CS510 Advanced Topics in Concurrency

Jonathan Walpole

Portland State
UNIVERSITY

Hazard Pointers: Safe Memory Reclamation of Lock-Free Objects

Maged M. Michael

The Problem

- Lock-free algorithms assume that threads can operate on any object at any time
- Freeing memory could break this assumption
- How can we free memory of deleted nodes in a safe and lock-free manner?

Prior Work

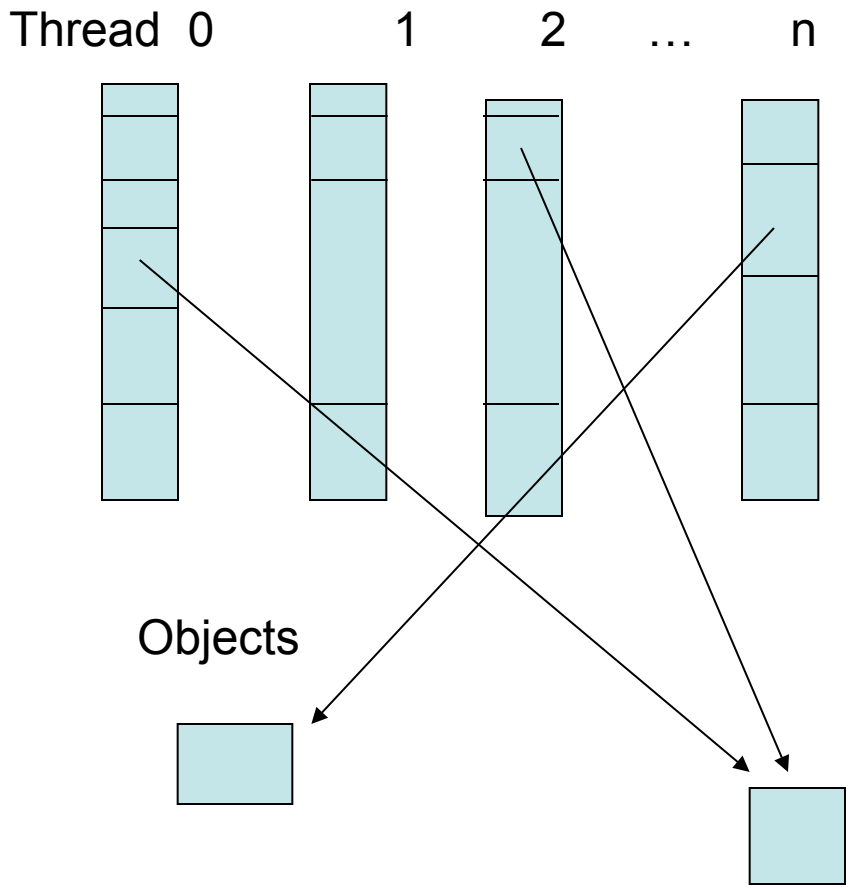
- No reuse
- Recycling of constant static storage
- Type safe memory
- Reference counts with DCAS or CAS2

Goals of Hazard Pointers

- Readers inform writers of references they hold
- Bound the number of unreclaimed objects
- Tolerance for thread failures (wait-free)
- No special hardware or kernel support needed
- Suitable for user or kernel use

Hazard Pointers

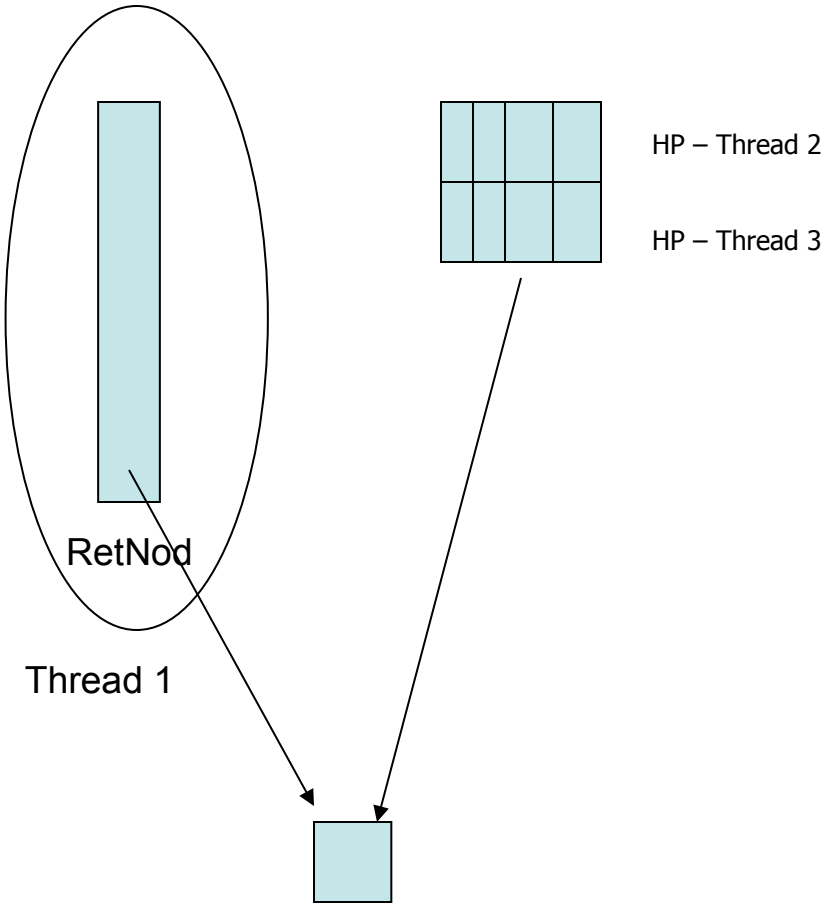
- Each thread has a set of pointers that only it writes, but other threads can read
- Each pointer is either null or points to an object the thread is currently using
- Before accessing any dynamic object a thread must first store a pointer to it in its list



Retired List

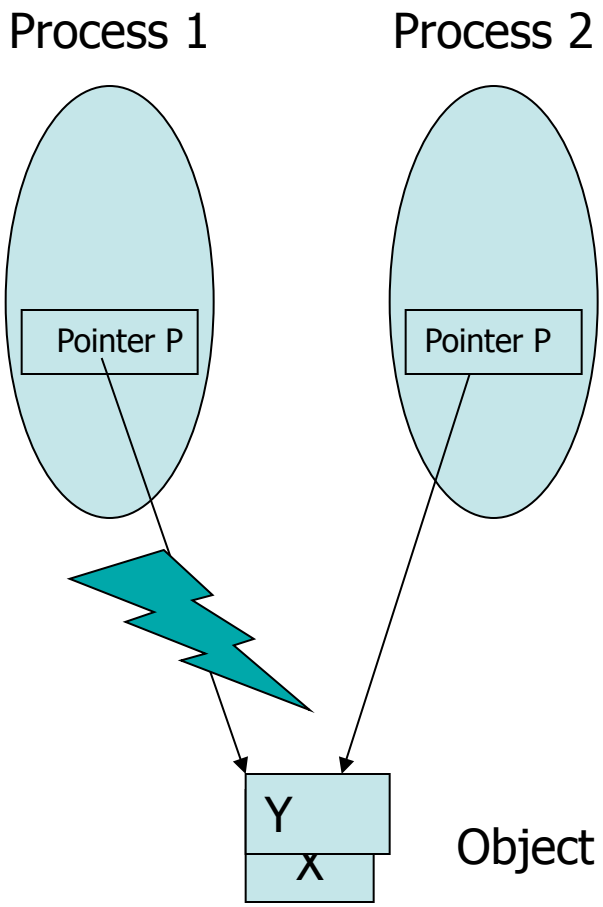
Each thread also has a list of nodes it has deleted and would like to free

When the length of this list reaches R, the thread scans the hazard pointer lists of all other threads to see if its safe to free any of its nodes.



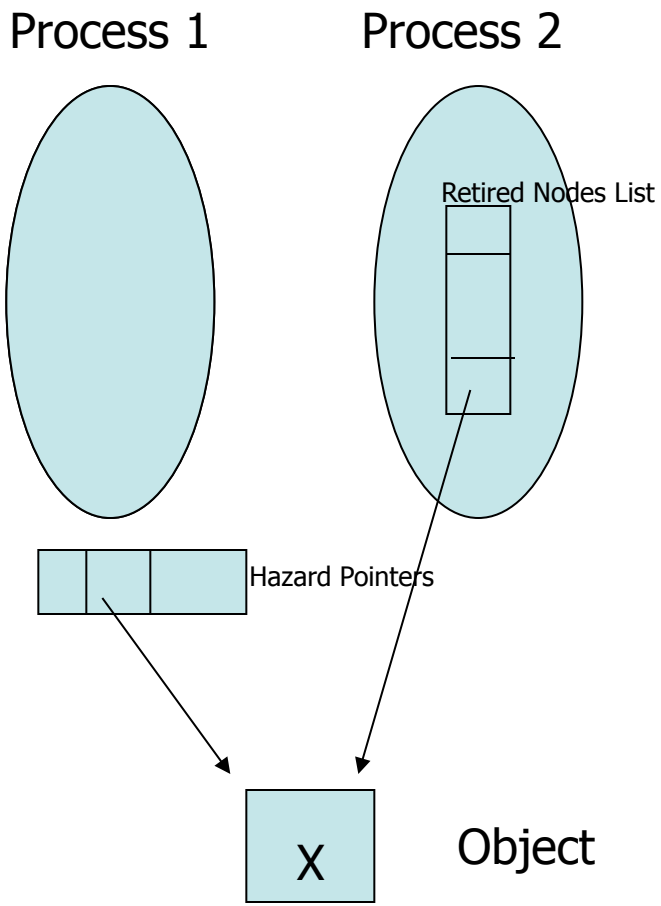
The ABA Problem

- Process 1 has a copy of a pointer to Object x
- Process 2 deletes object x, frees its memory, and reuses the same memory to create a new object y
- Process 1 applies CAS on the address of object x which is the same as that of object y
- The CAS succeeds but *the object has changed!*



ABA Problem Solved

- Process 1 has a hazard pointer to object x
- Process 2 deletes object x and moves object x to its retired nodes list
- Process 1 can still safely access object x
- Hazard Pointer disallows freeing the memory and hence reuse of the same memory
- The ABA problem cannot occur with Hazard Pointers



Hazard Pointer Types

```
// Hazard pointer record
    structure HPRecType
            { HP[K]:*NodeType; Next:*HPRecType; }
// The header of the HPRec list
    HeadHPRec : *HPRecType;
// Per-thread private variables
    rlist : listType; // initially empty
    rcount : integer; // initially 0
```

Retiring Nodes

```
RetireNode(node:*NodeType) {
    rlist.push(node);
    rcount++;
    if (rcount ≥ R)
        Scan(HeadHPRec);
}
```

Lock Free Dequeue

```
Dequeue : DataType(){
   while true {
4:       h <- Head;
4a:      *hp0 <- h;
4b:      if (Head != h) continue;
         t <- Tail;
5:       next <- h.next;
         *hp1 <- next;
6:       if(Head != h) continue;
7:       if(next == null) return EMPTY;
         if(h==t){
            CAS(&Tail , t ,next);continue;
         }
         data <- next.Data ;
8:       if CAS(&Head , h , next) break;
   }
9: RetireNode(h);
   return data;
}
```

Record Hazard Pointer for h

What if h was removed b/w 4 & 4a?

Access Hazard ! What if h was freed by some thread ?

ABA Hazard ! What if Head is pointing to some reincarnation of h?

ABA hazard on t !

Access Hazard using next !

ABA hazard on h !

Queues

```
// hp0 and hp1 are private ptrs to 2 of the thread's hazard ptrs

Enqueue(data:DataType) {
   1: node ← NewNode();
   2: node^.Data ← data;
   3: node^.Next ← null;
      while true {
   4:    t ← Tail;
  4a:    *hp0 ← t;
  4b:    if (Tail ≠ t) continue;
   5:    next ← t^.Next;
   6:    if (Tail ≠ t) continue;
   7:    if (next ≠ null) { CAS(&Tail,t,next); continue;}
   8:    if CAS(&t^.Next,null,node) break;
      }
   9: CAS(&Tail,t,node);
}
```

```
Dequeue() : DataType {
      while true {
  11:    h ← Head;
 11a:    *hp0 ← h;
 11b:    if (Head ≠ h) continue;
  12:    t ← Tail;
  13:    next ← h^.Next;
 13a:    *hp1 ← next;
  14:    if (Head ≠ h) continue;
  15:    if (next = null) return EMPTY;
  16:    if (h = t) { CAS(&Tail,t,next); continue; }
  17:    data ← next^.Data;
  18:    if CAS(&Head,h,next) break;
      }
  19: RetireNode(h); return data;
}
```

Lock Free Stack Push

```
   Push(data:DataType) {
1:     node <-NewNode();
2:     node.Data <- data ;
       while true {
3:         t <- Top ;
4:         node.Next <- t ;
5:         if CAS(&Top ,t ,node)
               return;
       }
}
```

Intialising the new node

Not an Accces Hazard !!

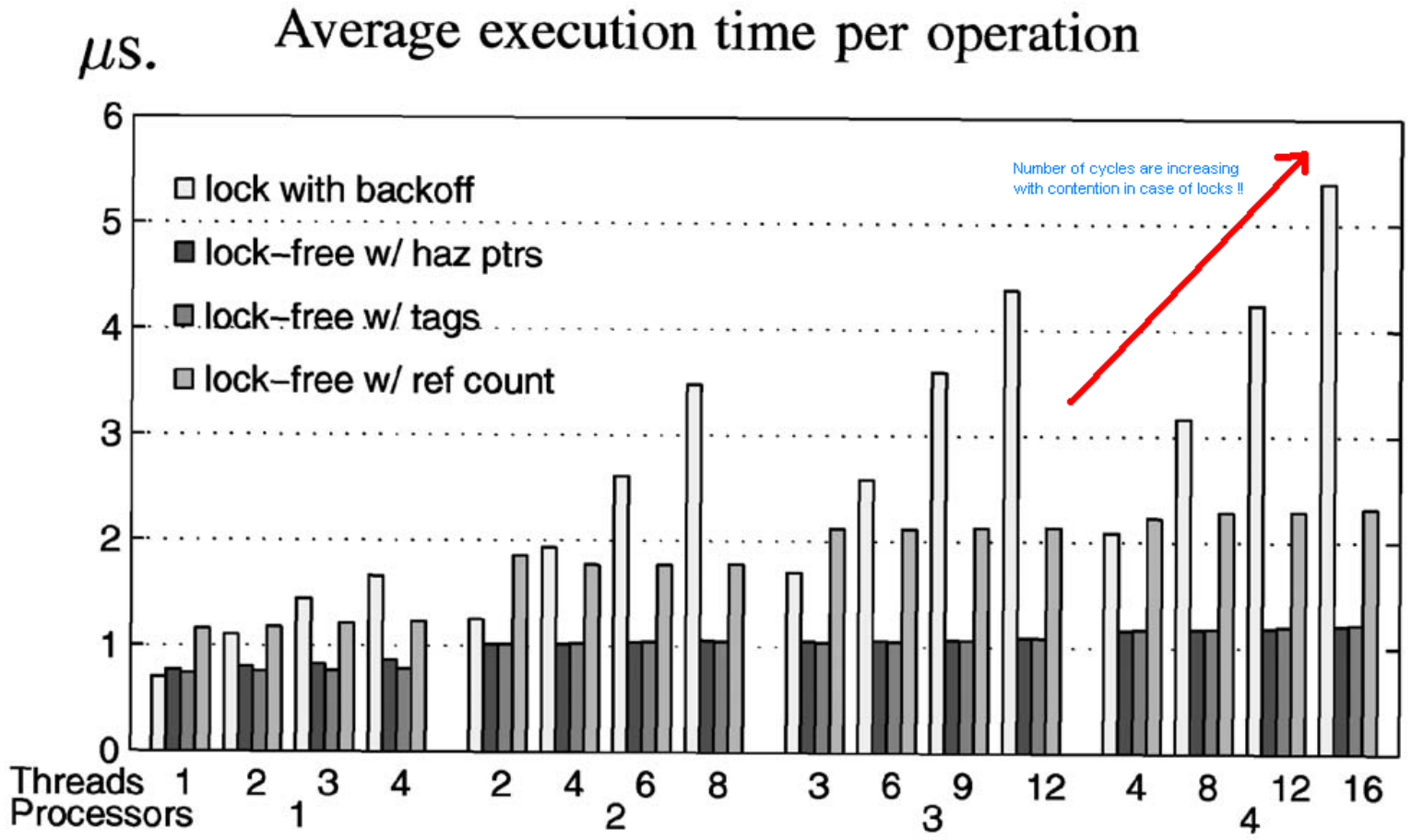
Not ABA Hazard as change of Top does not corrupt the Stack. !!

Lock Free Stack Pop

```
   Pop() : DataType {
        While true {
6:              t <- Top;
7:              if (t == null)
                  return EMPTY;
7a:             *hp <- t ;
7b:             If (Top != t) continue;
8:              next <- t.Next;
9:              if CAS (&Top , t ,next)
                  break;
        }
10:     data <- t.Data ;
10a:    RetireNode(t) ;
11:     return data;
   }
```

Access Hazard on t !
ABA Hazard on t!

Performance (Queues)



Conclusions

Safe memory reclamation

Solves ABA problem

Wait-free if the original algorithm is wait-free

Reasonable performance?

- readers must now write
- and these writes require memory barriers!

Stacks

```
structure NodeType { Data:DataType; Next:*NodeType; }
// Shared variables
Top:*NodeType;  // Initially null
// hp is a private ptr to one of the thread's hazard ptrs.

Push(data:DataType) {
 1:   node ← NewNode();
 2:   node^.Data ← data;
      while true {
 3:     t ← Top;
 4:     node^.Next ← t;
 5:     if CAS(&Top,t,node) return;
      }
}
```

```
Pop() : DataType {
      while true {
 6:     t ← Top;
 7:     if (t = null) return EMPTY;
 8:     *hp ← t;
 9:     if (Top ≠ t) continue;
10:     next ← t^.Next;
11:     if CAS(&Top,t,next) break;
      }
12:   data ← t^.Data;
13:   RetireNode(t); return data;
}
```