Lock-free Programming

Nathaniel Wesley Filardo

November 20, 2006

Outline

Introduction

Lock-Free Linked List Insertion

Lock-Free Linked List Deletion

Tradeoffs

Some real algorithms?

- Suppose some madman says "We shouldn't use locks!"
- You know that this results (eventually!) in inconsistent data structures.
  - Loss of invariants within the data structure
  - Live pointers to dead memory
  - Live pointers to undead memory (Hey, my type changed! Stop poking there!)
- Well, the madman insists, so here goes...

Lock-Free Linked List Insertion

Lock-Free Linked List Node

- Node definition is simple:

`void* data`
`void* next`

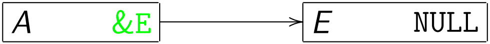
Insertion into a Linked List Without Locks

Insertion Code

```
insertAfter(label, data) {
        new = newNode(data);
        prev = findLabel(label);
        new->next = prev->next;
        prev->next = new;
}
```

Insertion into a Linked List Without Locks

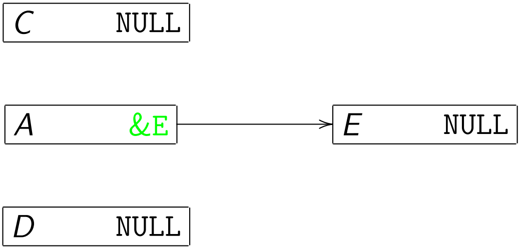
Precondition



- One list, two items on it: A and E.

Insertion into a Linked List Without Locks

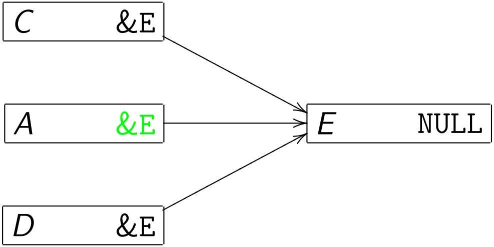
First step



- Two threads get two nodes, *C* and *D* and want to insert.
- Thread 1: `new = newNode(C);`
- Thread 2: `new = newNode(D);`
- `prev = findLabel(A); /* Gives &A to both */`

Insertion into a Linked List Without Locks

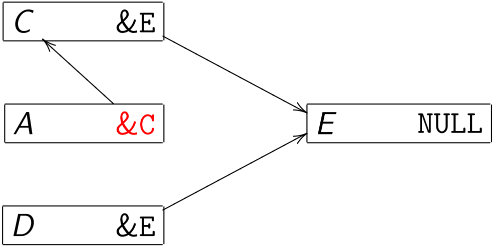
Second step



- Two threads point their respective nodes *C* and *D* into list at *E*
- `new->next = prev->next;`

Insertion into a Linked List Without Locks

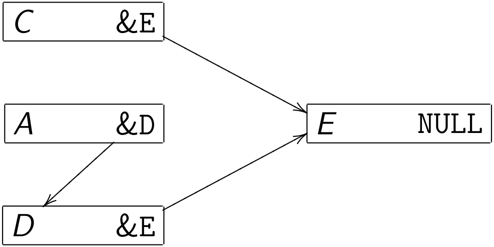
One thread goes



- One of the two goes (here the thread owning C)...
- `prev->next = new;`

Insertion into a Linked List Without Locks

And the other...



- And the other (owning *D*)...
- `prev->next = new;`
- This loses a node! (Nobody notices that *C* is no longer on the list)

Review of Atomic Primitives

- XCHG (ptr, val) atomically:
    - old_val = *ptr
    - *ptr = val
    - return old_val
- CAS (ptr, expect, new) atomically:
    - if (*ptr != expect) return *ptr;
    - else return XCHG (ptr, new);
- Note that CAS is no harder - it's a read and a write; the logic is free (it's on the chip).

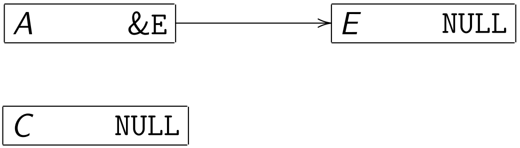
Review of Atomic Primitives

- Notice that we can use CAS to rescue this procedure.
- So let's rewrite that insertion code to be

```
insertAfter(label, data) {
        new = newNode(data);
        do {
              prev = findLabel(label);
              new->next = prev->next;
        } while
           ( CAS(&prev->next, new->next, new)
                   != new->next);
}
```

Insertion into a Lock-free Linked List: Simple case

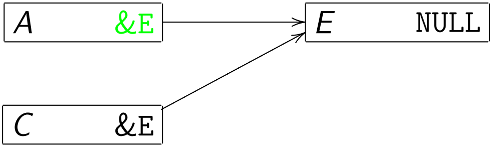
Setup



- Some thread constructs the bottom node C; wishes to place it between the two above, A and E.
- `new = newNode(C);`
- `prev = findLabel(A);`

Insertion into a Lock-free Linked List: Simple case

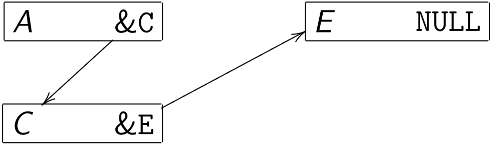
First step



- Thread points C node's next into list at E.
- `new->next = prev->next;`

Insertion into a Lock-free Linked List: Simple case

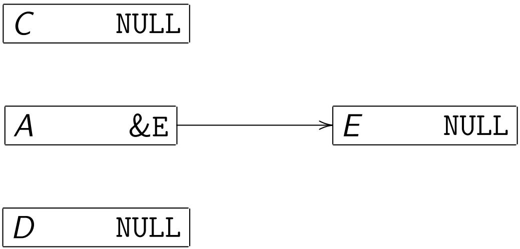
First step



- `CAS(&prev->next, new->next, new);`

Insertion into a Lock-free Linked List: Race case

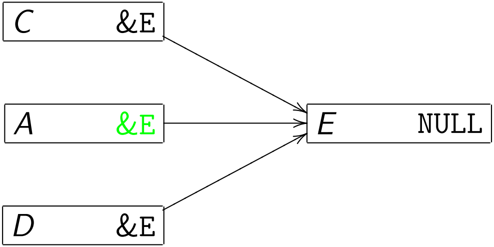
First step



- Two threads get their respective nodes C and D.
- `new = newNode(...);`
- `prev = findLabel(A);`

Insertion into a Lock-free Linked List: Race case

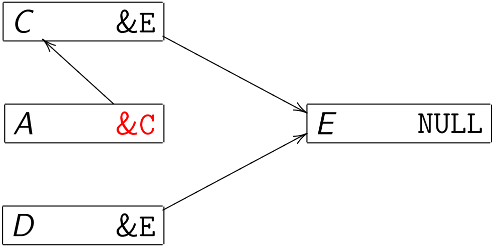
First step



- Both set their new node's next pointer.
- `new->next = prev->next;`

Insertion into a Lock-free Linked List: Race case

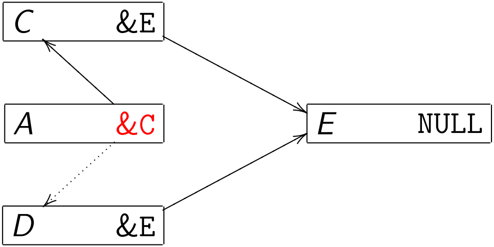
One thread goes



- One of the two goes (here the thread owning *C* won)...
- `CAS(&prev->next, new->next, new)`

Insertion into a Lock-free Linked List: Race case

And the other. . .



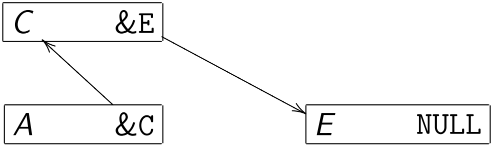
- And the other (owning *D*). . .
- `CAS(&prev->next, new->next, new)`
- Fails since `prev->next == &C` and `new->next == &E`.
- So this thread tries again.

That's great!

- This works fine for data structures supporting only insert and scan.
- How many data structures are like that?

Deletion is easy?

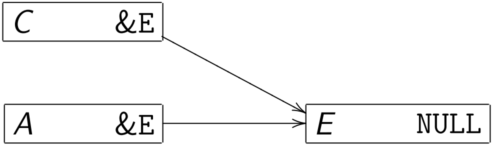
- Suppose we have



- And want to get rid of *C*.
- So `CAS(&A.next, &C, &E)`

Deletion is easy?

- Now we have

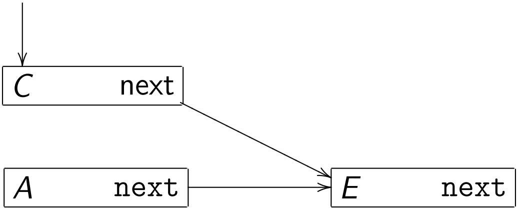


- Great, looks like deletion to me!

Deletion is easy?

Continued

- But imagine there was another thread accessing C (say, scanning the list).



- We have no way of knowing this, so for correctness we must not `free(C)`.

ABA Problem

- A problem of confused identity

global = malloc(sizeof(Foo))	
local_1 = global	local_2 = global
global = NULL	
free(local_1)	
global = malloc(sizeof(Foo))	
	/* Validity check */ if (global == local_2) global->foo_baz = ...

- Even though local_2 and global might share the same value, they don't *really* mean the same thing.

ABA Problem

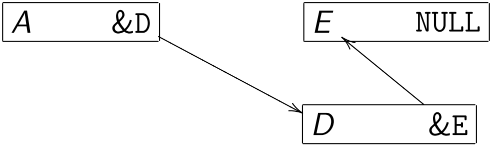
- So, for a “deleted” node (often “logically deleted node”)...
- Let’s just leave it detached from the list, marking it somehow as deleted.

- Other threads will fail their operations and restart.
- We might have a free list of available nodes, even...
  - Some published implementations do this, leaving as an exercise to syncrhonize all threads to delete the the list and free list when everybody’s done.
  - See [1] (linked & skip lists).

ABA Problem

Now reusing memory...

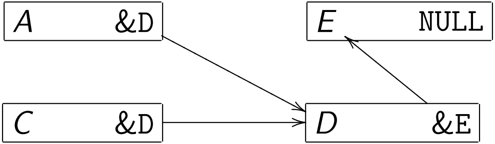
- We might have a somewhat complex case of a sorted list



ABA Problem

Now reusing memory...

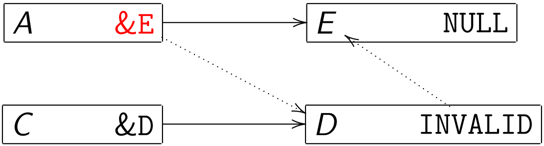
- Thread X trying to insert C after A starts up its dance...
- So we now have



ABA Problem

Now reusing memory…

- Somebody comes in and deletes *D*.
- So we now have

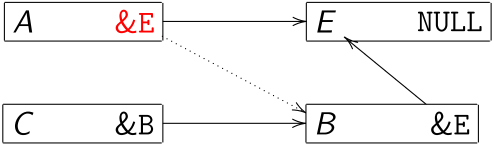


- There is a deleted node (*D*, bottom right) that was the `next` of *A* when thread *X* started running

ABA Problem

Now reusing memory (part 2)

- Another thread, Y, now reclaims deleted node, labels it B and points it to E.
- So now we have

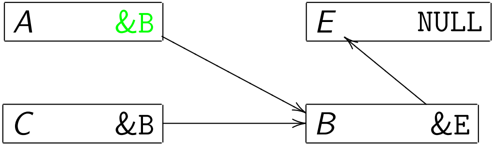


- Thread X still trying to insert C after A. Been preempted for "a while"

ABA Problem

Now reusing memory (part 3)

- Thread *Y* now inserts the reclaimed node where it belongs! (using CAS, of course)
- Trying for a sorted list with

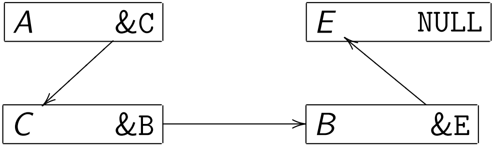


- Thread *X* still trying to insert "C" after "A". Been preempted for "a while"

ABA Problem

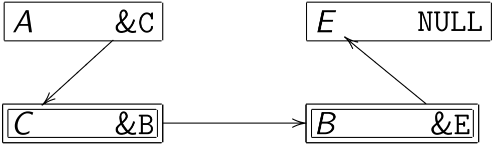
Now reusing memory (part 4)

- Thread X wakes up, and the CAS works (!) giving instead



ABA Problem

Woah, what just happened?



- But $\{A, C, B, E\}$ isn't sorted!

Fixing ABA

- It turns out that we need a more sophisticated delete (and maybe insert and lookup!) function. Look at [1] or [3] (or others) for more details.
- Generation counters are a simple way to solve ABA

Fixing ABA

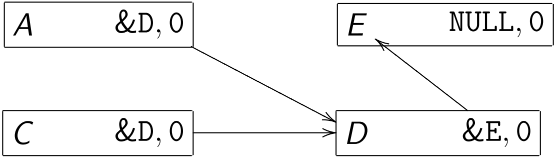
- Imagine that instead of CAS we had CAS2, which operates on two words at once:
  CAS (ptr, $\texttt{expect}_1$, $\texttt{expect}_2$, $\texttt{new}_1$, $\texttt{new}_2$)
  atomically:
  - if (*ptr != $\texttt{expect}_1$ || *(ptr+1) != $\texttt{expect}_2$)
    - return {*ptr, *(ptr+1)};
  - else
    - *ptr = $\texttt{new}_1$; *(ptr+1) = $\texttt{new}_2$;
    - return { $\texttt{expect}_1$, $\texttt{expect}_2$ };

Fixing ABA

- If we keep a generation counter at each site and CAS2 the pointer and the generation counter, some "reasonably large" number of pointer updates are all to unique values.

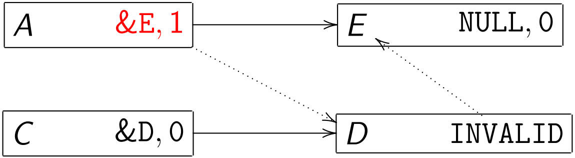
Fixing ABA

- From the above example, the initial list might have looked like



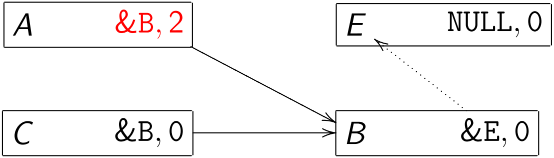
Fixing ABA

- Deletion of *D* might make it look like



Fixing ABA

- Insertion of *B* might make it look like



- 2 != 0 so we're saved!

Tradeoffs

Locks Can Be Expensive

- Consider XCHG style locks which use
  `while( xchg( &locked, LOCKED ) == LOCKED )`
  as their core operation.
- Each xchg flushes the processor pipeline...
- We could spend a long time here waiting or yielding...
- This implies we'll have very high latency *on contention*...

Tradeoffs

Locks Can Be Expensive

- That is, if N people are contending for a lock, $N - 1$ of them are `yield()`ing, just wasting time.
- Here they could all work at once . . .
- Only restarting on collision . . .
- And even then, *at least one* thread which collided has made progress.

Tradeoffs

Locks Can Be Expensive

- For a large data structure (*e.g.* linked list), we would *like* multiple *local* (independent) operations to be allowed concurrently.

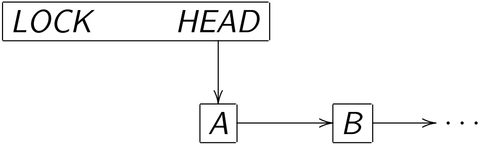
  insertafter(*label*, *node*)

- Can somewhat get this with a data structure full of locks
- . . . but order requirements mean that threads can still pile up while trying to get to their local site.

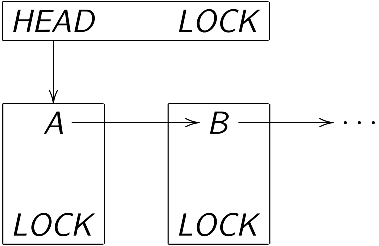
Tradeoffs

Locks Can Be Expensive

- That is, instead of



- We could have



Tradeoffs

Write Your Own?

- It's *extremely hard* to roll your own lockfree algorithm.
- But moreover, it's *almost impossible* to debug one.
- Thus all the papers are long not because the algorithms are hard, . . .
- . . . but because they prove the correctness of the algorithm so they can skip that step!

Tradeoffs

Large Systems

- We increase the number of atomic operations.
- Thus we starve processors for bus activity on Intel-like bus-locking systems.
- On systems with cache coherency protocols, we might livelock with no processor able to make progress due to cacheline stealing and high transit times.

Some real algorithms?

- [3] specifies a `CAS`-based lock-free list-based sets and hash tables using a technique called SMR to solve ABA and allow reuse of memory.
    - Their performance figures are worth looking at. Summary: fine-grained locks (lock per node) show linear-time increase with # threads, their algorithm shows essentially constant time.

Some real algorithms?

- Read-Copy-Update (RCU, [9]) uses techniques from lock-free programming.
- Is used in several OSes, including Linux.
- It's a bit more complicated than the examples given here, but worth reading about.

[1] Mikhail Fomitchev and Eric Ruppert, *Lock-free linked lists and skip lists*, PODC (2004July), no. 1-58113-802-4/04/0007, 50–60.

[2] Peter Memishian, *On locking*, Sun Microsystems, 2006.

[3] Maged M. Michael, *High performance dynamic lock-free hash tables and list-based sets*, SPAA (2002August), no. 1-58113-529-7/02/0008, 73–83.

[4] ______, *Safe memory reclamation for dynamic lock-free objects using atomic reads and writes*, PODC (2002July), no. 1-58113-485-1/02/0007, 1–10.

[5] ______, *Hazard pointers: Safe memory reclamation for lock-free objects*, IEEECS (2004Jan), no. TPDS-0058-0403, 1–10.

[6] H. Sundell, *Wait-free reference counting and memory management*, 2005April.

[7] Wikipedia, *Lock-free and wait-free algorithms*, 2006.

[8] ______, *Non-blocking synchronization*, 2006.

[9] ———, *Read-copy-update*, 2006.

Acknowledgements

- Dave Eckhardt (de0u) and Bruce Maggs (bmm) for moral support and big-picture guidance
- Jess Mink (jmink), Matt Brewer (mbrewer), and Mr. Wright (mrwright) for being victims of beta versions of this lecture.