Lockless Algorithms

- CAS based algorithms
- stack
- order linked list
- memory management (hazard pointers)

Obstruction, Lock and Wait Free Methods

- *obstruction free* – method guaranteed to complete in some bounded number of program steps if no other thread executes any steps during that same interval [easiest]
- *lock free* – a method M is said to be lock free if *some* thread is guaranteed to make progress in some bounded number of M's program steps
- *wait free* – if method M is guaranteed to complete in some bounded number of its own program steps - bound need not be statically known [hardest]

Obstruction, Lock and Wait Free Methods...

- a lock based solution is not obstruction free, if a thread sleeps holding the lock then NO other thread cannot make progress
- solutions based on CAS or LL/SC are normally lock free; the only way to prevent CAS succeeding is if some other CAS succeeds meaning that some other thread is making progress; solution may NOT be wait free as a particular thread's CAS may never succeed
- wait free solutions often based on helper functions, if a thread finds itself "blocked" by another thread, it completes the action on behalf of the other thread first [unblocks the blockage!]; implementations often idempotent as many threads may try to perform the same action which must only be effectively executed once.
- linearization point – instruction where method takes effect [eg. marking a node when removing node from concurrent CAS based ordered linked list]

LOCKLESS ALGORITHMS

Lockless Stack

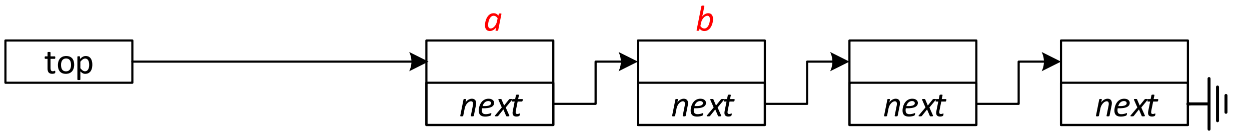
```
Node *top = NULL;      // top of stack

void push(Node *n) {
    do {
        Node *o = top;   // copy pointer
        n->next = o;
    } while (CAS(&top, o, n) == 0);
}
```

```
Node* pop() {
    Node *o, *n;
    do {
        o = top;            // copy pointer
        if (o == NULL)
            return NULL;
        n = o->next;
    } while (CAS(&top, o, n) == 0);
    return o;
}
```

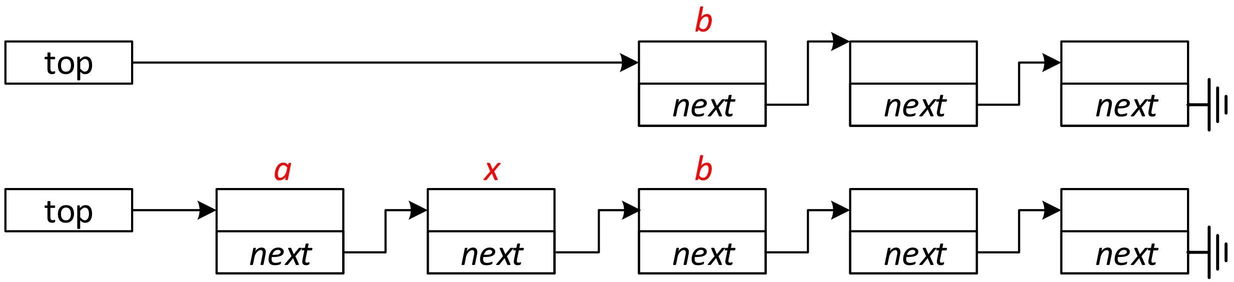
- CAS returns 1 if successful
- threads can push and pop nodes “concurrently”
- what can possibly go wrong? algorithm suffers from the **ABA** problem

Lockless Stack

- imagine the following stack and execution interleave...



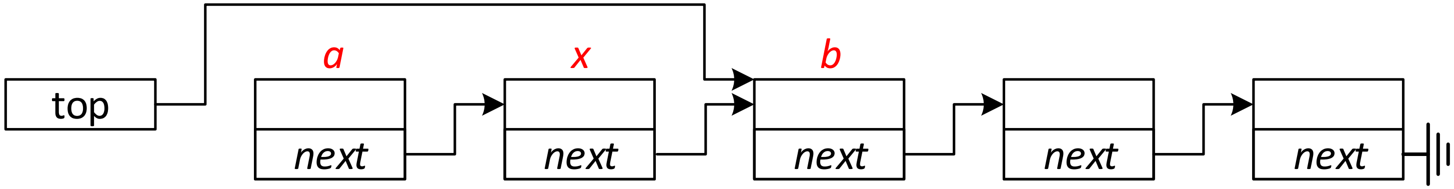
- thread 0 executes pop(), but gets <u>pre-empted</u> after executing n = o->next [*n = b*]
- thread 1 now pops node *a* from stack and <u>then</u> pushes nodes *x* and *a*, REUSING node *a*



- thread 0 is <u>then</u> rescheduled and executes its CAS which will succeed!

Lockless Stack

- CAS succeeds, BUT stack is left in the following state



- node *a* returned, but "two nodes popped from stack"
- called the ABA problem because *top* is assigned A, then B and then A again
    - the different A is not detected
- the Trieber [1986] stack uses a sequence count embedded in the top-of-stack pointer to avoid the ABA problem [counted pointer]
- needs a double length CAS - DCAS [IA32 - cmpxchg8b, x64 - cmpxchg16b]

cmpxchg8b (IA32) / cmpxchg16b (x64)

Operation

```
IF (64-Bit Mode and OperandSize = 64)
   THEN
      TEMP128 ← DEST
      IF (RDX:RAX = TEMP128)
         THEN
            ZF ← 1;
            DEST ← RCX:RBX;
         ELSE
            ZF ← 0;
            RDX:RAX ← TEMP128;
            DEST ← TEMP128;
            FI;
      FI
   ELSE
      TEMP64 ← DEST;
      IF (EDX:EAX = TEMP64)
         THEN
            ZF ← 1;
            DEST ← ECX:EBX;
         ELSE
            ZF ← 0;
            EDX:EAX ← TEMP64;
            DEST ← TEMP64;
            FI;
      FI;
FI;
```

DCAS(a, e , n)

a typically in a register rsi or rdi

rdx:rax = e

rcx:rbx = n

DCAS(a, e , n)

a typically in a register esi or edi

edx:eax = e

ecx:ebx = n

LOCKLESS ALGORITHMS

Trieber Lockless Stack

```
<Node*, int> top;     // Node* and count

void push(Node *n) {
    do {
        <o, c> = top;    // take atomic copy
        n->next = o;
    } while (CAS(&top, o, n) == 0);
}
```

```
Node* pop() {
    do {
        <o, c> = top;  // take atomic copy
        if (o == NULL)
            return NULL;
        n = o->next;
    } while (DCAS(&top, <o, c>, <n, c+1>) == 0);
    return o;
}
```

- pseudo C/C++
- count incremented each time a node popped from stack
- count wrap around is potentially a problem

- DCAS not necessary in push, CAS used instead
- original push code works with an ABA sequence - it doesn’t matter if the first Node has changed as always pushing on to front of stack

Trieber Lockless Stack…

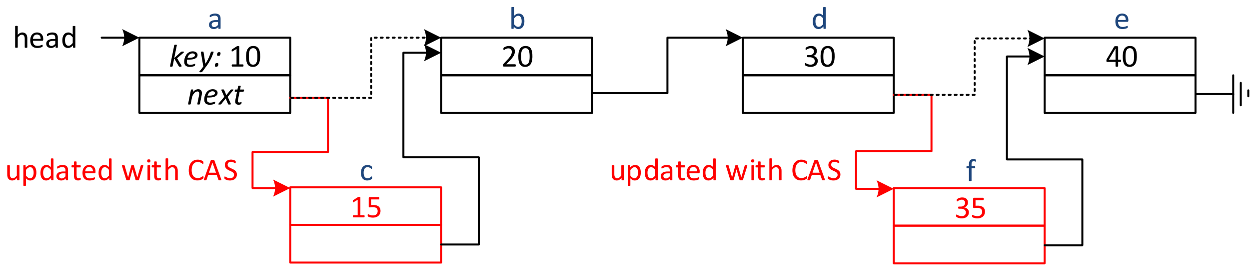
- lockless, but not much concurrency as access to *top* is a serious bottleneck
- is algorithm obstruction, lock or wait free?
- lock free since a thread could be in an endless loop trying to push a Node on to stack, BUT for its CAS to fail another thread must be making progress
- ABA problem will not occur if algorithm implemented using LL/SC – why?
  overwrite of top always detected
- alternatively, don't reuse node until threads don't have and cannot get a pointer to the node [discussed later in lockless ordered list implementation]

Lockless List

- ordered linked list or set
    - add(key)
    - remove(key)
- support concurrent *add(key)* and *remove(key)* operations
- would like number of operations per second to increase linearly with the number of threads
- need to consider memory management
    - if memory allocation / dealloction [new, delete, malloc and free] NOT lockless it could be a bottleneck
- can be hard to reason about an algorithm that works on a list which is concurrently being modified by other threads
- quite a challenge

Using CAS to add nodes

- use CAS to add nodes 15 and 35



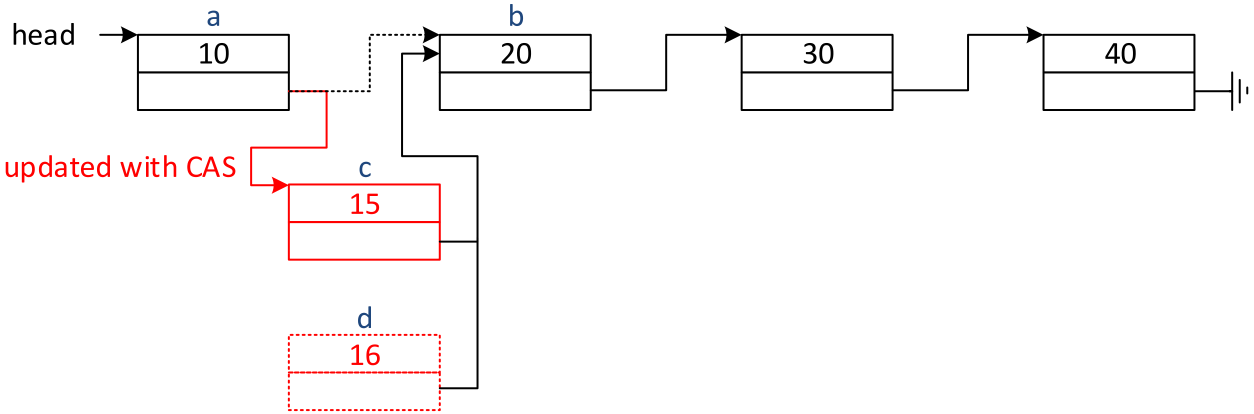
- search for insertion point, initialise next pointer and then execute CAS with correct parameters to insert node into list

```
CAS(&a->next, b, c);        // add node c between a and b
CAS(&d->next, e, f);        // add node f between d and e
```

- *disjoint-access parallelism*

Using CAS to add nodes...

- if 2 threads try to add nodes at the same position



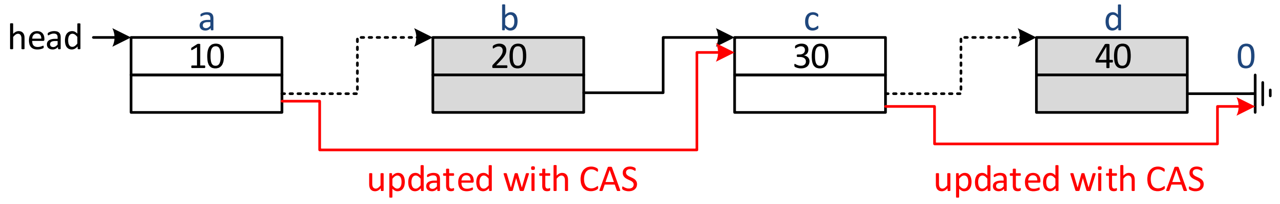
CAS(&a->next, b, c) *// first CAS executed will succeed…*
CAS(&a->next, b, d) *// and consequently second CAS executed will FAIL*

- first CAS executed succeeds, second will fail as *a->next != b*

- <u>RETRY on failure</u>, which means searching for insertion point AGAIN [costly if list long] and, if key not found, set up and re-execute CAS

Using CAS to remove nodes

- search for node and then execute CAS with correct parameters to remove node from list
- consider 2 threads removing non-adjacent nodes

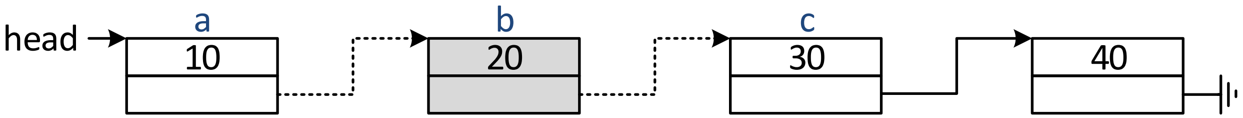


```
CAS(&a->next, b, c)        // remove node b (20)
CAS(&c->next, d, 0)        // remove node d (40)
```

- disjoint access parallelism

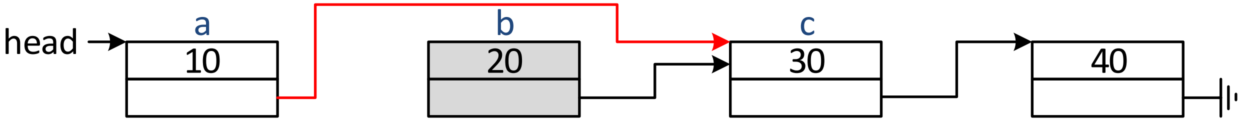
Using CAS to remove nodes...

- if two threads try to remove the same node



CAS(&a->next, b, c)
CAS(&a->next, b, c)

- first CAS executed succeeds

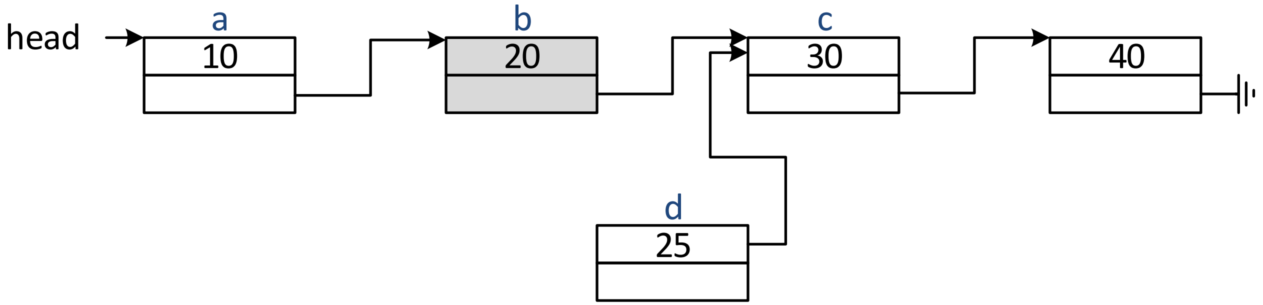


- second CAS executed fails as *a->next != b*

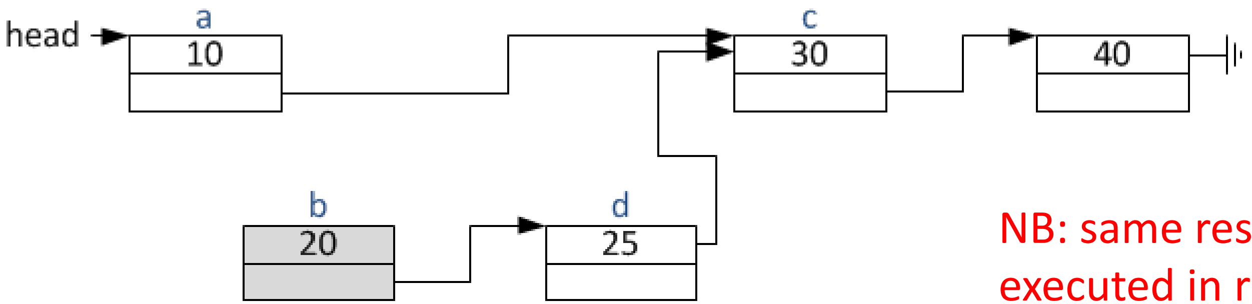
- RETRY on failure, which means searching AGAIN for node [which may not be found]

What can go wrong with remove?

- imagine removing node 20 and adding node 25 concurrently



```
CAS(&a->next, b, c);          // remove 20
CAS(&b->next, c, d);          // add 25
```

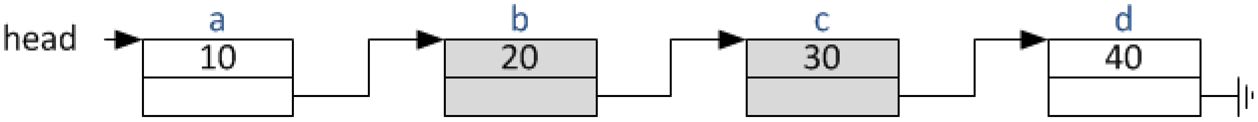


NB: same result if CAS instructions executed in reverse order

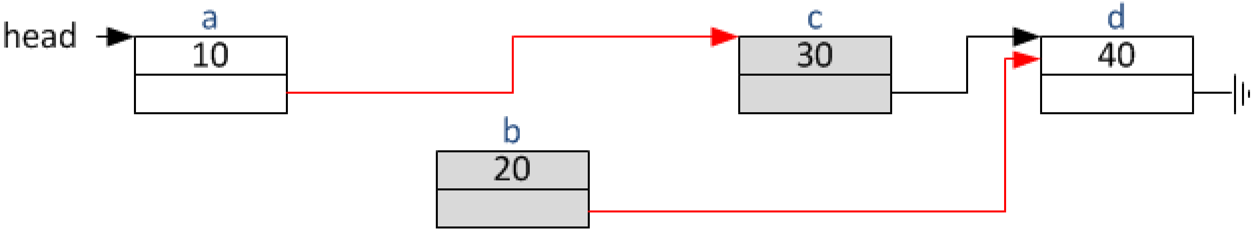
- NOT what was intended!

What else can go wrong with remove?

- consider deleting adjacent nodes



CAS(&a->next, b, c) // remove 20
CAS(&b->next, c, d) // remove 30

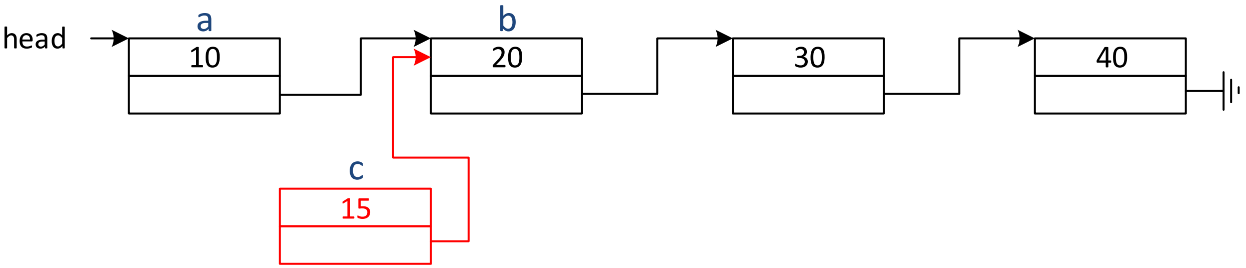


- AGAIN NOT what was intended

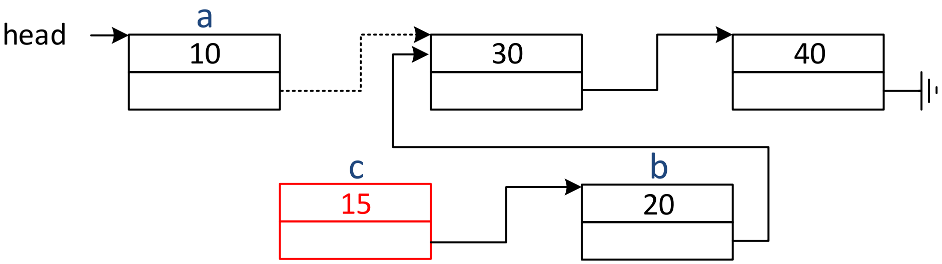
NB: similar result if CAS instructions executed in reverse order (nodes 20 and 30 swapped)

ABA Problem Again

- imagine insertion point found, BUT before CAS(&a->next, b, c) is executed, thread is pre-empted



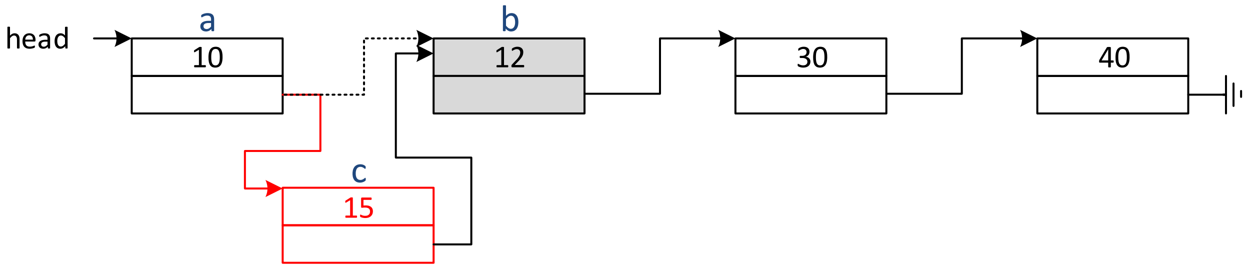
- another thread then removes b from list



- if thread adding 15 resumes execution, the CAS fails which is OK in this case
- BUT need to consider an alternative interleave where node b is reused

ABA Problem Again...

- if the memory used by b is reused, for example by a thread adding key 12 to the list before thread adding 15 resumes…

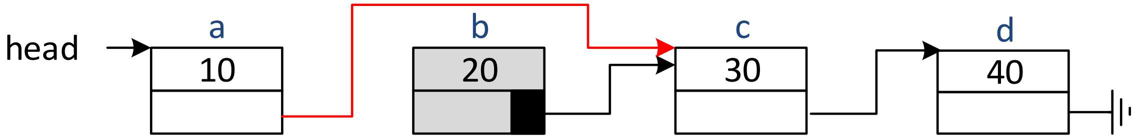


- when the thread adding 15 to list resumes, its CAS will succeed and 15 will be added into the list at the wrong position
- this is the ABA problem again
- nodes cannot be reused if any thread has or can get a pointer to the node

LOCKLESS ALGORITHMS

Lockless List

- used ideas from *A Pragmatic Implementation of Non-Blocking Linked Lists*, Tim Harris [2001], but code from...
- *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, Maged M. Michael [2004] [see Figure 9 in paper]
- initially ignore the ABA problem by not reusing nodes [will quickly run out of memory]
- two step removal eg. remove(20)

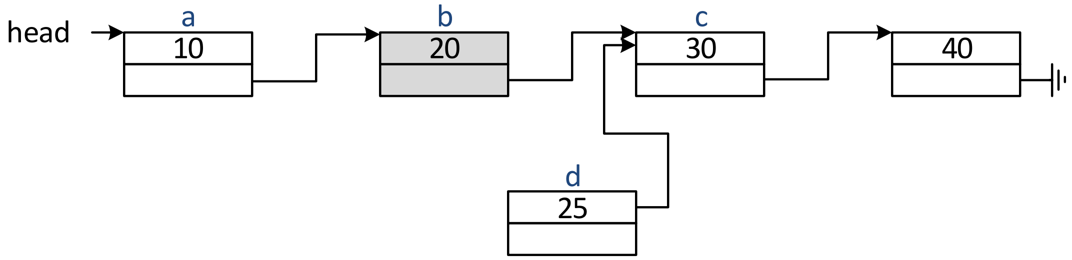


(1) atomically mark node by setting LSB of next pointer [logically removes node]
(2) remove node by updating next pointer using CAS

- avoids problem shown in slides 11 and 12 by detecting attempts to update the next field of a removed node

Revisit adding node [25] and removing node [20]?

- imagine adding node [25] and removing node [20] concurrently

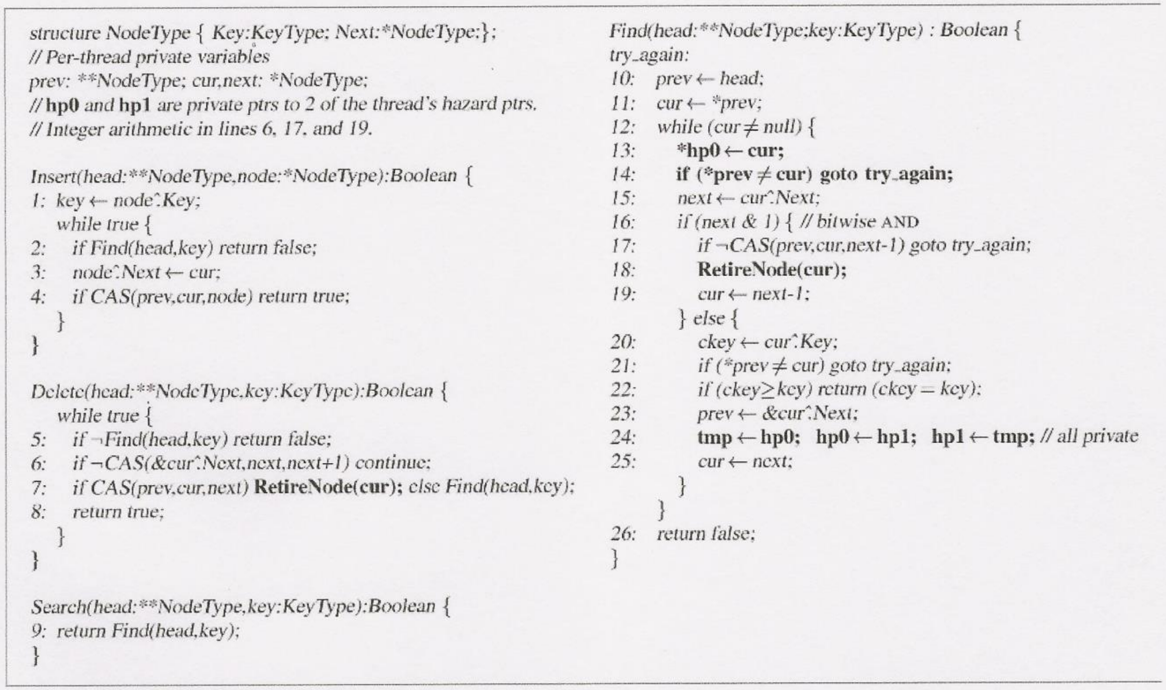


```
(1) CAS(&b->next, c, d);                    // add 25

    and

(2) if (CAS(&b->next, c, c + 1) == 1)   // MARK node b and then…
(3)     CAS(&a->next, b, c);                // remove b [20]
```

- if (1) executed first, (2) will fail as *b->next != c*
- if (2) executed first, (1) will fail as *b->next != c*
- if (3) fails, it means that a no longer points to b, BUT b is logically marked and can be removed later [OK for list to temporarily contain MARKED nodes]

Lockless List...

- Fig 9. from *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, Maged M. Michael [2004]



```
structure NodeType { Key:KeyType; Next:*NodeType;};
// Per-thread private variables
prev: **NodeType; cur,next: *NodeType;
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.
// Integer arithmetic in lines 6, 17, and 19.

Insert(head:**NodeType,node:*NodeType):Boolean {
1:  key ← node^.Key;
    while true {
2:    if Find(head,key) return false;
3:    node^.Next ← cur;
4:    if CAS(prev,cur,node) return true;
    }
}

Delete(head:**NodeType,key:KeyType):Boolean {
    while true {
5:    if ¬Find(head,key) return false;
6:    if ¬CAS(&cur^.Next,next,next+1) continue;
7:    if CAS(prev,cur,next) RetireNode(cur); else Find(head,key);
8:    return true;
    }
}

Search(head:**NodeType,key:KeyType):Boolean {
9:  return Find(head,key);
}

Find(head:**NodeType;key:KeyType) : Boolean {
try_again:
10:   prev ← head;
11:   cur ← *prev;
12:   while (cur ≠ null) {
13:     *hp0 ← cur;
14:     if (*prev ≠ cur) goto try_again;
15:     next ← cur^.Next;
16:     if (next & 1) { // bitwise AND
17:       if ¬CAS(prev,cur,next-1) goto try_again;
18:       RetireNode(cur);
19:       cur ← next-1;
        } else {
20:       ckey ← cur^.Key;
21:       if (*prev ≠ cur) goto try_again;
22:       if (ckey ≥ key) return (ckey = key);
23:       prev ← &cur^.Next;
24:       tmp ← hp0;  hp0 ← hp1;  hp1 ← tmp; // all private
25:       cur ← next;
        }
      }
26:   return false;
}
```

Fig. 9. Lock-free list-based set with hazard pointers.

Lockless List...

- Node class
  - int key
  - Node *next
- List implemented using global variable *head* and functions *add, remove and find*
  - Node *head // head of list
  - int add(Node **head, Node*node) // insert node
  - int remove(Node **head, int key) // remove node with key
  - int find(Node **head, int key) // find with key
- per thread local variables
  - prev **Node
  - Node *cur
  - Node *next

Lockless List...

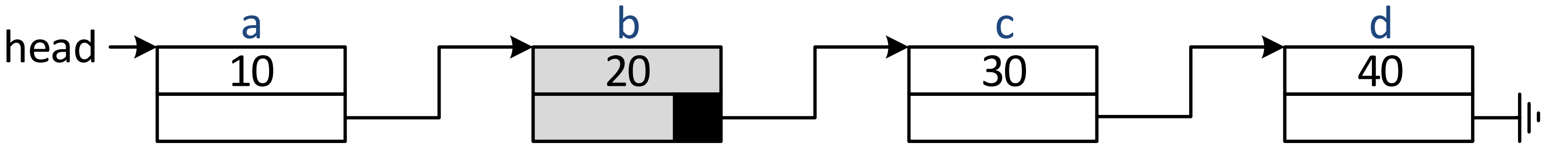
- MARKED node indicated by an ODD address in its next field
- OK as addresses normally aligned on at least a 4 byte boundary [2 or 3 LSBs normally 0]

- handle marked nodes as follows

```
if (n->next & 1) ...          // tests if node n MARKED
CAS(&n->next, v, v+1)         // MARK node n (assumes node NOT MARKED)
CAS(&n->next, v, v-1);        // UNMARK node n (assumes node MARKED)
```

- to atomically mark node b [logically remove] use

```
CAS(&b->next, c, c + 1);      // assumes node UNMARKED
```



find()

```
int find(Node **head, int key) {        // find insertion point or node to remove
retry:                                  // NB: thread local variables prev, cur and next
    prev = head;                        // NB: Node **prev, Node **head;
    cur = *prev;                        // *prev and hence cur will be unmarked

    while (cur != NULL) {

        next = cur->next;               // cur unmarked

        if (next & 1) {

            if (CAS(prev, cur, next-1) == 0)
                goto retry;
            cur = next-1;               // cur unmarked

        } else {

            int ckey = cur->key;
            if (*prev != cur)           // optimisation?? will fail sooner?
                goto retry;             // make sure key still in list and no nodes added between prev and cur OTHERWISE retry
            if (ckey >= key)
                return (ckey == key);
            prev = &cur->next;
            cur = next;

        }
    }
    return 0;
}
```

continue until end of list

test if marked node

try to remove marked node

move to next node

make copy of key

check that *prev == cur

return 1 if key found, 0 otherwise

move to next node

LOCKLESS ALGORITHMS

add()

```
int add(Node **head, Node *node)  {
    key = node->key;                // NB: thread local variables prev, cur and next
    while (1) {
        if (find(head, key))
            return 0;
        node->next = cur;
        if (CAS(prev, cur, node) == 1)
            return 1;
    }
}
```

return 0 if key already in list

set up new node's next pointer

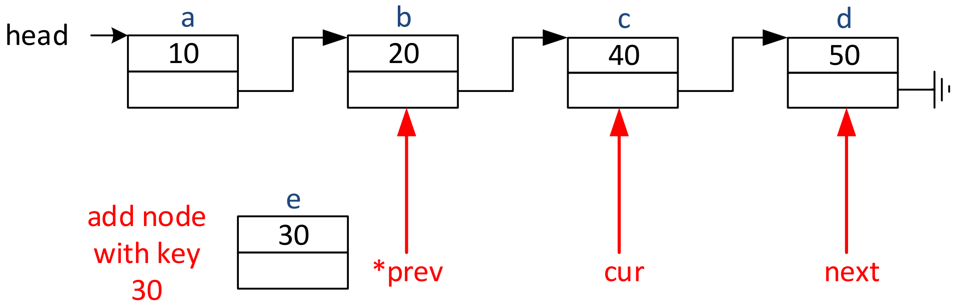
add node between prev and cur

returns 1 on success

keep trying until successful

add() …

- add(key) calls the find() function



NB: Node **prev;
Node *cur, *next;

- find() returns thread local pointers such that the new node should be added between **prev* and *cur*
- if CAS(prev, cur, node) succeeds, it must mean that prev still pointed to cur [nodes have not been added between prev and cur]
- a node CANNOT be added by linking to a MARKED node [logically removed] thus avoiding the problem discussed in slides 12 and 13

LOCKLESS ALGORITHMS

remove()

```
int remove(Node ** head, int key) {
    while (1) {                        // NB: thread local variables prev, cur and next

        if (find(head, key) == 0)      // cur and prev will be unmarked
            return 0;

        if (CAS(&cur->next, next, next+1) == 0)
                continue;

        if (CAS(prev, cur, next) == 0)
            find(head, key);

        return 1;

    }
}
```

keep trying until successful

returns 1 if key found
returns prev, cur and next
curr Node removed

return 0 if key not in list

try to MARK UNMARKED node
once marked node is logically removed

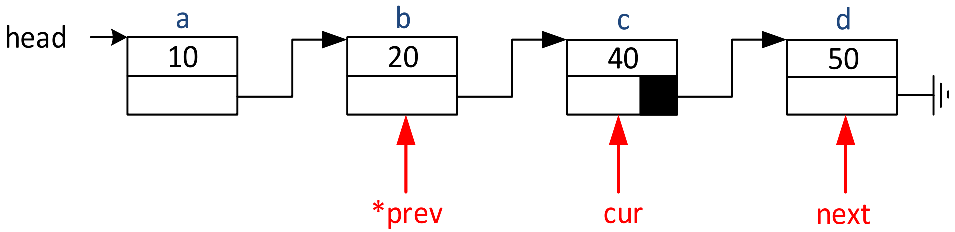
try to remove node from list

if CAS fails, use find() to remove marked node from list

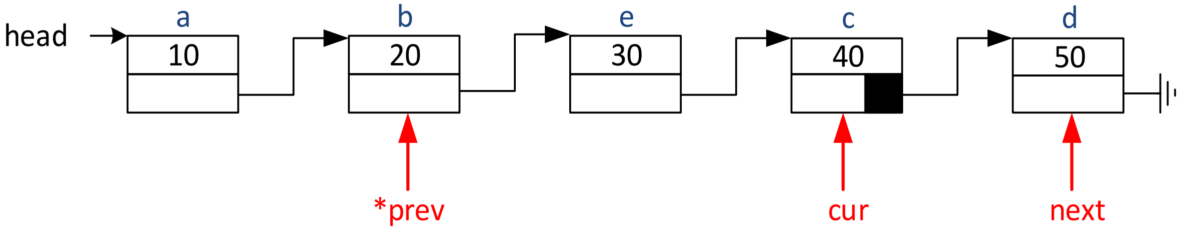
- calls find to remove marked node if CAS fails AND if find fails to remove the marked node, it can be removed by future calls to find (in add and remove)

remove() …

- assume initial search has returned *prev, cur and next AND cur has been MARKED [logically removed]



- imagine that before CAS(prev, cur, next) is executed to remove node, another thread inserted a node between prev and cur



- CAS(prev, cur, next) will FAIL

remove() …

- since node is logically removed, there is NO requirement to ensure that the node is removed from the linked list immediately

- BUT by calling find() again, any MARKED node(s) up to and including *key* will be removed

- NOT calling find() here, simply means that the MARKED node will remain in list until

    - a node is inserted after key

  OR

    - a node after or including key is removed

LOCKLESS ALGORITHMS

What still needs to be done?

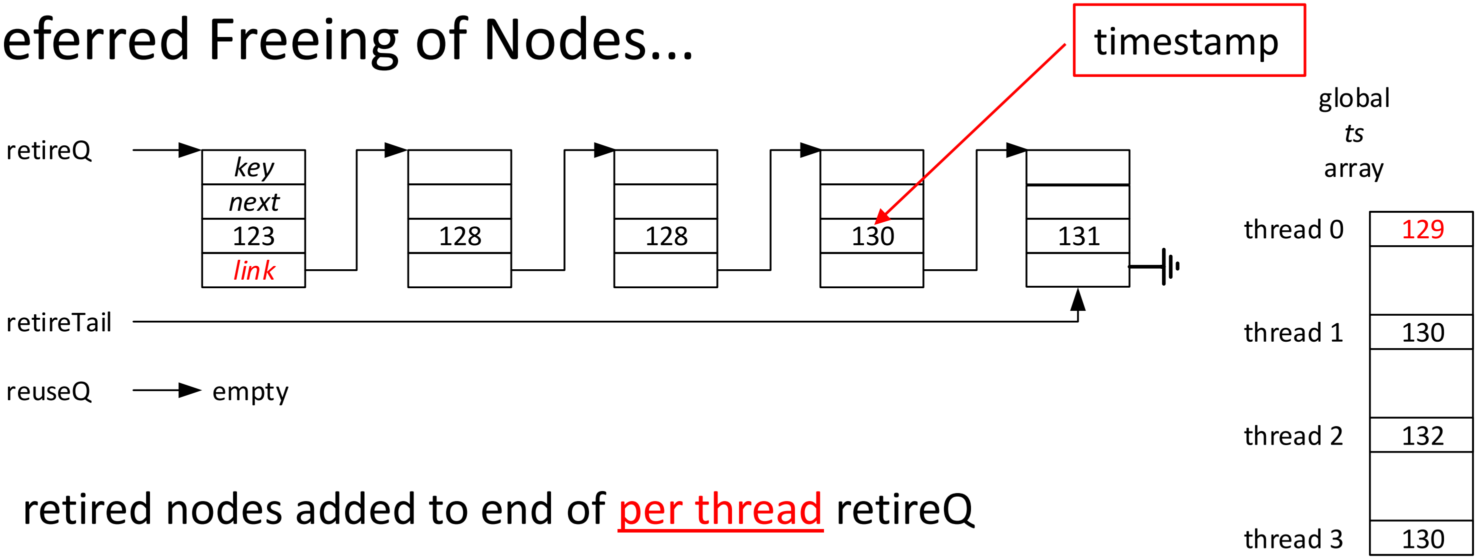
- current solution avoids the ABA problem by NOT re-using nodes
- there is no code for freeing or reusing nodes
- ONLY a partial solution without memory management
    - garbage collection [supported by Java, but not yet by C++]
    - reference counting [perhaps by using smart pointers, reported to be slow]
    - method proposed by Harris [section 6 of paper]
    - hazard pointers [Michael]

Deferred Freeing of Nodes [Harris]

- see end of section 6 in Harris paper
- each node has an additional (1) *link* so that node can be added to a per thread *retireQ* or *reuseQ* and (2) a *timestamp*
- before starting an add or remove operation, each thread obtains a global timestamp and saves it in a global *ts* array indexed by the thread number [best if each timestamp is stored in its own cache line]
- can use clock() or the __rdtsc() intrinsic or … to obtain timestamp
- a remove operation retires the node by adding it to a per thread retireQ and sets the node's timestamp by reading its global timestamp
- when a thread needs a node and the reuseQ is empty, it can traverse the *retireQ* and transfer nodes to the reuseQ if their *timestamp* is less than the minimum *ts* of any thread because this means that NO thread can still have a reference to the node
- allows nodes to be recycled

LOCKLESS ALGORITHMS

Deferred Freeing of Nodes...



- retired nodes added to end of per thread retireQ
- if thread needs a node and reuseQ empty, try to transfer nodes from retireQ to reuseQ
- in example, minimum thread *ts* is 129
- can transfer all nodes in retireQ with *ts* < 129 to per thread reuseQ [first three nodes]
- allocate nodes from reuseQ and ONLY call new if reuseQ empty
- why is a *link* needed? why not use *next*?
  - thread may need to follow next in order to traverse node even when on retireQ

Deferred Freeing of Nodes...

- memory management algorithm is NOT obstruction free

- if a thread pre-empted, its global *timestamp* will NOT change [*stuck*]
- per thread *timestamp* also *stuck* if thread never calls add() or remove()!

  - if thread not running for *20ms*, then *20ms* worth of removed nodes will be added to the retireQs before they can be transferred to the reuseQs

  - can result in many allocated nodes, especially when threads > CPUs

- also need to make sure algorithm works when some threads are producers and others are consumers

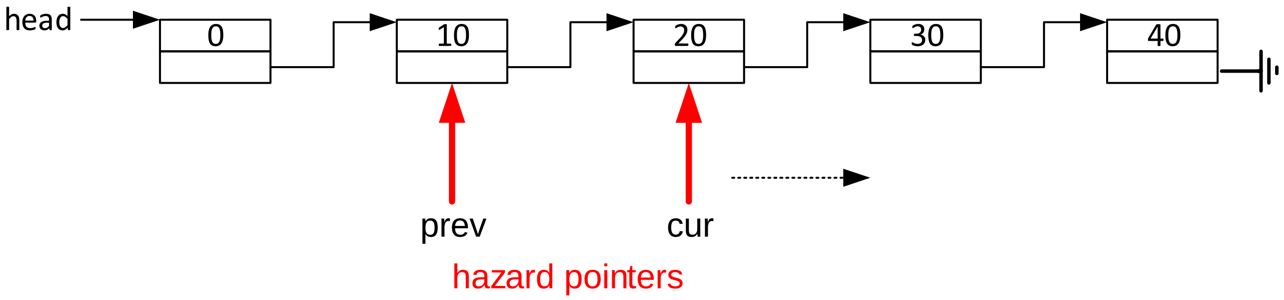
  - nodes added to consumer reuseQs needed by producers
  - need to free nodes on reuseQ so nodes can be reused by producers
  - nodes recirculated

- implementation simplified by using per thread Qs

LOCKLESS ALGORITHMS

Hazard Pointers

- *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*
- with an ordered linked list, two active pointers are used to traverse the list during a find operation [number of active pointers depends on algorithm]
- corresponds to the prev and cur pointers



- at each step, copy prev and cur into an array of per thread hazard pointers
- idea is not to reuse or free nodes if they have hazard pointers pointing to them
- once a node has been removed from list, no thread is able to get a pointer to the node unless it has a pointer to it already

LOCKLESS ALGORITHMS

Hazard Pointers…

- maintain a global array of per thread hazard pointers [best if each thread saves its hazard pointers in its own cache line(s)]

- use a per thread retireQ and reuseQ as in previous example

- a thread retires a node by adding it to its retireQ

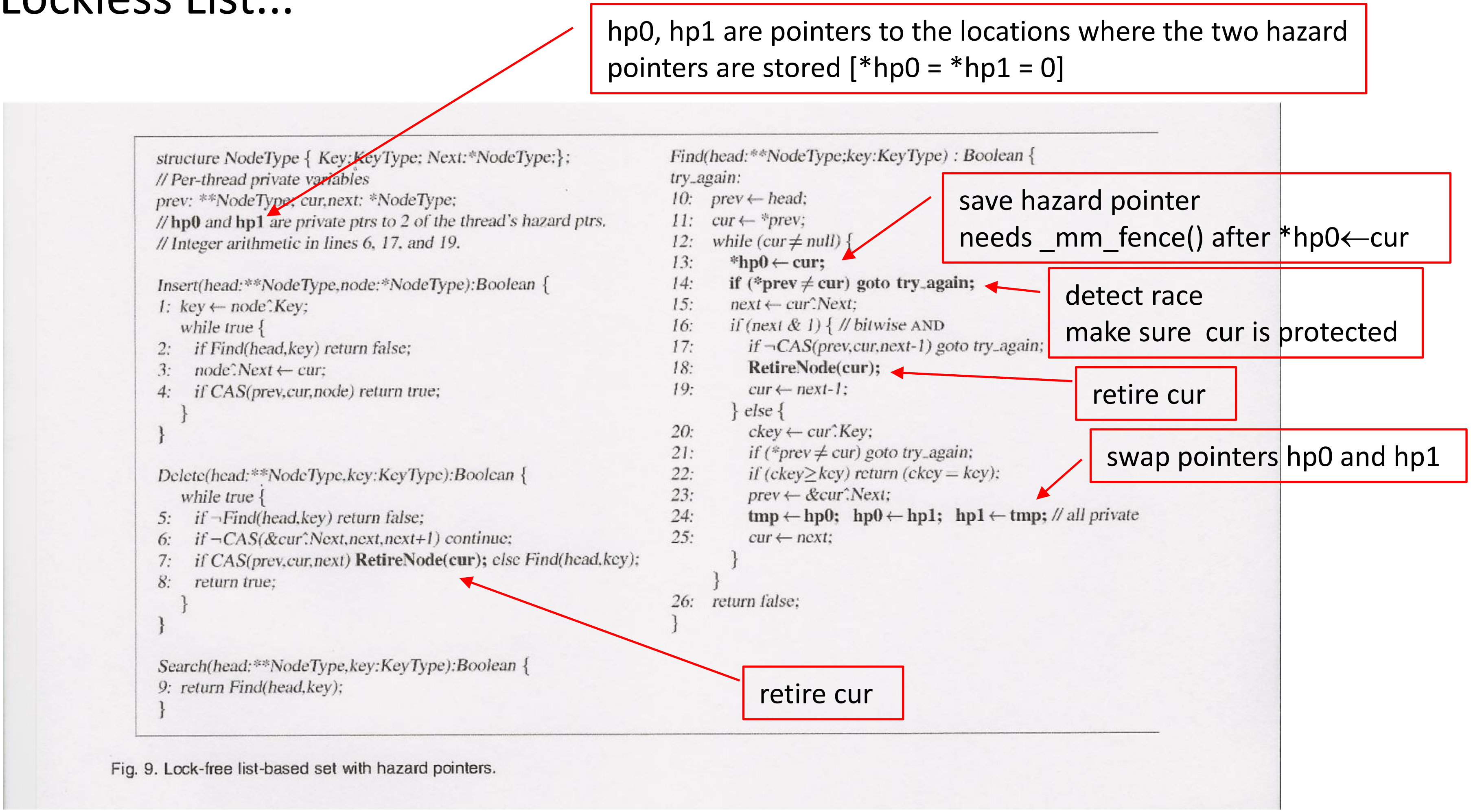
- when the length of its retireQ is >= 2*nthreads*HAZARDSPERTHREAD
  - take a local snapshot of ALL hazard pointers and store in a local array
  - optionally sort hazard pointers [in local array]
  - for each node on its retireQ, if address of node doesn't match any of the hazard pointers in the local array then transfer to reuseQ [at least half of the nodes should be transferred]
  - allows nodes to be recycled

- only need to call new if per thread reuseQ is empty

- can delete nodes instead of placing them on reuseQ

LOCKLESS ALGORITHMS

Lockless List...



hp0, hp1 are pointers to the locations where the two hazard pointers are stored [*hp0 = *hp1 = 0]

```
structure NodeType { Key:KeyType; Next:*NodeType;};
// Per-thread private variables
prev: **NodeType; cur,next: *NodeType;
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs.
// Integer arithmetic in lines 6, 17, and 19.

Insert(head:**NodeType,node:*NodeType):Boolean {
1:  key ← node^.Key;
    while true {
2:    if Find(head,key) return false;
3:    node^.Next ← cur;
4:    if CAS(prev,cur,node) return true;
    }
}

Delete(head:**NodeType,key:KeyType):Boolean {
    while true {
5:    if ¬Find(head,key) return false;
6:    if ¬CAS(&cur^.Next,next,next+1) continue;
7:    if CAS(prev,cur,next) RetireNode(cur); else Find(head,key);
8:    return true;
    }
}

Search(head:**NodeType,key:KeyType):Boolean {
9:  return Find(head,key);
}

Find(head:**NodeType;key:KeyType) : Boolean {
try_again:
10:   prev ← head;
11:   cur ← *prev;
12:   while (cur ≠ null) {
13:     *hp0 ← cur;
14:     if (*prev ≠ cur) goto try_again;
15:     next ← cur^.Next;
16:     if (next & 1) { // bitwise AND
17:       if ¬CAS(prev,cur,next-1) goto try_again;
18:       RetireNode(cur);
19:       cur ← next-1;
        } else {
20:       ckey ← cur^.Key;
21:       if (*prev ≠ cur) goto try_again;
22:       if (ckey ≥ key) return (ckey = key);
23:       prev ← &cur^.Next;
24:       tmp ← hp0;  hp0 ← hp1;  hp1 ← tmp; // all private
25:       cur ← next;
        }
      }
26:   return false;
}
```

Fig. 9. Lock-free list-based set with hazard pointers.

save hazard pointer
needs _mm_fence() after *hp0←cur

detect race
make sure cur is protected

retire cur

swap pointers hp0 and hp1

retire cur

- hazard pointers used to protect prev and cur

LOCKLESS ALGORITHMS

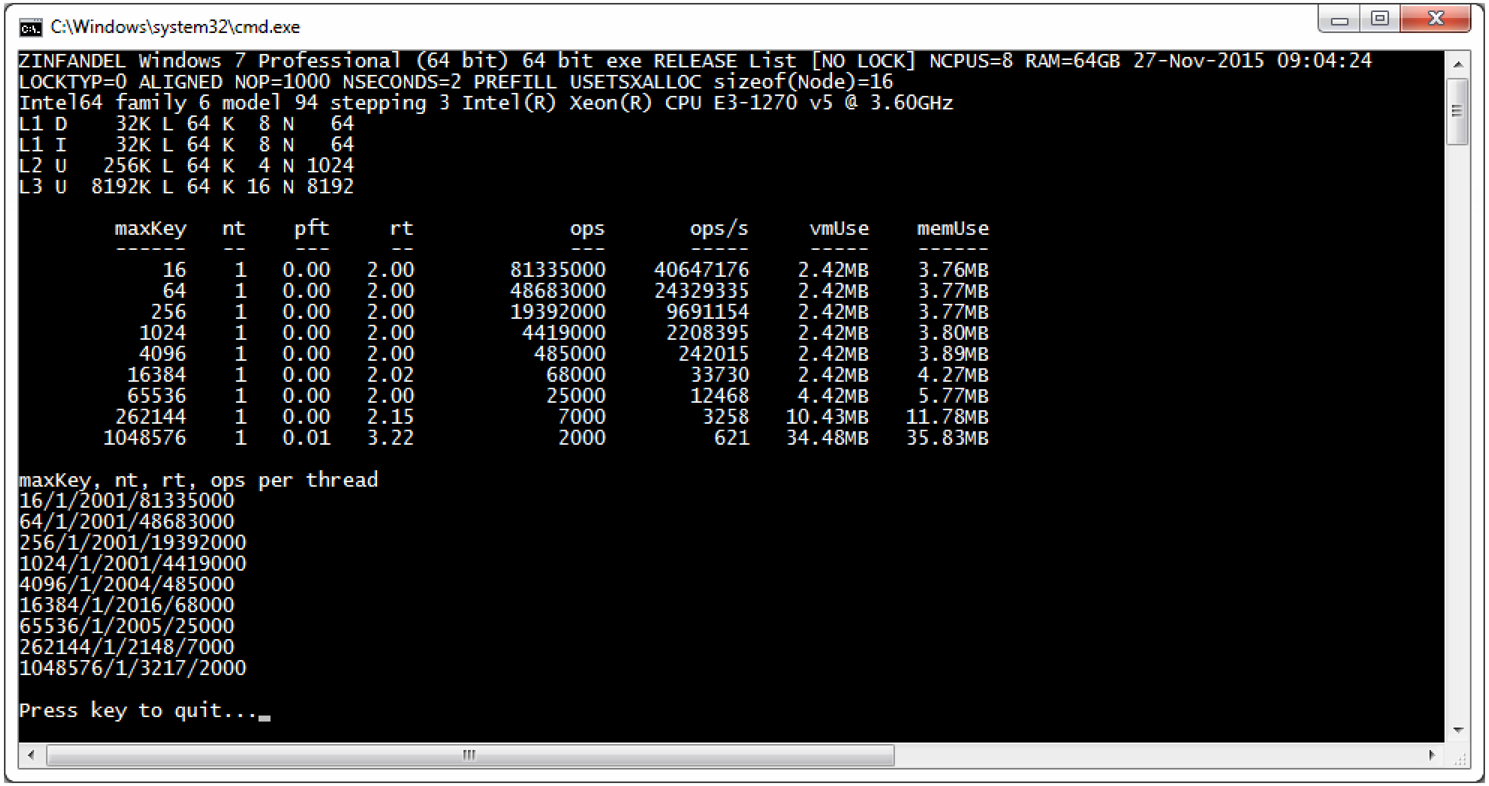
Lockless List...

- detect race

detect race
make sure cur is protected

- between cur being assigned [cur = …] and protected by hazard pointer [*hp0 = cur], cur could be moved to the retireQ or reuseQ, reused or even freed
- most straightforward way to make sure cur is protected by hazard pointer is to check that it is still in the list [*prev == cur]
- if cur is reused between cur = … and *hp0 = cur AND *prev == cur, it is of NO consequence because at this particular point in the algorithm a comparison has not been made with cur->key
- if cur is freed between cur = … and *hp0 = cur, accessing the node pointed to by cur could be result in a invalid memory access
- the _mm_fence() is to make sure that the hazard pointer is visible to ALL other threads thus protecting cur

Baseline Performance Single Threaded Ordered List



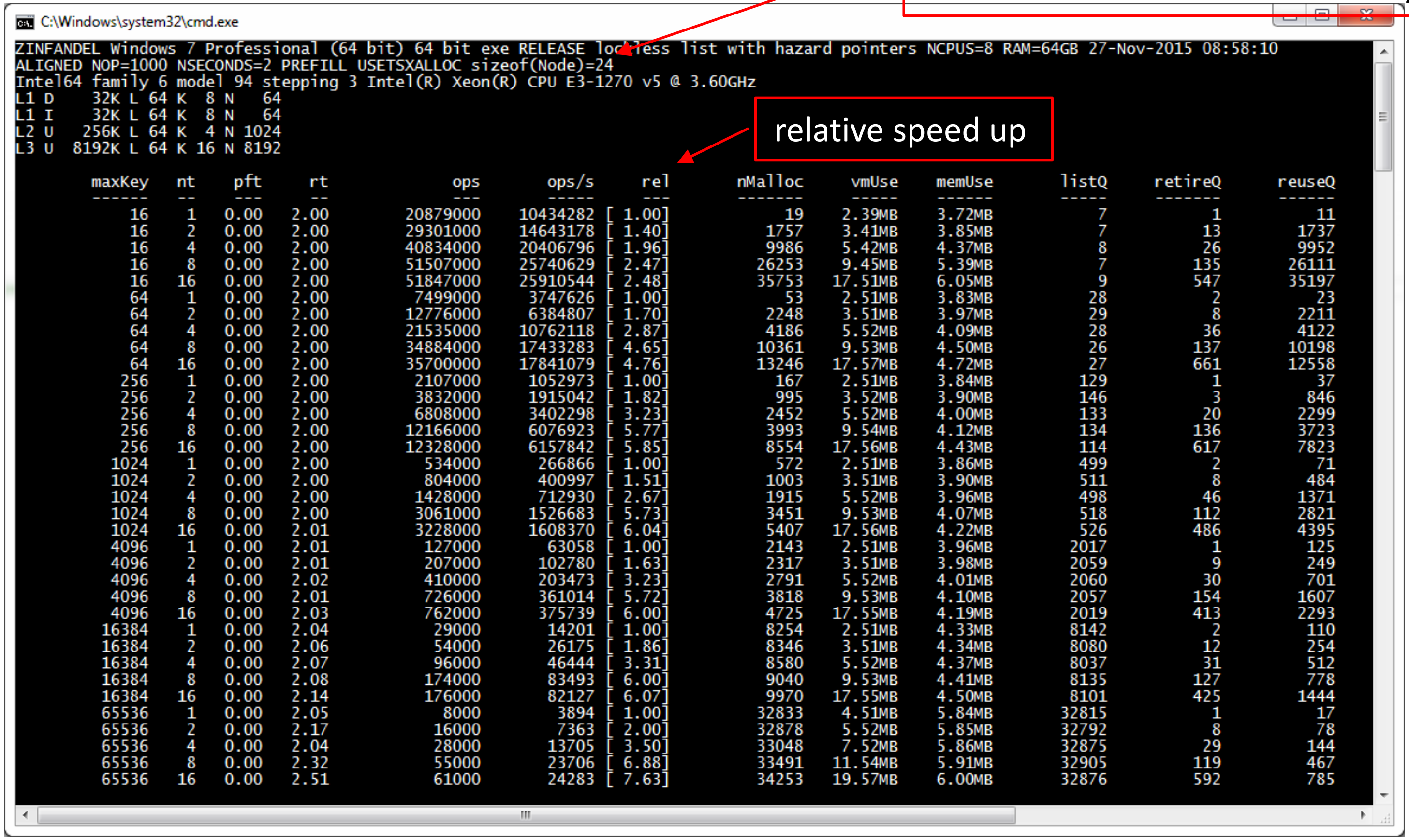
```
ZINFANDEL Windows 7 Professional (64 bit) 64 bit exe RELEASE List [NO LOCK] NCPUS=8 RAM=64GB 27-Nov-2015 09:04:24
LOCKTYP=0 ALIGNED NOP=1000 NSECONDS=2 PREFILL USETSXALLOC sizeof(Node)=16
Intel64 family 6 model 94 stepping 3 Intel(R) Xeon(R) CPU E3-1270 v5 @ 3.60GHz
L1 D    32K L 64 K  8 N   64
L1 I    32K L 64 K  8 N   64
L2 U   256K L 64 K  4 N 1024
L3 U  8192K L 64 K 16 N 8192

        maxKey   nt    pft      rt              ops        ops/s     vmUse     memUse
        ------   --    ---      --              ---        -----     -----     ------
            16    1   0.00    2.00         81335000     40647176    2.42MB     3.76MB
            64    1   0.00    2.00         48683000     24329335    2.42MB     3.77MB
           256    1   0.00    2.00         19392000      9691154    2.42MB     3.77MB
          1024    1   0.00    2.00          4419000      2208395    2.42MB     3.80MB
          4096    1   0.00    2.00           485000       242015    2.42MB     3.89MB
         16384    1   0.00    2.02            68000        33730    2.42MB     4.27MB
         65536    1   0.00    2.00            25000        12468    4.42MB     5.77MB
        262144    1   0.00    2.15             7000         3258   10.43MB    11.78MB
       1048576    1   0.01    3.22             2000          621   34.48MB    35.83MB

maxKey, nt, rt, ops per thread
16/1/2001/81335000
64/1/2001/48683000
256/1/2001/19392000
1024/1/2001/4419000
4096/1/2004/485000
16384/1/2016/68000
65536/1/2005/25000
262144/1/2148/7000
1048576/1/3217/2000

Press key to quit..._
```

- ALIGNED: each node in its own cache line
- PREFILL: list prefilled with odd integers for quick start up
- 50% add random key and 50% remove random key
- decreasing ops/s as list length increases [average list length maxKey/2]

LOCKLESS ALGORITHMS

Lockless list with Hazard Pointers

node needs a link field as it can effectively be in list and on retireQ simultaneously



C:\Windows\system32\cmd.exe

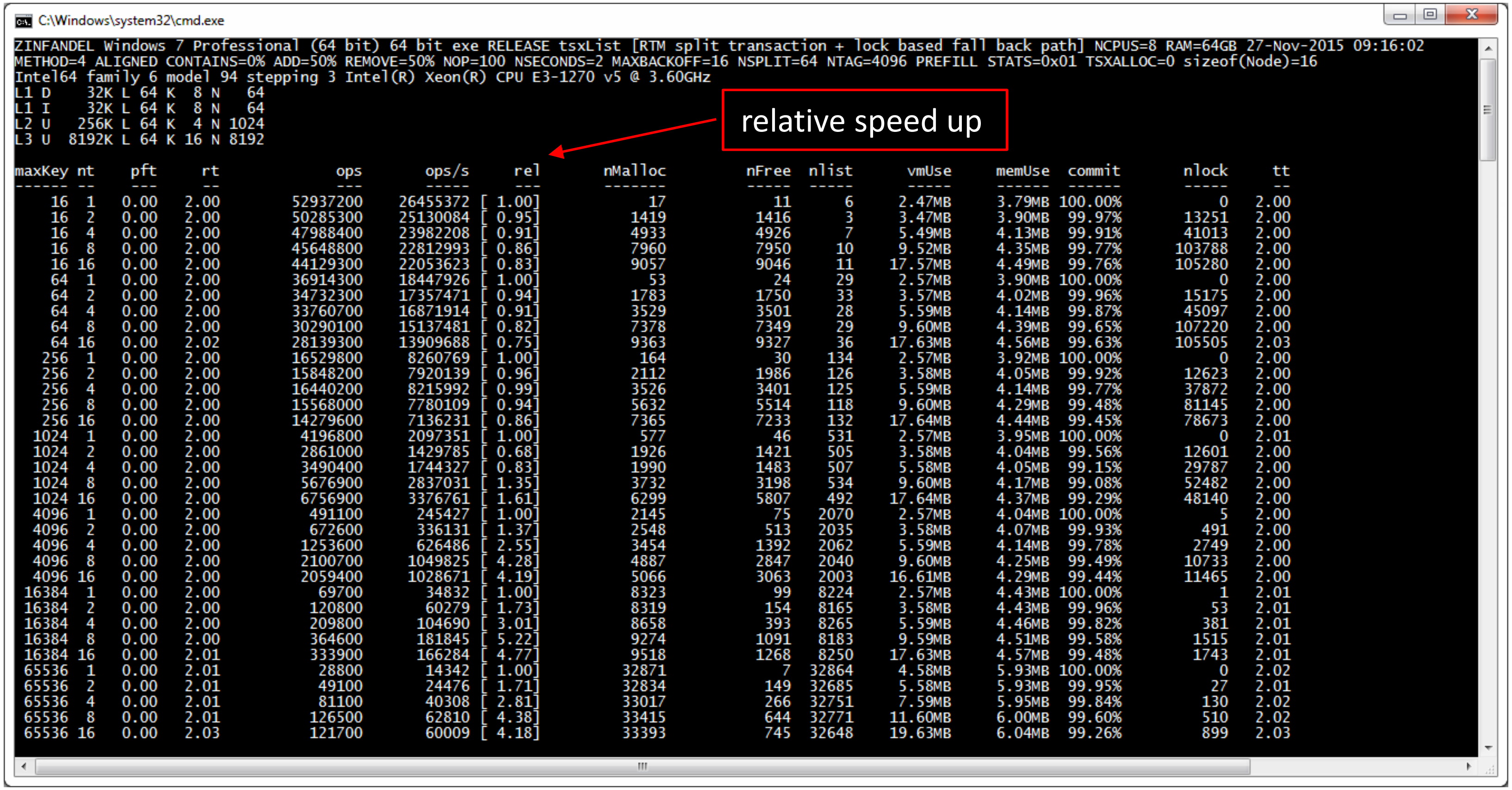
```
ZINFANDEL Windows 7 Professional (64 bit) 64 bit exe RELEASE lockless list with hazard pointers NCPUS=8 RAM=64GB 27-Nov-2015 08:58:10
ALIGNED NOP=1000 NSECONDS=2 PREFILL USETSXALLOC sizeof(Node)=24
Intel64 family 6 model 94 stepping 3 Intel(R) Xeon(R) CPU E3-1270 v5 @ 3.60GHz
L1 D    32K L 64 K  8 N   64
L1 I    32K L 64 K  8 N   64
L2 U   256K L 64 K  4 N 1024
L3 U  8192K L 64 K 16 N 8192

       maxKey   nt    pft    rt            ops      ops/s     rel     nMalloc     vmUse    memUse     listQ   retireQ     reuseQ
       ------   --    ---    --            ---      -----     ---     -------     -----    ------     -----   -------     ------
           16    1   0.00  2.00       20879000   10434282 [ 1.00]          19    2.39MB    3.72MB         7         1          11
           16    2   0.00  2.00       29301000   14643178 [ 1.40]        1757    3.41MB    3.85MB         7        13        1737
           16    4   0.00  2.00       40834000   20406796 [ 1.96]        9986    5.42MB    4.37MB         8        26        9952
           16    8   0.00  2.00       51507000   25740629 [ 2.47]       26253    9.45MB    5.39MB         7       135       26111
           16   16   0.00  2.00       51847000   25910544 [ 2.48]       35753   17.51MB    6.05MB         9       547       35197
           64    1   0.00  2.00        7499000    3747626 [ 1.00]          53    2.51MB    3.83MB        28         2          23
           64    2   0.00  2.00       12776000    6384807 [ 1.70]        2248    3.51MB    3.97MB        29         8        2211
           64    4   0.00  2.00       21535000   10762118 [ 2.87]        4186    5.52MB    4.09MB        28        36        4122
           64    8   0.00  2.00       34884000   17433283 [ 4.65]       10361    9.53MB    4.50MB        26       137       10198
           64   16   0.00  2.00       35700000   17841079 [ 4.76]       13246   17.57MB    4.72MB        27       661       12558
          256    1   0.00  2.00        2107000    1052973 [ 1.00]         167    2.51MB    3.84MB       129         1          37
          256    2   0.00  2.00        3832000    1915042 [ 1.82]         995    3.52MB    3.90MB       146         3         846
          256    4   0.00  2.00        6808000    3402298 [ 3.23]        2452    5.52MB    4.00MB       133        20        2299
          256    8   0.00  2.00       12166000    6076923 [ 5.77]        3993    9.54MB    4.12MB       134       136        3723
          256   16   0.00  2.00       12328000    6157842 [ 5.85]        8554   17.56MB    4.43MB       114       617        7823
         1024    1   0.00  2.00         534000     266866 [ 1.00]         572    2.51MB    3.86MB       499         2          71
         1024    2   0.00  2.00         804000     400997 [ 1.51]        1003    3.51MB    3.90MB       511         8         484
         1024    4   0.00  2.00        1428000     712930 [ 2.67]        1915    5.52MB    3.96MB       498        46        1371
         1024    8   0.00  2.00        3061000    1526683 [ 5.73]        3451    9.53MB    4.07MB       518       112        2821
         1024   16   0.00  2.01        3228000    1608370 [ 6.04]        5407   17.56MB    4.22MB       526       486        4395
         4096    1   0.00  2.01         127000      63058 [ 1.00]        2143    2.51MB    3.96MB      2017         1         125
         4096    2   0.00  2.01         207000     102780 [ 1.63]        2317    3.51MB    3.98MB      2059         9         249
         4096    4   0.00  2.02         410000     203473 [ 3.23]        2791    5.52MB    4.01MB      2060        30         701
         4096    8   0.00  2.01         726000     361014 [ 5.72]        3818    9.53MB    4.10MB      2057       154        1607
         4096   16   0.00  2.03         762000     375739 [ 6.00]        4725   17.55MB    4.19MB      2019       413        2293
        16384    1   0.00  2.04          29000      14201 [ 1.00]        8254    2.51MB    4.33MB      8142         2         110
        16384    2   0.00  2.06          54000      26175 [ 1.86]        8346    3.51MB    4.34MB      8080        12         254
        16384    4   0.00  2.07          96000      46444 [ 3.31]        8580    5.52MB    4.37MB      8037        31         512
        16384    8   0.00  2.08         174000      83493 [ 6.00]        9040    9.53MB    4.41MB      8135       127         778
        16384   16   0.00  2.14         176000      82127 [ 6.07]        9970   17.55MB    4.50MB      8101       425        1444
        65536    1   0.00  2.05           8000       3894 [ 1.00]       32833    4.51MB    5.84MB     32815         1          17
        65536    2   0.00  2.17          16000       7363 [ 2.00]       32878    5.52MB    5.85MB     32792         8          78
        65536    4   0.00  2.04          28000      13705 [ 3.50]       33048    7.52MB    5.86MB     32875        29         144
        65536    8   0.00  2.32          55000      23706 [ 6.88]       33491   11.54MB    5.91MB     32905       119         467
        65536   16   0.00  2.51          61000      24283 [ 7.63]       34253   19.57MB    6.00MB     32876       592         785
```

relative speed up

- good speed up with # threads [max speed up 7.63]
- BUT algorithm slow compared with baseline [64K 1 thread 12,468 : 3,894]
- 64K baseline 12,468, lockless 8 threads 23,706 [almost twice as fast]

LOCKLESS ALGORITHMS

Preview - lockless list using transactional memory (TSX)



relative speed up

```
ZINFANDEL Windows 7 Professional (64 bit) 64 bit exe RELEASE tsxList [RTM split transaction + lock based fall back path] NCPUS=8 RAM=64GB 27-Nov-2015 09:16:02
METHOD=4 ALIGNED CONTAINS=0% ADD=50% REMOVE=50% NOP=100 NSECONDS=2 MAXBACKOFF=16 NSPLIT=64 NTAG=4096 PREFILL STATS=0x01 TSXALLOC=0 sizeof(Node)=16
Intel64 family 6 model 94 stepping 3 Intel(R) Xeon(R) CPU E3-1270 v5 @ 3.60GHz
L1 D    32K L 64 K  8 N   64
L1 I    32K L 64 K  8 N   64
L2 U   256K L 64 K  4 N 1024
L3 U  8192K L 64 K 16 N 8192

maxKey nt    pft     rt             ops       ops/s     rel        nMalloc         nFree  nlist       vmUse      memUse  commit       nlock     tt
------ --    ---     --             ---       -----     ---        -------         -----  -----       -----      ------  ------       -----     --
    16  1   0.00   2.00        52937200    26455372 [ 1.00]            17            11      6     2.47MB     3.79MB 100.00%           0   2.00
    16  2   0.00   2.00        50285300    25130084 [ 0.95]          1419          1416      3     3.47MB     3.90MB  99.97%       13251   2.00
    16  4   0.00   2.00        47988400    23982208 [ 0.91]          4933          4926      7     5.49MB     4.13MB  99.91%       41013   2.00
    16  8   0.00   2.00        45648800    22812993 [ 0.86]          7960          7950     10     9.52MB     4.35MB  99.77%      103788   2.00
    16 16   0.00   2.00        44129300    22053623 [ 0.83]          9057          9046     11    17.57MB     4.49MB  99.76%      105280   2.00
    64  1   0.00   2.00        36914300    18447926 [ 1.00]            53            24     29     2.57MB     3.90MB 100.00%           0   2.00
    64  2   0.00   2.00        34732300    17357471 [ 0.94]          1783          1750     33     3.57MB     4.02MB  99.96%       15175   2.00
    64  4   0.00   2.00        33760700    16871914 [ 0.91]          3529          3501     28     5.59MB     4.14MB  99.87%       45097   2.00
    64  8   0.00   2.00        30290100    15137481 [ 0.82]          7378          7349     29     9.60MB     4.39MB  99.65%      107220   2.00
    64 16   0.00   2.02        28139300    13909688 [ 0.75]          9363          9327     36    17.63MB     4.56MB  99.63%      105505   2.03
   256  1   0.00   2.00        16529800     8260769 [ 1.00]           164            30    134     2.57MB     3.92MB 100.00%           0   2.00
   256  2   0.00   2.00        15848200     7920139 [ 0.96]          2112          1986    126     3.58MB     4.05MB  99.92%       12623   2.00
   256  4   0.00   2.00        16440200     8215992 [ 0.99]          3526          3401    125     5.59MB     4.14MB  99.77%       37872   2.00
   256  8   0.00   2.00        15568000     7780109 [ 0.94]          5632          5514    118     9.60MB     4.29MB  99.48%       81145   2.00
   256 16   0.00   2.00        14279600     7136231 [ 0.86]          7365          7233    132    17.64MB     4.44MB  99.45%       78673   2.00
  1024  1   0.00   2.00         4196800     2097351 [ 1.00]           577            46    531     2.57MB     3.95MB 100.00%           0   2.01
  1024  2   0.00   2.00         2861000     1429785 [ 0.68]          1926          1421    505     3.58MB     4.04MB  99.56%       12601   2.00
  1024  4   0.00   2.00         3490400     1744327 [ 0.83]          1990          1483    507     5.58MB     4.05MB  99.15%       29787   2.00
  1024  8   0.00   2.00         5676900     2837031 [ 1.35]          3732          3198    534     9.60MB     4.17MB  99.08%       52482   2.00
  1024 16   0.00   2.00         6756900     3376761 [ 1.61]          6299          5807    492    17.64MB     4.37MB  99.29%       48140   2.00
  4096  1   0.00   2.00          491100      245427 [ 1.00]          2145            75   2070     2.57MB     4.04MB 100.00%           5   2.00
  4096  2   0.00   2.00          672600      336131 [ 1.37]          2548           513   2035     3.58MB     4.07MB  99.93%         491   2.00
  4096  4   0.00   2.00         1253600      626486 [ 2.55]          3454          1392   2062     5.59MB     4.14MB  99.78%        2749   2.00
  4096  8   0.00   2.00         2100700     1049825 [ 4.28]          4887          2847   2040     9.60MB     4.25MB  99.49%       10733   2.00
  4096 16   0.00   2.00         2059400     1028671 [ 4.19]          5066          3063   2003    16.61MB     4.29MB  99.44%       11465   2.00
 16384  1   0.00   2.00           69700       34832 [ 1.00]          8323            99   8224     2.57MB     4.43MB 100.00%           1   2.01
 16384  2   0.00   2.00          120800       60279 [ 1.73]          8319           154   8165     3.58MB     4.43MB  99.96%          53   2.01
 16384  4   0.00   2.00          209800      104690 [ 3.01]          8658           393   8265     5.59MB     4.46MB  99.82%         381   2.01
 16384  8   0.00   2.00          364600      181845 [ 5.22]          9274          1091   8183     9.59MB     4.51MB  99.58%        1515   2.01
 16384 16   0.00   2.01          333900      166284 [ 4.77]          9518          1268   8250    17.63MB     4.57MB  99.48%        1743   2.01
 65536  1   0.00   2.01           28800       14342 [ 1.00]         32871             7  32864     4.58MB     5.93MB 100.00%           0   2.02
 65536  2   0.00   2.01           49100       24476 [ 1.71]         32834           149  32685     5.58MB     5.93MB  99.95%          27   2.01
 65536  4   0.00   2.01           81100       40308 [ 2.81]         33017           266  32751     7.59MB     5.95MB  99.84%         130   2.02
 65536  8   0.00   2.01          126500       62810 [ 4.38]         33415           644  32771    11.60MB     6.00MB  99.60%         510   2.02
 65536 16   0.00   2.03          121700       60009 [ 4.18]         33393           745  32648    19.63MB     6.04MB  99.26%         899   2.03
```

- good speed up with # threads [max speed up 5.22]
- algorithm almost as quick as baseline [64K 1 thread 12,468 : 14,324]
- 64K baseline 12,468, lockless 8 threads 62,810 [almost 5x faster]

Learning Outcomes

- you are now able to:
  - explain the difference between obstruction, lock and wait free algorithms
  - explain the operation of the Compare and Swap (CAS) instruction
  - implement a lockless stack using CAS
  - explain the ABA problem and some possible solutions
  - implement a lockless ordered list using CAS
  - assess the difficulty of adding memory management to a CAS based concurrent algorithm
  - add memory management to a lockless algorithm using hazard pointers