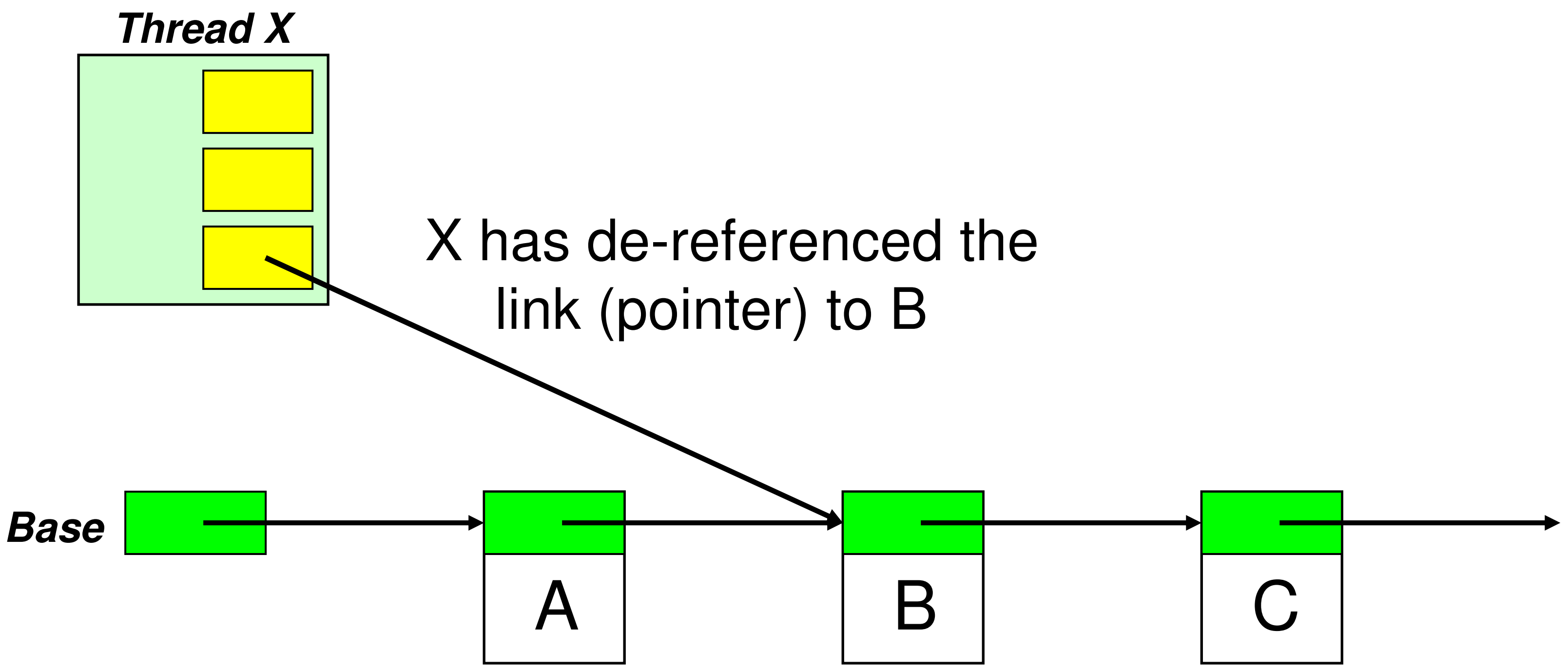
Can nodes be deleted and reused safely?



The Lock-Free Memory Reclamation Problem



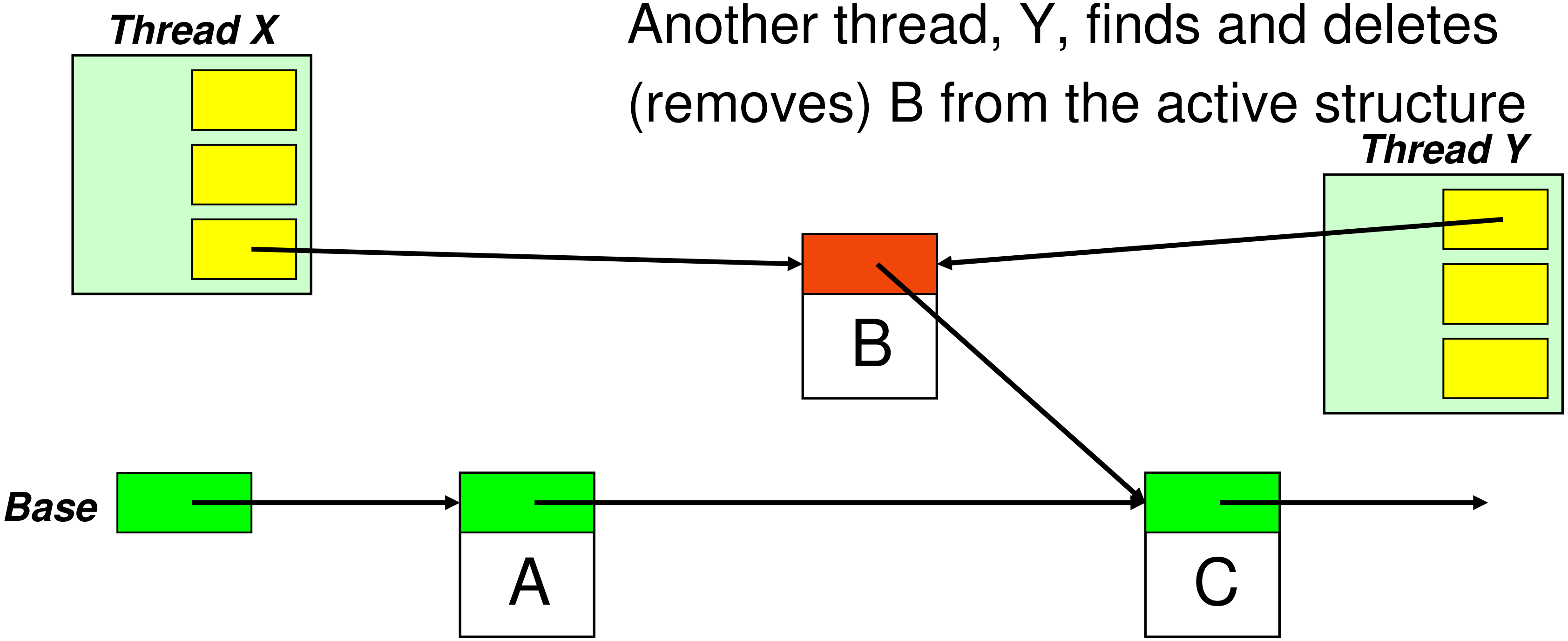
The Lock-Free Memory Reclamation Problem



X has de-referenced the link (pointer) to B

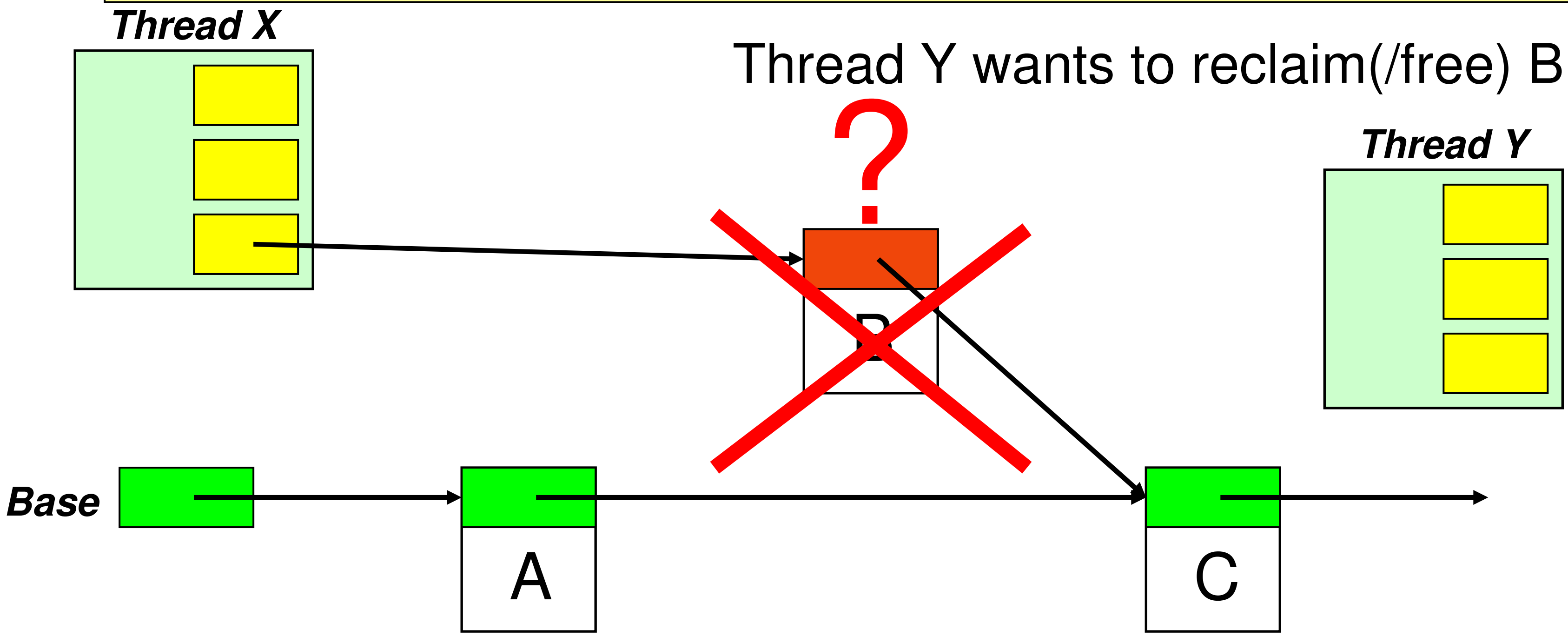
The Lock-Free Memory Reclamation Problem

Another thread, Y, finds and deletes (removes) B from the active structure



The Lock-Free Memory Reclamation Problem

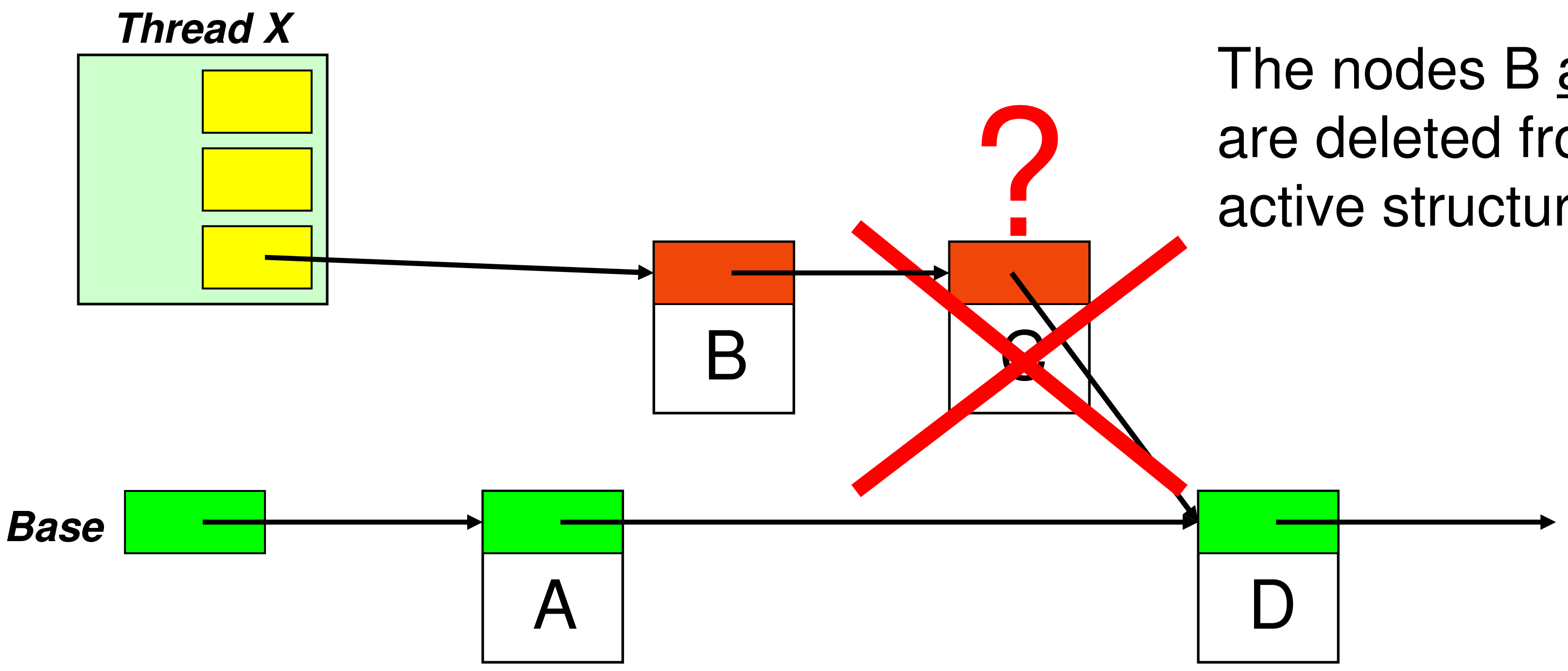
Property I: A node accessible via a private reference (i.e. *dereferenced*) should not be reclaimed



Thread Y wants to reclaim(/free) B

The Lock-Free Memory Reclamation Problem

Property II: Links in a dereferenced node should always be dereferencable.

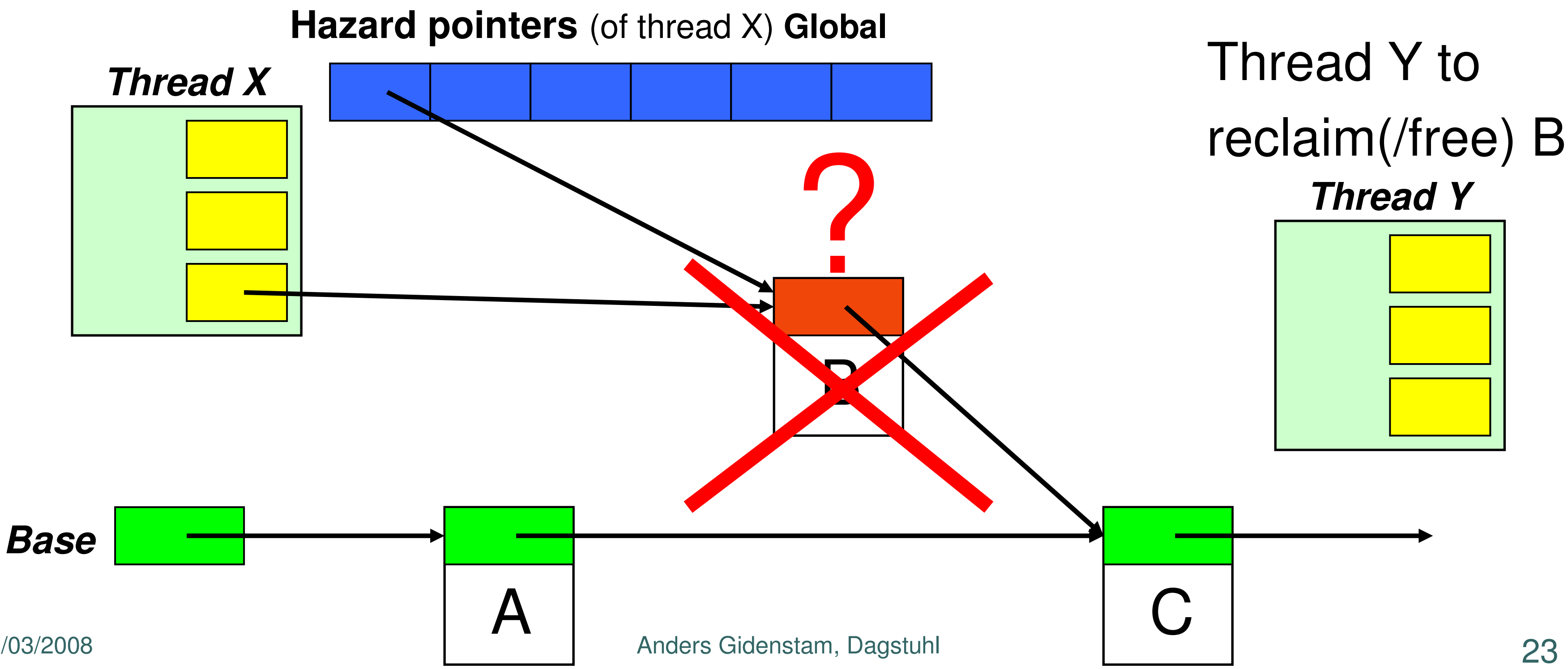


The nodes B <u>and</u> C are deleted from the active structure.

The Lock-Free Memory Reclamation Problem

Solutions providing Property I but not Property II:

- Hazard Pointers [Michael 2002]
- Pass the Buck [Herlihy 2002]



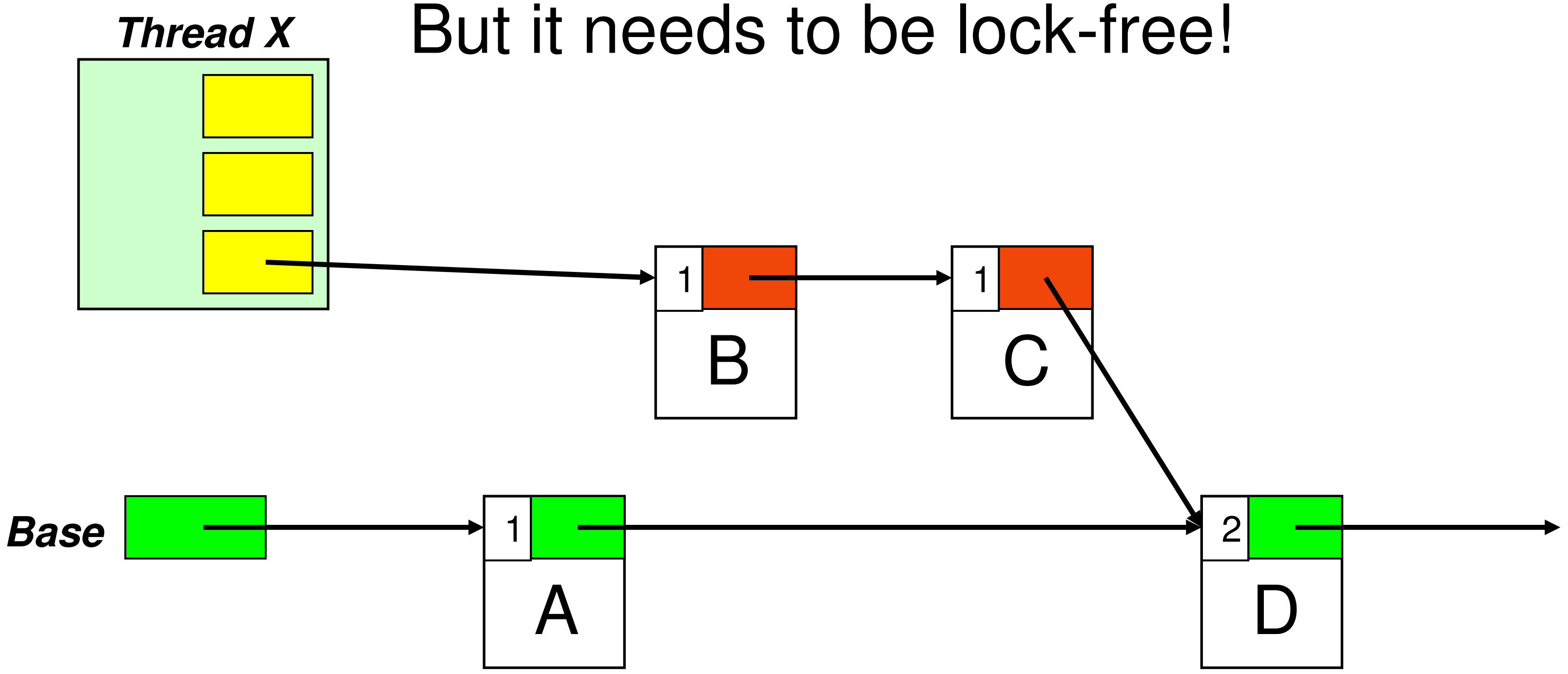
Thread Y to reclaim(/free) B

The Lock-Free Memory Reclamation Problem

Reference counting can guarantee

- Property I
- Property II

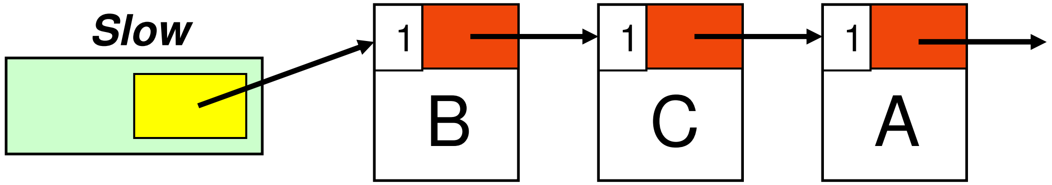
But it needs to be lock-free!



Lock-free reference counting solutions

- [Valois et al, 1995] Reference-count field MUST remain writable forever.
- LFRC [Detlefs et al, 2001] Needs double word CAS.
- SLFRC [Herlihy et al, 2002/2005] Pure reference counting (RC).
- LFMR [Gidenstam et al, 2005] RC + application guidance.

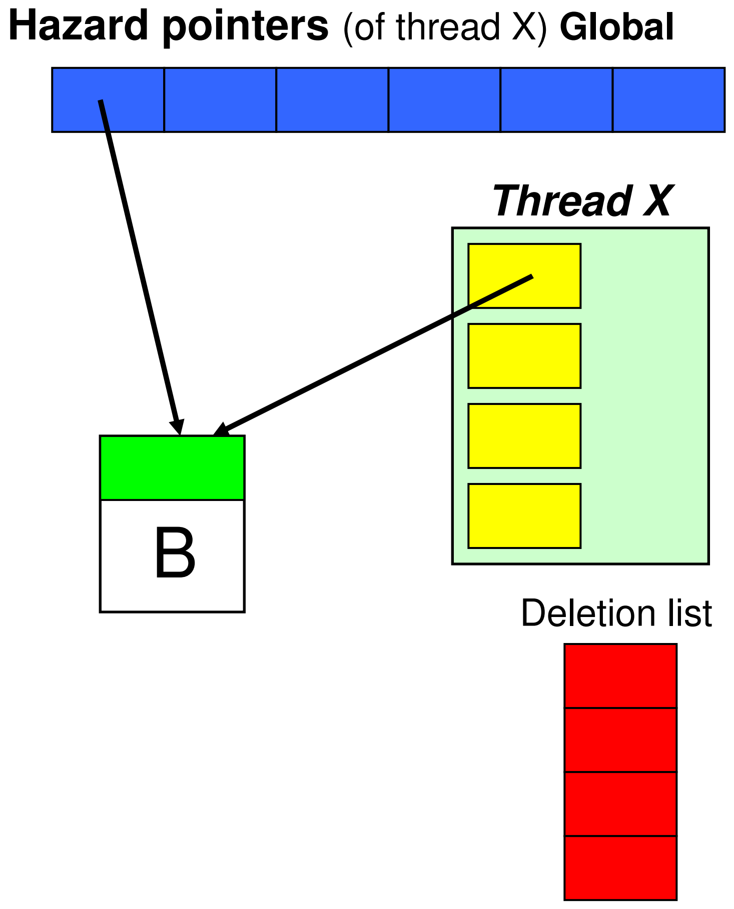
- Issues with pure reference counting
  - A slow thread with a private reference might prevent reclamation
  - Cyclic garbage



Our approach – The basic idea

Joint work w. M. Papatriantafilou, H. Sundell & P. Tsigas

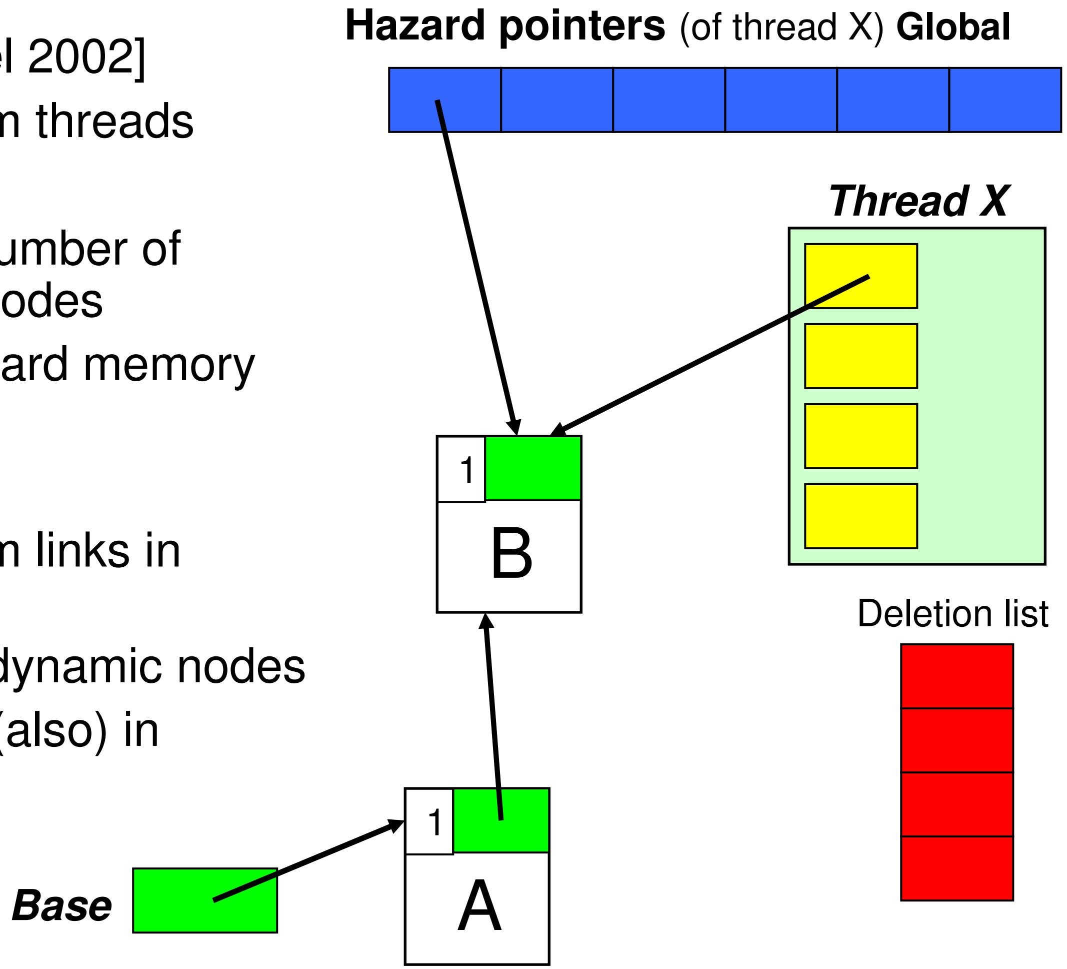
- Combine the best of
  - Hazard pointers [Michael 2002]
    - Tracks references from threads
    - Fast de-reference
    - Upper bound on the number of unreclaimed deleted nodes
    - Compatible with standard memory allocators



Our approach – The basic idea

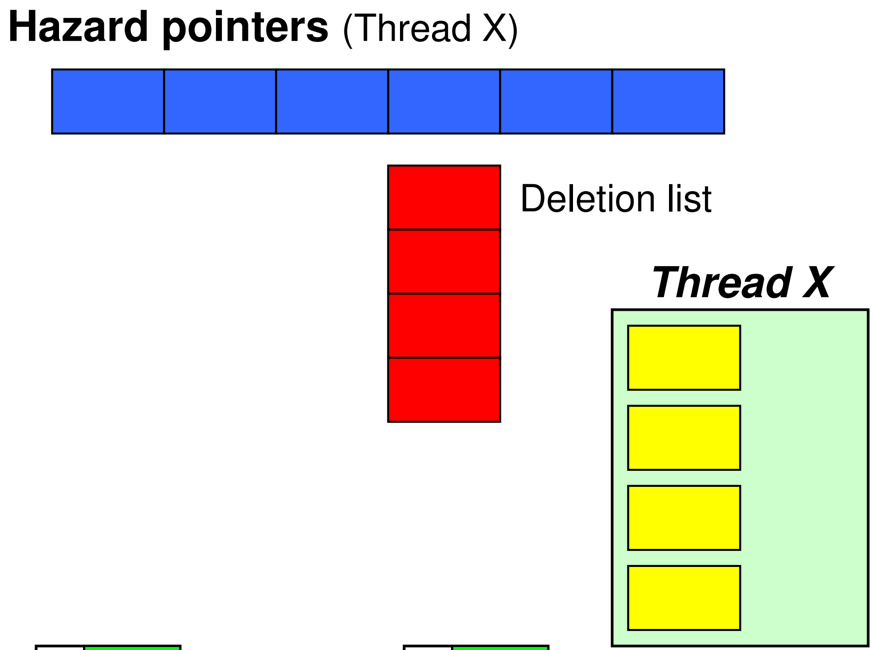
Joint work w. M. Papatriantafilou, H. Sundell & P. Tsigas

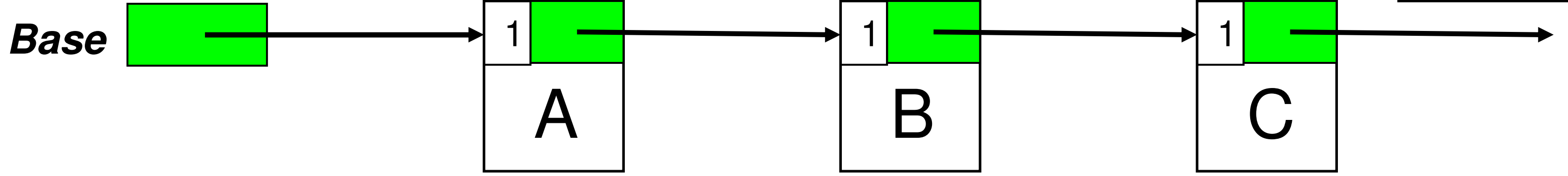
- Combine the best of
  - Hazard pointers [Michael 2002]
    - Tracks references from threads
    - Fast de-reference
    - Upper bound on the number of unreclaimed deleted nodes
    - Compatible with standard memory allocators
  - Reference counting
    - Tracks references from links in shared memory
    - Manages links within dynamic nodes
    - Safe to traverse links (also) in deleted nodes



The basic idea

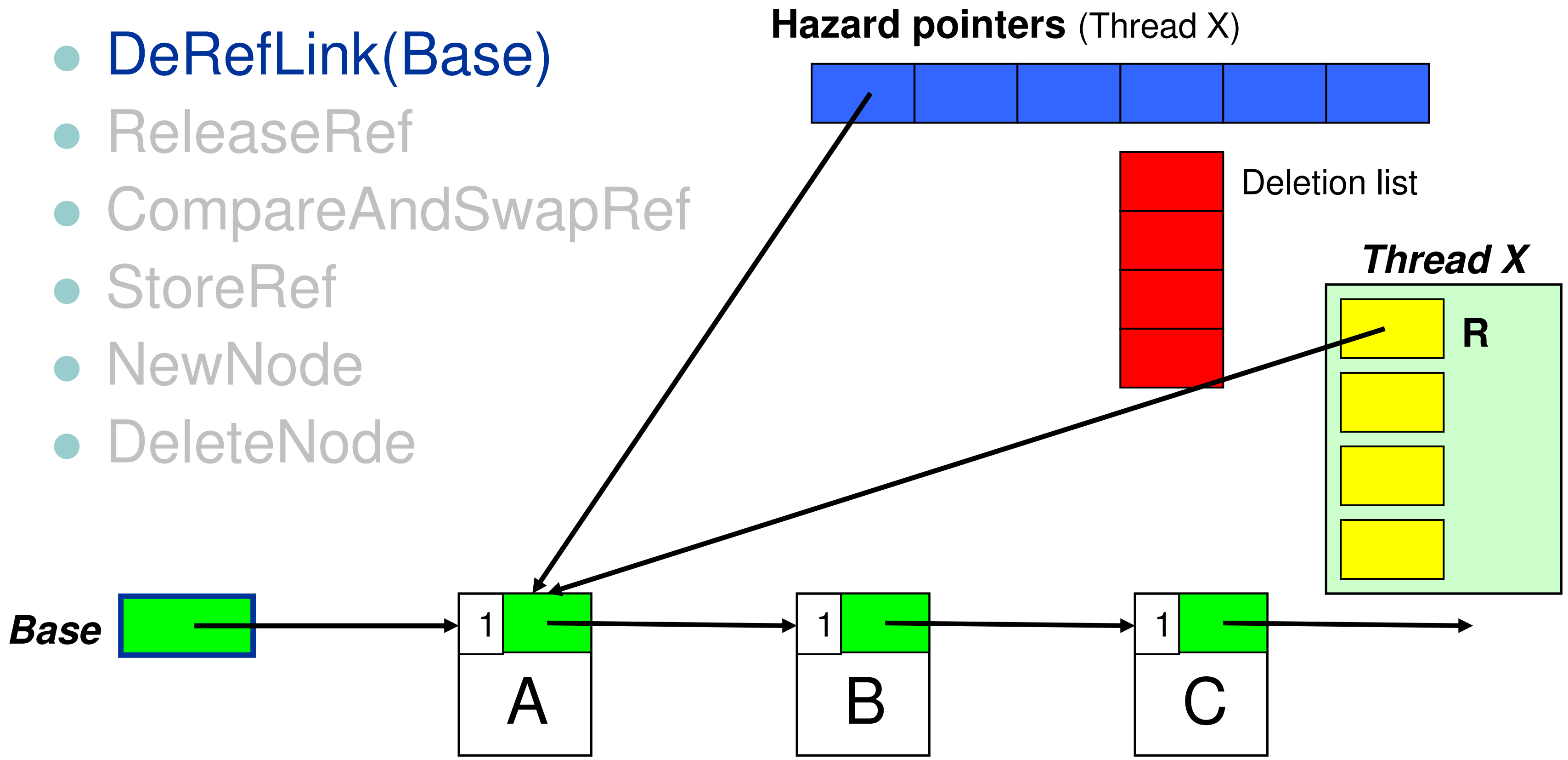
- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode





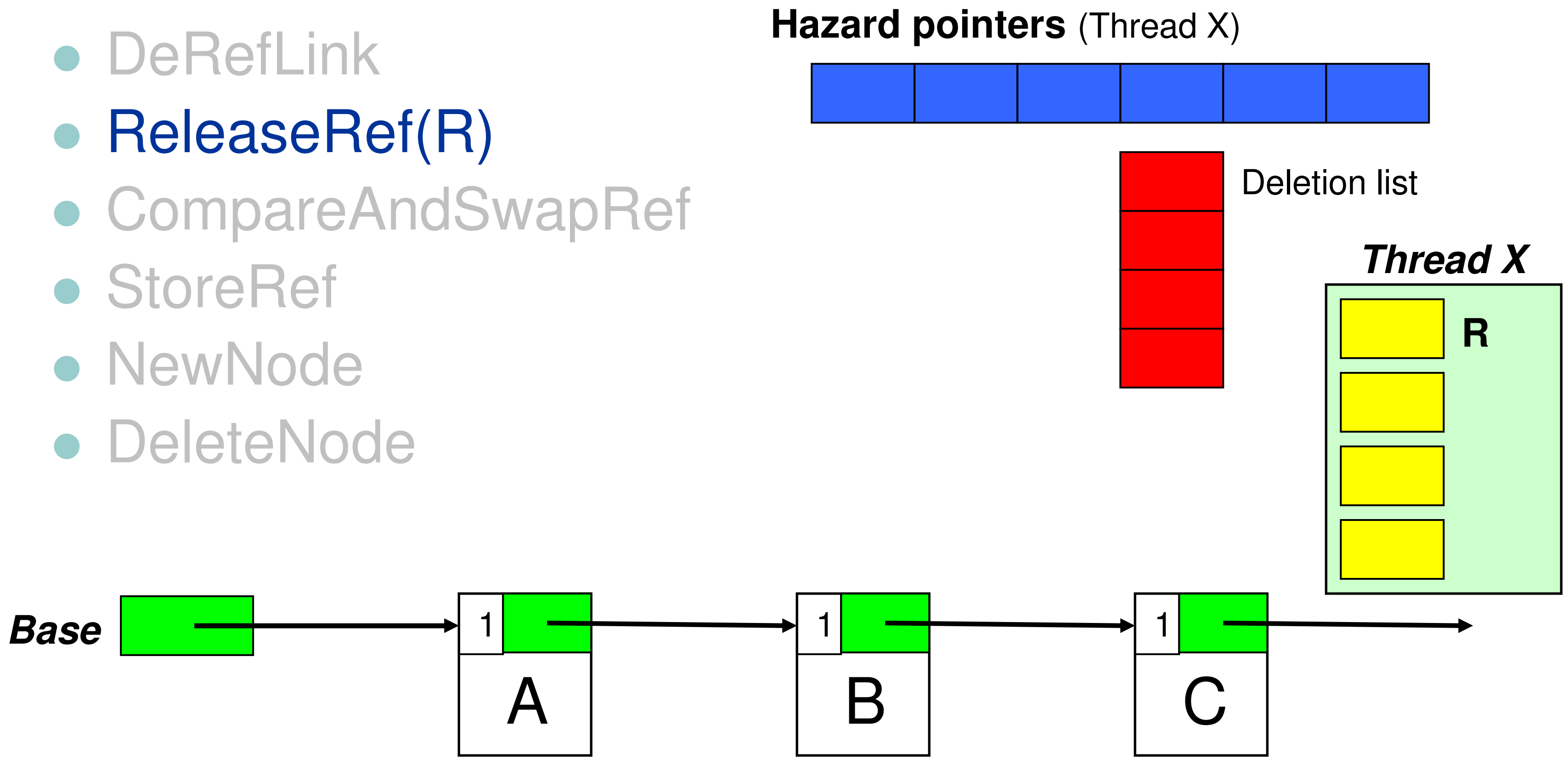
The basic idea

- API
  - DeRefLink(Base)
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



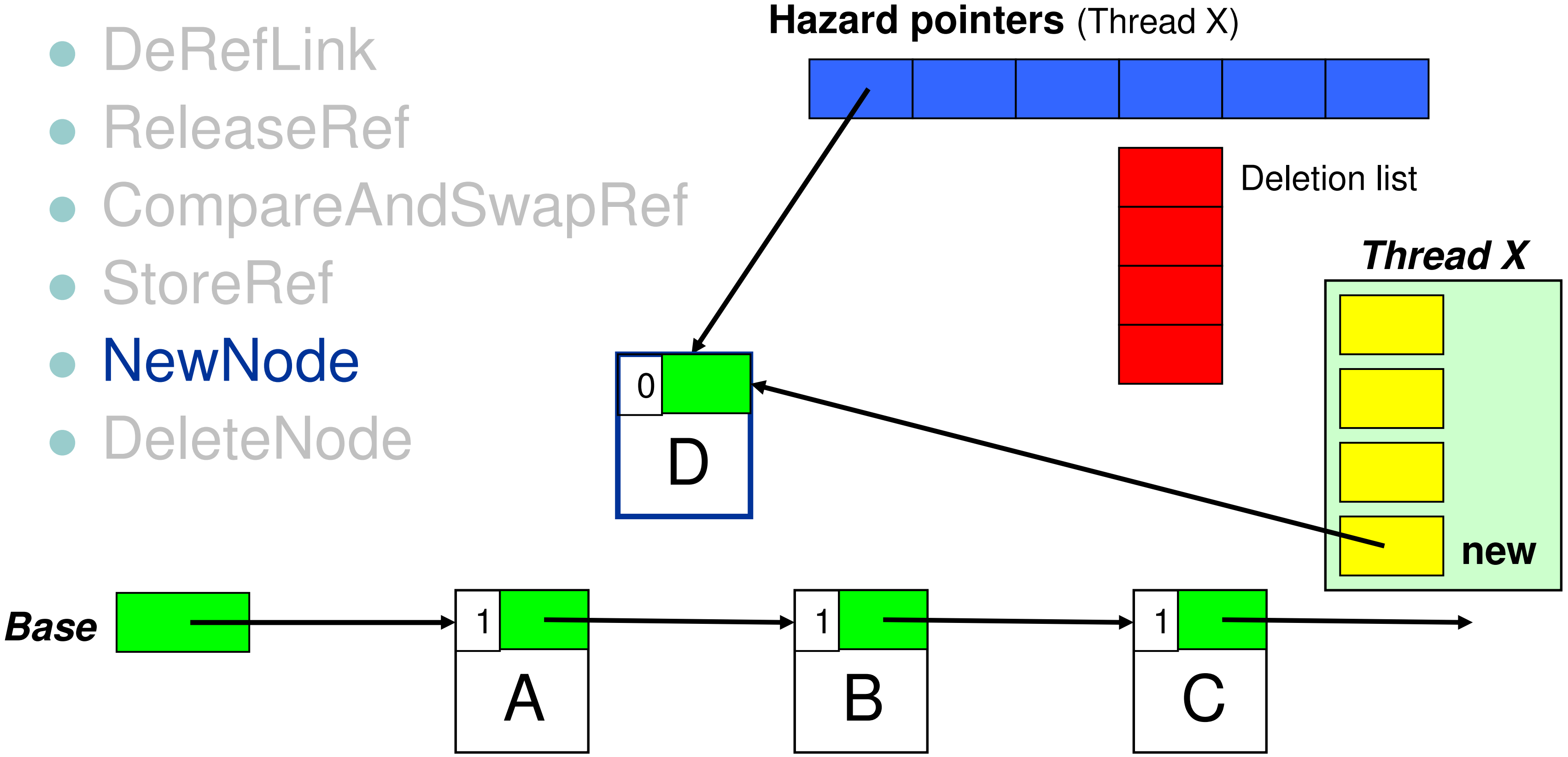
The basic idea

- API
  - DeRefLink
  - ReleaseRef(R)
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



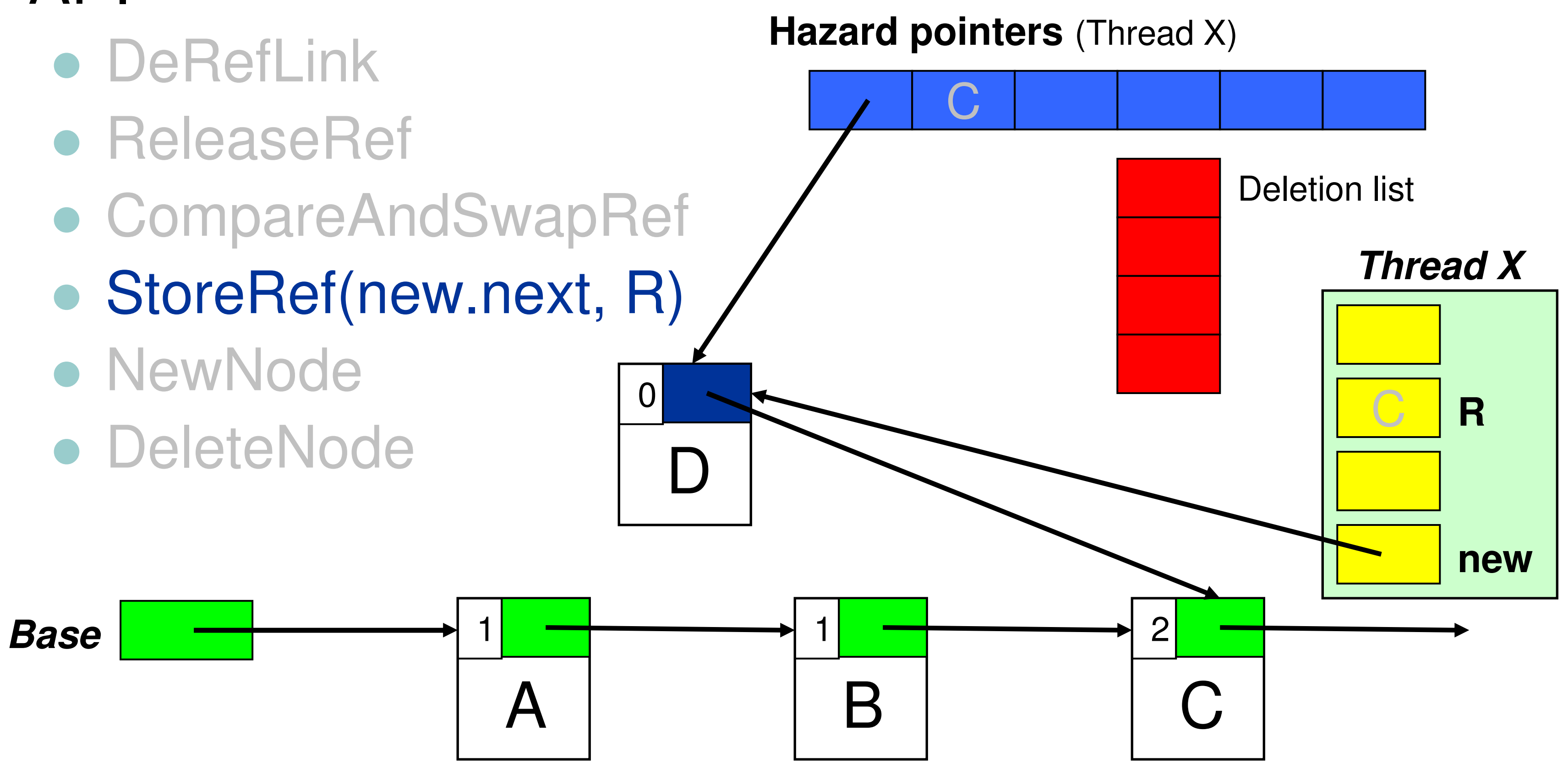
The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode



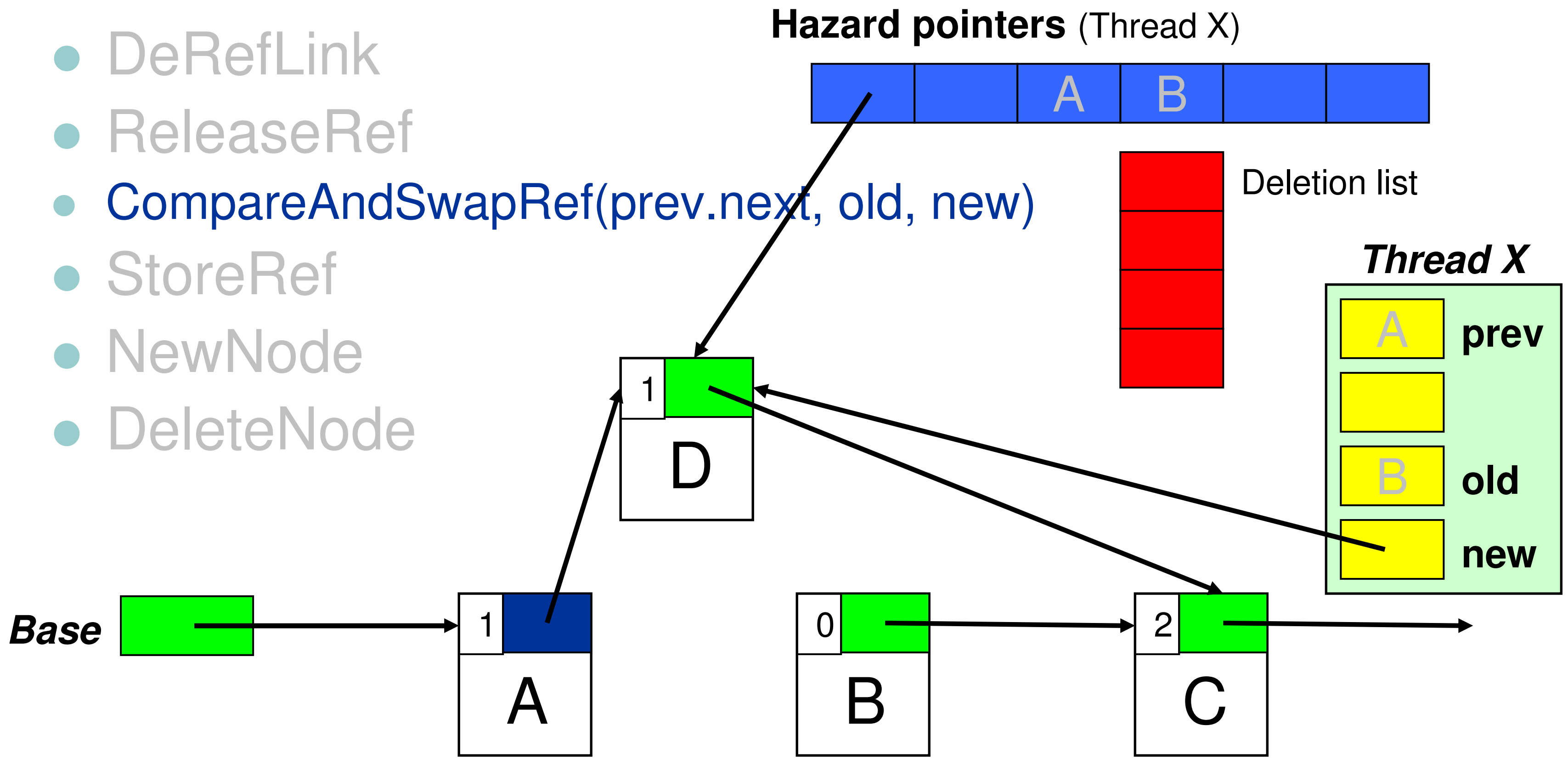
The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef(new.next, R)
  - NewNode
  - DeleteNode



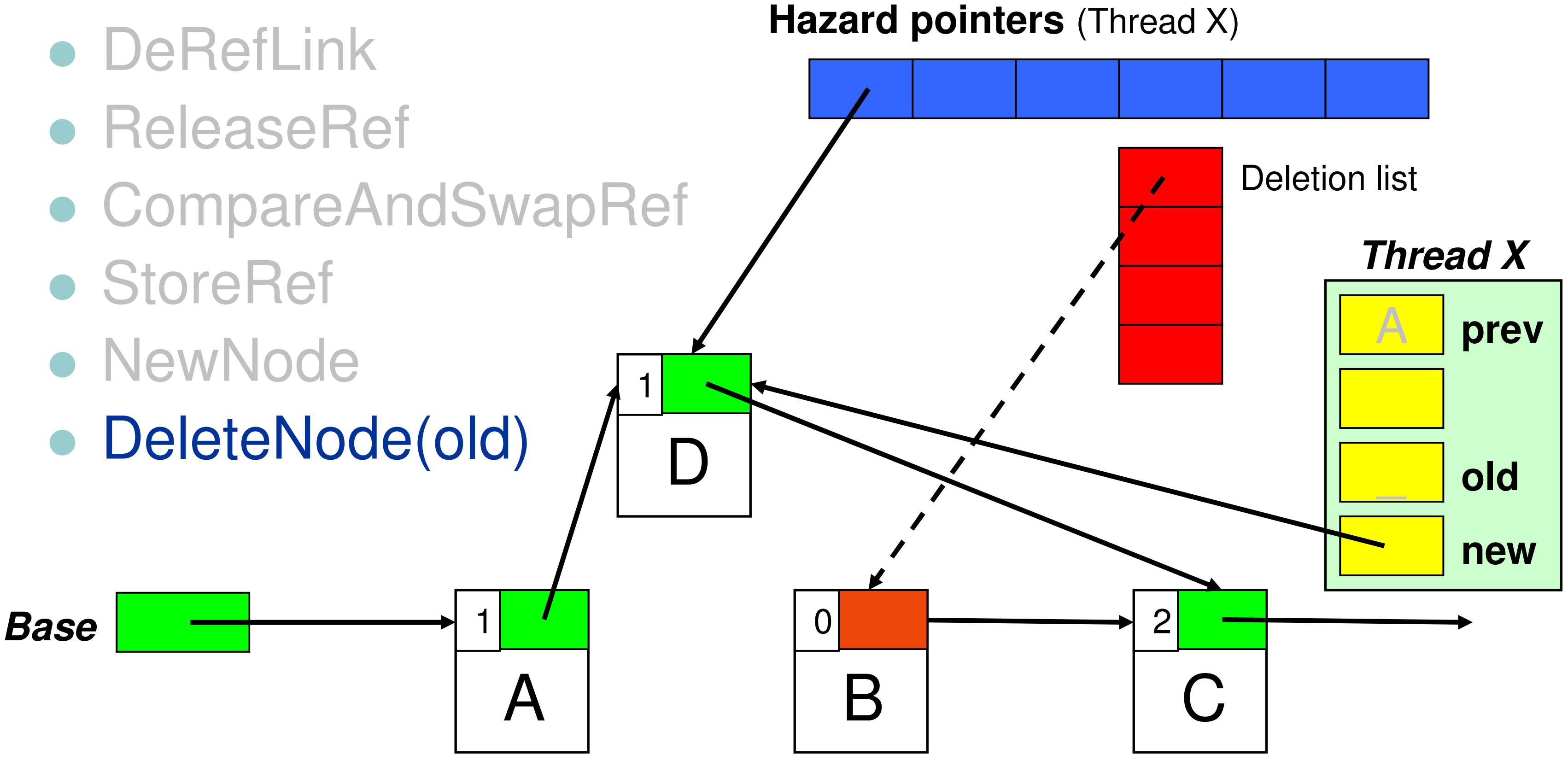
The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef(prev.next, old, new)
  - StoreRef
  - NewNode
  - DeleteNode



The basic idea

- API
  - DeRefLink
  - ReleaseRef
  - CompareAndSwapRef
  - StoreRef
  - NewNode
  - DeleteNode(old)



Bound on #unreclaimed nodes

Theorem: The maximum number of deleted but not yet reclaimed nodes in the system is never more than

$$N^2 \cdot (k + l_{max} + a + 1)$$

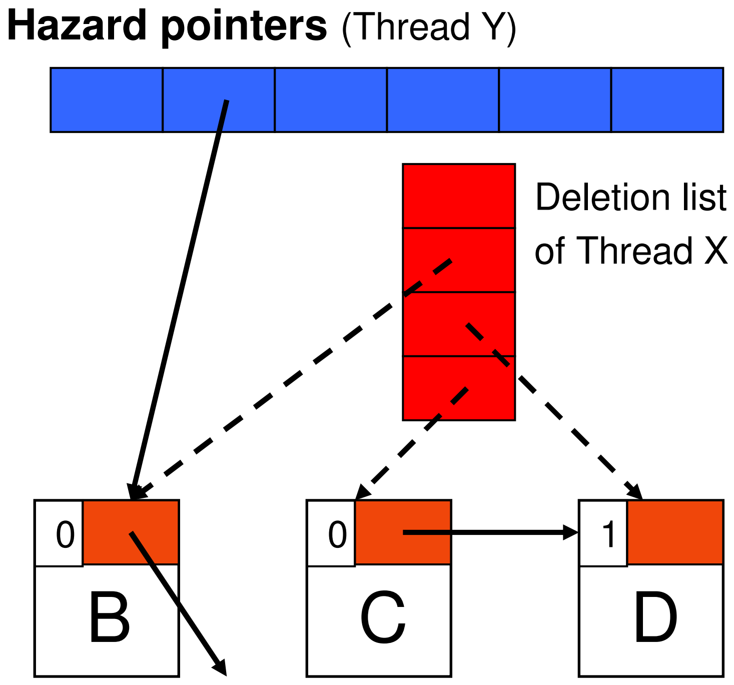
where

N is the number of threads in the system,

k is the number of hazard pointers per thread,

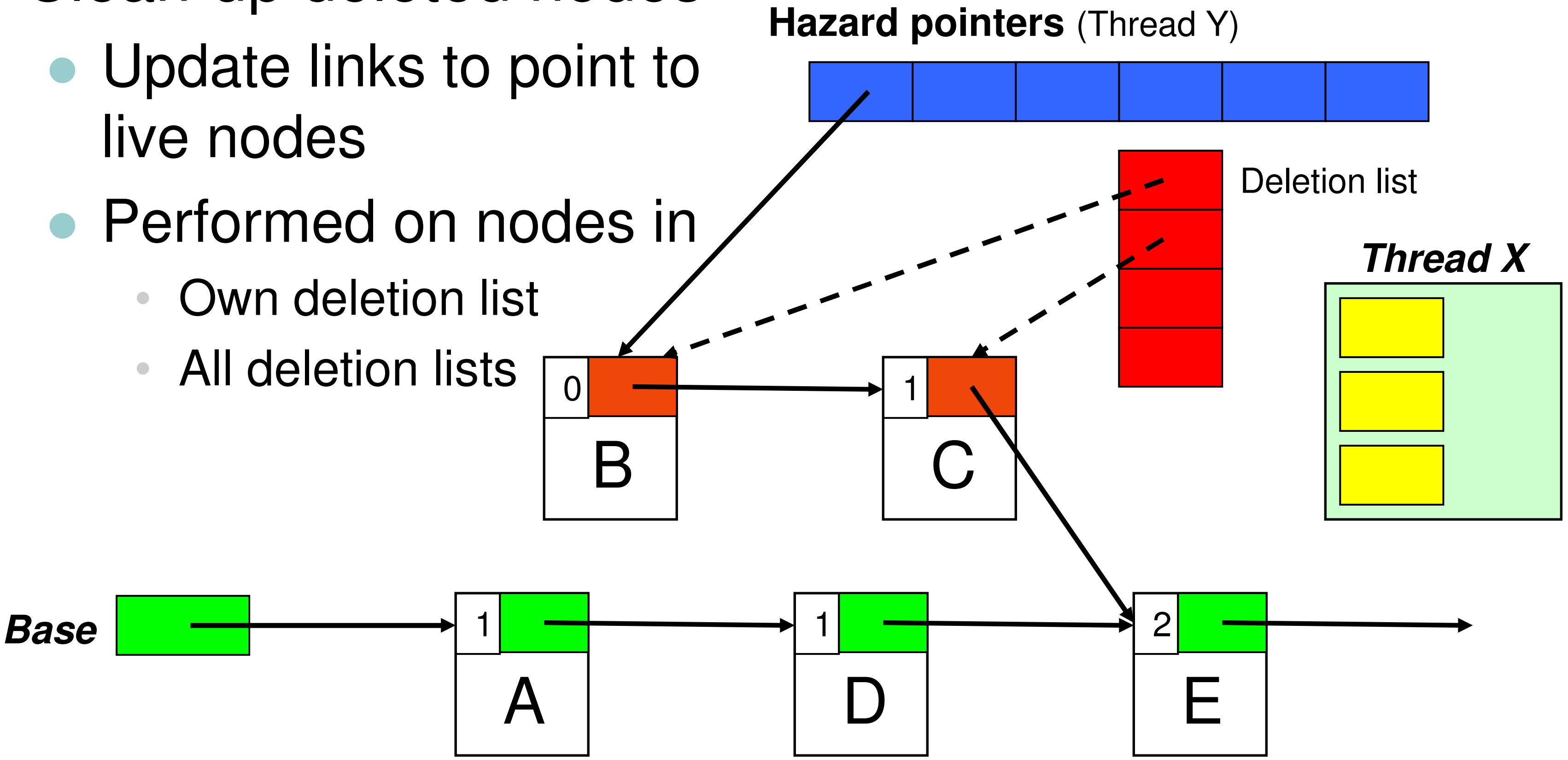
l_{max} is the maximum number of links a node can contain and

a is the maximum number of links in live nodes that may transiently point to a deleted node during an operation.



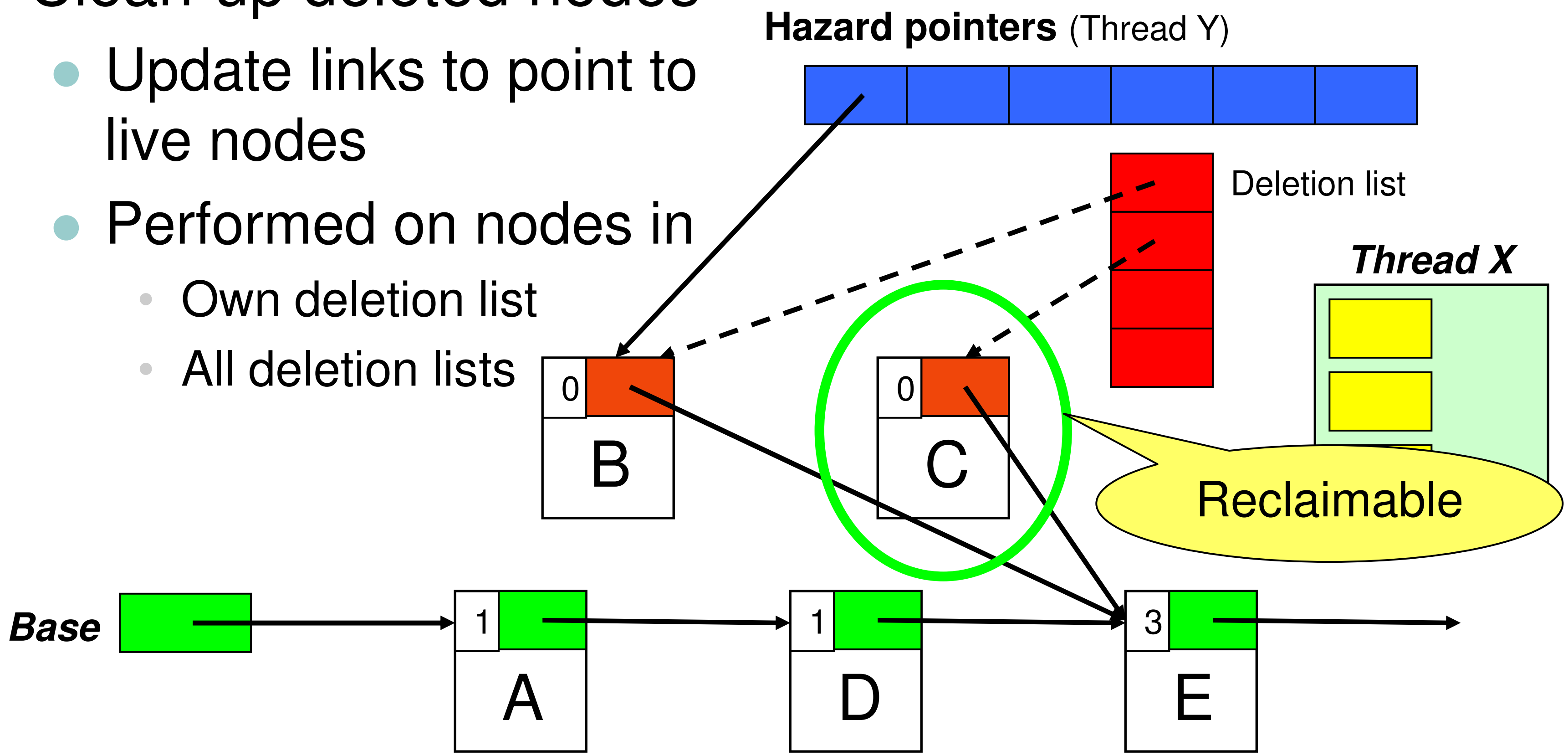
Breaking chains of garbage

- Clean-up deleted nodes
  - Update links to point to live nodes
  - Performed on nodes in
    - Own deletion list
    - All deletion lists



Breaking chains of garbage

- Clean-up deleted nodes
  - Update links to point to live nodes
  - Performed on nodes in
    - Own deletion list
    - All deletion lists



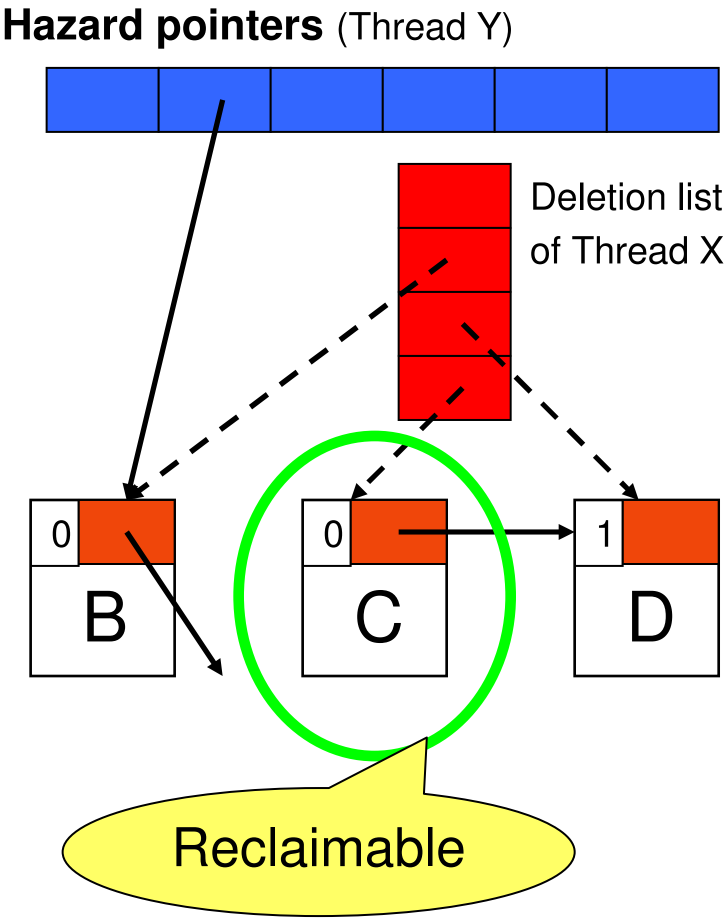
Bound on #unreclaimed nodes

- A deleted node is unreclaimable if
  - A hazard pointer points to it
    - Limited #hazard pointers: $N \cdot k$
  - Its reference count is nonzero
    - Limited #links in live nodes pointing to deleted nodes: $N \cdot a$
    - The links in most deleted nodes can be cleaned by any process.
      Exception: each thread can “hide” one node during a Delete operation.
      - #links in any node is bounded

      => $N \cdot l_{max}$
  - It is being cleaned by another thread: $N \cdot 1$
- => The maximum size of a thread’s deletion list is bounded by

  $N \cdot (k + l_{max} + a + 1)$
- => The total number of unreclaimable deleted nodes is bounded by

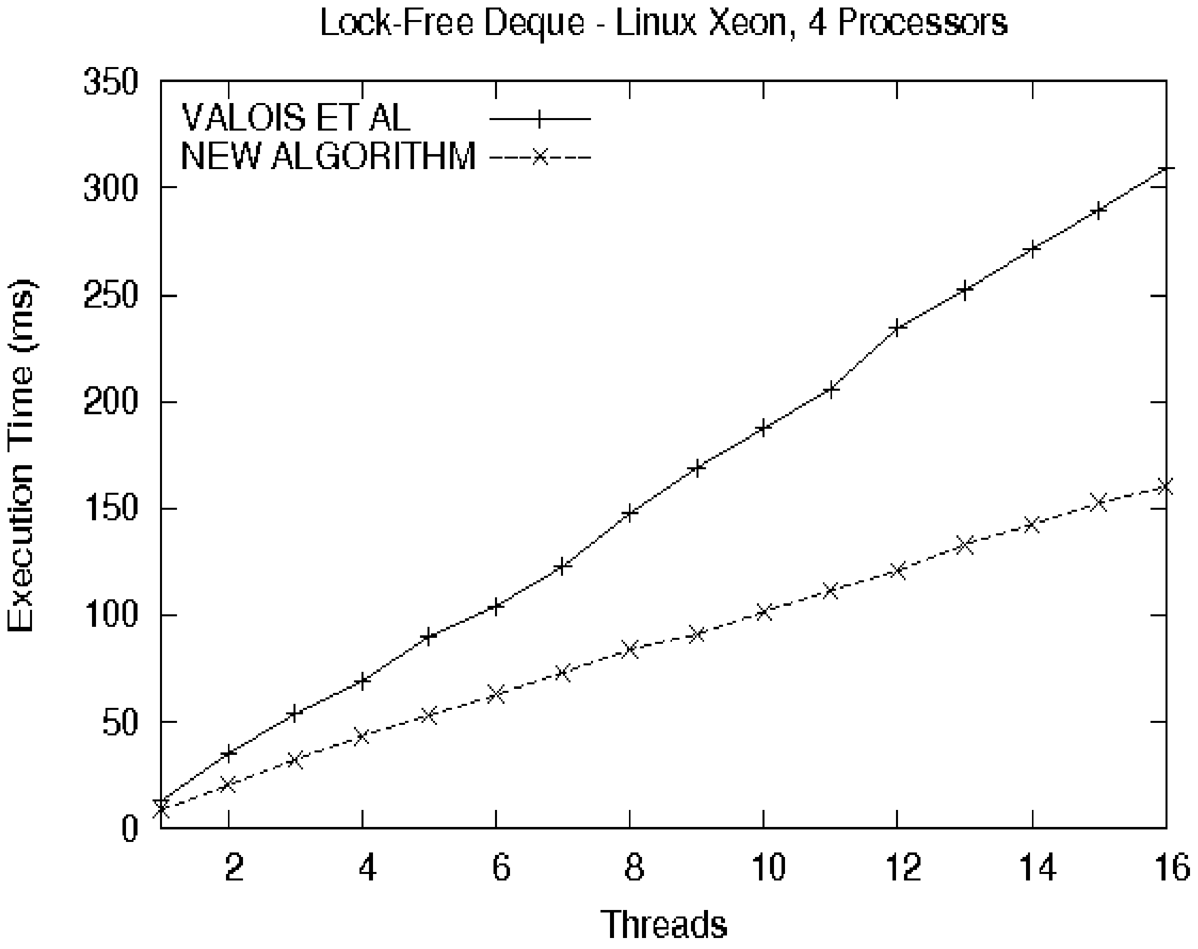
  $N^2 \cdot (k + l_{max} + a + 1)$



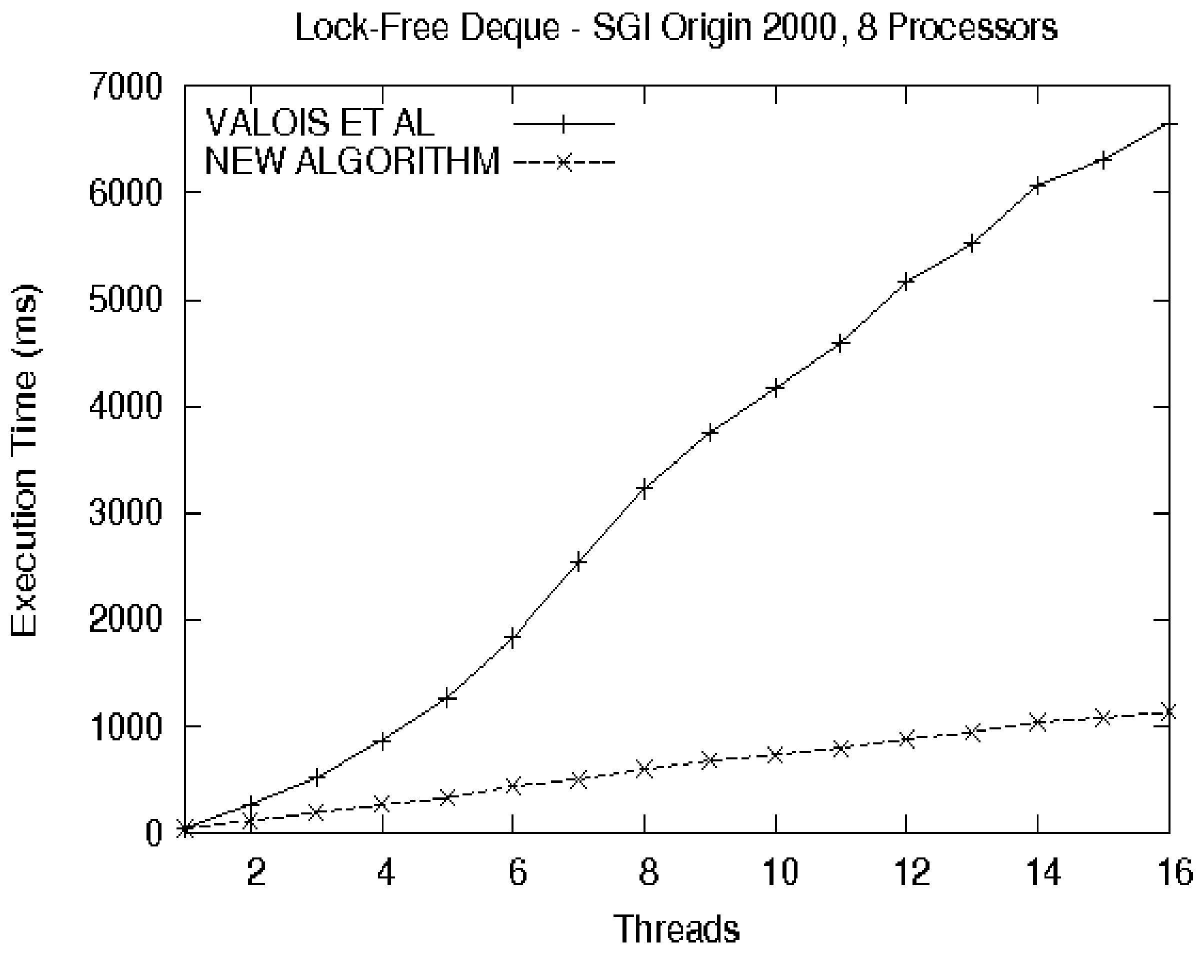
Experimental evaluation

- Lock-free deque [Sundell and Tsigas 2004]
  (deque – double-ended queue)
  - The algorithm needs traversal of deleted nodes
  - Time for 10000 random operations/thread
- Tested memory reclamation schemes
  - Reference counting, Valois et al.
  - LFRM (a.k.a. the new algorithm)
- Systems
  - 4 processor Xeon PC / Linux (UMA)
  - 8 processor SGI Origin 2000 / IRIX (NUMA)

Experimental evaluation



Experimental evaluation



Some lock-free data-structures

Stacks	**Memory man.**
[IBM 1983],[Treiber,1986]	version #s / HP
Queues	
[Valois, 1994]	RC
[MS, 1996]	RC / HP
[TZ, 2001]	bounded size
[Hoffman et al, 2007]	version #s / RC
Double Ended Queues / Deques	
[M, ?]	?
[ST, 2004]	RC
Priority Queues	
[Barnes, 1994]	bounded size
[ST 2003]	RC
Sets/Dictionaries	
[Michael, 2002]	HP
[TS 2004]	RC

Linked lists	**Memory man.**
[Valois, 1995]	RC
[Harris, 2001]	RC
Doubly linked lists	
[TS, 2007]	RC
Hash tables	
[Michael, 2002]	HP
[Shalev Shavit 2006]	?
Trees (Binary, Red-Black)	
[Fraser, 2004]	M-CAS / STM / RC

Current Work

- Memory management
  - Unified and easy to use interfaces to the memory management algorithms [NBAda library]
    - Easier to implement lock-free data-structures
    - Data-structure user could choose memory management method.
- Lock-free data-structures
  - Develop new (Red-Black trees with 1-CAS?)
  - Software library: NBAda
    - Ought to be here: http//www.mpi-inf.mpg.de/~andersg/ For now: Ask me.