Lock-free Programming

Nathaniel Wesley Filardo

April 10, 2006

Outline

Motivation

Lock-Free Linked List Insertion

Lock-Free Linked List Deletion

Some real algorithms?

Motivation

Review of atomic primitives
Locks can be expensive

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).

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...

Lock-Free Linked List Insertion

Lock-Free Linked List Node
Insertion into a Lock-free Linked List: Successful case
Insertion into a Lock-free Linked List: Race case

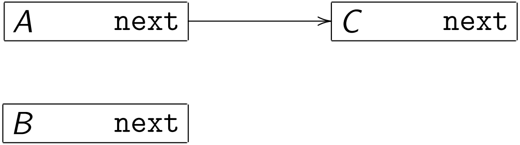
Lock-Free Linked List Node

- Node definition is simple:

`void* data`
`void* next`

Insertion into a Lock-free Linked List: Successful case

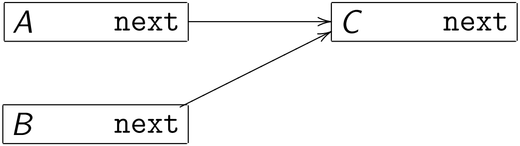
Setup



- Some thread constructs the bottom node B; wishes to place it between the two above, A and C.

Insertion into a Lock-free Linked List: Successful case

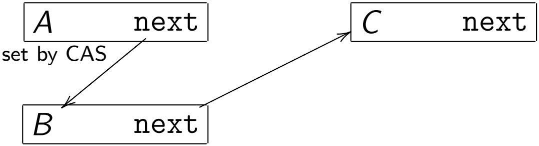
First step



- Thread points B node's next into list at C.

Insertion into a Lock-free Linked List: Successful case

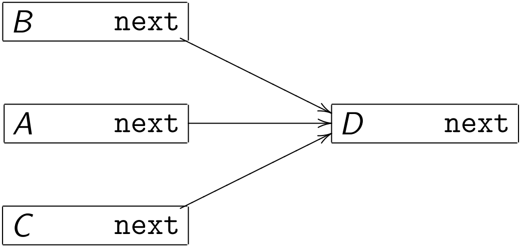
First step



- CAS used to point previous node *A* to new node *B*.
- ...
- So wait, what's the cleverness?

Insertion into a Lock-free Linked List: Race case

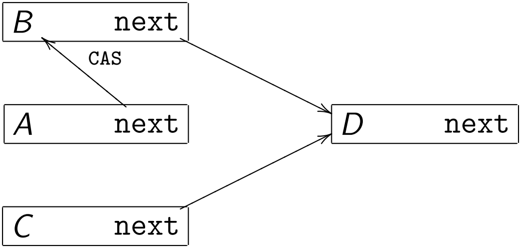
First step



- Two threads point their respective nodes B and C into list at D
- Both of them try to `CAS` the previous node's (A's) next pointer...

Insertion into a Lock-free Linked List: Race case

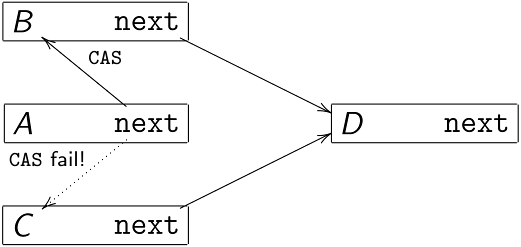
One thread goes



- One of the two goes (here the thread owning *B* won)...

Insertion into a Lock-free Linked List: Race case

And the other…



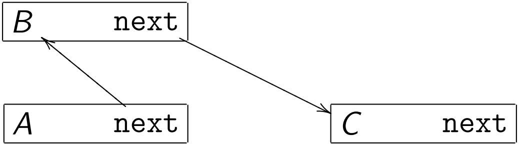
- And the other (owning C)…
- But the `expect` value doesn't match, so the linked list structure is OK.
- So this thread tries again and does the same dance…

That's great!

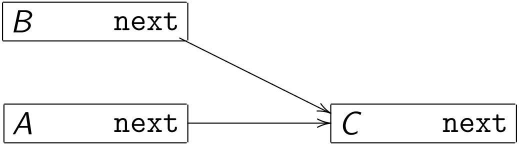
- Yes, if we want an insert-and-read only list, then it's fine!
- How many datastructures are like that?

Deletion is easy?

- Can we just prune the node?
- Given



- Can't we just transition via `CAS` to

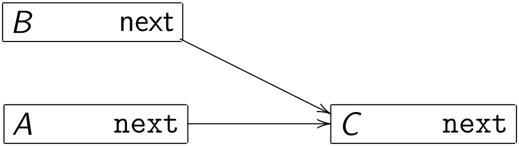


- Yes, but can we reclaim that memory?

Deletion is easy?

Continued

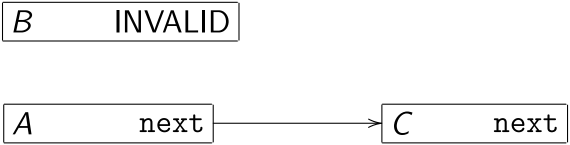
- Can't we just transition via `CAS` to



- There might be another thread touching the upper node (*B*)!
  - Can't touch that memory at all!
  - In particular, can't free() it!

Compromise?

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

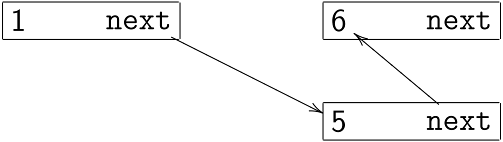


- Other threads will fail their operations and restart.
- We might have a free list of available nodes, even...
  - Some real-world implementations do this, leaving as an exercise to syncrhonize all threads to delete the the list and free list when everybody’s done.

Compromise?

Now reusing that memory…

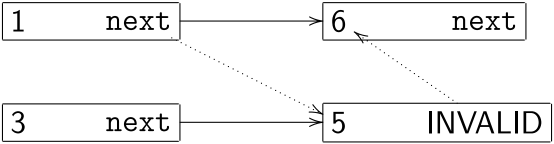
- We might have a somewhat complex case of a sorted list



Compromise?

Now reusing that memory…

- Thread X trying to insert "3" after "1" races against somebody deleting "5".
- So we now have

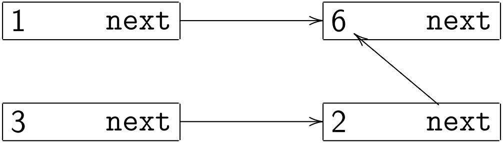


- There is a deleted node ("5", bottom right) that was the `next` of "1" when thread X started running

Compromise?

Now reusing that memory (part 2)

- Thread *Y* now reclaims deleted node, pushes in “2” and points to “6”.
- Trying for a sorted list with

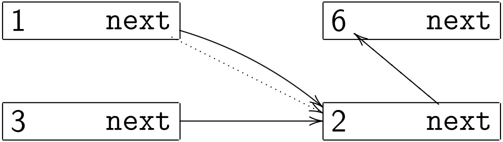


- Thread *X* still trying to insert “3” after “1”. Been preempted for “a while”
- Anybody see the problem yet?

Compromise?

Now reusing that memory (part 3)

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

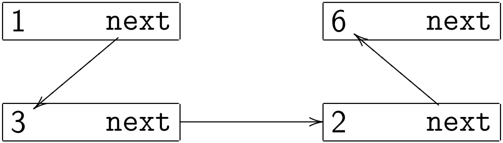


- Thread *X* still trying to insert “3” after “1”. Been preempted for “a while”
- The dotted line indicates what *X* expects to see!
- How about now?

Compromise?

Now reusing that memory (part 4)

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



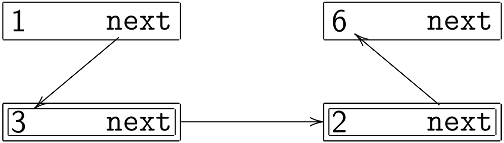
Compromise?

Earth-Shattering KABOOM!

Figure: There was supposed to be an . . . [mar()]

Compromise?

Woah, what just happened?



- But, but, but... $\{1, 3, 2, 6\}$ isn't sorted!
- This is called *The ABA problem*: the pointer changed *meaning* but we didn't notice.

Full fledged deletion & reclaim

OK, so how do we actually do this?

- It turns out that we need a more sophisticated delete function. Look at [Fomitchev and Ruppert(2004)] or [Michael(2002a)] (or others) for more details.
    - Generation counters are a simple way to solve ABA (usually requires use of `CASn` - acts on *n* words at once; much slower than `CAS`)
- But that doesn't solve memory reclaim - for these we need more sophisticated algorithms (which also solve ABA for us):
    - Hazard Pointers ("Safe Memory Reclaimation" or just "SMR") [Michael(2002b)] and [Michael(2004)]
    - Wait-free reference counters [Sundell(2005)]

Some real algorithms?

- [Fomitchev and Ruppert(2004)] gives a simple, non-reclaimable lock-free linked/skip-list algorithm.
- [Michael(2002a)] specifies a `CAS`-based lock-free list-based sets and hash tables using SMR as a refinement of the above.
    - 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!

- *Marvin the martian*, URL `http://www.snowflake-designs.com/images/Marvin%20Martian%201.jpg`.

- M. Fomitchev and E. Ruppert, PODC pp. 50–60 (2004), URL `http://www.research.ibm.com/people/m/michael/podc-2002.pdf`.

- M. M. Michael, SPAA pp. 73–83 (2002a), URL `http://portal.acm.org/ft_gateway.cfm?id=564881\&type=pdf\&coll=GUIDE\&dl=ACM\&CFID=73232202\&CFTOKEN=1170757`.

- M. M. Michael, PODC pp. 1–10 (2002b), URL `http://www.research.ibm.com/people/m/michael/podc-2002.pdf`.

- M. M. Michael, IEEECS pp. 1–10 (2004), URL `http://www.research.ibm.com/people/m/michael/podc-2002.pdf`.
- H. Sundell (IEEE, 2005), 1530-2075/05, URL `http://ieeexplore.ieee.org/iel5/9722/30685/01419843.pdf?tp=\&arnumber=1419843\&isnumber=30685`.
- Wikipedia, *Lock-free and wait-free algorithms* (2006a), URL `http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms`.
- Wikipedia, *Non-blocking synchronization* (2006b), URL `http://en.wikipedia.org/wiki/Non-blocking_synchronization`.

Acknowledgements

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