Lock-free Programming

Nathaniel Wesley Filardo
David A. Eckhardt

November 20, 2015

Context

- Things recently have been confusing!

Context

- Things recently have been confusing!
  - Instructions run “out-of-order” on every CPU!
  - Single data items are cached many times on many CPUs!
  - Causality is violated between variables!
- How can any program work??

Context

- Within a CPU
  - Instructions run "out-of-order"
  - *Data dependencies* delay when instructions start
  - Instruction outcomes are *published* when they are safe
  - → It is possible to write single-threaded code.

Context

- Within a CPU
  - Instructions run "out-of-order"
  - *Data dependencies* delay when instructions start
  - Instruction outcomes are *published* when they are safe
  - → It is possible to write single-threaded code.
- Cache coherence
  - Caches talk to each other with a MSI-like protocol
  - → All caches return an up-to-date version of each cache line

Context

- Within a CPU
  - Instructions run "out-of-order"
  - *Data dependencies* delay when instructions start
  - Instruction outcomes are *published* when they are safe
  - → It is possible to write single-threaded code.
- Cache coherence
  - Caches talk to each other with a MSI-like protocol
  - → All caches return an up-to-date version of each cache line
- Memory consistency
  - *Barrier instructions* separate code regions that import/export data across threads
  - → Programs can depend on causality across multiple cache lines

Today

- Lock-free programming
  - A particular kind of multi-threaded code
  - Multi-threaded access to a single data structure - without locks!

Today

- Lock-free programming
  - A particular kind of multi-threaded code
  - Multi-threaded access to a single data structure - without locks!
  - Something people might expect you to know about
  - An example to help think about modern machines

Today

- Lock-free programming
  - A particular kind of multi-threaded code
  - Multi-threaded access to a single data structure - without locks!
  - Something people might expect you to know about
  - An example to help think about modern machines
  - (Not a kind of code most people write.)

Outline

Introduction

Lock-Free Linked List Insertion

Lock-Free Linked List Deletion

Read-Copy-Update Mutual Exclusion

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 Might Take A While

- Consider `XCHG` style locks which use
  ```
  while( xchg( &locked, LOCKED ) == LOCKED )
  ```
  as their core operation.

Introduction
Locks Might Take A While

- Consider `XCHG` style locks which use
  ```
  while( xchg( &locked, LOCKED ) == LOCKED )
  ```
  as their core operation.
- We could spend an unbounded amount of time here spinning. . .
- *Contended* locks will have very high latency. . .

Introduction
Locks Might Take A While

- Consider `XCHG` style locks which use
  ```
  while( xchg( &locked, LOCKED ) == LOCKED )
  ```
  as their core operation.
- We could spend an unbounded amount of time here spinning...
- *Contended* locks will have very high latency...
- Locks *by definition* reduce parallelism.

Introduction
Locks Might Take A While

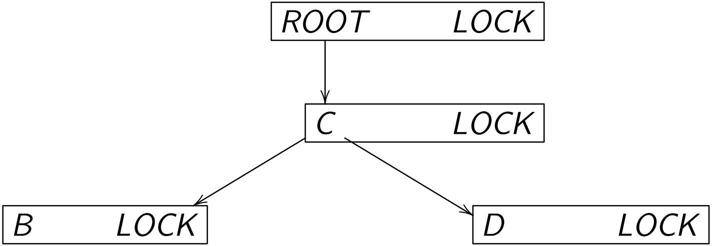
- Locks *by definition* reduce parallelism.
  - If N people are contending for a lock, $N - 1$ of them are just wasting time.
  - "It would be nice" if they could all work at once ...
  - ... but this requires a way to "handle" data-structure conflicts.

Introduction
Locks Might Take A While

- For a large data structure, we would like multiple *local* (independent) operations to be allowed concurrently.
    - *e.g.* "lookup" and "insert" in parallel threads
- Approaches:
    - "Data structure full of locks" — today
    - Lock-free data structures — today
    - "Hardware transactional memory" [Her] — not today

Introduction
Locks Can Be. . . Not So Bad?

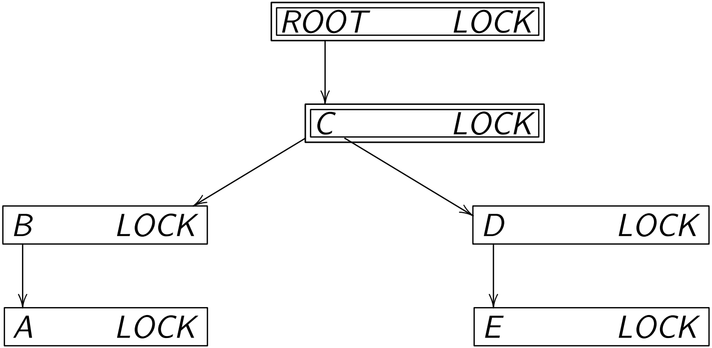
- Instead of a lock around a tree, we could have a tree with locks:



- The protocol: lock the root, then (lock child & unlock parent) as you go down.
  - This kind of *lock handoff* is a very common design.

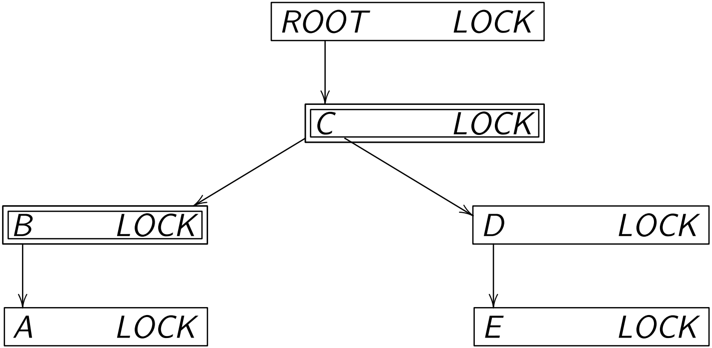
Introduction
Locks Can Be...Not So Bad?

- Trying to find node A.
- Step 1: lock root pointer and top node



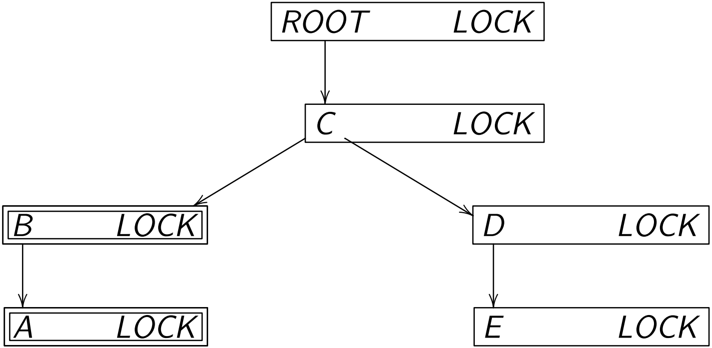
Introduction
Locks Can Be…Not So Bad?

- Trying to find node A.
- Step 2: lock left child and unlock parent.



Introduction
Locks Can Be…Not So Bad?

- Trying to find node A.
- Step 3: lock left child and unlock parent



Introduction

- This dance is sometimes called “hand-over-hand locking”.
- In a binary tree, each traversal by one thread “opens half of the tree” for other threads.

Introduction

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

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"

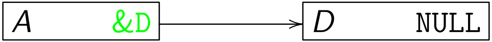
insertAfter(A,B)	insertAfter(A,C)
prev = &A	
B.next=A.next	
A.next=&B	
	prev = &A
	C.next=A.next
	A.next=&C

Insertion into a Linked List Without Locks
Race trace

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

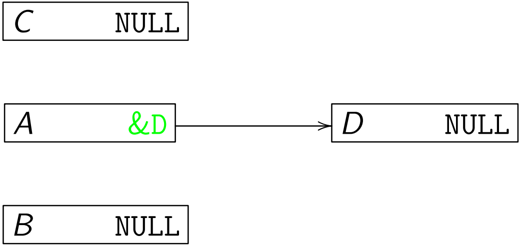
- Either of these assignments makes sense in isolation, but one of them will override the other!

Insertion into a Linked List Without Locks
Precondition



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

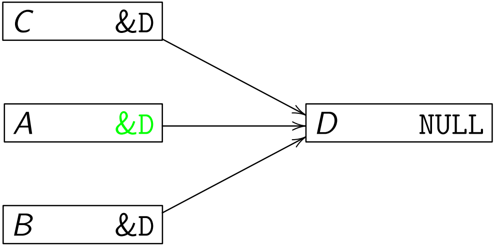
Insertion into a Linked List Without Locks
First step



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

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

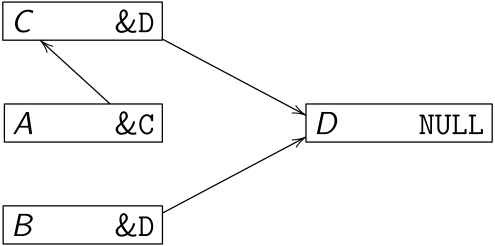
Insertion into a Linked List Without Locks
Second step



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

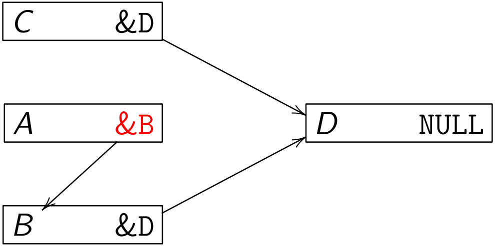
B.next=&D	C.next=&D

Insertion into a Linked List Without Locks
One thread goes



- Suppose the thread owning *C* completes its assignment first.

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`" (oops!)

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`" (oops!)
- How to fix that?

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`" (oops!)
- How to fix that?
    1. 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...

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`" (oops!)
- How to fix that?
  1. 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...
  2. The pattern for today
     - 2.1 *Assume* update collisions happen rarely
     - 2.2 *Detect* when they do happen — hardware support
     - 2.3 Figure out how to *"try again"*

Insertion into a Linked List Without Locks
The Lock Free / Transactional Approach

```
while(not done)
    Determine preconditions for the update
    Prepare for update
    ATOMICALLY
      if(preconditions still hold)
        make update;
        done = true;
```

- Does this pattern finish in bounded time?

Insertion into a Linked List Without Locks
The Lock Free / Transactional Approach

```
while(not done)
    Determine preconditions for the update
    Prepare for update
    ATOMICALLY
      if(preconditions still hold)
        make update;
        done = true;
```

- Does this pattern finish in bounded time?
  - No: could “encounter trouble” unboundedly.
- But if threads “almost never” spatially collide...
  - We gain “a lot” of parallelism by deleting locks.
  - We pay “a little” work handling retries.

Insertion into a Linked List Without Locks

- Re-writing list-insert in this pattern:

insertAfter(A,B)	insertAfter(A,C)
while(!done)	while(!done)
findLabel(A)	findLabel(A)
ATOMICALLY if (A->next == D) A->next = B done = 1	*ATOMICALLY* if (A->next == D) A->next = C 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

  ```
  // "lock bus" (not really)
  old_val = *ptr;
  *ptr = val;
  // "unlock bus" (not really)
  return old_val;
  ```
- Summary: one fetch and one store under (mini) lock.

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:

- Still one read, one write under same lock.
- (logic time $\ll$ memory time)

Insertion into a Lock-free Linked List

- Our assignments were really supposed to be

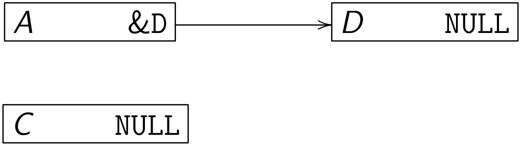
insertAfter(A,B)	insertAfter(A,C)
while(!done)	while(!done)
findLabel(A)	findLabel(A)
ATOMICALLY if (A->next == D) A->next = B done = 1	*ATOMICALLY* if (A->next == D) A->next = C done = 1

- This translates into

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

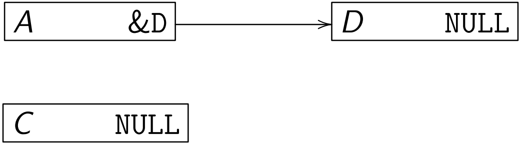
- CAS will assign if match, or bail otherwise.

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

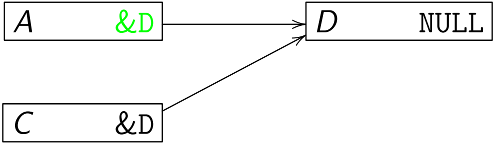


- Some thread constructs the bottom node C; wishes to place it between the two above, A and D.
- `new = newNode(C);`
- `prev = findLabel(A); /* == &A */`

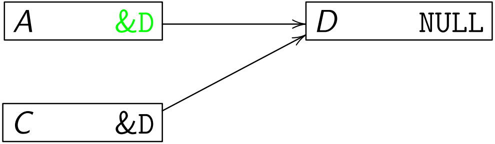
Insertion into a Lock-free Linked List
Simple case, first step



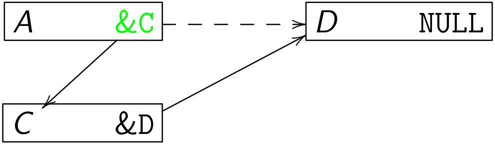
- Thread points C node's next into list at D.
- `C.next = A.next;`



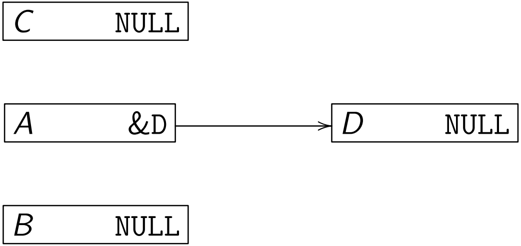
Insertion into a Lock-free Linked List
Simple case, second step



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



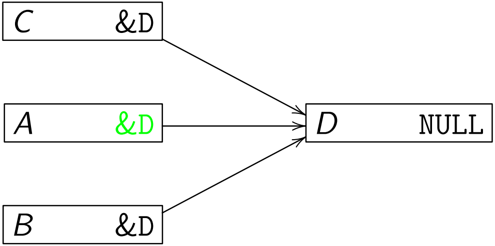
Insertion into a Lock-free Linked List
Race case, setup



- Two threads get their respective nodes *B* and *C*.

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

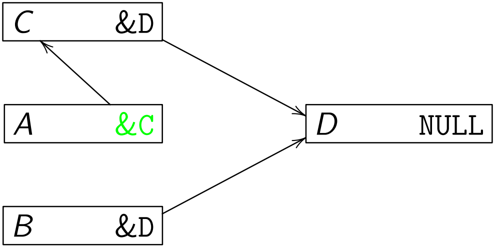
Insertion into a Lock-free Linked List
Race case, first step



- Both set their new node's next pointer.

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

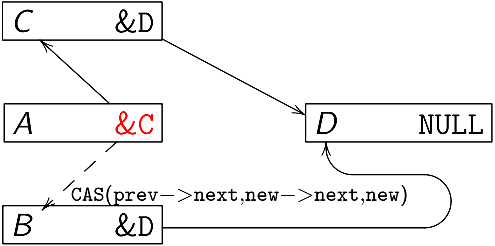
Insertion into a Lock-free Linked List
Race case, first thread



- 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

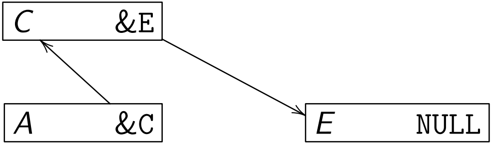
```
insertAfter(after, newlabel) {
   new = newNode(newlabel);
   do {
     prev = findLabel(after);
     expected = new->next = prev->next;
   } while
     ( CAS(&prev->next, expected, new)
                != expected);
}
```

That's great!

- It works!
  - No locks!
  - Threads can simultaneously scan and scan the list...
  - Threads can simultaneously scan and *grow* the list!
  - Threads can simultaneously *grow* and grow 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?
  - Have we implemented all the standard operations?

Deletion is easy?

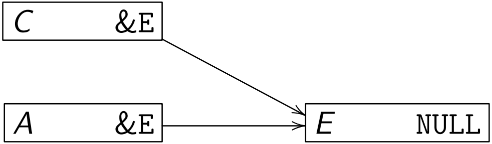
- Suppose we have



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

Deletion is easy?

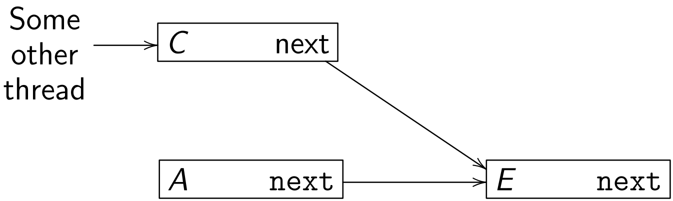
- Now we have



- Great, looks like deletion to me!
    - It's off the data-structure (*logically deleted*) ⋯
- But not *freed* (*"actually" deleted / reclaimed*).

Deletion is easy?
Continued

- 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?

- We need *some* way to reclaim that memory for reuse..
- Some implementations cheat and assume a stop-the-world garbage collector.
    - (That's like a giant lock!)
- Doing deletion honestly is remarkably tricky!
    - We're not going to really have time to cover it.

Deletion is easy?
What's to be done?

- Assume: once some memory is committed to being a LF list node that it's OK if it's *always* a LF list node.
- So we can have two lists: the "real" list and a "free" list.
  - This is not real `free()` but is hard enough.
- In particular, we run into the "ABA problem".

ABA Problem: Introduction

- 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 = ...

ABA Problem: Introduction

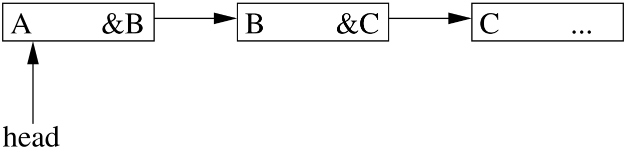
- A problem of confused identity

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 point to the same address, they don't *really* mean the same thing.

ABA Problem: Introduction
Preliminaries

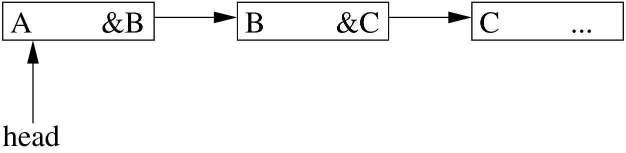
- We begin with an innocent linked list:



- 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: Introduction
Pop

- We begin with a linked list:



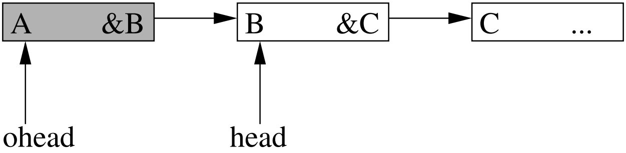
- Removing the head looks like

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

- If not, retry.

ABA Problem: Introduction
Pop

- If successful,



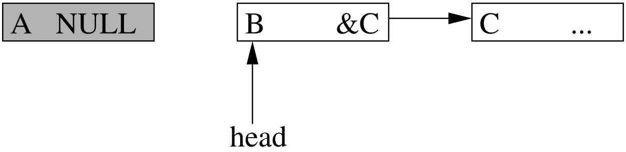
- is the result of

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

- If not, retry.

ABA Problem: Introduction
Push

- We begin with a linked list and private item



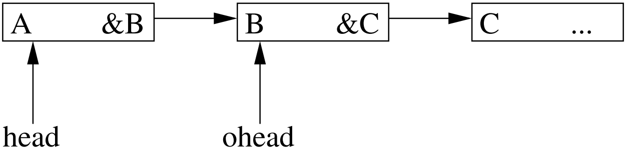
- Inserting at the head looks like

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

- If not, retry.

ABA Problem: Introduction
Push

- If that works, we get



- from

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

- If not, retry.

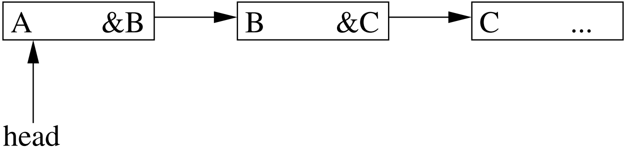
ABA Problem: Things go south
And now it breaks!

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

<table>
<tr><th>Thread 1</th><th>Thread 2</th><th>Thread 3</th></tr>
<tr><td rowspan="3">Pop</td><td>Pop</td><td></td></tr>
<tr><td></td><td>Pop</td></tr>
<tr><td>Push</td><td></td></tr>
<tr><td colspan="3">BANG!</td></tr>
</table>

- An extremely slow pop is racing against
  - A thread which pops and then immediately pushes.
  - A third which thread executes a pop.

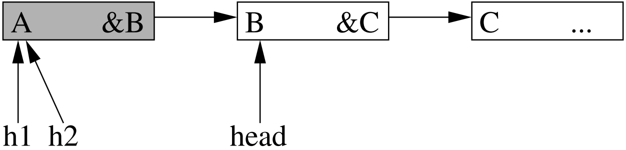
ABA Problem: Things go south



- 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!

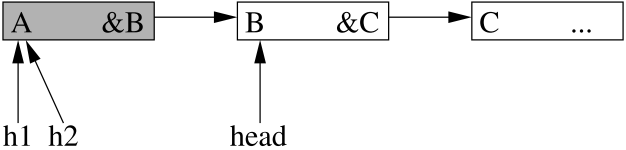
ABA Problem: Things go south



- The first thread got one instruction into its pop, while
- The second thread completed its pop operation.

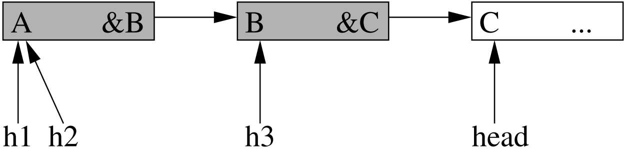
`h1 = head`	`h2 = head`	`== &A`
	`n2 = h2->next`	`== &B`
	`CAS(head, h2, n2)`	Success!

ABA Problem: Things go south



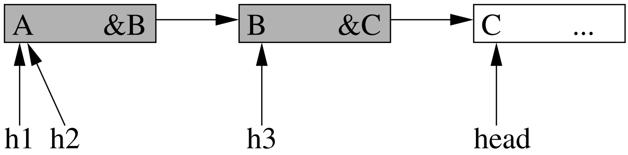
- The third thread executes a pop operation.

ABA Problem: Things go south



- The third thread executed a pop operation.

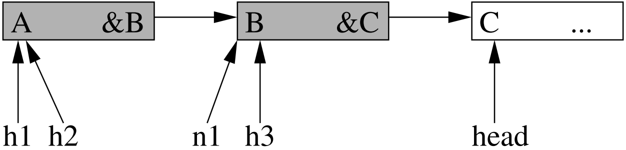
ABA Problem: Things go south



And the slower thread gets a few more instructions:

`n1 = h1->next;`		`== &B`

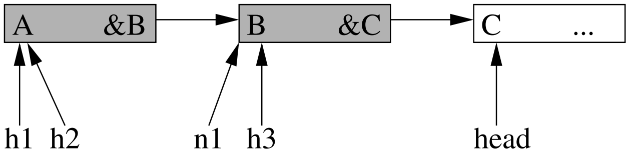
ABA Problem: Things go south



And the slower thread got a few more instructions:

`n1 = h1->next;`		`== &B`

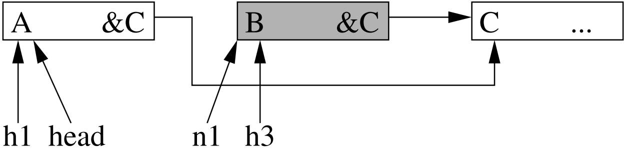
ABA Problem: Things go south



Now the second thread does its push operation...

	`h2 = head;`	`== &C`
	`h2->next = h2;`	`A.next` $\leftarrow$ `&C`
	`CAS(head, h2, &A)`	Success!

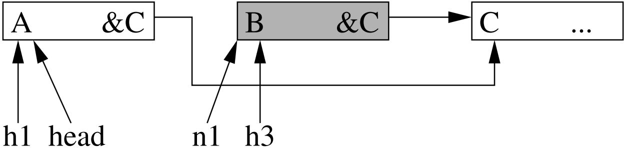
ABA Problem: Things go south



Now the second thread did its push operation...

	`h2 = head;`	`== &C`
	`h2->next = h2;`	`A.next` ← `&C`
	`CAS(head, h2, &A)`	Success!

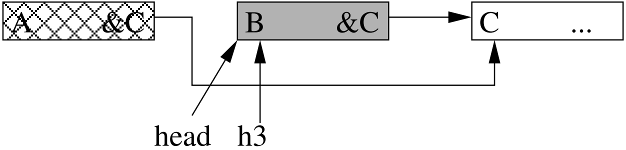
ABA Problem: Things go south



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

`CAS(head, h1, n1)`		Success!

ABA Problem: Things go south



And the slower thread finally completed its pop operation...

`CAS(head, h1, n1)`		Success?

B, which was well and quite off the list, and not owned by Thread 1, is now at the head!

ABA Problem: Things go south

- Thread 1 missed its chance to be notifed of having stale data.
    - All that matters is that *A* ended up back on the list head when Thread 1 was CAS-ing.
- There's relatively little that *thread 1* can do about this!
- For fun, try designing a different failure case.
    - Try getting a circular list.

Fixing ABA

- Generation counters are a simple way to solve ABA
  - Let's replace all pointers with

```
struct versioned_ptr {
    void * p; /* Pointer */
    unsigned int v; /* Version */
};
```

- 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 ((head.p == &C) && (head.v == 4))
  head.p = &D
  head.v = 5
```

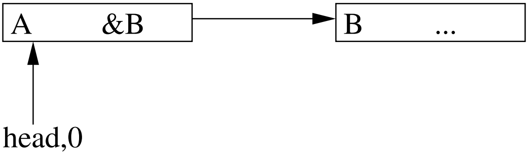
Fixing ABA

- Like CAS, we want a CAS2, which operates on two (adjacent) words at once:

```
CAS2(*curs, *expects, *news) atomically:
 olds[0] = curs[0]; olds[1] = curs[1];
 if (curs[0]==expects[0] && curs[1]==expects[1])
    curs[0] = news[0]; curs[1]= news[1];
 return { olds[0], olds[1] };
```

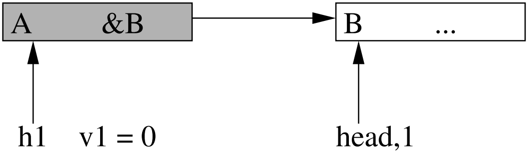
- CAS2 looks more expensive than CAS?
  - Two reads, two writes.
  - With luck, it's one cache line; without, it could be two.
  - May be $(1 + \epsilon)$ times as hard as CAS...
  - May be ∞ times as hard as CAS...

Fixing ABA
2^{nd} thread pops...



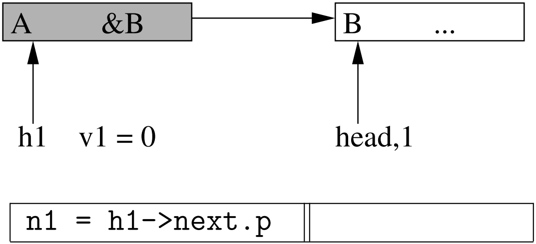
h1 = head.p v1 = head.v	h2 = head.p	== &A
	n2 = h2->next.p v2 = head.v	== &B == 0
	CAS2(head,{h2,v2},{n2,v2+1})	Success!

Fixing ABA
2^{nd} thread popped. . .



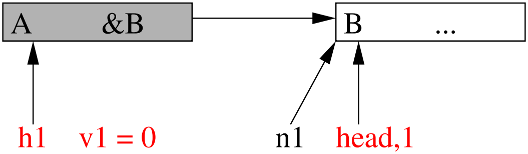
`h1 = head.p`	`h2 = head.p`	`== &A`
	`n2 = h2->next.p`	`== &B`
	`v2 = head.v`	`== 0`
	`CAS2(head,{h2,v2},{n2,v2+1})`	Success!

Fixing ABA
1st thread reads `n1`



- `n1` and `v1` are just local variables in preparation for... `CAS2(head,{h1,v1},{n1,v1+1})`

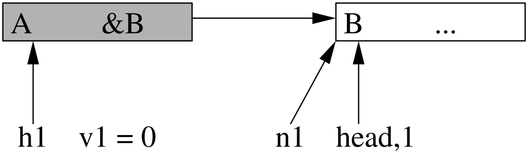
Fixing ABA
1st thread read n1



`n1 = h1->next.p`	

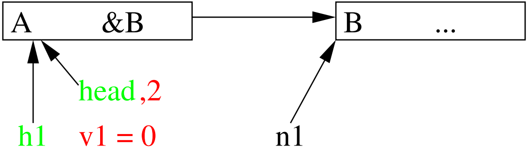
- `n1` and `v1` are just local variables in preparation for... `CAS2(head,{h1,v1},{n1,v1+1})`
- So if that were to happen right now...

Fixing ABA
2nd thread pushes...



	`h2 = head.p;` `v2 = head.v;`
	`A.next = h2;`
	`CAS2(head,{h2,v2},{&A,v2+1})`

Fixing ABA
2nd thread pushed; here's where it broke before



	`h2 = head.p;` `v2 = head.v;`
	`A.next.p = h2;`
	`CAS2(head,{h2,v2},{&A,v2+1})`

- `CAS2(head,{h1,v1},{n1,v1+1})`
- `head == h1` but `v1 == 0` $\neq$ 2. Hooray!

Fixing ABA For Real

- Generation counters kinda stink.
- Be more clever:
  - Find some way to wait until the coast is clear.
  - Look at [FR04] or [Mic02a] (or others) for more details.
- Or use different hardware ("make the EEs do it"):
  - Old world: "Load-Linked/Store-Conditional/Validate"
  - New world: Hardware Transactional Memory
  - These assure you of no ABA because the $A \rightarrow B$ transition nullifies your ability to successfully store (aborts the transaction), even if B turns back into A.
  - To the EEs in the room: no missed edges!

Real-world applications

- CAS-based LF algorithms are relatively rare in the wild.
- But: motivation for transactional memory, which appears to finally be here to stay.
- So: forever more, you will be able to run a chunk of code touching (increasingly large amounts of) memory and "see if it worked."
- A very powerful tool for concurrency design.
  - [RHP$^+$] shows potential neat uses of HTM in Linux.

Read-Copy-Update Mutual Exclusion
Preliminaries

- The deletion problem would 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* (*atomic delete* or *logical delete*) phase from the *private use* (e.g. `free()`) phase.
- And ensure that no readers hold a critical section that might see the update *and* private phases.
    - Seeing one or the other is OK!

Read-Copy-Update Mutual Exclusion
Preliminaries

- Read-Copy-Update (RCU, [Wikc, McK03]; 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 assumptions:
  - Many more readers than writers.
  - Reader critical sections are *short*:
    - No `yield()`, `malloc()`, page faults, ...
  - One writer at a time is OK.
  - Some consistency requirements can be relaxed.
    - Use-after-free, pointers to garbage: definitely bad.
    - Double-linked-list invariant `node->next->prev != node` may be OK if violated during reader execution.
- Big feature: writers can tell when all "earlier" readers are done.

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 function(s):
  - `void * rcu_assign(void *, void *);` is used to assign a new value to an RCU protected pointer.
  - (Other architectures may require more)
- Writer function:
  - `void rcu_wait(void);` called after updates are complete.
    - Move from "update" to "private" 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 = list;
list_node_t *node = llist->head;
while(node != NULL) {
  ... /* Do something reader-like */
  node = node->next;
}
rcu_read_unlock();
```

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

- Example: delete the head of the same global list, `list`.
- Use writer exclusion mutex, `list_wlock`.
- Updates use `rcu_assign()`, finish with `rcu_wait()`.

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

Read-Copy-Update Mutual Exclusion API: Summary

- Like rwlock:
  - It allows an arbitrary number of readers to run together.
  - 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 (at most one!) writer!

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

Read-Copy-Update Mutual Exclusion Implementation: Key Ideas

- "All the magic is inside `rcu_wait()`" ...
- The deletion problem (like 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 consider memory private.
- Being safe for *private use* 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 `looking[i]` values out there, with each thread having its own index...
- Each reader increments `looking[me]` when done.
- The writer then scans waiting for each to change.
- The writer then knows that no readers have stale references, and is now OK to `free` deleted item(s).
- Nice idea, but doesn't work (how sad!)

Read-Copy-Update Mutual Exclusion Implementation

- So how does RCU *actually* do this?
  - "All the magic is inside `rcu_wait()`" ...
- `rcu_read_lock()` simply disables interrupts.
  - So we need readers that won't call `yield()`.
- `rcu_assign()` ensures ordering of writes.
- Too much detail for today's lecture.
- It's "the right kind of write".
- (Inserts a write memory barrier *before* it does the assignment requested.)

Read-Copy-Update Mutual Exclusion Implementation

- Given all of this, what does `rcu_wait()` do?
- It waits until every CPU takes an interrupt!
  - Could just have a counter per CPU and wait for each to fire, or...
- Or! Each `rcu_wait` runs sequentially on each CPU.
  - Because readers are non-preemptible, waiting until all CPUs preempt means that all readers must have dropped their "lock" and so have forgotten any pointers to memory we want to free.

Read-Copy-Update Mutual Exclusion
Confessions of an Instructor

Real-world RCU once upon a time worked this way but more recent implementations are much fancier. For the really enthusiastic, see things like Linux's “Sleepable RCU” implementation [McK06].

Conclusion

- Discussed...
  - “Tree of locks”
  - The lock-free pattern
    - “replace locks with luck (plus detection and fixup)”
  - CAS/CAS2 as “mini-transactions”
  - A simple wrong idea
    - “address == meaning”
  - The “ABA problem”
    - “RCU: Wait for people to leave the room”
- Note: “classical” LF may be replaced by HTM (another lecture)

Conclusion
Words of Warning

- 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 at least don't have to debug that.

Thanks. Questions?

- Mikhail Fomitchev and Eric Ruppert, *Lock-free linked lists and skip lists*, PODC (2004), no. 1-58113-802-4/04/0007, 50–60, http://www.research.ibm.com/people/m/michael/podc-2002.pdf.
- Maurice Herlihy, *Does hardware transactional memory change everything?*
- Paul McKenney, *Kernel Korner - Using RCU in the Linux 2.5 Kernel*, http://www.linuxjournal.com/article/6993.
- Paul McKenny, *Sleepable RCU*, http://lwn.net/Articles/202847/.
- Peter Memishian, *On locking*, July 2006, http://blogs.sun.com/meem/entry/on_locking.

- Maged M. Michael, *High performance dynamic lock-free hash tables and list-based sets*, SPAA (2002), no. 1-58113-529-7/02/0008, 73–83, http://portal.acm.org/ft_gateway.cfm?id=564881&type=pdf&coll=GUIDE&dl=ACM&CFID=73232202&CFTOKEN=1170757.
- ______, *Safe memory reclamation for dynamic lock-free objects using atomic reads and writes*, PODC (2002), no. 1-58113-485-1/02/0007, 1–10, http://www.research.ibm.com/people/m/michael/podc-2002.pdf.
- ______, *Hazard pointers: Safe memory reclamation for lock-free objects*, IEEECS (2004), no. TPDS-0058-0403, 1–10,

http://www.research.ibm.com/people/m/michael/podc-2002.pdf.

Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel, *TxLinux: using and managing hardware transactional memory in an operating system*, ACM SIGOPS Operating Systems Review, vol. 41, ACM, p. 87102.

H. Sundell, *Wait-free reference counting and memory management*, International Parallel and Distributed Processing Symposium, no. 1530-2075/05, IEEE, April 2005, http://ieeexplore.ieee.org/iel5/9722/30685/01419843.pdf?tp=&arnumber=1419843&isnumber=30685.

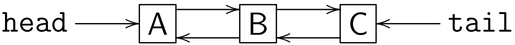
- Wikipedia, *Lock-free and wait-free algorithms*, http://en.wikipedia.org/wiki/Lock-free_and_wait-free_algorithms.
- ______, *Non-blocking synchronization*, http://en.wikipedia.org/wiki/Non-blocking_synchronization.
- ______, *Read-copy-update*, http://en.wikipedia.org/wiki/Read-copy-update.

Acknowledgements

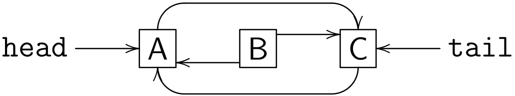
- Dave Eckhardt (de0u) has seen this lecture about as often as I have, and has produced useful commentary on every release.
- Bruce Maggs (bmm) for moral support and big-picture guidance
- Jess Mink (jmink), Matt Brewer (mbrewer), and Mr. Wright (mrwright) for being victims of beta versions of this lecture.
- [Nobody on this list deserves any of the blame, but merely credit, for this lecture.]

Pictures for RCU
Writer view

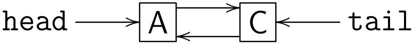
- Let's again take a linked list, this time a doubly linked one.



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

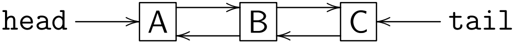


- Now the writer synchronizes, forcing all readers with references to B out of the list. Only then can B be reclaimed!

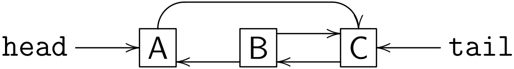


Pictures for RCU
Reader View

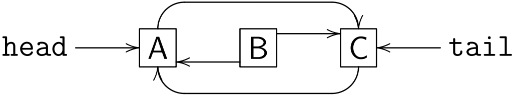
- Looking at that again, from the reader's side now. Originally



- The writer first sets it to



- And then



Pictures for RCU
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!
  - (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.

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 Reclamation" or just "SMR") [Mic02b] and [Mic04]
  - Wait-free reference counters [Sun05]
- 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!

Some real algorithms?

[Mic02a] 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.

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 [Mem06]:

> *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.*

Miscellany
Locking vs. RCU

- Interestingly, this kind of RCU tends to decrease the number of (bus) atomic operations.
  - Uses scheduler to get per-CPU atomicity.
- 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, like lockfree, suffers a slowdown from cache line shuffling, but will make progress due to having at most one writer.

Miscellany
Lockfree vs. Locking.

- Most lock-free algorithms increase the number of atomic operations, compared to the lockful variants.
- Thus we may 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.