Lock-free Programming

Nathaniel Wesley Filardo

November 10, 2008

Outline

Introduction

- 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!)

Introduction
Locks Can Be Expensive

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

Introduction
Locks Can Be Expensive

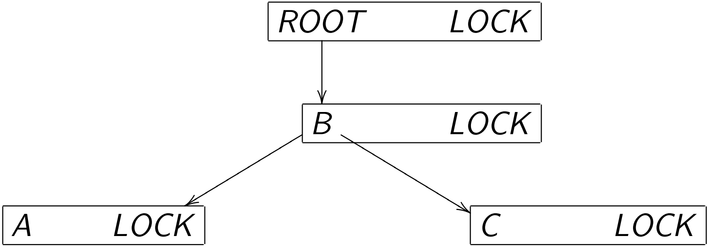
- That is, if N people are contending for a lock, $N-1$ of them are just wasting time.
- It would be nice if they could all work at once ...
- ... being careful not to step on each other when there was actually a problem.

Introduction
Locks Can Be Expensive

- For a large data structure, we would *like* multiple *local* (independent) operations to be allowed concurrently.
  - *e.g.* "lookup" and "insert" in parallel threads
- Can somewhat get this with a data structure full of locks (think: big tree)
- ...but order requirements mean that threads can still pile up while trying to get to their local site.

Introduction
Locks Can Be Expensive

- Instead of a lock around a tree, we could have a tree with locks:



- The protocol is lock the root, then (lock child & unlock parent) as you go down.
  - This kind of *lock handoff* is a very common design.
- Here every time a thread decides to go down one branch, it gets out of roughly half of the others' ways.

Introduction

- But let's see what we can do without any locks at all.

Lock-Free Linked List Insertion

Lock-Free Linked List Node

- Node definition is simple:

label_t label
void* next

- When drawing, we'll use a shorthand:

label_t label = A
void* next = &B

$\Leftrightarrow$

A	*&B*

Insertion into a Linked List Without Locks
Insertion Code

```
insertAfter(after, newlabel) {
  //lockList();
  new = newNode(newlabel);
  prev = findLabel(after);
  new->next = prev->next;
  prev->next = new;
  //unlockList();
}
```

Insertion into a Linked List Without Locks
Good trace in 410 notation

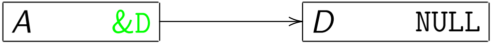
insertAfter(A,B)	insertAfter(A,C)
prev = &A	
B.next=A.next	
A.next=B	
	prev = &A
	C.next=A.next
	A.next=C

Insertion into a Linked List Without Locks
Race trace in 410 notation

insertAfter(A,B)	insertAfter(A,C)
prev = &A	
B.next = A.next	
	prev = &A
	C.next = A.next
A.next = B	A.next = C

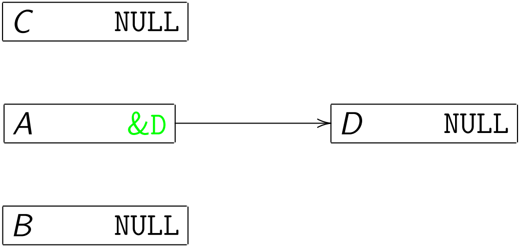
- Either of these assignments makes sense in isolation, but one of them will override the other!

Insertion into a Linked List Without Locks
Precondition



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

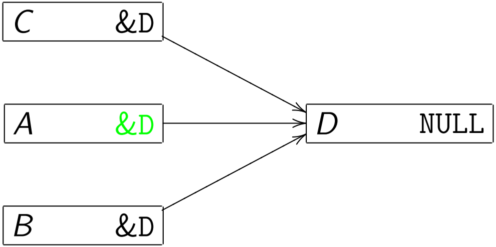
Insertion into a Linked List Without Locks
First step



- Two threads get two nodes, B and C, and want to insert.

new = newNode(B);	new = newNode(C);
prev = &A	prev = &A

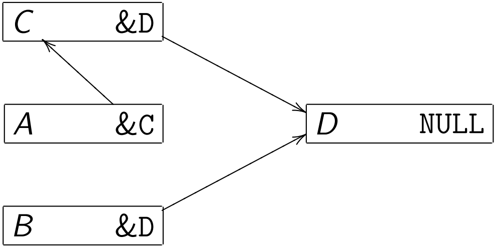
Insertion into a Linked List Without Locks
Second step



- Two threads point their respective nodes C and B into list at D

B.next=&D	C.next=&D

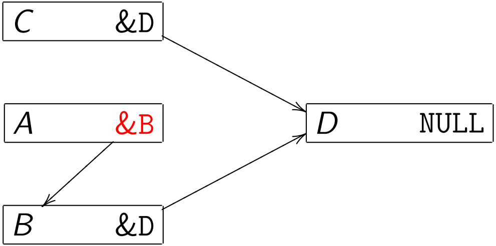
Insertion into a Linked List Without Locks
One thread goes



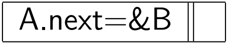
- Suppose the thread owning C completes its assignment first.

| | A.next=&C |

Insertion into a Linked List Without Locks
And the other...



- And the other (owning B) completes second, overwriting ...



- Node C is unreachable!

Insertion into a Linked List Without Locks

- What went wrong?
  1. Thread B observed that `&A->next == D`
  2. Thread C observed that `&A->next == D`
  3. Thread C changed `&A->next` "from D to C"
  4. Thread B changed `&A->next` "from D to B"
     - But it was C not D!
- How to fix that?
  - Give B and C critical sections and serialize them
    - Then there is no gap between observation and changing
    - But that requries locking, which we are avoiding...
  - Take two: assume mistakes are rare, clean up afterward!

Insertion into a Linked List Without Locks
The Lock Free Approach

```
while(not done)
   Prepare data structure update (e.g. new node)
   Determine preconditions for the update
   ATOMICALLY
     if(preconditions hold)
       make update
         done = 1
```

- Unlike critical sections, this is not (really) bounded
  - Could keep encountering trouble over and over...
- But as long as threads “almost always” don’t do spacially overlapping updates...
  - Then we gain in parallelism by having not locked.

Insertion into a Linked List Without Locks

- Our assignments were really supposed to be

insertAfter(A,B)	insertAfter(A,C)

while(!done)
   ATOMICALLY
     if A->next == D
       A->next = B
     else
       done = 1
	while(!done)
	   ATOMICALLY
	     if A->next == D
	       A->next = C
	     else
	       done = 1

- If we do that, one critical section will *safely* fail out and tell us to try again.
- How do we do this ATOMICALLY without locking?

Review of Atomic Primitives

- Remember our old friend XCHG?
- XCHG (ptr, val) atomically:

```
old_val = *ptr;
*ptr = val;
return old_val;
```

Review of Atomic Primitives

XCHG(ptr,new)	CAS(ptr, expect, new)
ATOMICALLY	ATOMICALLY
old = *ptr;	old = *ptr;
	if(old == expect)
*ptr = new;	*ptr = new;
return old;	return old;

- Note that CAS is no harder - it's a read and a write; the logic is free (it's on the CPU).

Insertion into a Lock-free Linked List

- Our assignments were really supposed to be

insertAfter(A,B)	insertAfter(A,C)

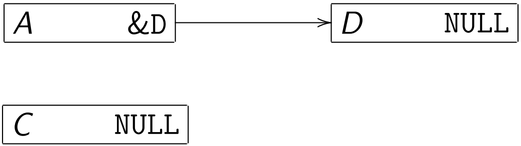
while(!done)
  ATOMICALLY
    if A->next == D
      A->next = B
    else
      done = 1
	while(!done)
	  ATOMICALLY
	    if A->next == D
	      A->next = C
	    else
	      done = 1

- This translates into

```
while(!done)
   n = A->next;
   done = (CAS(&A->next,n,B) == n)
```

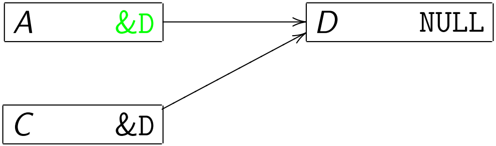
- CAS will let us do assignment when the data matches and will bail out when it doesn't!

Insertion into a Lock-free Linked List
Simple case, setup



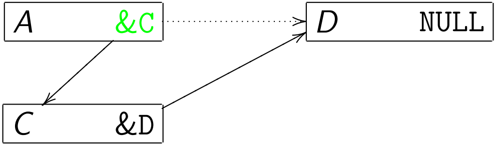
- Some thread constructs the bottom node *C*; wishes to place it between the two above, *A* and *D*.
- `new = newNode(C);`
- `prev = findLabel(A); /* == &A */`

Insertion into a Lock-free Linked List
Simple case, first step



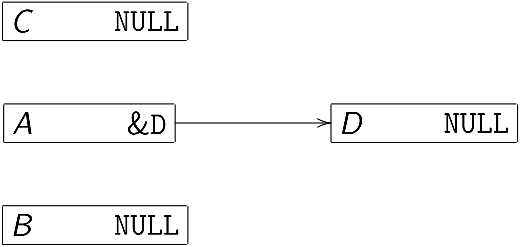
- Thread points *C* node's next into list at *D*.
- `C.next = A.next;`

Insertion into a Lock-free Linked List
Simple case, second step



- `CAS(&A.next, &D, &C);`

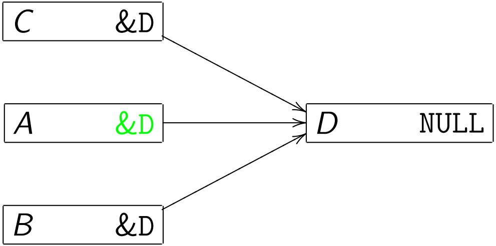
Insertion into a Lock-free Linked List
Race case, setup



- Two threads get their respective nodes B and C.

new = newNode(B);	new = newNode(C);
prev = &A	prev = &A

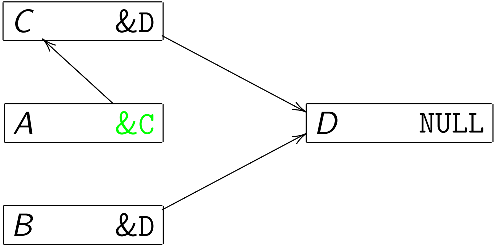
Insertion into a Lock-free Linked List
Race case, first step



- Both set their new node's next pointer.

B.next=&D	C.next=&D

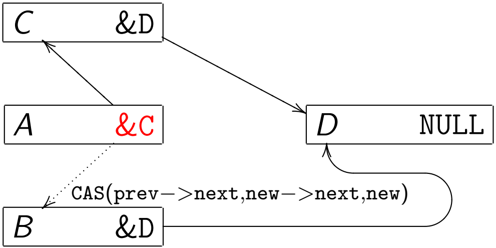
Insertion into a Lock-free Linked List
Race case, first thread



- Thread C goes first ...

```
CAS(&A->next, D, C)
```

Insertion into a Lock-free Linked List
Race case, second thread



- And the other (owning *B*)...

```
CAS(&A->next, D, B)
```

- Fails since `A->next == C`, not `D`.
- So this thread tries again.

Insertion into a Lock-free Linked List

- Rewrite the insertion code to be

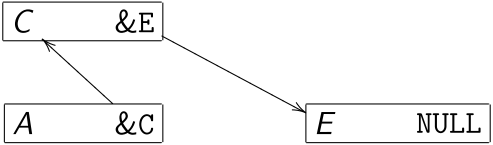
```
insertAfter(after, newlabel) {
   new = newNode(newlabel);
   do {
     prev = findLabel(after);
     expected = new->next = prev->next;
   } while
     ( CAS(&prev->next, expected, new)
                != expected);
}
```

That's great!

- It works!
  - No locks!
  - Threads can simultaneously scan and scan the list...
  - Threads can simultaneously scan and *modify* the list!
  - Threads can simultaneously *modify* and modify the list!
- All those while loops... (retrying over and over?)
  - Remember, mutexes had while loops too...
    - maybe even around `CAS()`!
  - Here, whenever we retry we *know* somebody else got work done!
- Are we done?
  - Most data structures need to support deletion as well ...

Deletion is easy?

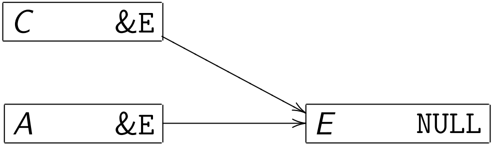
- Suppose we have



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

Deletion is easy?

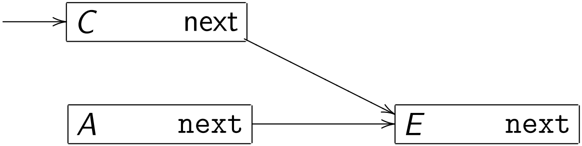
- Now we have



- Great, looks like deletion to me!

Deletion is easy?
Continued

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



- We don't know when that thread is done with C!
- So we can never `free(C);`

Deletion is easy?
What's to be done?

- Deletion turns out to be connected with the infamous "ABA problem."
- We need *some* way to reclaim that memory for reuse..
- (Some implementations cheat and assume as stop-the-world garbage collector.)
- Doing this honestly is remarkably tricky!

ABA Problem

- A problem of confused identity

global = malloc(sizeof(Foo))		//0x1337
local_1 = global	local_2 = global	
global = NULL		
free(local_1)		//0x1337
global = malloc(sizeof(Foo))		//0x1337
	/* 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

- We begin with an innocent linked list:

- Where `head` is a a global pointer to the list.
- We're just going to do operations at the head – treating the list like a stack.

ABA Problem
Pop

- We begin with a linked list:

- Removing the head looks like

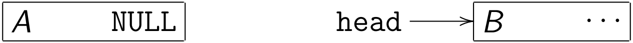
`lhead = head`	`/* == &A */`
`lnext = lhead->next`	`/* == &B */`
`CAS(head, lhead, lnext);`	

- If the `CAS` is successful, we are done, and the list is

- If not, start over.

ABA Problem
Push

- We begin with a linked list and private item



- Inserting at the head looks like

`lhead = head`	`/* == &B */`
`A.next = lhead`	`/* A points at B */`
`CAS(head, lhead, &A);`	

- If the `CAS` is successful, we are done, and the list is

- If not, start over.

ABA Problem
And now it breaks!

Here's a 30,000 foot look at how this is going to break.

Thread 1	**Thread 2**
P	Pop
o	Use memory
p	Push
BANG!	

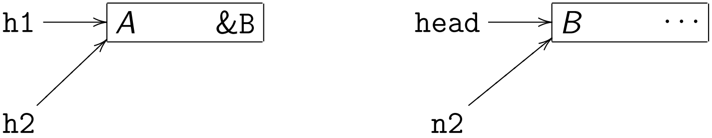
- In words: An extremely, agonizingly slow pop is racing against a pop and a push, with some scribbling in the middle.
- All operations are going to be aimed at the same node, *A*.
- The end is catastrophe.

ABA Problem

- The first thread gets one instruction into its pop, while
- The second thread completes its pop operation:

`h1 = head`	`h2 = head`	`== &A`
	`n2 = h2->next`	`== &B`
	`CAS(head, h2, n2)`	Success!

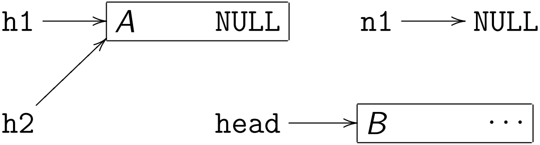
- The world now looks like



ABA Problem

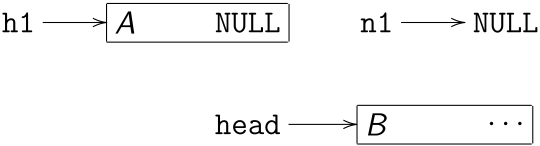
- Now the faster thread is going to do something to the node it just popped, and the slower one is going to make a little more progress...

	`h2->next = NULL;`	Use memory
`n1 = h1->next`		`== NULL`

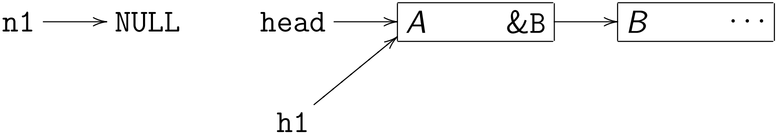


ABA Problem

- Now the faster thread does its push operation...

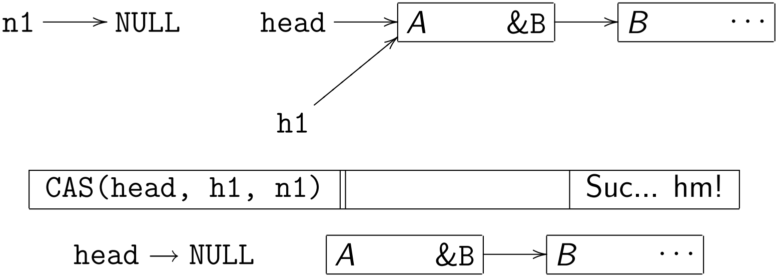


	`h2 = head;`	`== &B`
	`A.next = h2;`	`== &B`
	`CAS(head, h2, &A)`	Success!



ABA Problem

- And the slower thread finally completes its pop operation...



ABA Problem

- The left thread missed its chance to be notifed of having stale data.
    - Notice that the choice of writing NULL was arbitrary.
    - In particular, we might have instead done a much larger series of operations.
    - All that matters is that *A* ended up back on the list head when Thread 1 was CAS-ing.
- In punishment, the datastructure is now broken!

Fixing ABA

- It turns out that we need a more sophisticated delete (and maybe insert and lookup!) function. Look at [Fomitchev and Ruppert(2004)] or [Michael(2002a)] (or others) for more details.
- Generation counters are a simple way to solve ABA
    - Let's replace all pointers with

```
struct versioned_ptr {
   void * p; /* Pointer */
   unsigned int c; /* Counter */
};
```

- This will allow a "reasonably large" number of pointer updates before we have to worry.

Fixing ABA

- Suppose we had a primitive which let us write things like
  ATOMICALLY

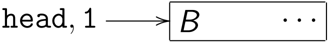
```
if ((A.next.p == &C) && (A.next.version == 4))
  A.next.p = &D
  A.next.version = 5
```

Fixing ABA

- Like CAS, we want a CAS2, which operates on two (adjacent) words at once:
  CAS2(ptr[2], expect[2], new[2]) atomically:
```
 old[0] = ptr[0]; old[1] = ptr[1];
 if (ptr[0] == expect[0] && ptr[1] == expect[1])
   ptr[0] = new[0]; ptr[1]= new[1];
 return { old[0], old[1] };
```
- CAS2 looks more expensive than CAS?
  - Two reads, two writes.
  - With luck, it's one cache line; without, it could be two.
  - May be $(1+\epsilon)$ times as hard as CAS...
  - May be ∞ times as hard as CAS...

Fixing ABA

`h1 = head.p`	`h2 = head.p`	`== &A`
	`n2 = h2->next.p`	`== &B`
	`c2 = head.c`	`== 0`
	`CAS2(head,{h2,c2},{n2,c2+1})`	Success!



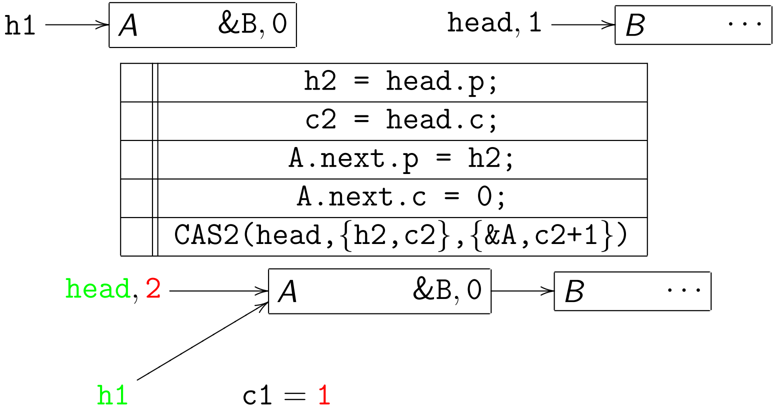
Fixing ABA

n1 = h1->next.p	
c1 = head.c	

- These are just local variables in preparation for...
  `CAS2(head,{h1,c1},{n1,c1+1})`
- If that were to happen *right now* ...

c1 = 1

Fixing ABA



- `CAS2(head,{h1,c1},{n1,c1+1})`
- `head == h1` but `c1 == 1` $\neq$ 2. Hooray!

Some real algorithms?

- [Michael(2002a)] 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.
    - SMR actually solves ABA as a side effect of safely reclaiming memory. Instead of blocking the writer until everybody leaves a critical section, it can efficiently scan to see if threads are interested in a particular chunk 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.

Read-Copy-Update Mutual Exclusion
Preliminaries

- The ABA problems would all be solved if we could wait for everyone who might have read what is now a stale pointer to complete.
- Phrased slightly differently, we need to separate the *memory update* phase from the *reclaim* (`free()`) phase.
- And ensure that no readers hold a critical section that might see the update *and* reclaim phases.
  - Seeing one or the other is OK!

Read-Copy-Update Mutual Exclusion
Preliminaries

- Read-Copy-Update (RCU, [Wikipedia(2006a), McKenney(2003)]; earlier papers) uses techniques from lock-free programming.
- Is used in several OSes, including Linux.
- It's a bit more complicated than the examples given here and not truly lock-free, but certainly interesting.

Read-Copy-Update Mutual Exclusion
Preliminaries

- Looks like a reader-writer lock from $30,000$ ft.
- Key observations:
  - Many more readers than writers.
  - Readers frequently can complete critical sections in bounded time.
    - In particular: never have to `yield()`.
  - Readers want to see a consistent datastructure.
  - The ABA problems would all be solved if we could force everybody who might have read what is now a stale pointer to complete.

Read-Copy-Update Mutual Exclusion
Preliminaries

- Many more readers than writers.
  - So we should make sure that the readers don't have to do much.
  - Kind of like a rwlock.
- Readers frequently can complete critical sections in bounded time.
  - Required property of RCU readers.
  - We'll see why this is important in a bit.
- Readers want to see a consistent datastructure.
  - Not all consistency guarantees need to be kept, but, for example, we want to avoid use-after-free and the possibility of faulting.
  - But it might be the case that we let `node->next->prev != node` as readers only use these pointers to traverse.

Read-Copy-Update Mutual Exclusion
Preliminaries

- Disclaimer: function names have been changed from, *e.g.*, the Linux implementation, to make the meanings more clear.
- Disclaimer 2: RCU comes in many flavors - the one here is a small toy model but works on real hardware (like Pebbles).

Read-Copy-Update Mutual Exclusion API

- Reader critical section functions.
  - `void rcu_read_lock(void);`
  - `void rcu_read_unlock(void);`
  - Note the absence of parameters (how odd!).
- Accessor functions:
  - `void * rcu_fetch(void *);` is used to fetch a pointer from an RCU protected data structure.
  - `void * rcu_assign(void *, void *);` is used to assign a new value to an RCU protected pointer.
- Synchronization points:
  - `void rcu_synchronize(void);` is used once a writer is finished to signal that updates are complete.
    - Moves from “update” to “reclaim” phase.

Read-Copy-Update Mutual Exclusion API: Reader's View

- Suppose we have a global list, called `list`, that we want to read under RCU.
- The code for iteration looks like

```
rcu_read_lock();
list_head_t *llist = rcu_fetch(list);
list_node_t *node = rcu_fetch(llist->head);
while(node != NULL) {
  ... /* Do something reader-like */
  node = rcu_fetch(node->next);
}
rcu_read_unlock();
```

Read-Copy-Update Mutual Exclusion
API: Writer's View

- Suppose we want to delete the head of the same global list, `list`.
- We need to give it a writer exclusion mutex, `list_wlock`.

```
void delete_head_of_list() {
   list_node_t *head;
   mutex_lock(&list_wlock); // No other writers
     head = list->head; // No rcu_fetch()
     list_node_t *next = head->next;
     rcu_assign(list, next);
     rcu_synchronize();
   mutex_unlock(&list_wlock);
   free(head); /* Reclaim phase */
}
```

Read-Copy-Update Mutual Exclusion
API: Summary

- This is kinda like a rwlock:
    - It allows an arbitrary number of readers to run against each other.
    - It prevents multiple writers from writing at once.
- It is absolutely unlike a rwlock because
    - readers and writers do not exclude each other!

Read-Copy-Update Mutual Exclusion API: Wait, WHAT*?*

- Readers can run alongside writers! There's no mechanism in the reader to serialize against the writer! See:

CPU 1 (reader)	CPU 2 (writer)
`rcu_read_lock();`	`mutex_lock(...);`
`llist = rcu_fetch(list);`	`...`
	`rcu_assign(list, new);`
	`rcu_synchronize();`
`rcu_fetch(llist->head);`	

Read-Copy-Update Mutual Exclusion Implementation: Key Ideas

- "All the magic is inside `rcu_synchronize()`" ...
- The deletion problem, and ABA, was a problem of not knowing when nobody had a stale reference.
- If
    - readers agree to drop *all* references in bounded time
    - AND writers can tell when readers have dropped references
- Then we know when it is safe to reclaim (*i.e.* `free()`) memory.
- Being safe for *reclaim* is exactly the same as being safe for *reuse*.

Read-Copy-Update Mutual Exclusion Implementation: Approximation

- Want:
  - readers agree to drop *all* references in bounded time
  - AND writers can tell when readers have dropped references
- You can imagine that there's an array of `reading[i]` values out there, with each thread having its own index...
- Each reader sets `reading[me] = 1`, reads, then sets `reading[me] = 0`.
- The writer then scans the array looking for all flags to be 0.
- When this happens, the writer knows that no readers have stale references, and all is well!

Read-Copy-Update Mutual Exclusion Implementation

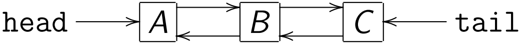
- So how does RCU *actually* do this?
  - "All the magic is inside `rcu_synchronize()`" ...
- `rcu_read_lock()` simply disables the local CPU's preemptive scheduler.
  - So we need readers that won't call `yield()`.
- `rcu_assign()` inserts a write memory barrier ("write fence") to force all writes in the out-of-order buffers to be made visible *before* it does the assignment requested.
- `rcu_fetch(x)` is just `x` on most architectures.
  - There are exceptions (notably, DEC ALPHA systems).

Read-Copy-Update Mutual Exclusion Implementation

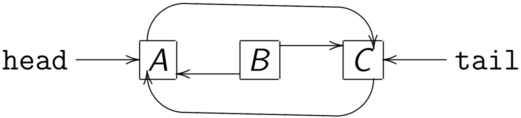
- Given all of this, what does `rcu_synchronize()` do?
- It waits until every CPU undergoes a context switch!
  - Could just have a context switch counter per CPU and wait for each to fire, or...
  - Ensure that the thread calling synchronize gets run on every CPU before the synchronize returns (using something like `move_me_to_cpu(int cpunum);`)
- Because readers are non-preemptible, this will force all critical sections that began before the synchronize to complete before the writer can enter reclaim phase.
- That enables safe reclaim and as a side-effect solves the ABA problem for us!

Read-Copy-Update Mutual Exclusion
Pictures: Writer view

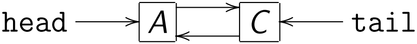
- Let's again take a linked list, this time a doubly linked one.



- Now suppose the writer acquires the write lock and updates to delete B:

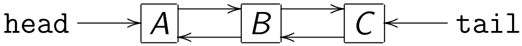


- Now the writer synchronizes, forcing all readers with references to B out of the list. Only then can B be reclaimed!

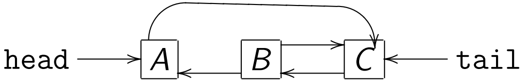


Read-Copy-Update Mutual Exclusion
Pictures: Reader View

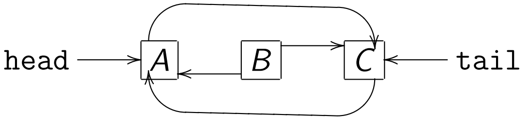
- Looking at that again, from the reader's side now. Originally



- The writer first sets it to



- And then



Read-Copy-Update Mutual Exclusion
Pictures

- The writer forced memory consistency (fencing) between each update.
- So each reader's dereference occurred *entirely before* or *entirely after* each write.
- So the reader's traversal in either direction is entirely consistent!
- Though moving back and forth might expose the writer's action.
- But it's OK, because we'll just see a disconnected node.
- It's not *gone* yet, just disconnected.
- It won't be reclaimed until we drop our critical section.

Tradeoffs
What Have We Learned?

- We can replace fixed-time lock-based critical sections with "almost-always-fixed-time" compare-and-swap loops...
  - Note that getting the lock was not fixed-time, just the critical section.
  - `CAS` is a kind of critical mini-section in hardware.
- Because many threads may have references into a data structure, knowing when something has no references is both very *important* and very *difficult*.
  - But all is not lost!
  - Generation counters and RCU offer paths to salvation.
  - There are others, for the curious.

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 the debugging step!

Tradeoffs
Lockfree vs. Locking.

- Most lock-free algorithms increase the number of atomic operations, compared to the lockful variants.
- Thus we starve processors for bus activity on 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.
    - Nobody can get all the cachelines to execute an instruction before a request comes in and and steals one of the ones they had.

Tradeoffs
Locking vs. RCU

- Interestingly, RCU tends to decrease the number of atomic operations.
  - It can because it requires readers to be non-blocking and can interact with the scheduler.
- RCU requires the ability to force a thread to run on every CPU or at least observe when every CPU has context switched.
  - Difficult to use RCU in userland!
- RCU still suffers a slowdown from cache line shuffling, but will make progress due to having at most one writer.

Conclusion

- Lock-free datastructures are extremely cool.
- Understanding them
  - Uses clever hardware features
    - This is probably good for one's soul anyway.
    - Hardware is only going to get more "clever."
  - Leads to real-world tools like RCU.
  - Gives a topic for conversation at parties.
- Lock-free algorithms proper have their place, but that place is somewhat small.
  - Generally more complex than standard lockful algorithms.
  - Much harder ("impossible?") to debug.
  - Usually used only when there is no other option.

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

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

Wikipedia, *Read-copy-update* (2006a), http://en.wikipedia.org/wiki/Read-copy-update.

P. McKenney (2003), http://www.linuxjournal.com/article/6993.

Wikipedia, *Lock-free and wait-free algorithms* (2006b), http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms.

- Wikipedia, *Non-blocking synchronization* (2006c), http://en.wikipedia.org/wiki/Non-blocking_synchronization.
- M. M. Michael, PODC pp. 1–10 (2002b), http://www.research.ibm.com/people/m/michael/podc-2002.pdf.
- M. M. Michael, IEEECS pp. 1–10 (2004), http://www.research.ibm.com/people/m/michael/podc-2002.pdf.
- H. Sundell, in *International Parallel and Distributed Processing Symposium* (IEEE, 2005), 1530-2075/05, http://ieeexplore.ieee.org/iel5/9722/30685/01419843.pdf?tp=&arnumber=1419843&isnumber=30685.

P. Memishian, *On locking* (2006),
http://blogs.sun.com/meem/entry/on_locking.

Acknowledgements

- Dave Eckhardt (de0u) has seen this lecture about as often as I have, and has produced useful commentary on every release.
- 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.
- [Nobody on this list deserves any of the blame, but merely credit, for this lecture.]

Full fledged deletion & reclaim

- Even though we might be able to solve ABA, it still doesn't solve memory reclaim!
- Imagine that instead of being reclaimed by the list, the deleted node before had been reclaimed by something else...
    - A different list
    - A tree
    - For use as a thread control block

Full fledged deletion & reclaim

- What if we looked at ABA differently ...
- It only matters if there is the possibility of confusion.
- In particular, might demonstrate strong interest in things that might confuse me
    - Hazard Pointers ("Safe Memory Reclaimation" or just "SMR") [Michael(2002b)] and [Michael(2004)]
    - Wait-free reference counters [Sundell(2005)]
- These are ways of asking "If I, Thread 189236, were to put something here, would anybody be confused?"
- This solves ABA, but really as a side effect: it lets us reclaim address space (and therefore memory) because we know nobody's using it!

The SMR Algorithm

- Every thread comes pre-equipped with a *finite* list of "hazards"
- Memory reclaim involves scanning everybody's hazards to see if there's a collision
- Threads doing reclaim `yield()` (to the objecting thread) until the hazard is clear
- Difficulty
    - Show that hazards can only decrease when deletions are pending
    - Show that deletions eventually succeed (can't deadlock on hazards)
    - Managing the list of threads' hazards is difficult

Observation On Object Lifetime

Instance of a general problem [Memishian(2006)]:

> *Things get tricky when the object must go away. [...] Any thread looking up the object – by definition – does not yet have the object and thus cannot hold the object's lock during the lookup operation. [...] Thus, whatever higher-level synchronization is used to coordinate the threads looking up the object must also be used as part of removing the object from visibility.*