Practical and Lock-Free Doubly Linked Lists

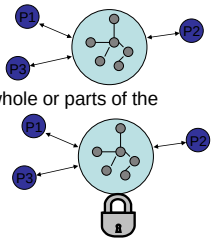
Håkan Sundell
Philippas Tsigas

Outline

- Synchronization Methods
- Doubly Linked Lists
- Concurrent Doubly Linked Lists
  - Previous results
  - New Lock-Free Algorithm
- Experiments
- Conclusions

Synchronization

- Shared data structures needs synchronization
- Synchronization using Locks
  - Mutually exclusive access to whole or parts of the data structure

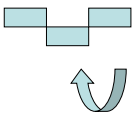


Blocking Synchronization

- Drawbacks
  - Blocking
  - Priority Inversion
  - Risk of deadlock
- Locks: Semaphores, spinning, disabling interrupts etc.
  - Reduced efficiency because of reduced parallelism

Non-blocking Synchronization

- Lock-Free Synchronization
  - Optimistic approach (i.e. assumes no interference)

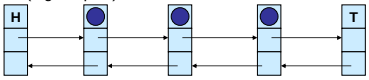


1. The operation is prepared to later take effect (unless interfered) using hardware atomic primitives
2. Possible interference is detected via the atomic primitives, and causes a retry
   - Can cause starvation

- Wait-Free Synchronization
  - Always finishes in a finite number of its own steps.

Doubly Linked Lists

- Fundamental data structure
  - Can be used to implement various abstract data types (e.g. deques)



- Unordered List, i.e. the nodes are ordered only relatively to each other.
  - Supports Traversals
  - Supports Inserts/Deletes at arbitrary positions
- Operations: *InsertAfter, InsertBefore, Delete, Prev, Next, Read, First, Last*

Previous Non-blocking Doubly Linked Lists

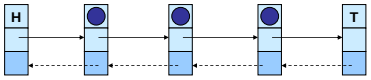
- M. Greenwald, “Two-handed emulation: how to build non-blocking implementations of complex data structures using DCAS”, PODC 2002
- O. Agesen et al., “DCAS-based concurrent deques”, SPAA 2000
  - D. Detlefs et al., “Even better DCAS-based concurrent deques”, DISC 2000
  - P. Martin et al. “DCAS-based concurrent deques supporting bulk allocation”, TR, 2002
  - *Errata:* S. Doherty et al. “DCAS is not a silver bullet for nonblocking algorithm design”, SPAA 2004

Previous Non-blocking Doubly Linked Lists

- H. Attiya and E. Hillel, “Built-in coloring for highly-concurrent doubly linked lists”, DISC 2006
- J. Valois, Ph.D. Thesis, 1995
- P. Martin (D. Lea) “A practical lock-free doubly-linked list”, 2004
- Problems (unified, all prev. pub have subset):
  - DCAS not available in contempory sys.
  - No disjoint access parallelism
  - No traversals from deleted nodes.
  - No delete operation
  - Not consistent data structure when idle

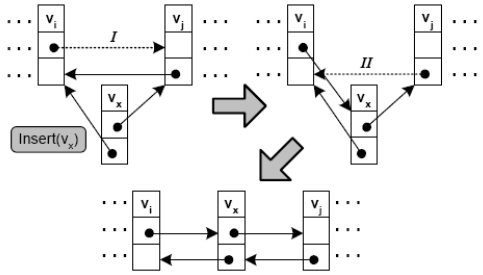
New Lock-Free Concurrent Doubly Linked List (*Deque at OPODIS 2004*)

- Treat the doubly linked list as a singly linked list with auxiliary information in each node about its predecessor!

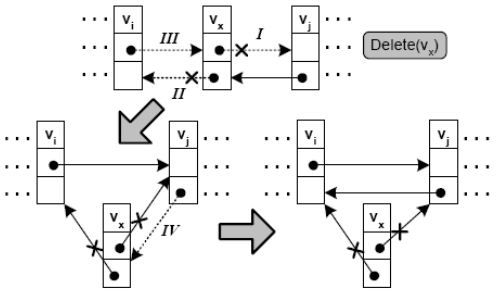


- Singly Linked Lists
  - T. Harris, “A pragmatic implementation of non-blocking linked lists”, DISC 2001
    - Marks pointers using spare bit
    - Needs only standard CAS

Lock-Free Doubly Linked Lists - INSERT



Lock-Free Doubly Linked Lists - DELETE



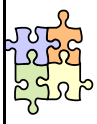
Lock-Free Doubly Linked List – Traversal and Positions

- **Informal:** From a deleted node (i.e. earlier position) there should be a path that leads into the active list (i.e. a new position).
- **Definition 1** *The position of a cursor that references a node that is present in the list is the referenced node. The position of a cursor that references a deleted node, is represented by the node that was directly to the next of the deleted node at the very moment of the deletion (i.e. the setting of the deletion mark). If that node is deleted as well, the position is equal to the position of a cursor referencing that node, and so on recursively. The actual position is then interpreted to be at an imaginary node directly previous of the representing node.*

Lock-Free Doubly Linked List - Memory Management

- The information about neighbor nodes should also be accessible in partially deleted nodes!
  - Enables helping operations to find
  - Enables continuous traversals
- A. Gidenstam et. al, “Efficient and reliable lock-free memory reclamation based on reference counting”, ISPAN 2005
  - Combines Hazard Pointers with Reference Count.
    - Allows fully dynamic memory management
    - Allows (also in deleted) in-node safe pointers
    - Breaks cyclic garbage
    - High performance

New Lock-Free Doubly Linked List - Techniques Summary



- General Doubly Linked List Structure
  - Treated as singly linked lists with extra info
- Uses CAS atomic primitive
- Lock-Free memory management
  - Gidenstam et. al: (HP + Ref. Count).
- Helping scheme
- Back-Off strategy
- All together proved to be linearizable

Experiments

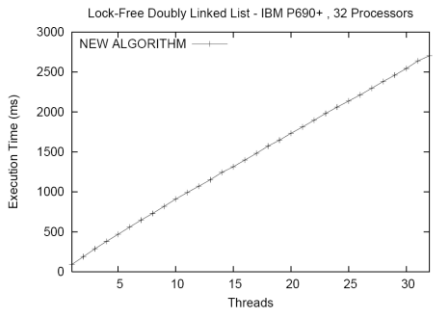
- Implemented in C on two high-performance computers
  - Sun Fire 15K, Solaris, Ultrasparc III 900 MHz, 48-way
  - IBM p690+ Regatta, AIX, 1.7 GHz, 32-way
- 50 000 random operations / thread
  - 30% traversal, 10% insertbefore, 10% insertafter, 50% delete
- 1 upto 32 threads
- Measured average execution time
  - Scales lineary with increasing number of threads

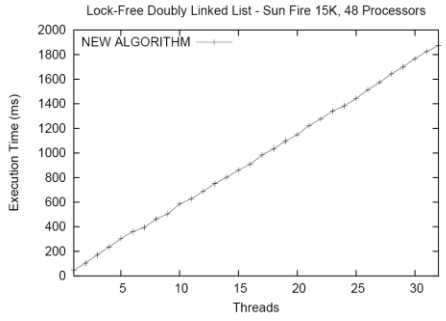
Conclusions

- Implements a general doubly linked list, the first lock-free using CAS.
  - Allows disjoint-access parallelism.
  - Uses lock-free memory management.
  - Uses atomic primitives available in contemporary systems.
  - Allows fully dynamic list sizes.
  - Supports traversals of deleted nodes.

Questions?

- Contact Information:
  - Address:
    - Håkan Sundell, School of Business and Informatics, University College of Borås, Sweden
    - Philippas Tsigas, Computing Science, Chalmers University of Technology, Sweden
  - Email:
    - Hakan.Sundell@hb.se
    - tsigas@cs.chalmers.se
  - Web:
    - http://www.cs.chalmers.se/~noble





ORÅS

```
function Next(cursor: pointer to pointer to Node): boolean
NT1   while true do
NT2      if *cursor = tail then return false;
NT3      next:=(*cursor).next;
NT4      d := next.next.d;
NT5      if d = true and (*cursor).next ≠ ⟨next,true⟩ then
NT6         SetMark(&next.prev);
NT7         CAS(&(*cursor).next,next,next.next.p);
NT8         continue;
NT9      *cursor:=next;
NT10     if d = false and next ≠ tail then return true;
```

UNIVERSITY COLLEGE OF BORÅS

```
function Prev(cursor: pointer to pointer to Node): boolean
PV1   while true do
PV2      if *cursor = head then return false;
PV3      prev:=(*cursor).prev;
PV4      if prev.next = ⟨*cursor,false⟩ and (*cursor).next.d = false then
PV5         *cursor:=prev;
PV6         if prev ≠ head then return true;
PV7      else if (*cursor).next.d = true then Next(cursor);
PV8      else prev:=CorrectPrev(prev,*cursor);
```

UNIVERSITY COLLEGE OF BORÅS

```
function Read(cursor: pointer to pointer to Node): pointer to word
RD1   if *cursor = head or *cursor = tail then return ⊥;
RD2   value:=(*cursor).value;
RD3   if (*cursor).next.d = true then return ⊥;
RD4   return value;
```

UNIVERSITY COLLEGE OF BORÅS

```
procedure InsertBefore(cursor: pointer to pointer to Node
  , value: pointer to word)
IB1    if *cursor = head then return InsertAfter(cursor,value);
IB2    node:=CreateNode(value);
IB3    prev:=(*cursor).prev;
IB4    while true do
IB5       while (*cursor).next.d = true do
IB6          Next(cursor);
IB7          prev:=CorrectPrev(prev,*cursor);
IB8       next:=(*cursor);
IB9       node.prev:=⟨prev,false⟩;
IB10      node.next:=⟨next,false⟩;
IB11      if CAS(&prev.next,⟨(*cursor),false⟩,⟨node,false⟩) then break;
IB12      prev:=CorrectPrev(prev,*cursor);
IB13      Back-Off
IB14   *cursor:=node;
IB15   prev:=CorrectPrev(prev,next);
```

UNIVERSITY COLLEGE OF BORÅS

```
procedure InsertAfter(cursor: pointer to pointer to Node
  , value: pointer to word)
IA1    if *cursor = tail then return InsertBefore(cursor,value);
IA2    node:=CreateNode(value);
IA3    prev:=(*cursor);
IA4    while true do
IA5       next:=prev.next;
IA6       node.prev:=⟨prev,false⟩;
IA7       node.next:=⟨next,false⟩;
IA8       if CAS(&(*cursor).next,⟨next,false⟩,⟨node,false⟩) then break;
IA9       if prev.next.d = true then
IA10         return InsertBefore(cursor,value);
IA11      Back-Off
IA12   *cursor:=node;
IA13   prev:=CorrectPrev(prev,next);
```

UNIVERSITY COLLEGE OF BORÅS

```
function Delete(cursor: pointer to pointer to Node): pointer to word
D1    node:=*cursor;
D2    if node = head or node = tail then return ⊥;
D3    while true do
D4       next:=(*cursor).next;
D5       if next.d = true then return ⊥;
D6       if CAS(&node.next,next,⟨next.p,true⟩) then
D7          while true do
D8             prev:=node.prev;
D9             if prev.d = true or CAS(&node.prev,prev,⟨prev.p,true⟩)
                 then break;
D10         prev:=CorrectPrev(prev.p,next);
D11         value:=node.value;
D12         return value;
```



```
function CorrectPrev(prev, node: pointer to Node):pointer to Node
CP1   lastlink:=⊥;
CP2   while true do
CP3      link1:=node.prev; if link1.d = true then break;
CP4      prev2:=prev.next;
CP5      if prev2.d = true then
CP6         if lastlink ≠ ⊥ then
CP7            SetMark(&prev.prev);
CP8            CAS(&lastlink.next,prev,⟨prev2.p,false⟩);
CP9            prev:=lastlink; lastlink:=⊥;
CP10           continue;
CP11     prev2:=prev.prev;
CP12     prev:=prev2;
CP13     continue;
CP14  if prev2 ≠ node then
CP15     lastlink:=prev;
CP16     prev:=prev2;
CP17     continue;
CP18  if CAS(&node.prev,link1,⟨prev,false⟩) then
CP19     if prev.prev.d = true then continue;
CP20     break;
CP21  Back-Off
CP22  return prev;
```



```
procedure SetMark(link: pointer to pointer to Node)
SM1   while true do
SM2      node:=*link;
SM3      if node.d = true or CAS(link,node,⟨node.p,true⟩) then break;

procedure TerminateNode(node: pointer to Node)
TN1   node.prev:=⊥;
TN2   node.next:=⊥;

procedure CleanUpNode(node: pointer to Node)
CU1   while true do
CU2      prev:=node.prev;
CU3      if prev.prev.d = false then break
CU4      prev2:=prev.prev;
CU5      CAS(&node.prev,⟨prev.p,true⟩,⟨prev2.p,true⟩);
CU6   while true do
CU7      next:=node.next;
CU8      if next.next.d = true then break
CU9      next2:=next.next;
CU10     CAS(&node.next,⟨next.p,true⟩,⟨next2.p,true⟩);
```



```
union Link
        _: word
        ⟨p, d⟩: ⟨pointer to Node, boolean⟩

structure Node
        value: pointer to word
        prev: union Link
        next: union Link

// Global variables
head, tail: pointer to Node
// Local variables
node,prev,prev2,next,next2,lastlink: pointer to Node

function CreateNode(value: pointer to word):pointer to Node
CN1   node:=NEW_NODE();
CN2   node.value:=value;
CN3   return node;
```



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