

Semantics and tools for low-level concurrent programming

January, 14-18 2013



Mark Batty

U. Cambridge



Formalisation of the C11/C++11
concurrency model

Discovered and fixed
several unsoundness in the standard

Alastair Donaldson

Imperial College

Techniques to find bugs,
or prove correctness,
of graphics processing unit (GPU) kernels

Software performance optimisation
for multicore processors



Martin Vechev

ETH Zurich



Software reliability

Program analysis and synthesis
for building robust systems

Applications to concurrent algorithms

Francesco Zappa Nardelli

INRIA

Formalisation of
hardware and
programming language
memory models

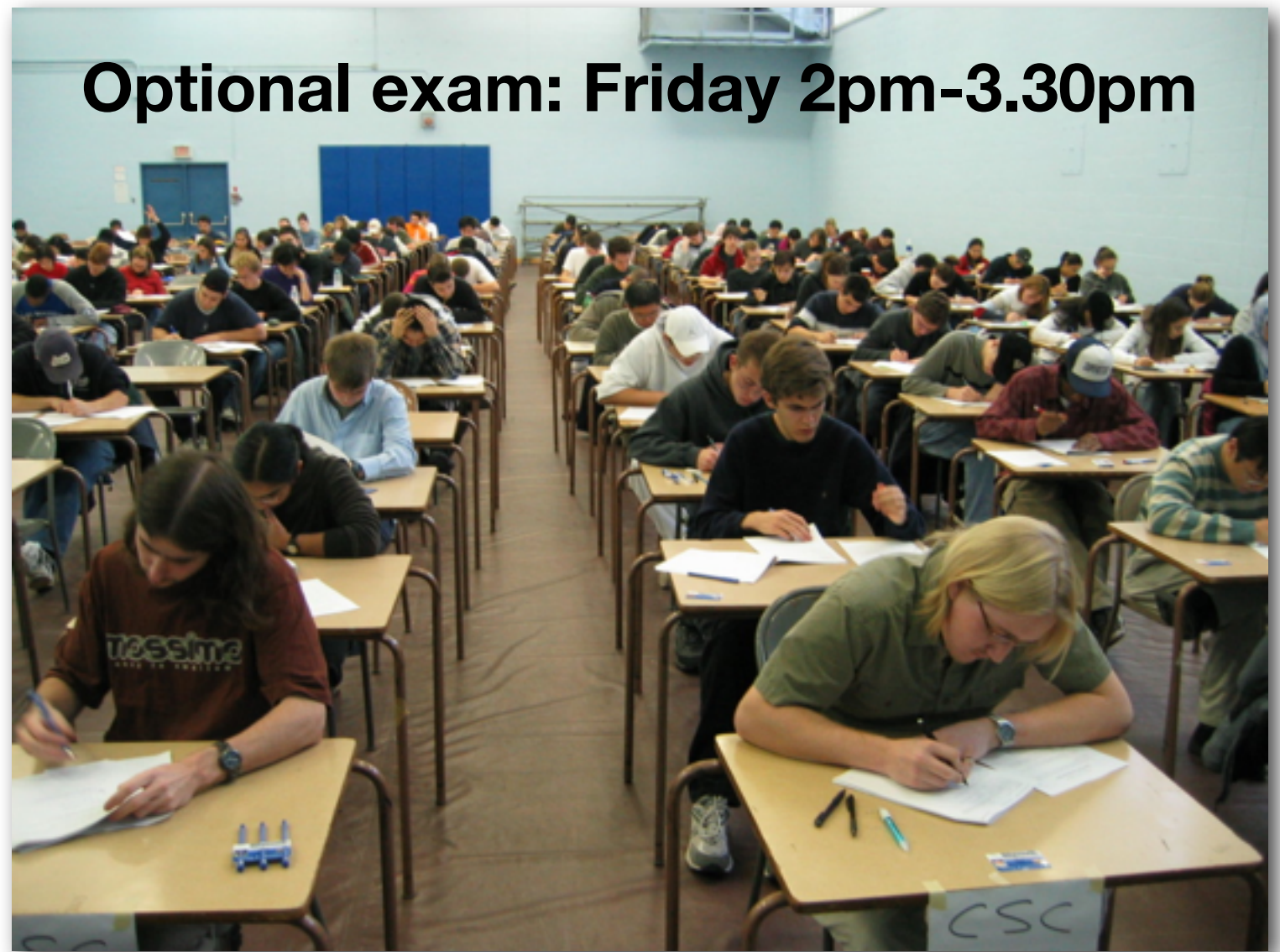
Compilation and concurrency



Don't be
LATE

Lectures start at 9am.

Optional exam: Friday 2pm-3.30pm



**Do not forget your laptop
for the TD sessions.**

Thursday afternoon: free



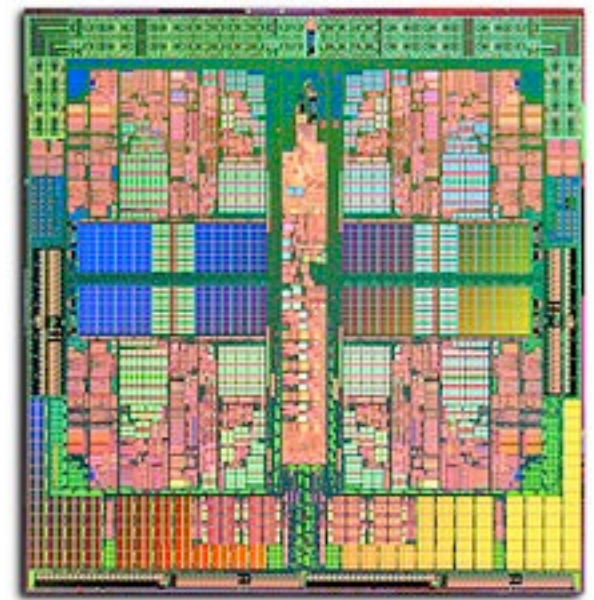
G. Eijffels
Kent Colman

Do not forget the presence sheet

G. Eiffer
Kent Colman

Do not forget the presence sheet

...and
enjoy the school!



Semantics and tools for low-level concurrent programming

Mark Batty

U. Cambridge



Alastair Donaldson

Imperial College



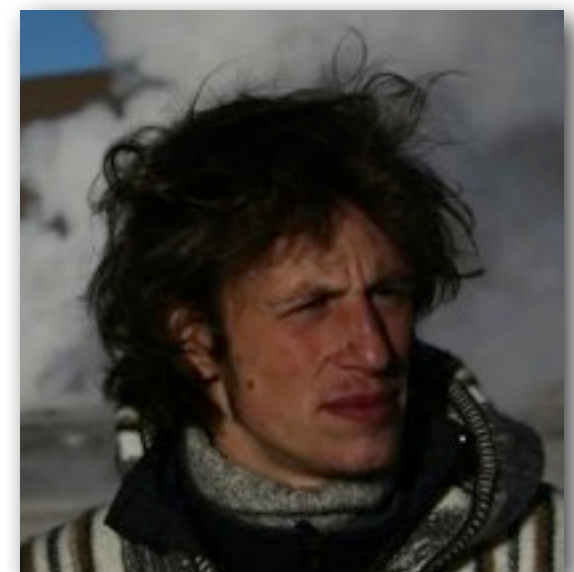
Martin Vechev

ETH Zurich

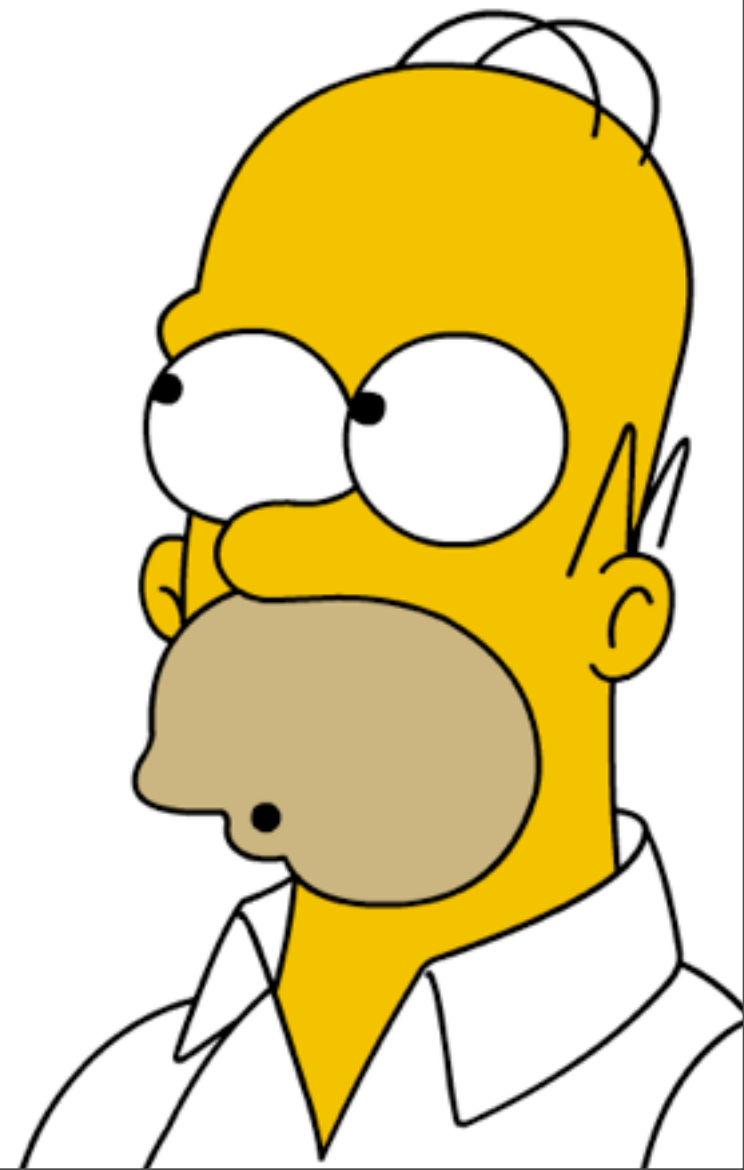


Francesco Zappa Nardelli

INRIA



concurrent programming



```

void __lockfunc _##op##_lock(locktype##_t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(_raw_##op##_trylock(lock)))
            break;
        preempt_enable();

        if (!(lock)->break_lock)
            (lock)->break_lock = 1;
        while (!op##_can_lock(lock) && (lock)->break_lock)
            _raw_##op##_relax(&lock->raw_lock);
    }
    (lock)->break_lock = 0;
}

```

excerpt from Linux spinlock.c

```

void __lockfunc _##op##_lock(locktype##_t *lock)
{
    for (;;) {
        preempt_disable();
        if (likely(_raw_##op##_trylock(lock)))
            break;
        preempt_enable();

        if (!(lock)->break_lock)
            (lock)->break_lock = 1;
        while (!op##_can_lock(lock) && (lock)->break_lock)
            _raw_##op##_relax(&lock->raw_lock);
    }
    (lock)->break_lock = 0;
}

```

excerpt from Linux spinlock.c

excerpt from
www.javaconcurrencyinpractice.com

```

/**
 * LazyInitRace
 *
 * Race condition in lazy initialization
 *
 * @author Brian Goetz and Tim Peierls
 */

@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }

```



```

52 ResourceResponse response;
53 unsigned long identifier = std::numeric_limits<unsigned long>::max();
54 if (document->frame())
55     identifier = document->frame()->loader()->loadResourceSynchronously(request, storedCredentials, error, response);
56
57 // No exception for file:/// resources, see <rdar://problem/4962298>.
58 // Also, if we have an HTTP response, then it wasn't a network error in fact.
59 if (!error.isNull() && !request.url().isLocalFile() && response.httpStatusCode() <= 0) {
60     client.didFail(error);
61     return;
62 }
63
64 // FIXME: This check along with the one in willSendRequest is specific to xhr and
65 // should be made more generic.
66 if (sameOriginRequest && !document->securityOrigin()->canRequest(response.url())) {
67     client.didFailRedirectCheck();
68     return;
69 }
70
71 client.didReceiveResponse(response);
72
73 const char* bytes = static_cast<const char*>(data.data());
74 int len = static_cast<int>(data.size());
75 client.didReceiveData(bytes, len);
76
77 client.didFinishLoading(identifier);
78 }

```

*/

excerpt from [WebKit](#)

excerpt from
www.javaconcurrencyinpractice.com

```

@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }

```

in practice

sequential code, interaction via *shared memory*, some OS calls.

Libraries may provide some abstractions (e.g. message passing).
However, somebody must still implement these libraries. And...

Programming is hard:
subtle algorithms, awful corner cases.

Testing is hard:
some behaviours are observed rarely and difficult to reproduce.

Warm-up: let's implement a shared stack!

excerpt from
www.javaconcurrencyinpractice.com

```
class ExpensiveObject { }
```

Setup

A program is composed by *threads* that communicate by writing and reading in a *shared memory*. No assumptions about the relative speed of the threads.

In this example we will use a *mild variant* of the *C programming language*:

- local variables: x, y, \dots (allocated on the stack, local to each thread)
- global variables: Top, H, \dots (allocated on the heap, shared between threads)
- data structures: arrays $H[i]$, records $n = t \rightarrow tl, \dots$
- an atomic *compare-and-swap* operation (e.g. CMPXCHG on x86):

```
bool CAS (value_t *addr, value_t exp, value_t new) {  
    atomic {  
        if (*addr == exp) then { *addr = new; return true; }  
        else return false;  
    }  
}
```

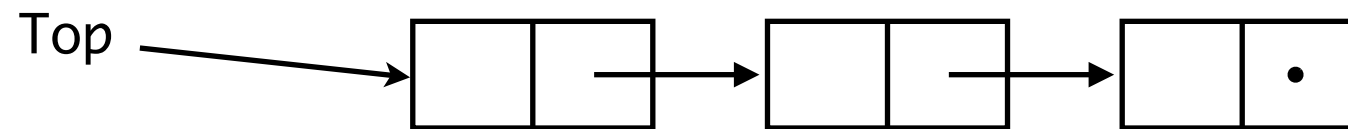
A stack

We implement a stack using a list living in the heap:

- each entry of the stack is a record of two fields:

```
typedef struct entry { value hd; entry *tl } entry
```

- the top of the stack is pointed by Top.



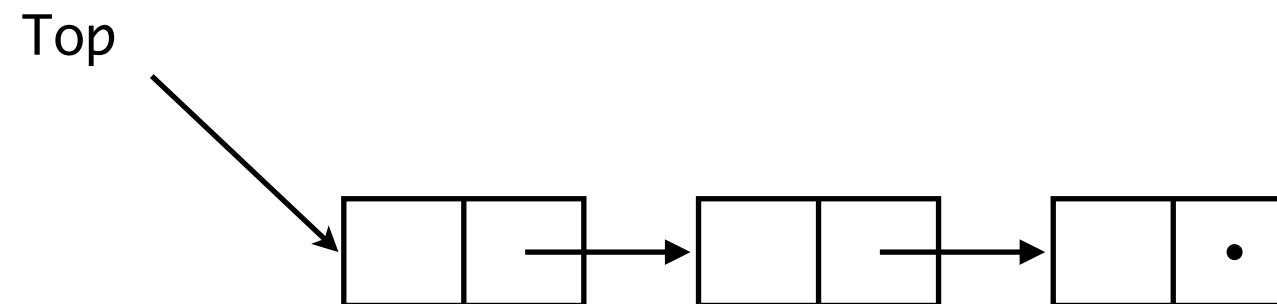
```
pop () {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```


A sequential stack: demo

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

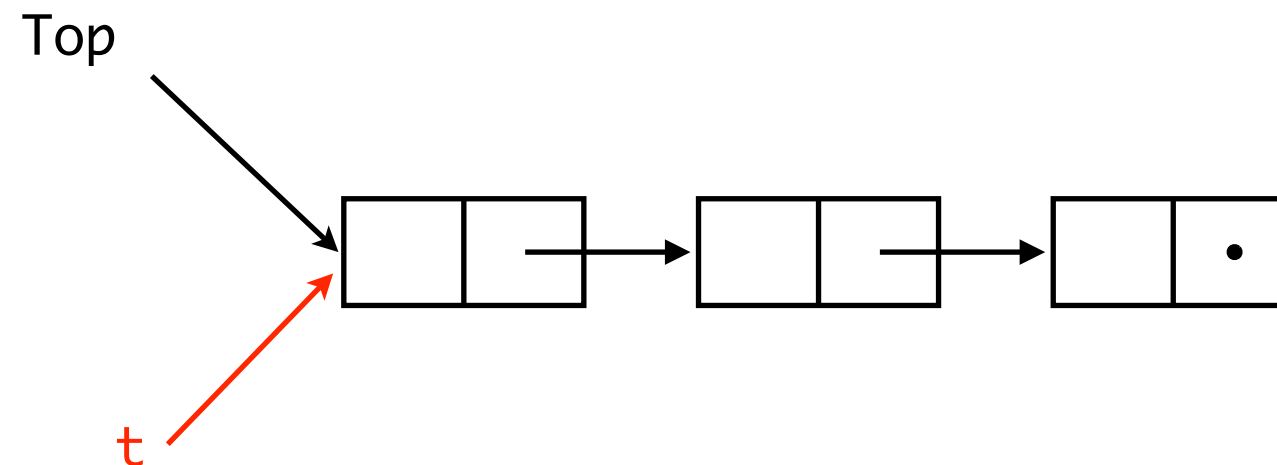
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

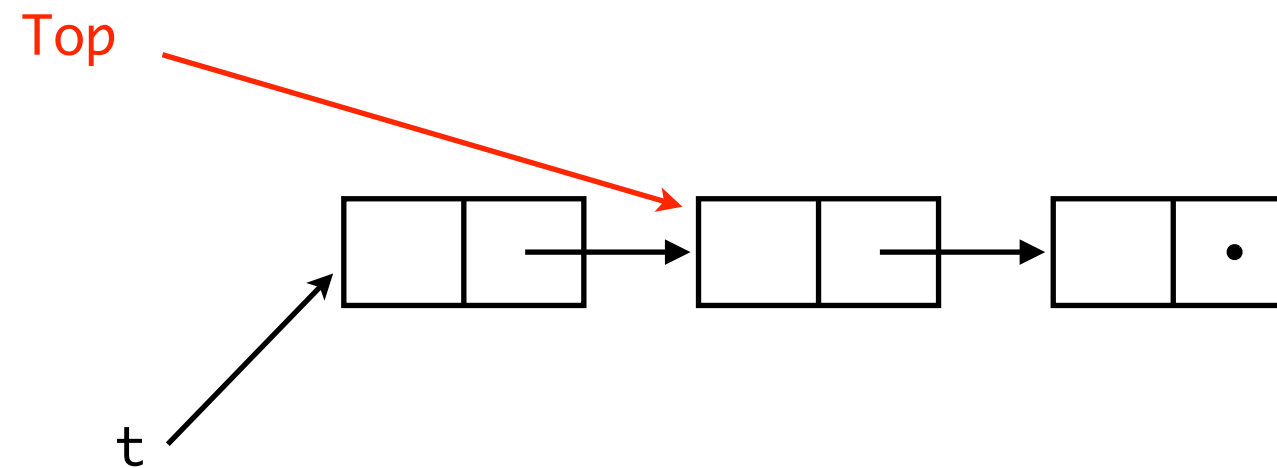
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

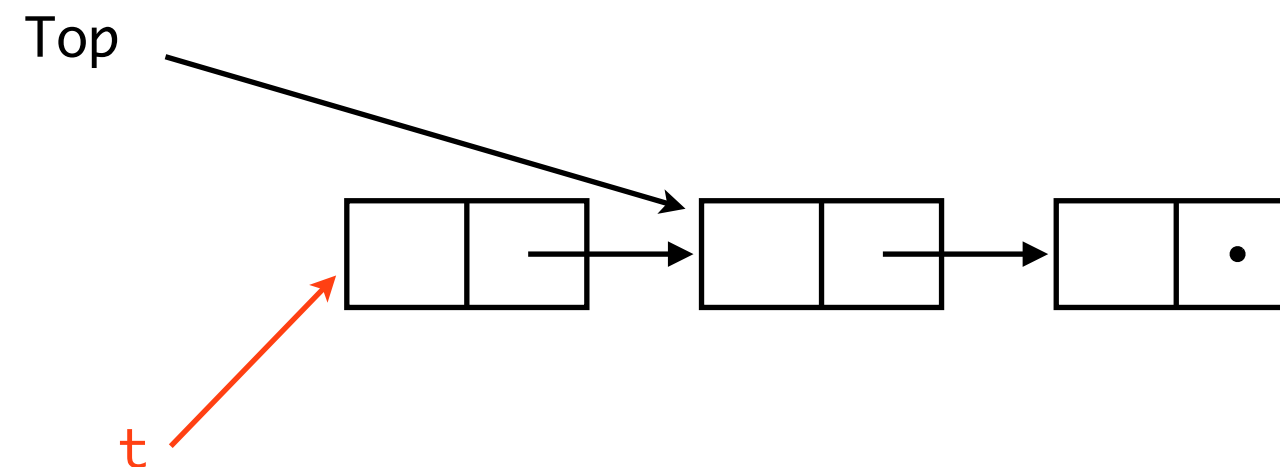
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: pop ()

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

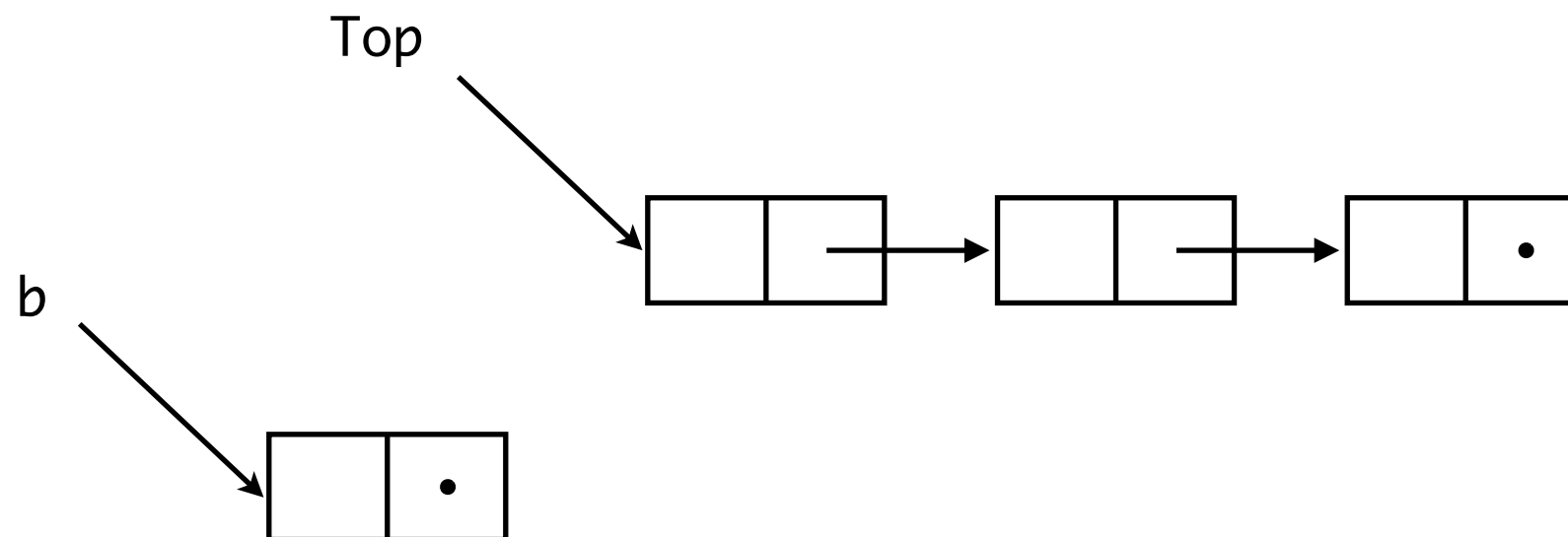
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

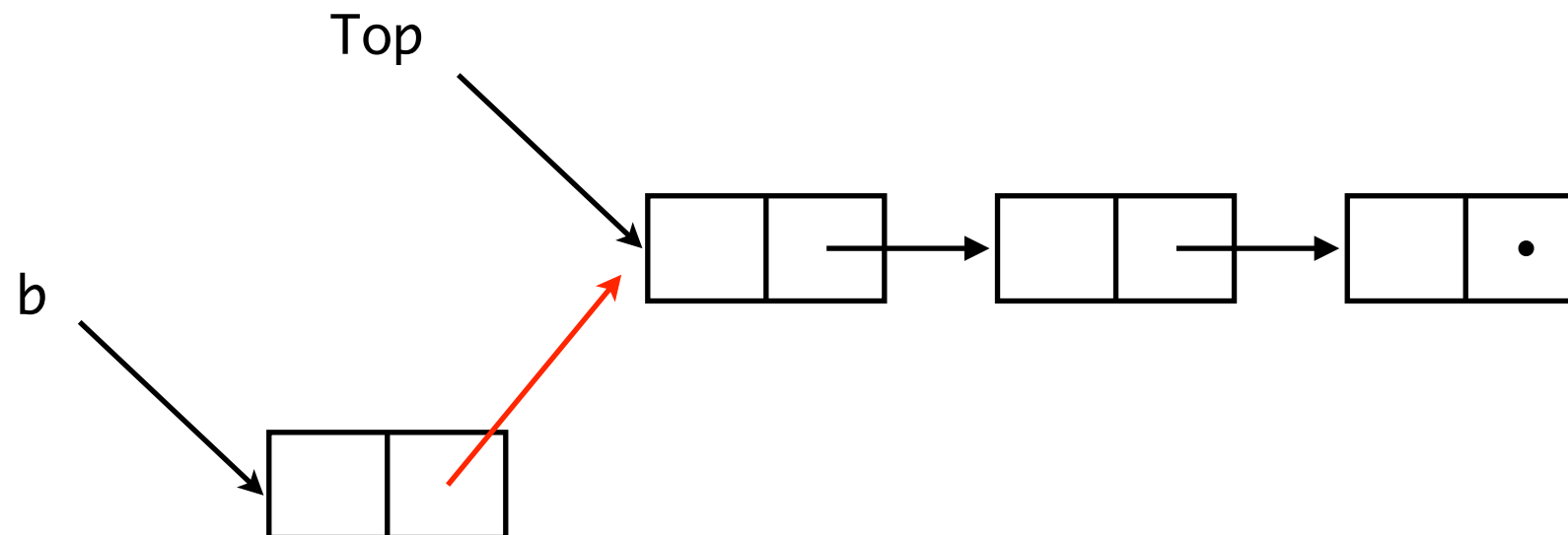
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

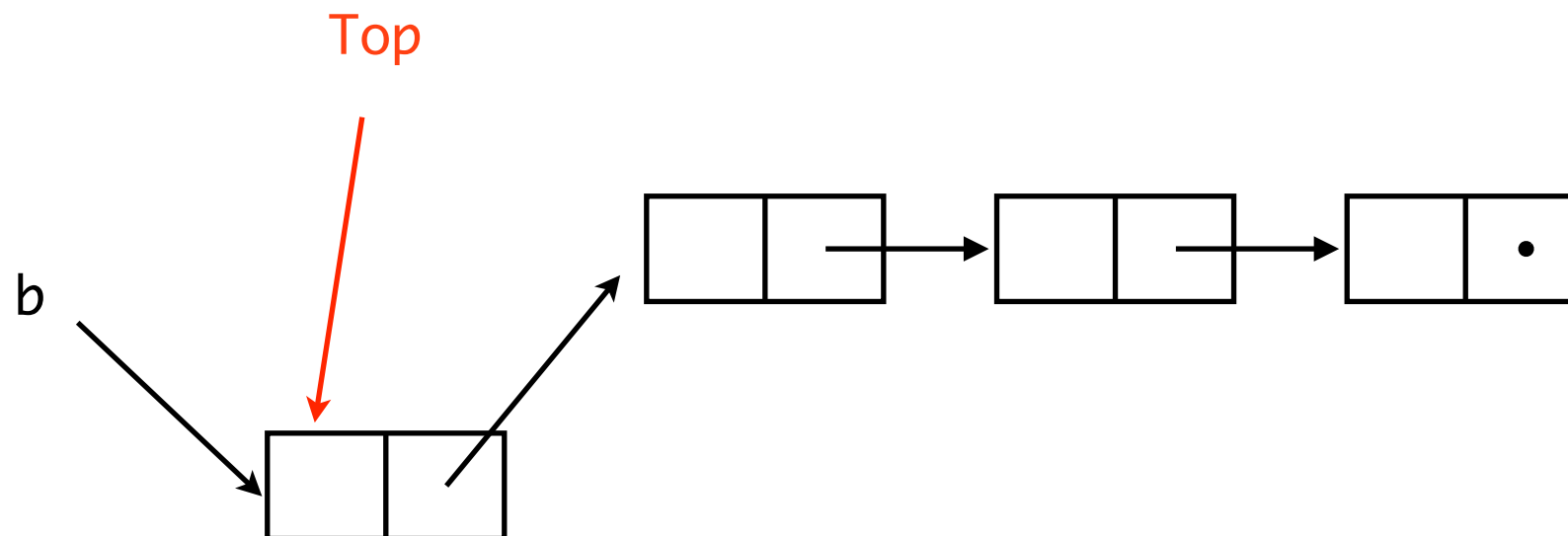
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

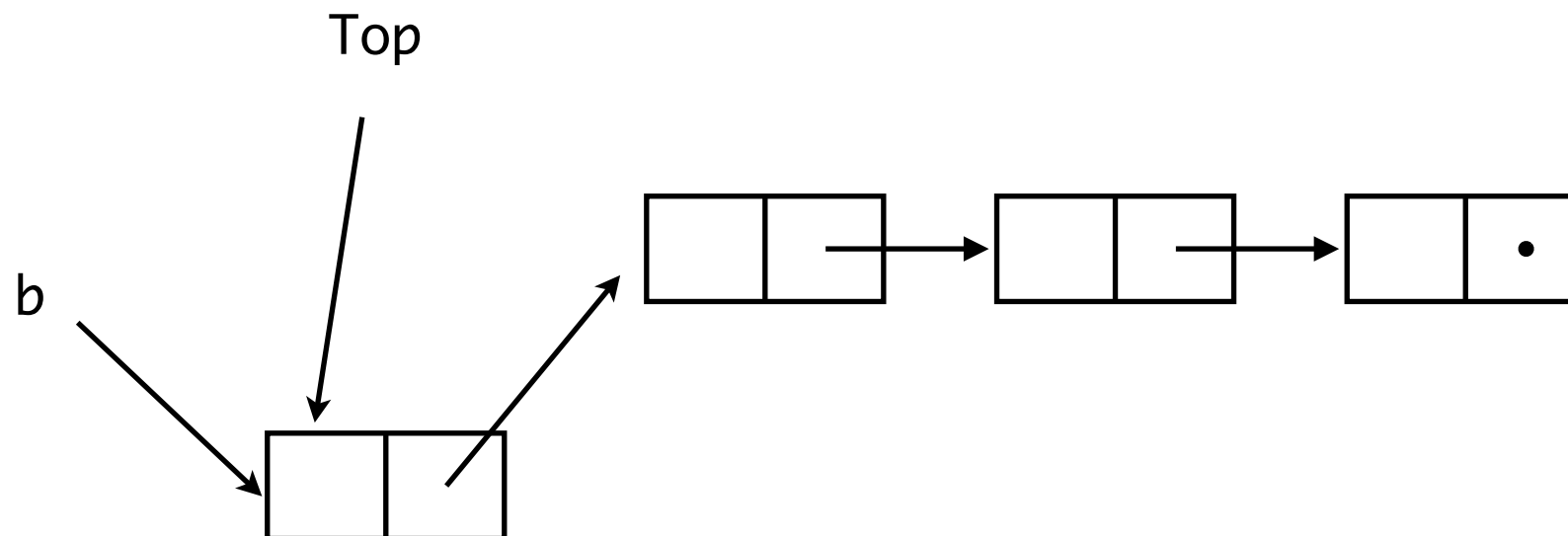
```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```



A sequential stack: push (b)

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

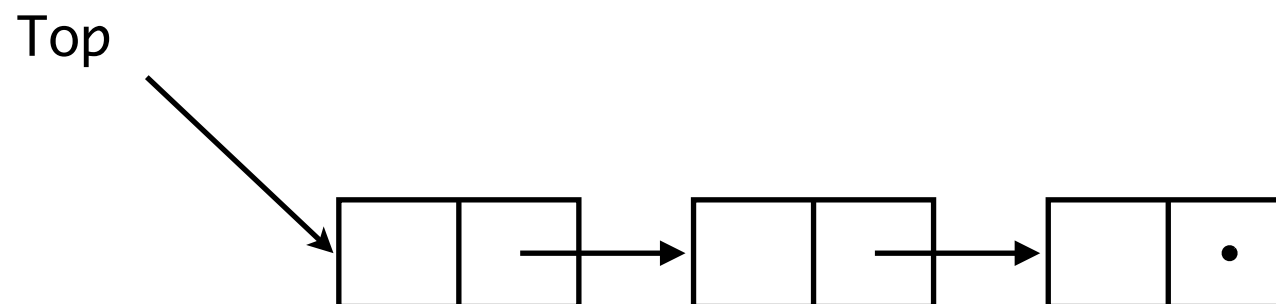


A sequential stack in a concurrent world

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

Imagine that two threads invoke pop() concurrently...

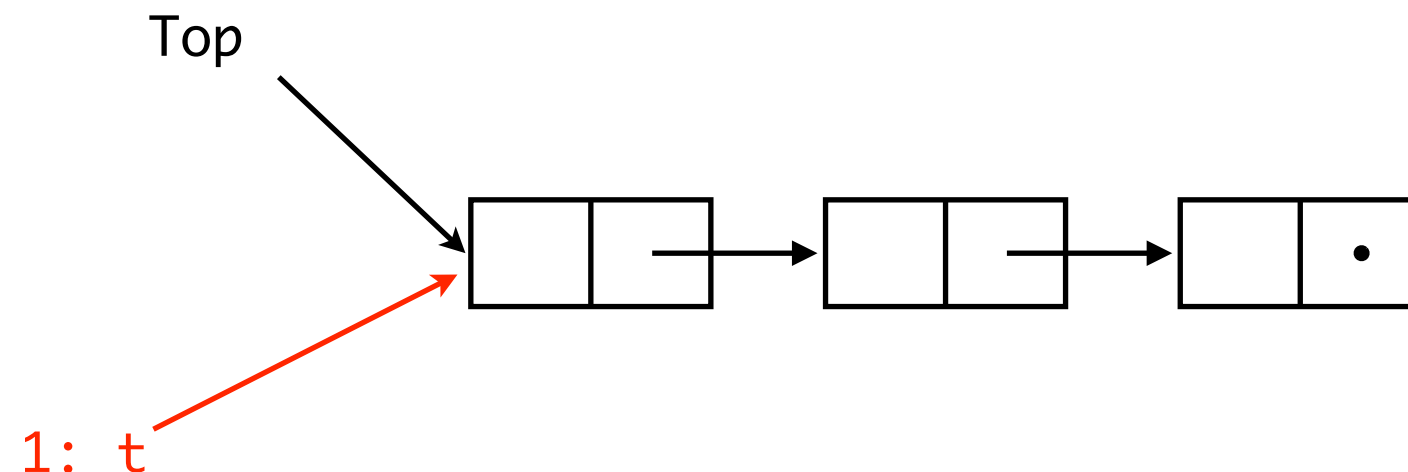


A sequential stack in a concurrent world

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

Imagine that two threads invoke pop() concurrently...

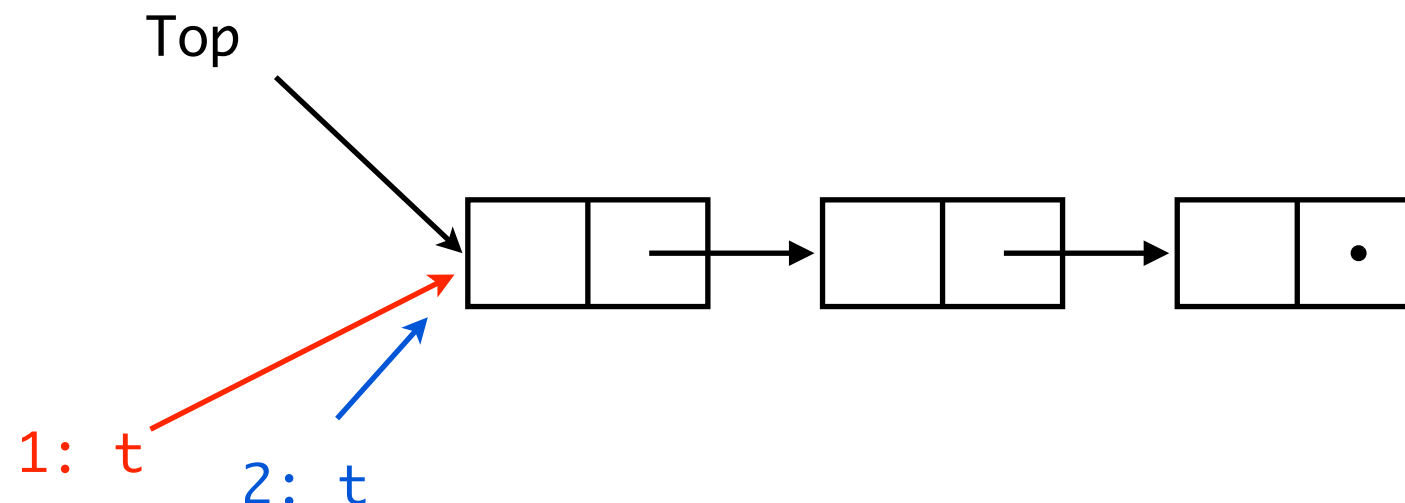


A sequential stack in a concurrent world

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

Imagine that two threads invoke pop() concurrently...

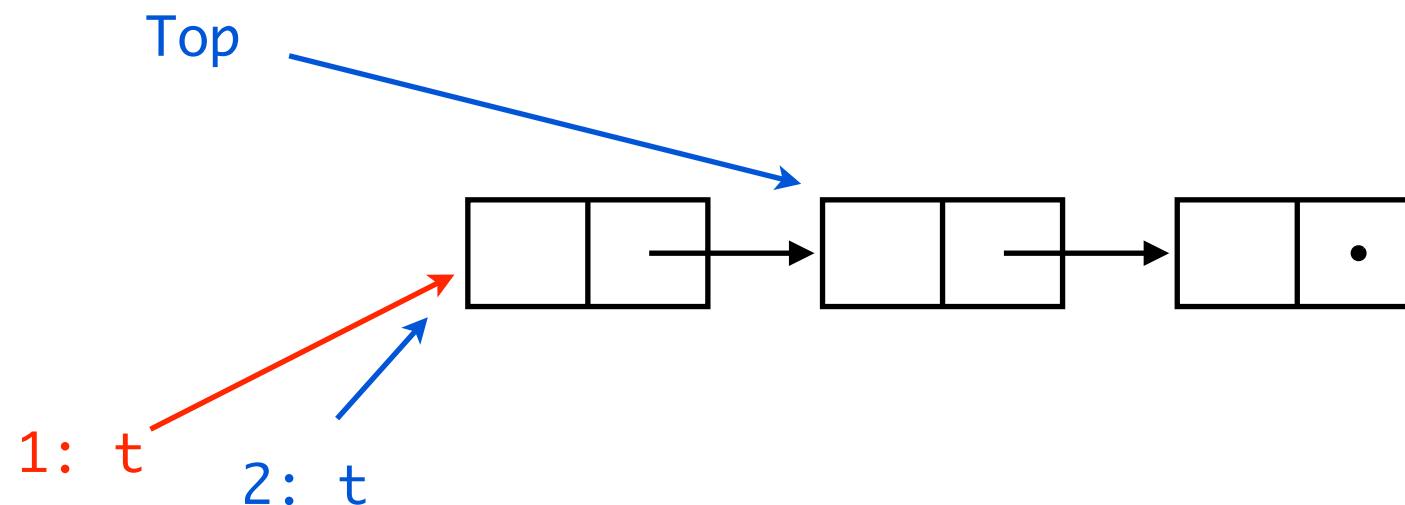


A sequential stack in a concurrent world

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

Imagine that two threads invoke pop() concurrently...

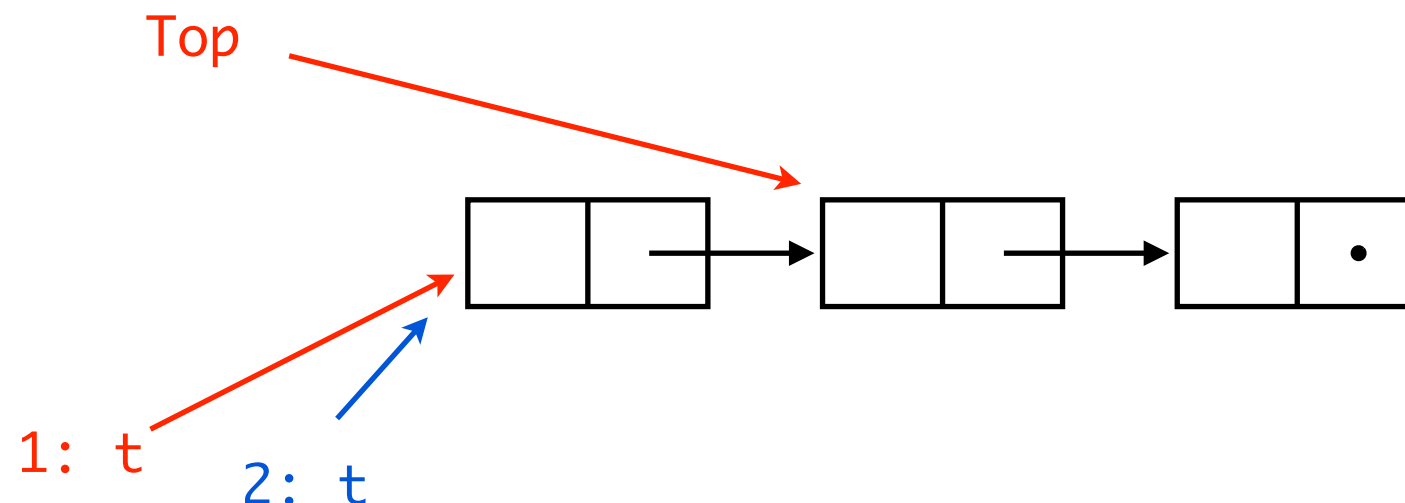


A sequential stack in a concurrent world

```
pop ( ) {  
    t = Top;  
    if (t != nil)  
        Top = t->tl;  
    return t;  
}
```

```
push (b) {  
    b->tl = Top;  
    Top = b;  
    return true;  
}
```

Imagine that two threads invoke pop() concurrently...
...the two threads might pop the same entry!



Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    return t;  
}
```

```
push (b) {  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

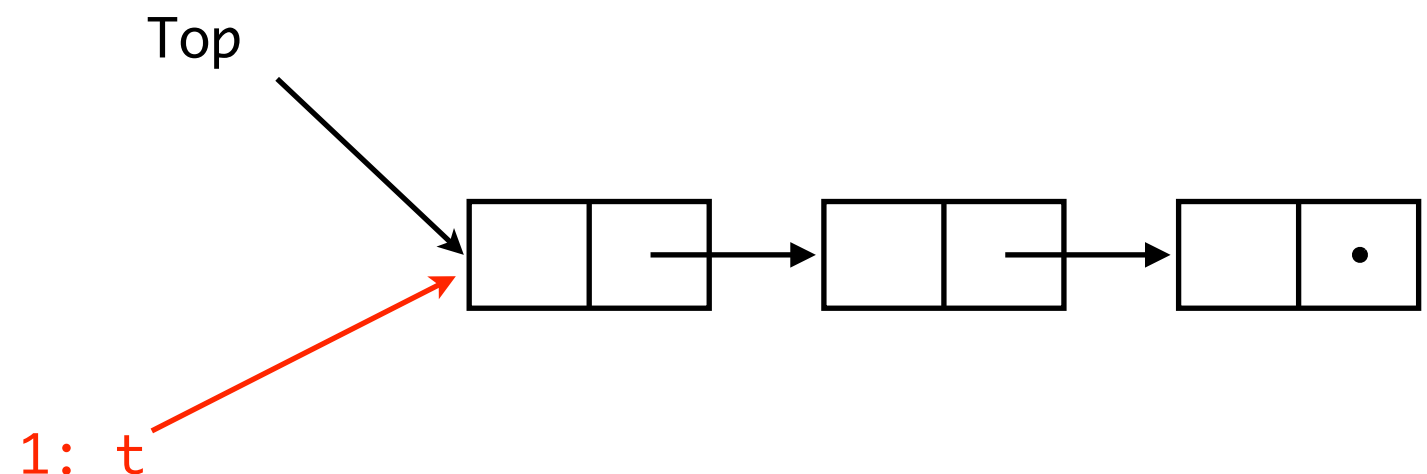
•

Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    return t;  
}
```

```
push (b) {  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Two concurrent pop() now work fine...

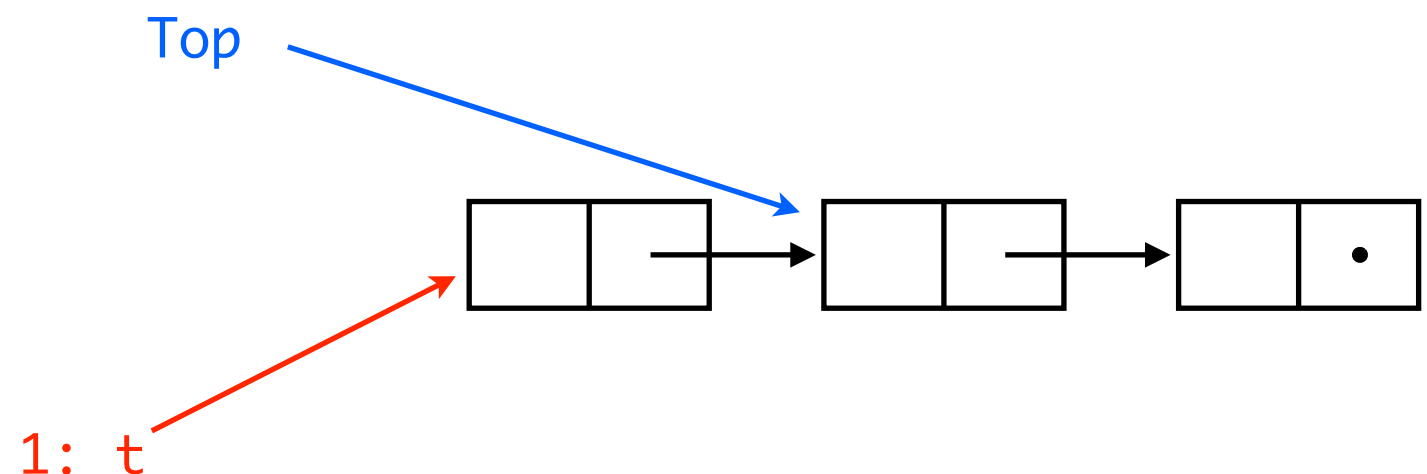


Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    return t;  
}
```

```
push (b) {  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Two concurrent pop() now work fine...



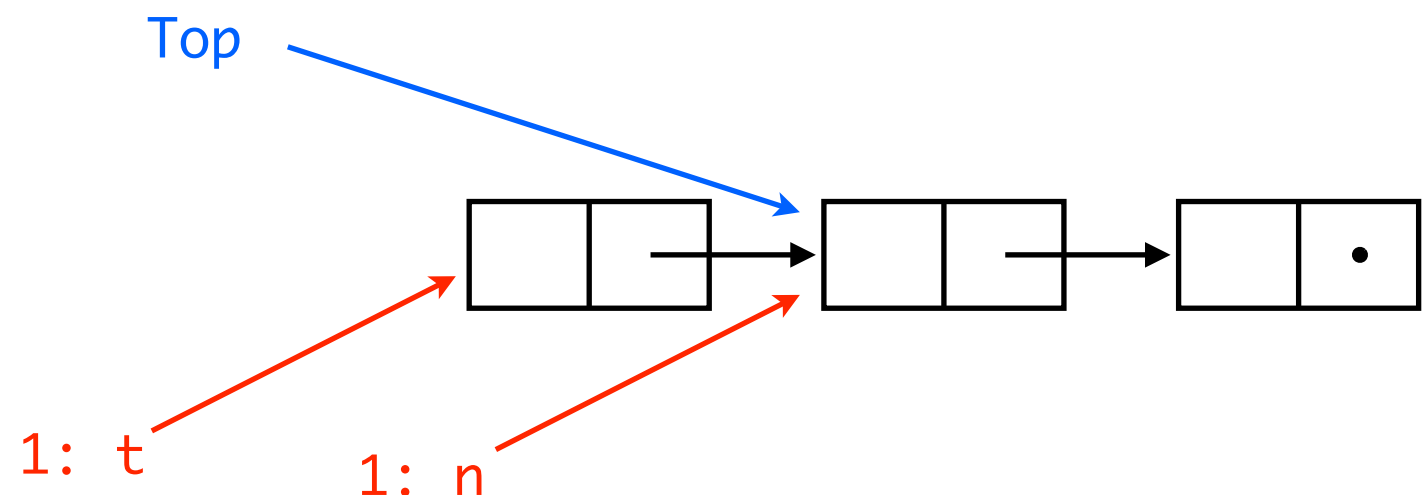
Idea 1: validate the Top pointer using CAS

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Two concurrent pop() now work fine...

The CAS of Th. 1 fails.

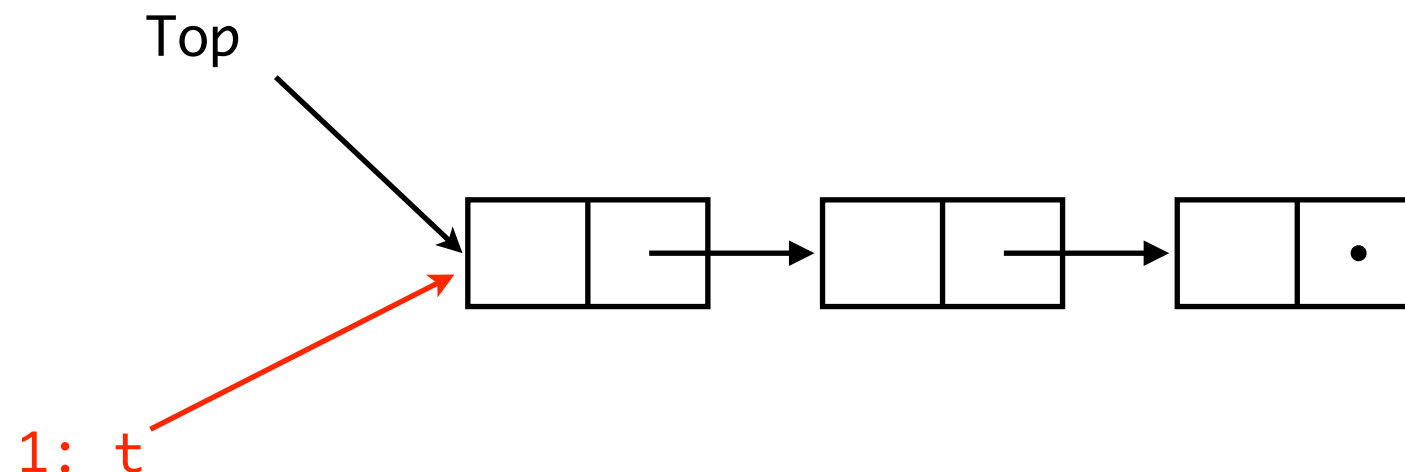


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 starts popping...

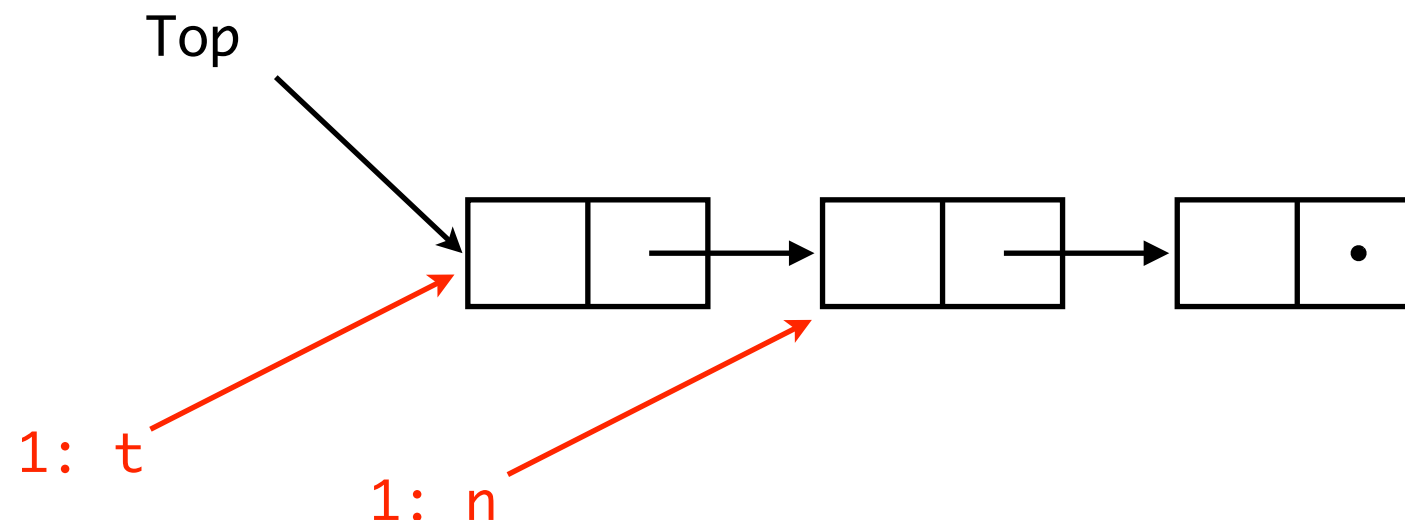


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 starts popping...

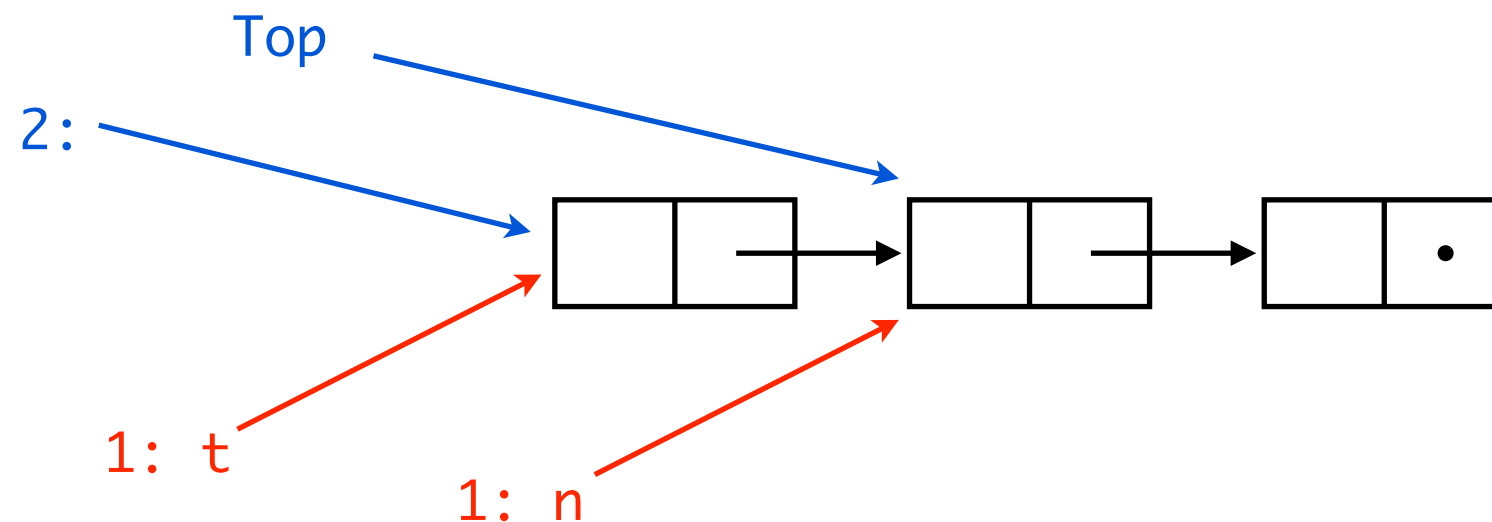


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pops...

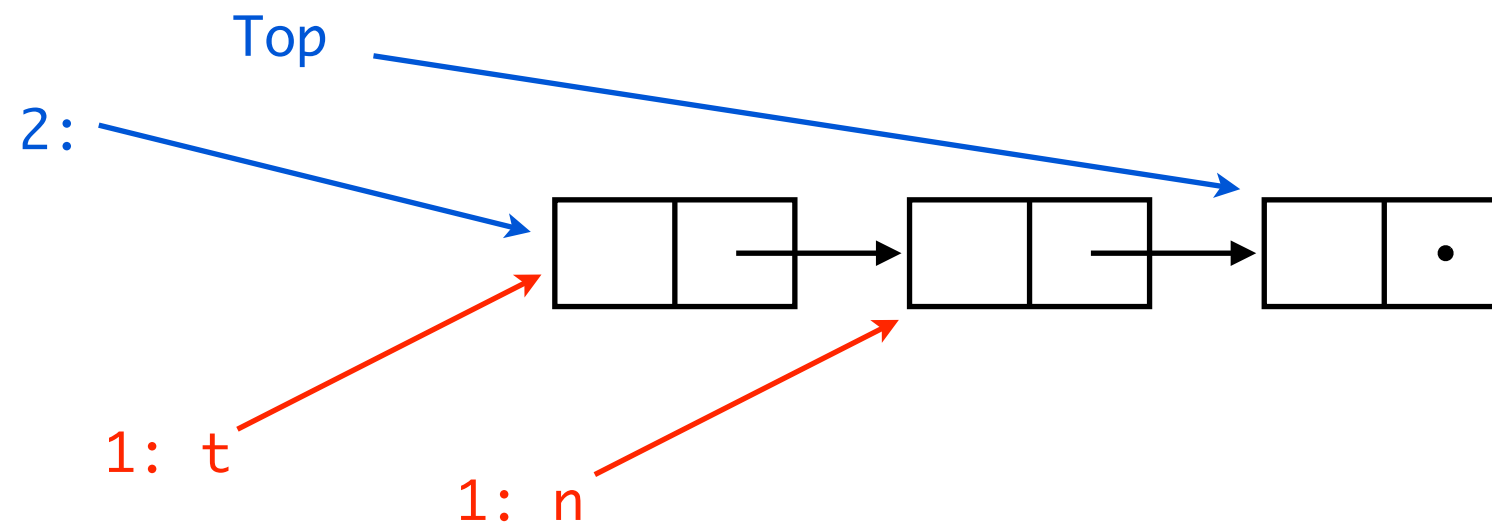


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pops again...

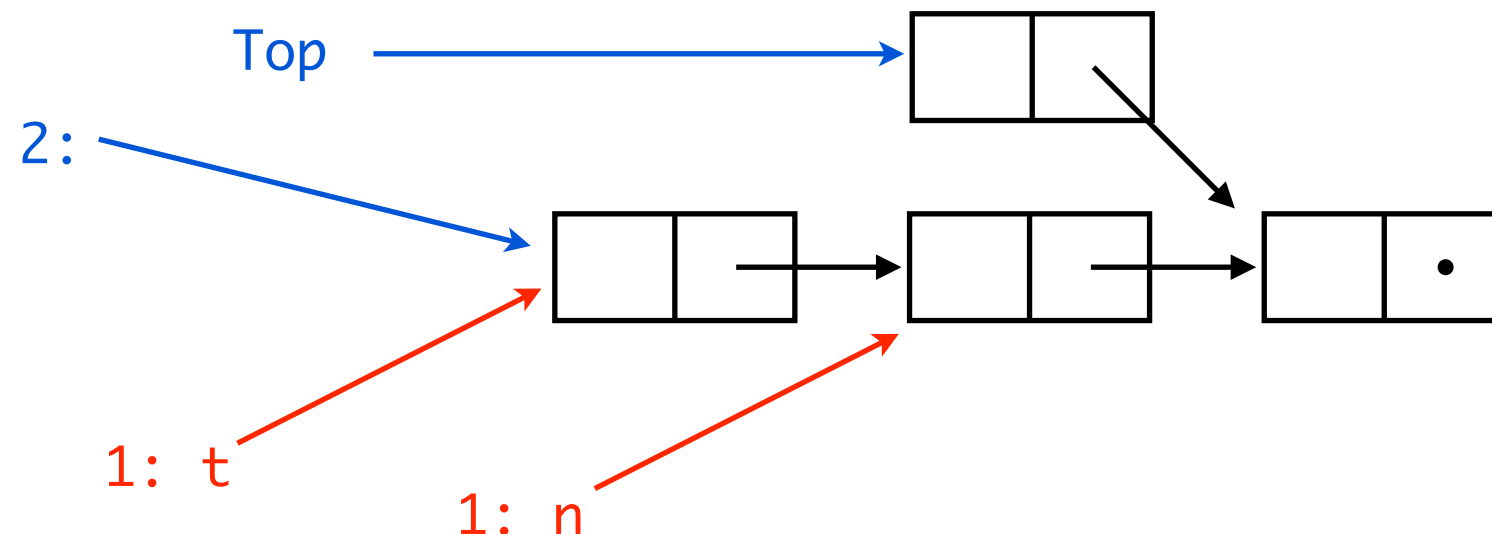


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pushes a new node...

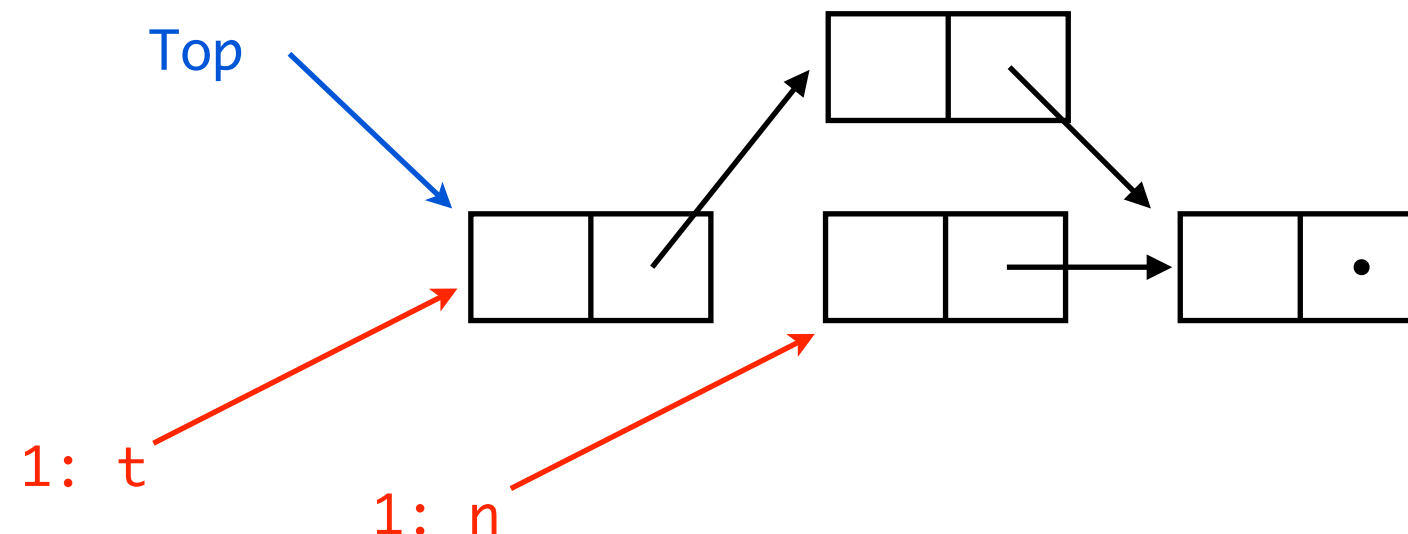


The ABA problem

```
pop ( ) {  
  while (true) {  
    t = Top;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  return t;  
}
```

```
push (b) {  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 2 pushes the old head of the stack...

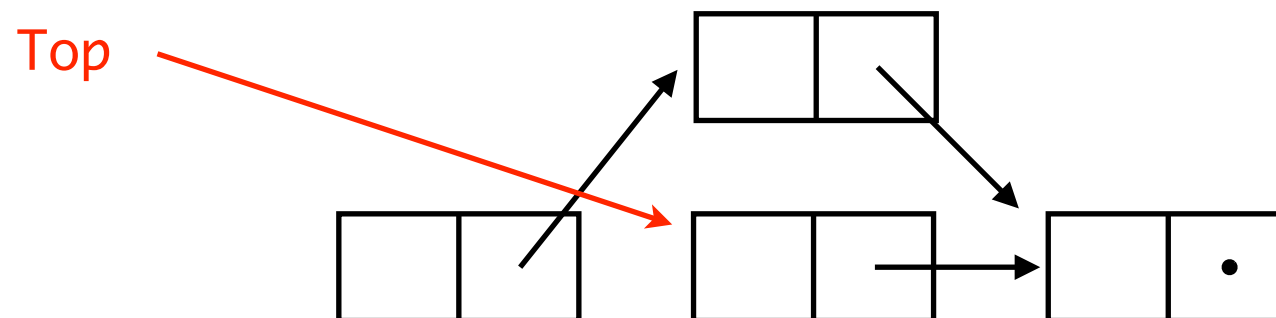


The ABA problem

```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    return t;  
}
```

```
push (b) {  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Th 1 corrupts the stack...



The hazard pointers methodology

Michael adds to the previous algorithm a *global array H of hazard pointers*:

- thread i alone is allowed to write to element $H[i]$ of the array;
- any thread can read any entry of H .

The algorithm is then modified:

- before popping a cell, a thread puts its address into its own element of H .
This entry is cleared only if CAS succeeds or the stack is empty;
- before pushing a cell, a thread checks to see whether it is pointed to from any element of H . If it is, push is delayed.

Michael's algorithm, simplified

```
pop ( ) {  
    while (true) {  
        atomic { t = Top;  
                 H[tid] = t; };  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    H[tid] = nil;  
    return t;  
}
```

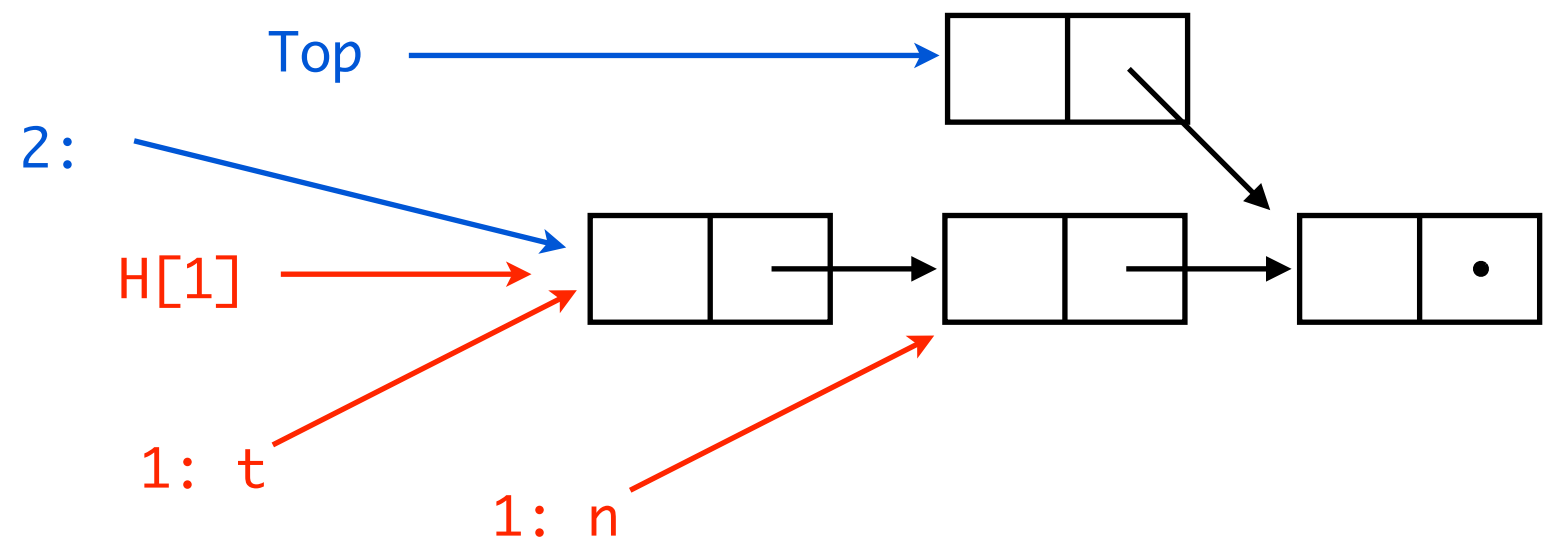
```
push (b) {  
    for (n = 0; n < no_threads, n++)  
        if (H[n] == b) return false;  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Michael's algorithm, simplified

```
pop ( ) {  
  while (true) {  
    atomic { t = Top;  
             H[tid] = t; };  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 2 cannot push the old head, because Th 1 has an hazard pointer on it...

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```



Key properties of Michael's simplified algorithm

- A node can be added to the hazard array only if it is reachable through the stack;
- a node that has been popped is not reachable through the stack;
- a node that is unreachable in the stack and that is in the hazard array cannot be added to the stack;
- while a node is reachable and in the hazard array, it has a constant tail.

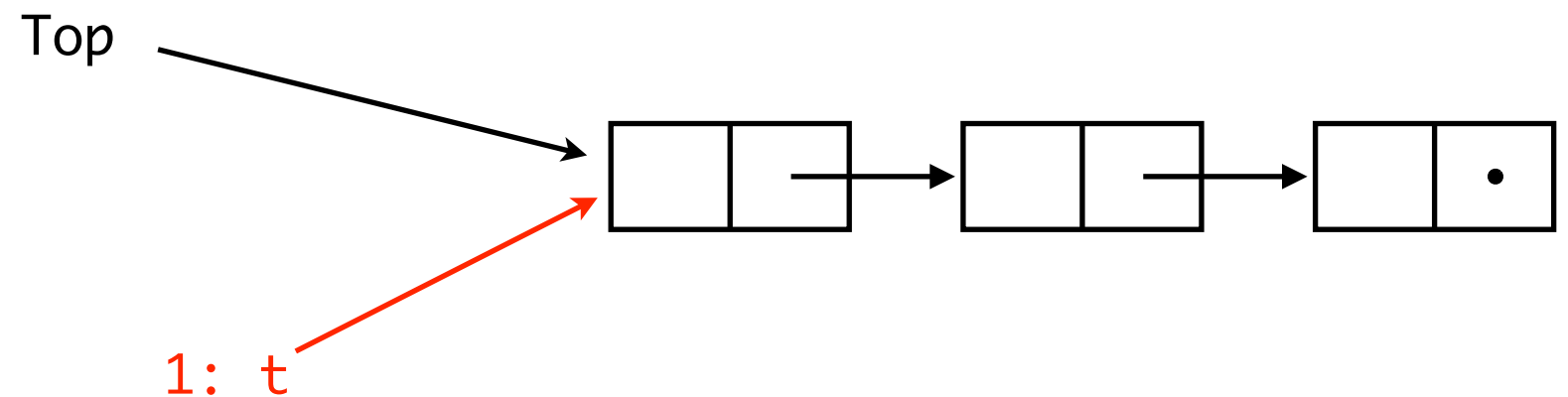
These are a good example of the properties we might want to state and prove about a concurrent algorithm.

The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

Th 1 copies Top...

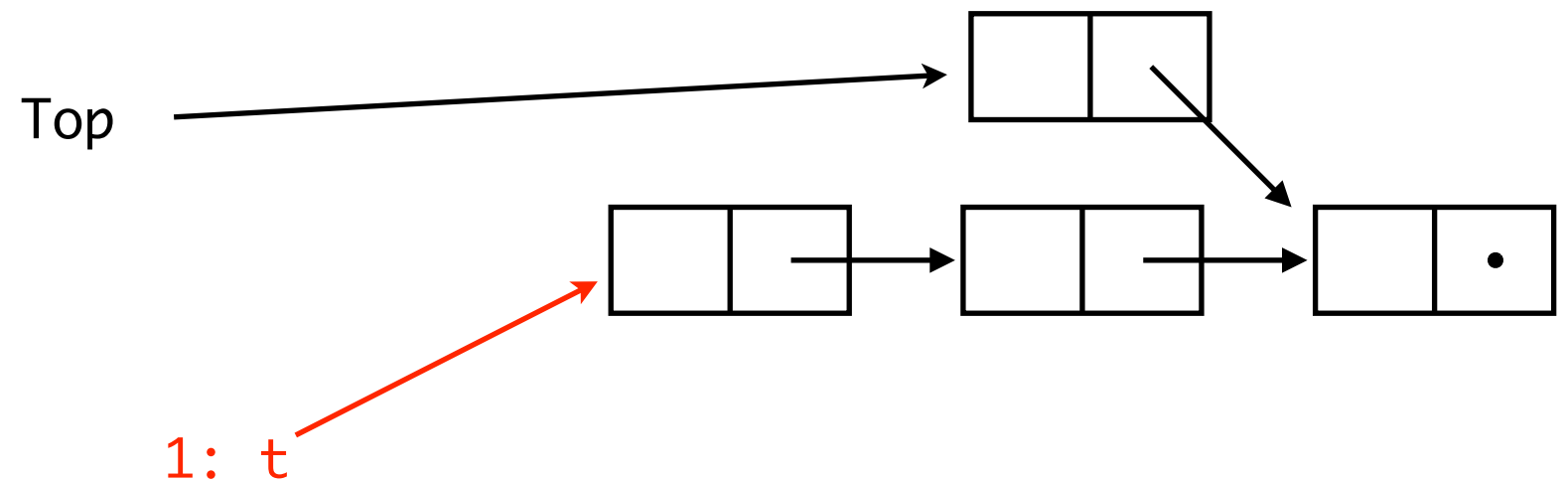


The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 2 pops twice, and
pushes a new node...

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```

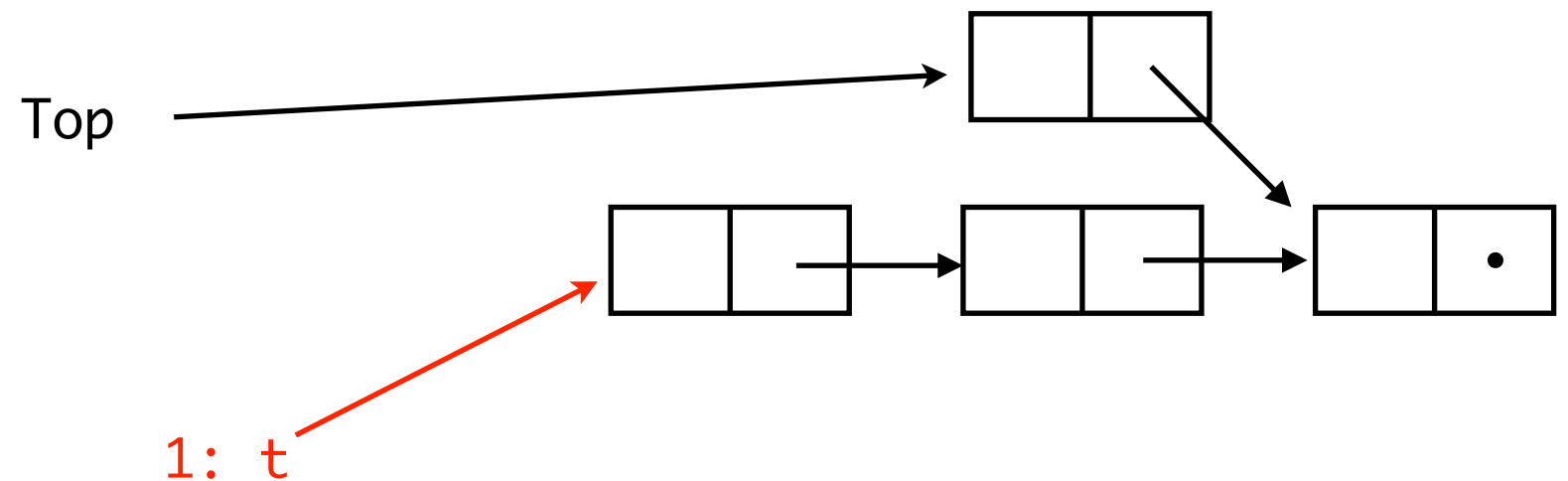


The role of *atomic*

```
pop ( ) {  
    while (true) {  
        t = Top;  
        H[tid] = t;  
        if (t == nil) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    H[tid] = nil;  
    return t;  
}
```

Th 2 starts pushing the old head, and is halfway in the for loop...

```
push (b) {  
    for (n = 0; n < no_threads, n++)  
        if (H[n] == b) return false;  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

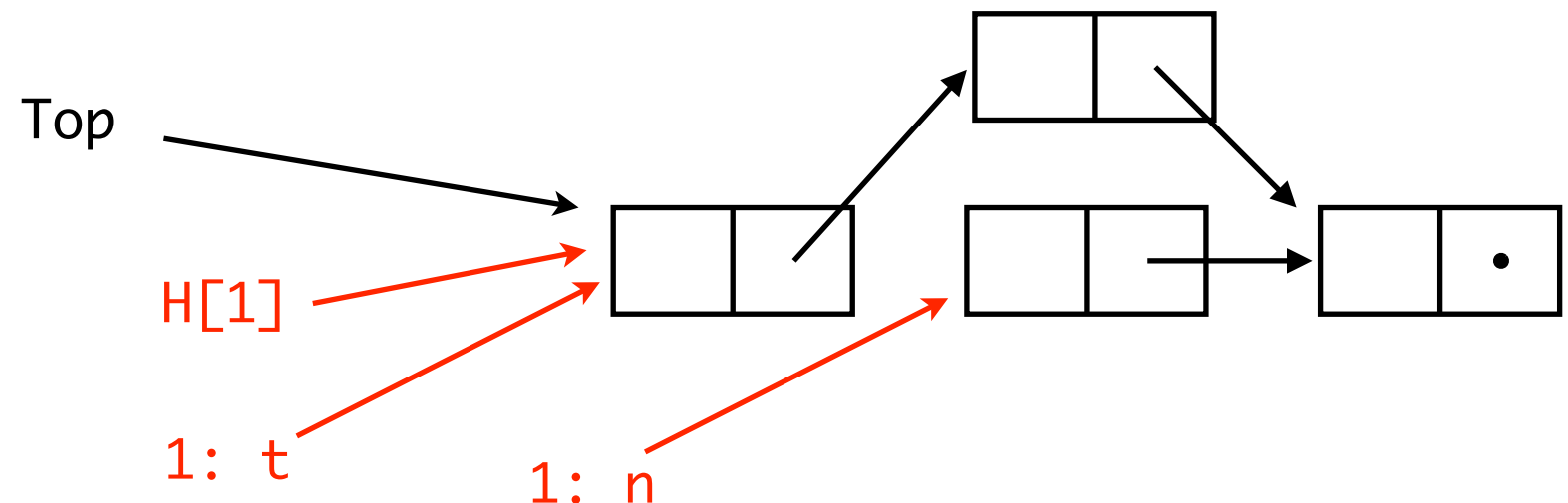


The role of *atomic*

```
pop ( ) {  
  while (true) {  
    t = Top;  
    H[tid] = t;  
    if (t == nil) break;  
    n = t->tl;  
    if CAS(&Top,t,n) break;  
  }  
  H[tid] = nil;  
  return t;  
}
```

Th 1 sets its hazard
pointer... but Th 2 might
not see the hazard pointer
of Th 1!

```
push (b) {  
  for (n = 0; n < no_threads, n++)  
    if (H[n] == b) return false;  
  while (true) {  
    t = Top;  
    b->tl = t;  
    if CAS(&Top,t,b) break;  
  }  
  return true;  
}
```



Michael shared stack

```
pop ( ) {  
    while (true) {  
        t = Top;  
        if (t == nil) break;  
        H[tid] = t;  
        if (t != Top) break;  
        n = t->tl;  
        if CAS(&Top,t,n) break;  
    }  
    H[tid] = nil;  
    return t;  
}
```

```
push (b) {  
    for (n = 0; n < no_threads, n++)  
        if (H[n] == b) return false;  
    while (true) {  
        t = Top;  
        b->tl = t;  
        if CAS(&Top,t,b) break;  
    }  
    return true;  
}
```

Trust me: if we validate t against the Top pointer before reading $t \rightarrow tl$, we get a correct algorithm.




Reaction 1.

That algorithm is insane... I will never use it in my everyday programming.



Reaction 1.

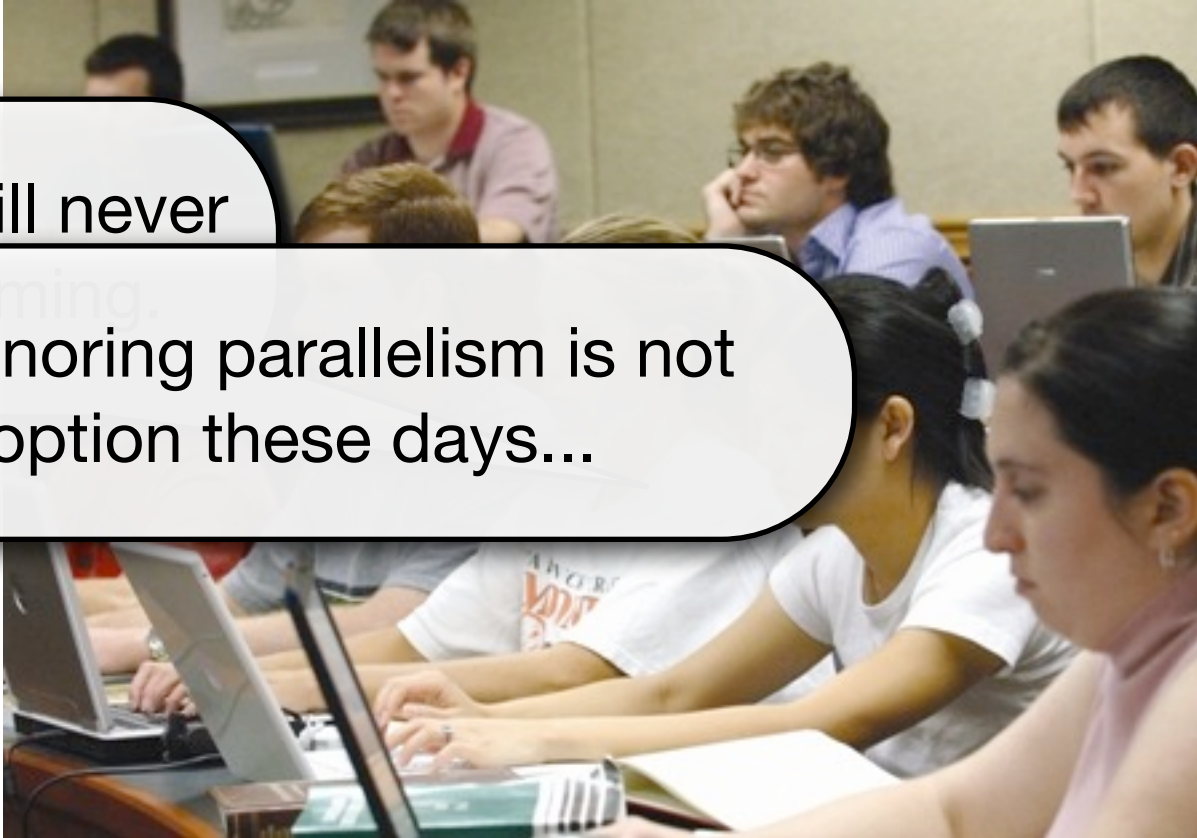


That algorithm is insane... I will never use it in my everyday programming.



Yes, you will! Michael algorithms are part of `java.util.concurrent`.

Reaction 1.



That algorithm is insane... I will never use it in my everyday programming.

...and ignoring parallelism is not an option these days...



Yes, you will! Michael algorithms are part of `java.util.concurrent`.

Reaction 2.

Shared memory?

***Wow, this is cool!
Does it really work?***

Will the compiler introduce errors?

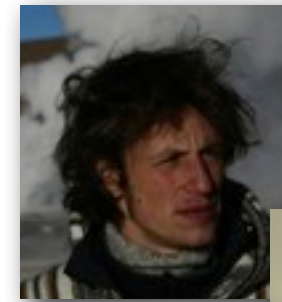
Which programming language?

What does the hardware execute?



Welcome !!!

1. Shared-memory concurrency,
from hardware to programming languages



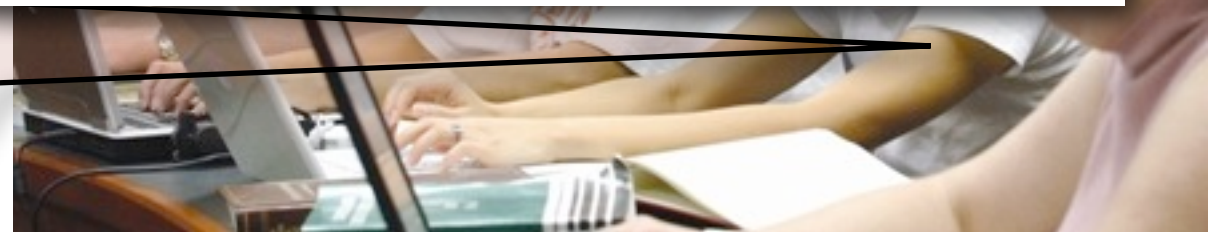
2. Machine assisted concurrent programming



3. GPU kernels



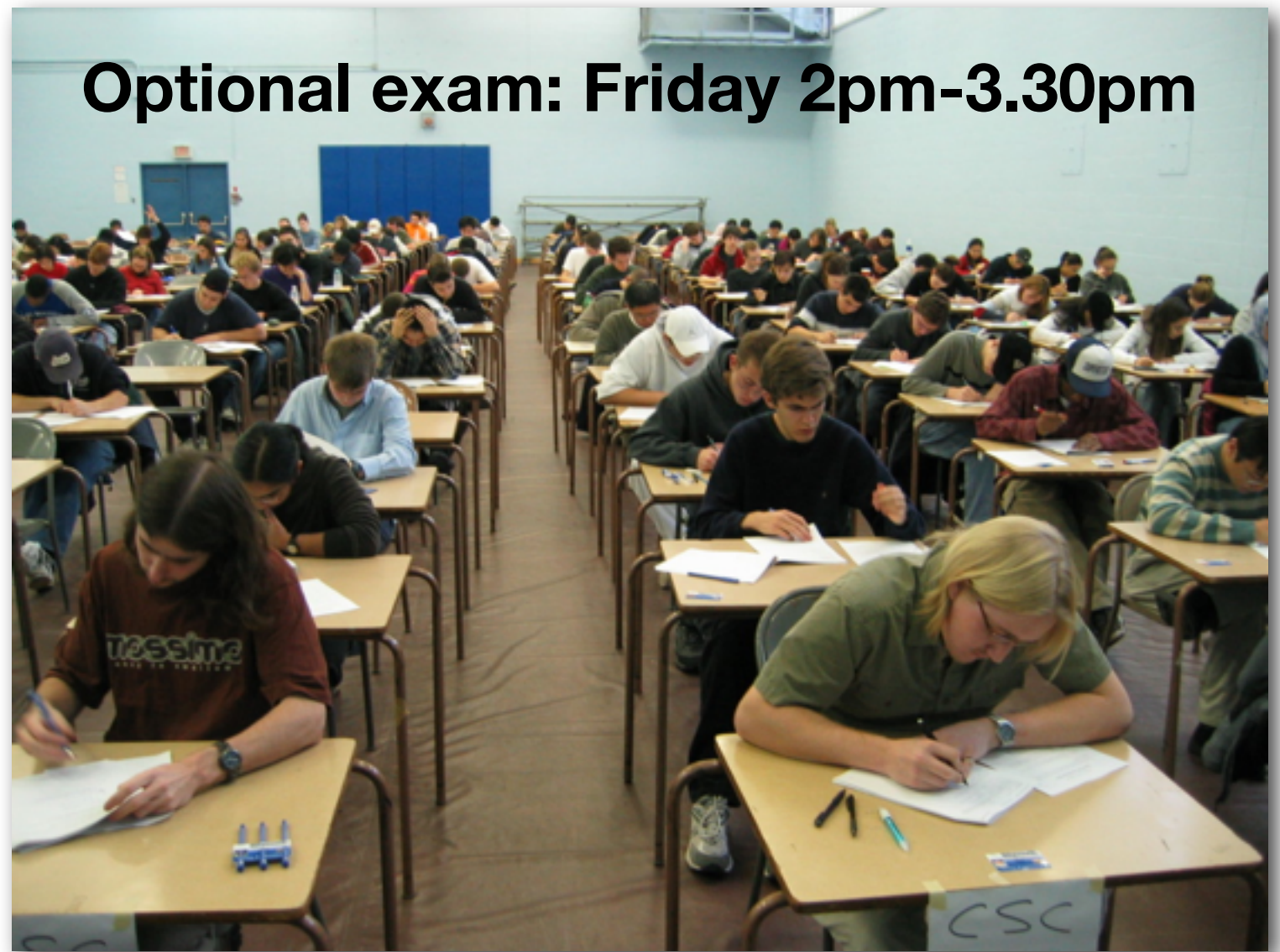
What does the hardware execute?



Don't be
LATE

Lectures start at 9am.

Optional exam: Friday 2pm-3.30pm



**Do not forget your laptop
for the TD sessions.**

Thursday afternoon: free



shared memory



Shared memory

(from Wikipedia)



In computer hardware, **shared memory** refers to a (typically) large block of [random access memory](#) (RAM) that can be accessed by several different [central processing units](#) (CPUs) in a [multiple-processor computer system](#).

A shared memory system is relatively easy to program since all processors share a single view of data and the communication between processors can be as fast as memory accesses to a same location. The issue with shared memory systems is that many CPUs need fast access to memory and will likely [cache memory](#), which has two complications:

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.
- [Cache coherence](#): Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data (see [cache coherence](#) and [memory coherence](#)). Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between multiple processors. On the other hand they can sometimes become overloaded and become a bottleneck to performance.

Shared memory

(from Wikipedia)



...relatively easy to program...

...all processors share a single view of data...

...bottleneck to performance...

- CPU-to-memory connection becomes a bottleneck. Shared memory computers cannot scale very well. Most of them have ten or fewer processors.
- **Cache coherence**: Whenever one cache is updated with information that may be used by other processors, the change needs to be reflected to the other processors, otherwise the different processors will be working with incoherent data (see **cache coherence** and **memory coherence**). Such coherence protocols can, when they work well, provide extremely high-performance access to shared information between m and become a bottleneck

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
MOV $[x] \leftarrow 1$ MOV EAX $\leftarrow [y]$	MOV $[y] \leftarrow 1$ MOV EBX $\leftarrow [x]$

Can you guess the final register values: EAX = ? EBX = ?

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
MOV $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MOV EAX $\leftarrow [y]$	MOV EBX $\leftarrow [x]$

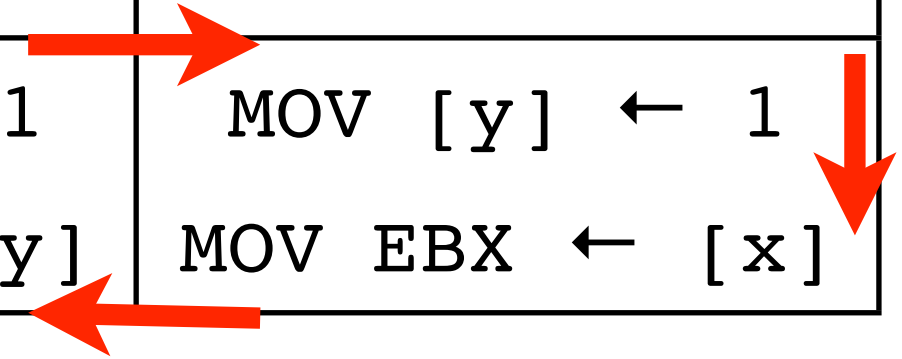
Can you guess the final register values: EAX = 1 EBX = 1

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
MOV $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MOV EAX $\leftarrow [y]$	MOV EBX $\leftarrow [x]$






Can you guess the final register values: EAX = 1 EBX = 1

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
 MOV $[x] \leftarrow 1$ MOV EAX $\leftarrow [y]$	 MOV $[y] \leftarrow 1$ MOV EBX $\leftarrow [x]$ 

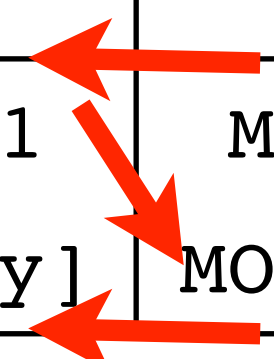
Can you guess the final register values: EAX = 1 EBX = 1

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
MOV $[x] \leftarrow 1$	MOV $[y] \leftarrow 1$
MOV EAX $\leftarrow [y]$	MOV EBX $\leftarrow [x]$






Can you guess the final register values: EAX = 1 EBX = 1

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
 MOV $[x] \leftarrow 1$ MOV EAX $\leftarrow [y]$	MOV $[y] \leftarrow 1$ MOV EBX $\leftarrow [x]$ 






Can you guess the final register values: EAX = 1 EBX = 0

Shared memory

Initial shared memory values: $[x]=0$ $[y]=0$

Per-processor registers: EAX EBX

Thread 0	Thread 1
 <code>MOV [x] ← 1</code>	<code>MOV [y] ← 1</code>
<code>MOV EAX ← [y]</code> 	<code>MOV EBX ← [x]</code> 

Can you guess the final register values: **EAX = 0 EBX = 1**

Shared memory

Thread 0	Thread 1
MOV [x] ← 1 MOV EAX ← [y]	MOV [y] ← 1 MOV EBX ← [x]

The possible outcomes should be:

- EAX : 1, EBX : 1
- EAX : 0, EBX : 1
- EAX : 1, EAX : 0

Shared memory



	Thread 0	Thread 1
MOV		
MOV		

Let's see...

The po

- EAX
- EAX
- EAX

Shared memory



Thread 0	Thread 1
MOV	
MOV	

The po

- EAX
- EAX
- EAX

We can observe

$$EAX = EBX = 0$$

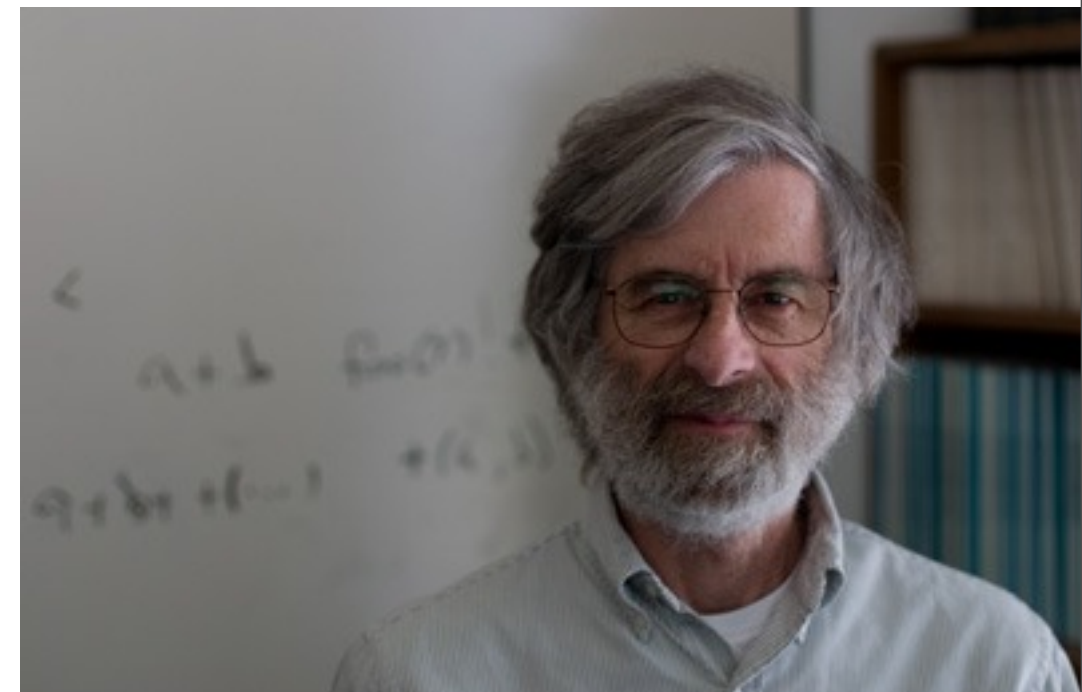
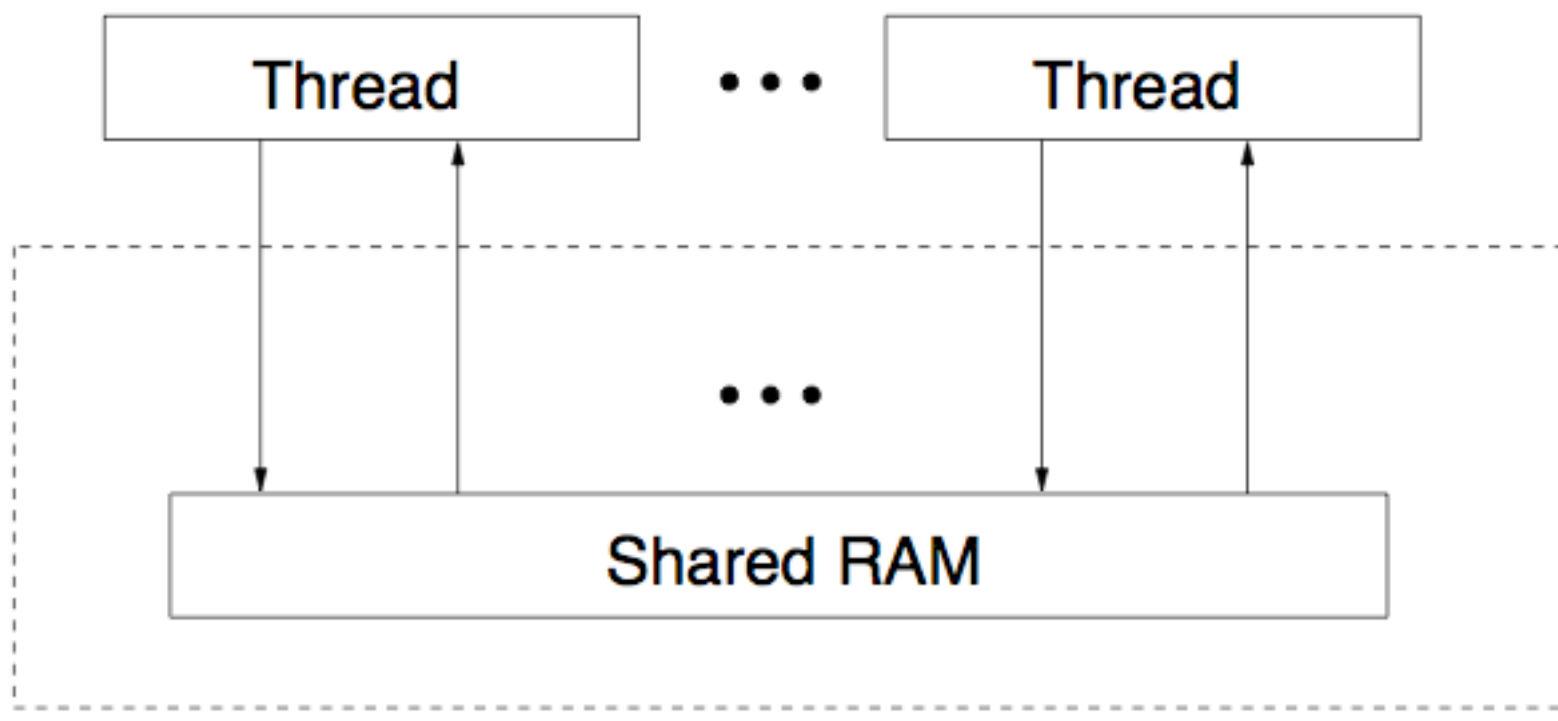
as well

According to most programmers

Multiprocessors have a *sequentially consistent* shared memory:

...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program...

Lamport, 1979.

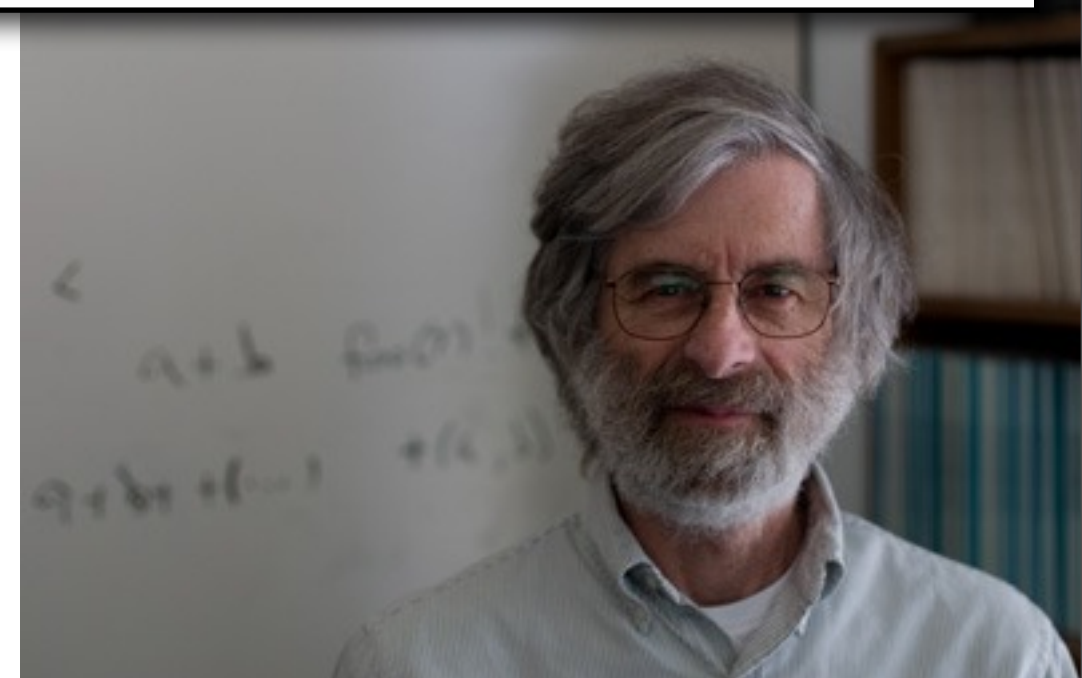
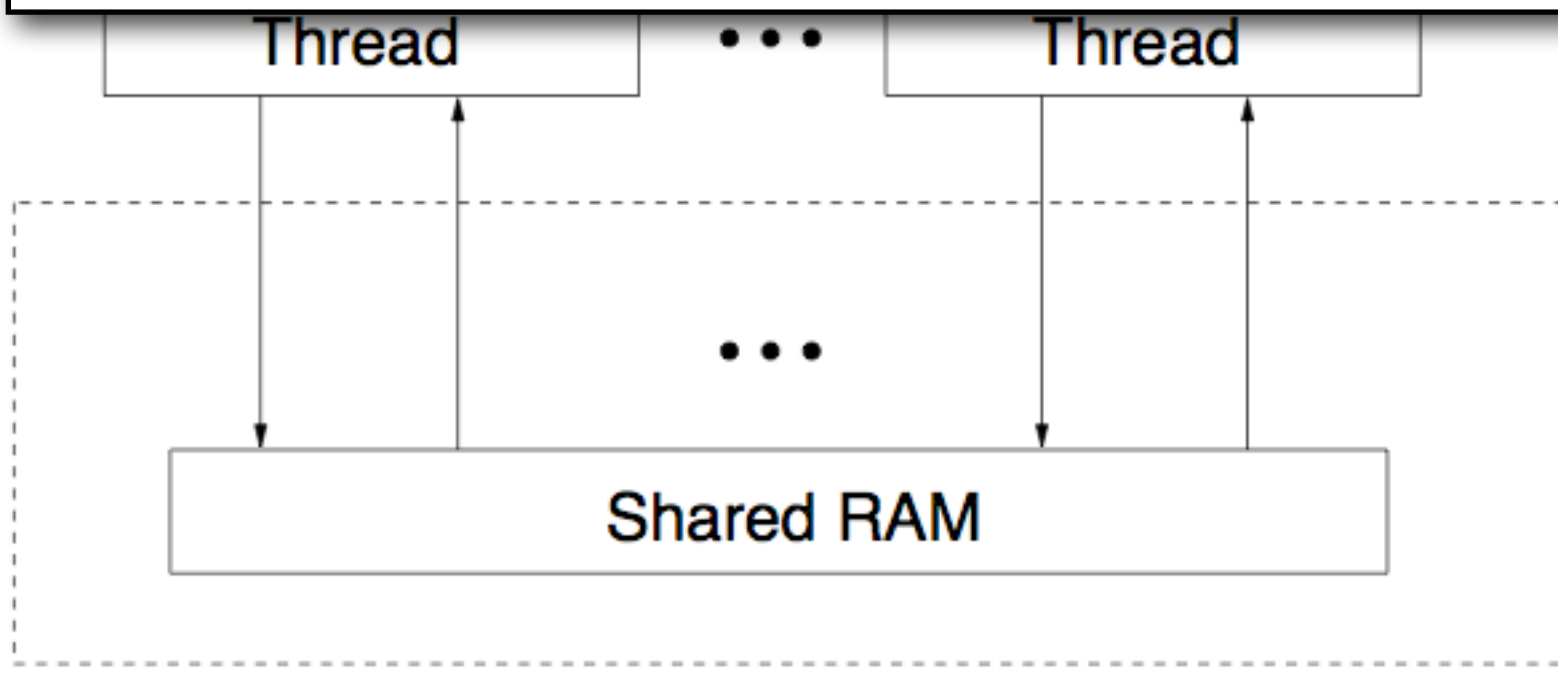


According to most programmers

Multiprocessors have a *sequentially consistent* shared memory:

Properties

- no thread local reordering
- each write becomes visible to all threads at the same time



According to most programmers

FALSE

Multiprocessors (and compilers) incorporate many
performance optimisations

(local store buffers, shadowing register files, hierarchies of caches, ...)

These are:

- unobservable by single-threaded programs;
- sometimes observable by concurrent code.

According to most programmers

Upshot:

only a relaxed (or weakly consistent)
view of the memory.

, ...)

Multi

(local s

These

• uno

• sometimes observable by concurrent code.

Not new



Multiprocessors since 1964 (Univac 1108A - or Burroughs, in '62)

Relaxed Memory since 1972 (IBM System 370/158MP)

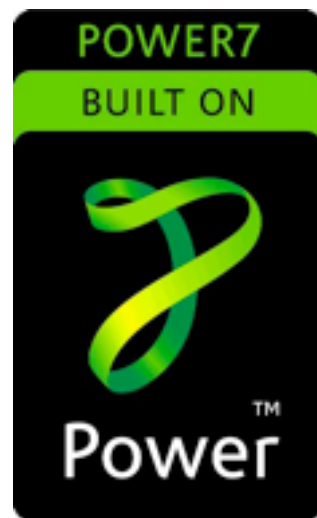
Eclipsed for a long time (except in high-end) by advances in performance:

- transistor counts (continuing)
- clock speed (hit power dissipation limit)
- ILP (hit smartness limit?)

Mass market multiprocessors since 2005



Intel Xeon E7
up to 20 hardware threads



IBM Power 795 server
up to 1024 hardware threads



Best quad core phone: 4 contenders examined

EARLY VIEW HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

Mass market multiprocessors since 2005



Intel Xeon E7
up to 20 hardware threads

Programming multiprocessors
no longer just for specialists



Best quad core phone: 4 contenders examined

EARLY VIEW HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

But it's hard!

1. Real memory models are subtle
2. Real memory models differ between architectures
3. Real memory models differ between languages

Almost none of the last 40 years' work on verification of concurrent code deals with relaxed memory (new trend in the last few years).

Much of the research on relaxed models does not address real processors and languages (new trend in the last few years).

But it's hard!

1. Real memory models are subtle

2.

3. Industrial processors and language specs
are often flawed

Also We've looked at the specs of x86, Power, ARM, Java, and C++

con

They all have problems

Mu

pro

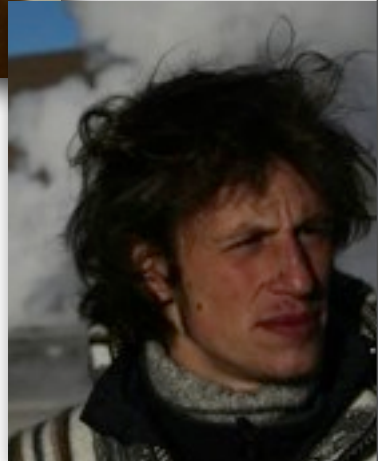
These lectures

Hardware models

- 1) why are industrial specs so often flawed?
focus on x86, with a glimpse of Power/ARM
- 2) usable models: x86-TSO, Power

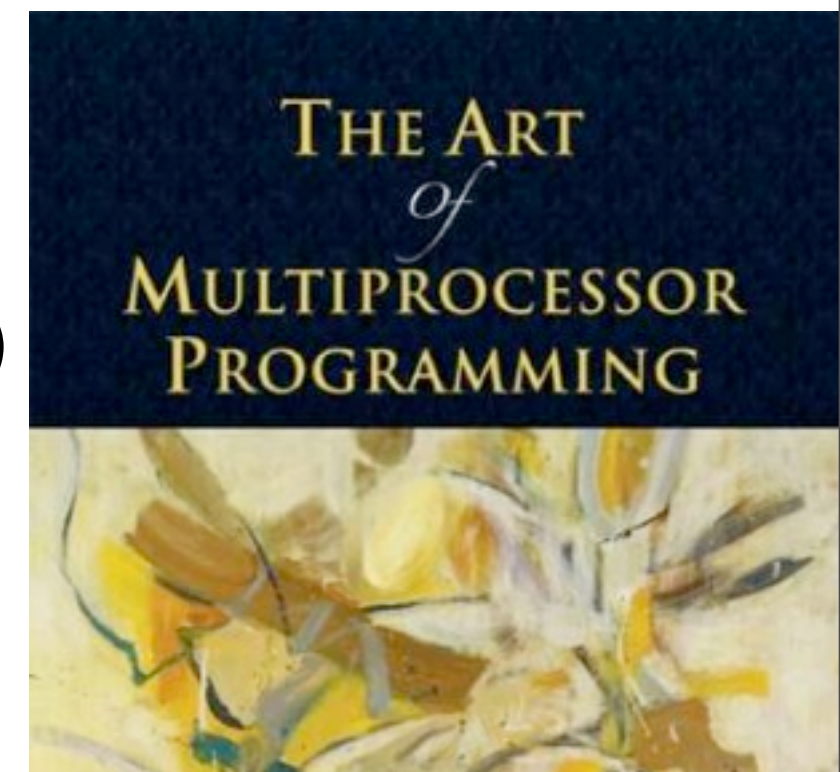
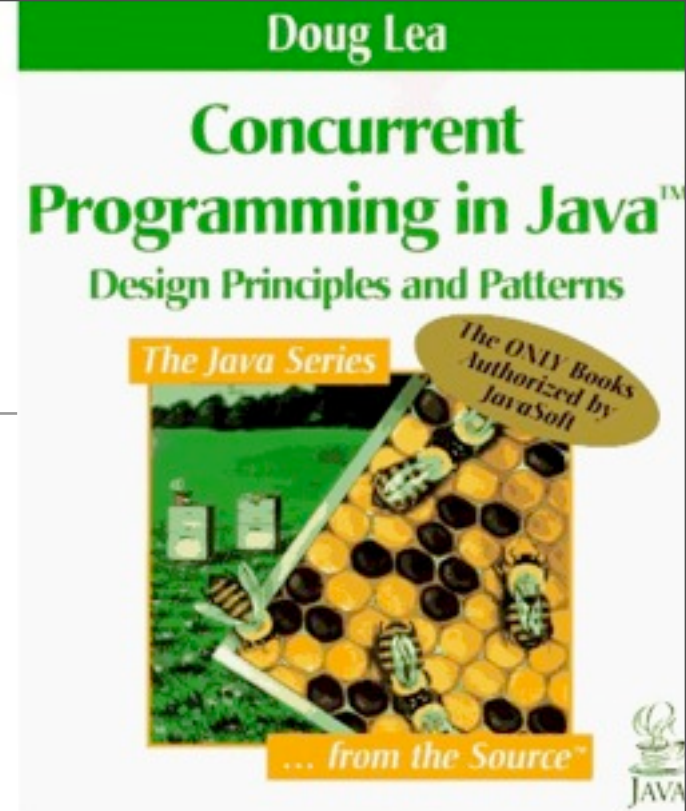
Programming language models

- 1) defining the semantics of a concurrent programming language
- 2) data-race freedom
- 3) soundness of compiler optimisations
- 4) an in-depth look at C11/C++11



Uses

1. how to code low-level concurrent datastructures
2. how to build concurrency testing and verification tools
3. how to specify and test multiprocessors
4. how to design and express high-level language definitions
- 5. to discover some ugly monsters still lurking in your multiprocessor / your favorite programming language (despite a lot of efforts)**



Hardware models

Architectures

Hardware manufacturers document **architectures**:

- **loose specifications**
- claimed to cover a **wide range** of past and future **processor implementations**.

Architectures should:

- **reveal enough** for effective programming;
- without **unduly constraining** future processor design.

Examples: Intel 64 and IA-32 Architectures SDM, AMD64 Architecture Programmer's Manual, Power ISA specification, ARM Architecture Reference Manual, ...



Intel® 64 and IA-32 Architectures
Software Developer's Manual



VOLUME 3A: System Programming Guide
Part 1



In practice

Architectures described by informal prose:

In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.

(Intel SDM, november 2006, vol3a, 10-5)

As we shall see, such descriptions are:

1) vague; 2) incomplete; 3) unsound.

Fundamental problem: prose specifications cannot be used to test programs or to test processor implementations.

Intel 64/IA32 and AMD64 - before Aug. 2007

Era of Vagueness

A model called **Processor Ordering**, informal prose.

Example: Linux kernel mailing list, 20 nov. - 7 déc. 1999 (143 posts).

A one-instruction programming question, a microarchitectural debate!

Keywords: speculation, ordering, causality, retire, cache...

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock_optimization\(i386\)](#)"
Topics: [BSD](#); [FreeBSD](#); [SMP](#)

People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you have to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that should have been serialized by the spinlock.

```
spin_unlock();
```

```
spin_lock();
```

```
a = 1;
/* cache miss satisfied, the "a" line is bouncing back and forth */
```

```
b gets the value 1
```

```
a = 0;
and it returns "1", which is wrong for any working spinlock.
```

Unlikely? Yes, definitely. Something we are willing to live with as a potential bug in any real kernel? Definitely not.

Manfred objected that according to the *Pentium Processor Family Developers Manual, Vol3, Chapter 19.2 Memory Access Ordering*, "to optimize performance, the Pentium processor allows memory reads to be reordered ahead of buffered writes in most situations. Internally, CPU reads (cache hits) can be reordered around buffered writes. Memory reordering does not occur at the pins, reads (cache miss) and writes appear in-order." He concluded from this that the second CPU would never see the spin_unlock() before the "b=a" line. Linus agreed that on a Pentium, Manfred was right. However, he quoted in turn from the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding." He explained:

A Pentium is a in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you'll never see the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the `_spinlock_` will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing) the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btcl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+xchgl" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imaging running the following test in a loop on multiple CPU's:

```
int test_locking(void){

static int a; /* protected by spinlock */
int b;
```

But a Pentium is also very uninteresting from a SMP standpoint these days. It's just too weak with too little per-CPU cache etc..

This is why the PPro has the MTRR's - exactly to let the core do speculation (a Pentium doesn't need MTRR's, as it won't re-order anything external to the CPU anyway, and in fact won't even re-order things internally).

Jeff V. Merkey added:

What Linus says here is correct for PPro and above. Using a mov instruction to unlock does work fine on a 486 or Pentium SMP system, but as of the PPro, this was no longer the case, though the window is so infinitesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is writtne to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the piplines are no longer invalid and the buffers should be blown out.

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

It will always return 0. You don't need "spin_unlock()" to be serializing.

The only thing you need is to make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you're changing something to be observable on other processors.

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" isnot serializing any more, so there is very little effective ordering between the two actions

```
b = a;spin_unlock();
```

as they access completely different data (ie no data dependencies in sight). So what you could end up doing is equivalent to

```
CPU#1
CPU#2
b = a; /* cache miss, we'll delay this.. */
```

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc absolutely have to have completed first, cache-miss or not.

He went on:

Since the instructions for the store in the spin_unlock have to have been externally observed for spin_lock to be aquired (presuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the value of "a" have to have completed first.

In general, IA32 is Processor Ordered for cacheable accesses. Speculation doesn't affect this. Also, stores are not observed speculatively on other processors.

There was a long clarification discussion, resulting in a complete turnaround by Linus:

Everybody has convinced me that yes, the Intel ordering rules are strong enough that all of this really is legal, and that's what I wanted. I've gotten sane explanations for why serialization (as opposed to just the simple locked access) is required for the lock() side but not the unlock() side, and that lack of symmetry was what bothered me the most.

Oliver made a strong case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

Thanks, guys, we'll be that much faster due to this..

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock optimization\(i386\)](#)"
Topics: [BSD](#); [FreeBSD](#); [SMP](#)
People: [Linus Torvalds](#); [Jeff V. Merkey](#); [Erich Boleyn](#); [Manfred Spraul](#); [Peter Samuelson](#); [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you have to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that should have been serialized by the spinlock.

```
spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the "a" line is bound to the cache,
b gets the value 1
```



A Pentium is an in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you'll never see the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the `_spinlock_` will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing) the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btrl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+xchgl" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imaging running the following test in a loop on multiple CPU's:

```
int test_locking(void){
static int a; /* protected by spinlock */
int b;
```

We can shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain.

this was no longer the case, though the window is so infinitesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is writtne to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the piplines are no longer invalid and the buffers should be blown out.

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

It will always return 0. You don't need "spin_unlock()" to be serializing.

The only thing you need is to make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you're changing something to be observable on other processors.

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" isnot serializing any more, so there is very little effective ordering between the two actions

```
b = a;spin_unlock();
```

as they access completely different data (ie no data dependencies in sight). So what you could end up doing is equivalent to

```
CPU#1
CPU#2
b = a; /* cache miss, we'll delay this.. */
```

The reason for this is that stores can only possibly be observed when all prior instructions have retired (i.e. the store is not sent outside of the processor until it is committed state, and the earlier instructions are already committed by that time), so the any loads, stores, etc absolutely have to have completed first, cache-miss or not.

He went on:

Since the instructions for the store in the spin_unlock have to have been externally observed for spin_lock to be aquired (presuming a correctly functioning spinlock, of course), then the earlier instructions to set "b" to the value of "a" have to have completed first.

In general, IA32 is Processor Ordered for cacheable accesses. Speculation doesn't affect this. Also, stores are not observed speculatively on other processors.

There was a long clarification discussion, resulting in a complete turnaround by Linus:

Everybody has convinced me that yes, the Intel ordering rules are strong enough that all of this really is legal, and that's what I wanted. I've gotten sane explanations for why serialization (as opposed to just the simple locked access) is required for the lock() side but not the unlock() side, and that lack of symmetry was what bothered me the most.

Oliver made a strong case that the lack of symmetry can be adequately explained by just simply the lack of symmetry wrt speculation of reads vs writes. I feel comfortable again.

Thanks, guys, we'll be that much faster due to this..

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock optimization\(i386\)](#)"
Topics: [BSD](#); [FreeBSD](#); [SMP](#)
People: [Linus Torvalds](#); [Jeff V. Merkey](#); [Erich Boleyn](#); [Manfred Spraul](#); [Peter Samuelson](#); [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you have to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical region (maybe it missed a cache line), and get a stale value for any of the reads that should have been serialized by the spinlock.

```
spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the "a" line is bound to the cache,
b gets the value 1
```



A Pentium is an in-order machine, without any of the interesting speculation wrt reads etc. So on a Pentium you'll never see the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing) the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

I might be proven wrong, but I don't think I am.

Note that another thing is that yes, "btrl" may be the worst possible thing to use for this, and you might test whether a simpler "xor+xchgl" might be better - it's still serializing because it is locked, but it should be the normal 12 cycles that Intel always seems to waste on serializing instructions rather than 22 cycles.

Elsewhere, he gave a potential (though unlikely) exploit:

As a completely made-up example (which will probably never show the problem in real life, but is instructive as an example), imagine running the

4% speed-up in a benchmark test, making the optimization very valuable. The same optimization cropped up in the FreeBSD mailing list.

We can shave about 22 ticks off the asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain.

this was no longer the case, though the window is so infinitesimally small, most kernels don't hit it (Netware 4/5 uses this method but it's spinlocks understand this and the code is written to handle it. The most obvious aberrant behavior was that cache inconsistencies would occur randomly. PPro uses lock to signal that the pipelines are no longer invalid and the buffers should be blown out.

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Erich Boleyn, an Architect in an IA32 development group at Intel, also replied to Linus, pointing out a possible misconception in his proposed exploit. Regarding the code Linus posted, Erich replied:

It will always return 0. You don't need "spin_unlock()" to be serializing.

The only thing you need is to make sure there is a store in "spin_unlock()", and that is kind of true by the fact that you're changing something to be observable on other processors.

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

And yes, the above CAN return 1 with the proposed optimization. I doubt you can make it do so in real life, but hey, add another access to another variable in the same cache line that is accessed through another spinlock (to get cache-line ping-pong and timing effects), and I suspect you can make it happen even with a simple example like the above.

The reason it can return 1 quite legally is that your new "spin_unlock()" is not serializing any more, so there is very little effective ordering between the two actions

```
b = a; spin_unlock();
```

as they access completely different data (ie no data dependencies in sight). So the "b = a" and "spin_unlock()" end up doing is equivalent to

```
/* we'll delay this.. */
```

is that stores can only possibly be observed when all prior stores have been retired (i.e. the store is not sent outside of the processor until the processor is in a consistent state, and the earlier instructions are already committed to the cache).

first, cache-m

He went on:

Since the instructions are not retired until the processor is in a consistent state, the externally observed value of "a" have to be the value of "a" that was last committed to the cache.

In general, IA32 is not a store-order machine. The order of stores doesn't affect this. The order of reads does affect this. processors.

There was a long chain of replies. Linus:

Everybody has come to the conclusion that all of the sane explanations for the problem (cache line ping-pong, lack of symmetry was violated, etc.) are not the cause of the problem.

Oliver made a strong case for the problem being explained by just stores not being retired. I feel comfortable with that.

Thanks, guys, we'll be that much faster due to this.



1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock optimization\(i386\)](#)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you _have_ to have a serializing instruction in order to make sure that the processor doesn't re-order things around the unlock.

For example, with a simple write, the CPU can legally delay a read that happened inside the critical section until it has read a stale value for any of the resources held by the spinlock.

```
spin_unlock();

spin_lock();

a = 1;
/* cache miss satisfied, the processor
b gets the value 1
```

```
a = 0;
and it reads the value 1

Unlikely, but a
bug in a
```

Manfred Spraul, in the Pentium Manual, writes in around 1995 that (cache miss) CPU would be a Pentium manual, for speculative reads and stores.

A Pentium is an in-order machine. It never sees the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing) the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

return 1 with the proposed optimization. I doubt you but hey, add another access to another variable accessed through another spinlock (to get cache-coherency), and I suspect you can make it happen even as above.

What is legally is that your new "spin_unlock()" is not very little effective ordering between the two

different data (ie no data dependencies in sight). So this is equivalent to

say this.. */

is that stores can only possibly be observed when all prior stores are retired (i.e. the store is not sent outside of the processor's cache state, and the earlier instructions are already committed to the bus). This is completed



It does NOT WORK!
Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

g the optimization very valuable. The same optimization cropped up in the FreeBSD mailing list.

lock for a simple "movl \$0,%0" instruction, a huge gain.

no longer the case, though the window is so infinitesimally small, most processors don't hit it (Network 4/5 uses this method but it's spinlocks and this and the code is written to handle it. The most obvious behavior was that cache inconsistencies would occur randomly. The lock to signal that the pipelines are no longer invalid and the buffers blown out.

in the behavior Linus describes on a hardware analyzer, BUT IN SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD is still be on older Pentium hardware and that's why they don't can bite in some cases.

Boleyn, an Architect in an IA32 development group at Intel, also replied to pointing out a possible misconception in his proposed exploit. Regarding Linus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

thing you need is to make sure there is a store in "spin_unlock()", which is kind of true by the fact that you're changing something to be observable on other processors.

first, cache-inconsistency. He went on:

Since the instruction is externally observable, a functioning spinlock would have the value of "a" have to

In general, IA32 is doesn't affect this. processors.

There was a long clarification from Linus:

Everybody has come enough that all of the sane explanations for (access) is required of symmetry was v

Oliver made a strong explained by just s writes. I feel comfort

Thanks, guys, we'll be that much faster due to them

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock_optimization\(1386\)](#)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)
Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!
Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.
The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.
The issue is not writes being issued in-order (although all the Intel CPU books warn you NOT to assume that in-order write behaviour - I bet it won't be the case in the long run).

The issue is that you _have_ to have a serializing instruction in order to ensure that the processor doesn't re-order things around the unlock.
For example, with a simple write, the CPU can legally delay a read that happened inside the critical section until after the write, leaving a stale value for any of the readers that spinlock.



A Pentium is an in-order machine. It does not reorder reads or writes. So on a Pentium you will never see the problem.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.
Note that the fact that it does not crash now is quite possibly because of either
we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)
the window is probably very very small, and you have to be unlucky to hit it.
Faster CPU's, different compilers, whatever.

```
spin_lock()
{
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

return 1 with the proposed optimization. I doubt you but hey, add another access to another variable accessed through another spinlock (to get cache-locks), and I suspect you can make it happen even as above.
The only thing that is legally is that your new "spin_unlock()" is not a serializing instruction, so there is very little effective ordering between the two

It does NOT WORK!
Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

According to the *Pentium Processor Family Developers Manual, Vol3, Chapter 19.2 Memory Access Ordering*, "to optimize performance, the Pentium processor allows memory reads to be reordered ahead of buffered writes in most situations. Internally, CPU reads (cache hits) can be reordered around buffered writes. Memory reordering does not occur at the pins, reads (cache miss) and writes appear in-order."

...longer the case, though the window is so infinitesimally small, most don't hit it (Network 4/5 uses this method but it's spinlocks and this and the code is written to handle it. The most obvious behavior was that cache inconsistencies would occur randomly. ...lock to signal that the pipelines are no longer invalid and the buffers blown out.
...n the behavior Linus describes on a hardware analyzer, BUT IN SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD is still be on older Pentium hardware and that's why they don't can bite in some cases.
...yn, an Architect in an IA32 development group at Intel, also replied to pointing out a possible misconception in his proposed exploit. Regarding Linus posted, Erich replied:

...ays return 0. You don't need "spin_unlock()" to be serializing.
...thing you need is to make sure there is a store in "spin_unlock()", which is kind of true by the fact that you're changing something to be observable on other processors.

functioning spinlock. The value of "a" have to be 0.
In general, IA32 is doesn't affect this. processors.
There was a long cl... Linus:
Everybody has con... enough that all of t... sane explanations f... access) is required... of symmetry was v...
Oliver made a stro... explained by just s... writes. I feel comf...



Thanks, guys, we'll be back much faster due to this.

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "spin_unlock_optimization(i386)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being warn you NOT to assume case in the long run).

The issue is that you _have sure that the processor do

For example, with a simple happened inside the critical stale value for any of the r spinlock.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)
the window is probably very very small, and you have to be unlucky to hit it. Faster CPU's, different compilers, whatever.

```
spin_lock()  
a = 1;  
mb();  
a = 0;  
mb();  
b = a;  
spin_unlock();  
return b;  
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into "b" and return it. So if we EVER returned anything else, the spinlock would obviously be completely broken, wouldn't you say?

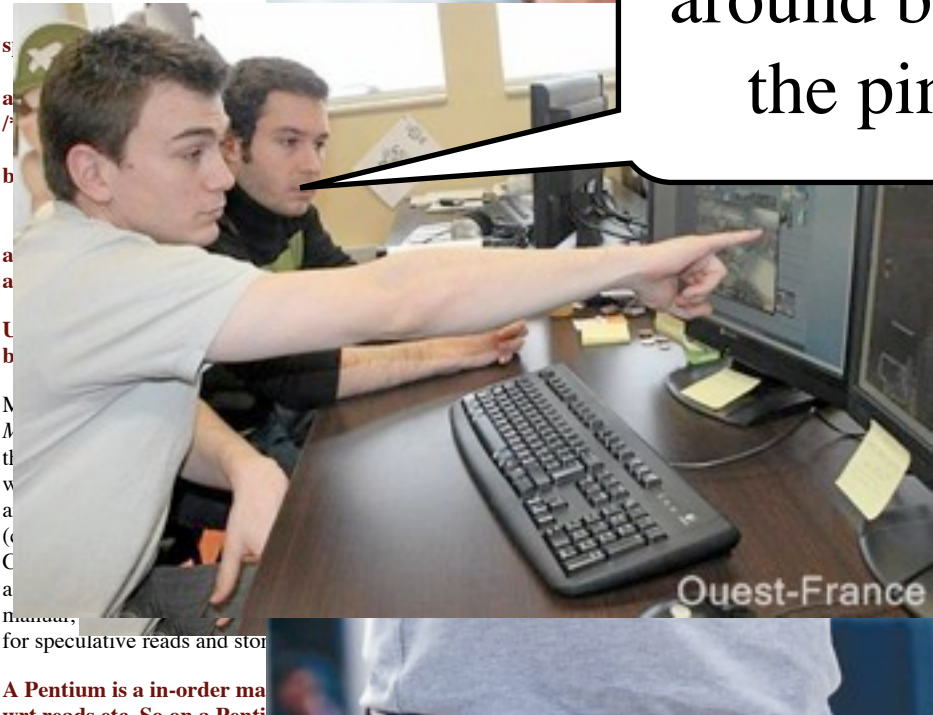
return 1 with the proposed optimization. I doubt you but hey, add another access to another variable accessed through another spinlock (to get cache-ects), and I suspect you can make it happen even e above.

ite legally is that your new "spin_unlock()" isnot is very little effective ordering between the two

It does NOT WORK!

From the Pentium Pro manual, "The only enhancement in the PentiumPro processor is the added support for speculative reads and store-buffer forwarding."

spin_unlock();



A Pentium is a in-order ma wrt reads etc. So on a Pentium you n never see the problem.

around buffered writes. Memory re the pins, reads (cache miss) and

on't hit it (Network 4/5 uses this method but it's spinlocks and this and the code is writtne to handle it. The most obvious behavior was that cache inconsistencies would occur randomly. lock to signal that the piplines are no longer invalid and the buffers blown out.

n the behavior Linus describes on a hardware analyzer, BUT N SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD ist still be on older Pentium hardware and that's why they don't can bite in some cases.

yn, an Architect in an IA32 development group at Intel, also replied to ntng out a possible misconception in his proposed exploit. Regarding inus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

hing you need is to make sure there is a store in "spin_unlock()", s kind of true by the fact that you're changing something to be observable on other processors.

ssor Family Developers
ary Access Ordering "to



So
y
d
at
ior
ted
d
b" to the
ation
er
und by
strong
otten
e locked
at lack
ely
ds vs

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "spin_unlock_optimization(i386)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people use it, and let them get faster timings. They will crash, eventually.

The window may be small, but if you do this, then suddenly spinlocks aren't reliable any more.

The issue is not writes being warn you NOT to assume case in the long run).

The issue is that you _have_ sure that the processor do

For example, with a simple happened inside the critical stale value for any of the r spinlock.

From the Pentium only enhanced processor is speculative

I have seen the behavior Linus describes on a hardware analyzer, BUT ONLY ON SYSTEMS THAT WERE PPRO AND ABOVE. I guess the BSD people must still be on older Pentium hardware and that's why they don't know this can bite in some cases.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.
Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

```
spin_lock()
a = 1;
mb();
a = 0;
mb();
b = a;
spin_unlock();
return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into b, so, the spinlock would

imization. I doubt you s to another variable spinlock (to get cache- make it happen even

"spin_unlock()" isnot ring between the two

So

"to

y

ed at

ior ted ed

b" to the

ation er

und by

strong otten e locked hat lack

ely ds vs



Quest-France

writes. Memory re s (cache miss) and

are 4/5 uses this method but it's spinlocks code is writtne to handle it. The most obvious hat cache inconsistencies would occur randomly. hat the piplines are no longer invalid and the buffers

Linus describes on a hardware analyzer, BUT HAT WERE PPRO AND ABOVE. I guess the BSD der Pentium hardware and that's why they don't e cases.

t in an IA32 development group at Intel, also replied to ble misconception in his proposed exploit. Regarding mus posted, Erich replied:

ays return 0. You don't need "spin_unlock()" to be serializing.

hing you need is to make sure there is a store in "spin_unlock()", s kind of true by the fact that you're changing something to be observable on other processors.



a manually, for speculative reads and stor

A Pentium is a in-order ma wrt reads etc. So on a Pentium you n never see the problem.

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock optimization\(i386\)](#)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people eventually.

The window may be small, but it's not reliable any more.

The issue is not writes but reads. I warn you NOT to assume the case in the long run).

The issue is that you _have_ to ensure that the processor

For example, with a simple read, it happened inside the critical section, stale value for any of the spinlock.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.

Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

```
spin_lock()  
{  
    a = 1;  
    mb();  
    a = 0;  
    mb();  
    b = a;  
    spin_unlock();  
    return b;  
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into b, so the spinlock would

It will always return 0. You don't need "spin_unlock()" to be serializing.

Linus describes on a BUT ONLY ON HERE PPRO AND D people must still be aware and that's why

processor is speculative

on older Pentium hardware they don't know



writes. s (cache

are 4/5 uses this method. The code is written to handle that cache inconsistency that the pipelines are no

Linus describes on a h HAT WERE PPRO AND der Pentium hardware ne cases.

t in an IA32 development ble misconception in his mus posted, Erich replied:

ays return 0. You don't need "spin_un

thing you need is to make sure there is s kind of true by the fact that you're changing something that's observable on other processors.



Intel guy

a a Manually, for speculative reads and stor

A Pentium is a in-order machine. It doesn't do out-of-order reads etc. So on a Pentium you never see the problem.

imization. I doubt you s to another variable spinlock (to get cache- make it happen even

"spin_unlock()" isnot ring between the two

So

"to

y

ed at

b" to the

ation er

und by

strong otten e locked hat lack

ely ds vs

1. spin_unlock() Optimization On Intel
20Nov1999-7Dec1999 (143 posts) Archive Link: "[spin_unlock optimization\(i386\)](#)"
Topics: [BSD](#), [FreeBSD](#), [SMP](#)
People: [Linus Torvalds](#), [Jeff V. Merkey](#), [Erich Boleyn](#), [Manfred Spraul](#), [Peter Samuelson](#), [Ingo Molnar](#)

Manfred Spraul thought he'd found a way to shave spin_unlock() down from about 22 ticks for the "lock; btrl \$0,%0" asm code, to 1 tick for a simple "movl \$0,%0" instruction, a huge gain. Later, he reported that Ingo Molnar noticed a 4% speed-up in a benchmark test, making the optimization very valuable. Ingo also added that the same optimization cropped up in the FreeBSD mailing list a few days previously. But Linus Torvalds poured cold water on the whole thing, saying:

It does NOT WORK!

Let the FreBSD people eventually.

The window may be small, but it's not reliable any more.

The issue is not writes but reads. I warn you NOT to assume the case in the long run).

The issue is that you _have_ to ensure that the processor

For example, with a spinlock, it happened inside the critical section that the stale value is read from the spinlock.

Note that I actually thought this was a legal optimization, and for a while I had this in the kernel. It crashed. In random ways.
Note that the fact that it does not crash now is quite possibly because of either

we have a lot less contention on our spinlocks these days. That might hide the problem, because the _spinlock_ will be fine (the cache coherency still means that the spinlock itself works fine - it's just that it no longer works reliably as an exclusion thing)

```
spin_lock()
{
    a = 1;
    mb();
    a = 0;
    mb();
    b = a;
    spin_unlock();
    return b;
}
```

Now, OBVIOUSLY the above always has to return 0, right? All accesses to "a" are inside the spinlock, and we always set it to zero before we read it into b. If not, the spinlock would

optimization. I doubt you can write a program that makes it happen even once.

"spin_unlock()" isn't atomic. It's a write followed by a read. If there's a write between the two, the spinlock would

So

rior
ted
ed

b" to the

ation
er

und by

strong
often
locked
that lack

ely
ds vs

It will always read 0. "spin_unlock" will be faster than a lock. Thanks, guys, we'll be that much faster due to this..

they don't know

writes.
s (cache

Intel guy

are 4/5 uses this method. The code is written to handle the case that cache inconsistencies exist, but that the pipelines are not

Linus describes on a hardware issue. THAT WERE PRO AND CON under Pentium hardware in some cases.

in an IA32 development. It's a subtle misconception in his code. I posted, Erich replied:

ays return 0. You don't need "spin_unlock"

thing you need is to make sure there is no kind of true by the fact that you're changing something that's observable on other processors.



Intel 64/IA32 and AMD64 - Aug. 2007 / Oct. 2008

Intel publishes a white paper, defining 8 informal-prose principles, e.g.

P1. Loads are not reordered with older loads.

P2. Stores are not reordered with older stores.

supported by 10 litmus test (illustrating allowed or forbidden behaviours), e.g.:

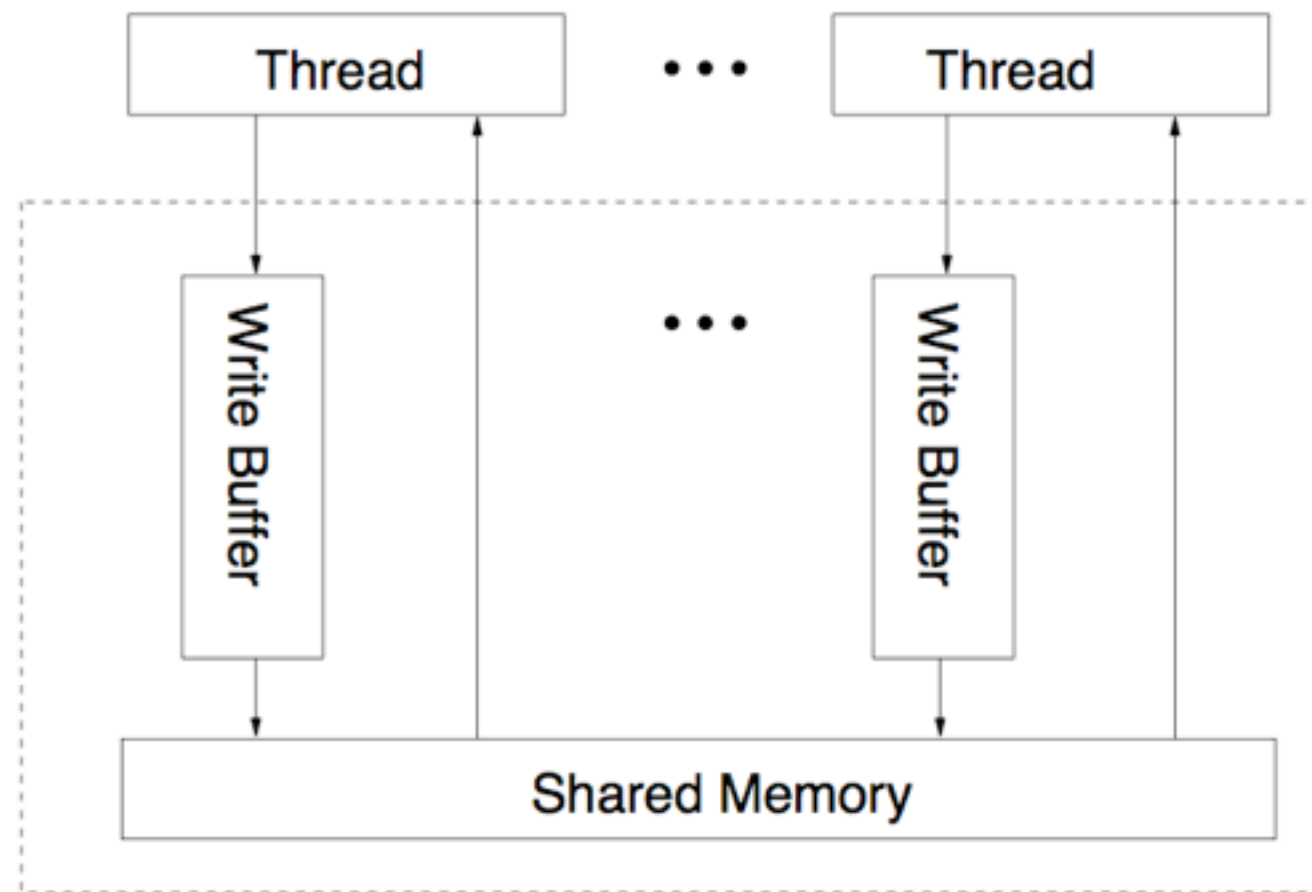
Thread 0	Thread 1
MOV [x] ← 1	MOV EAX ← [y] (1)
MOV [y] ← 1	MOV EBX ← [x] (0)
Forbidden final state: $EAX = 1 \wedge EBX = 0$	

P3. Loads may be reordered with older stores to different locations but not with older stores to the same location.

Thread 0	Thread 1
MOV [x] ← 1	MOV [y] ← 1
MOV EAX ← [y] (0)	MOV EBX ← [x] (0)
Allowed final state: $0:EAX = 0 \wedge 1:EBX = 0$	

P3. Loads may be reordered with older stores to different locations but not with older stores to the same location.

Thread 0	Thread 1
$\text{MOV } [x] \leftarrow 1$ $\text{MOV EAX} \leftarrow [y] \text{ (0)}$	$\text{MOV } [y] \leftarrow 1$ $\text{MOV EBX} \leftarrow [x] \text{ (0)}$
Allowed final state: $0:\text{EAX} = 0 \wedge 1:\text{EBX} = 0$	

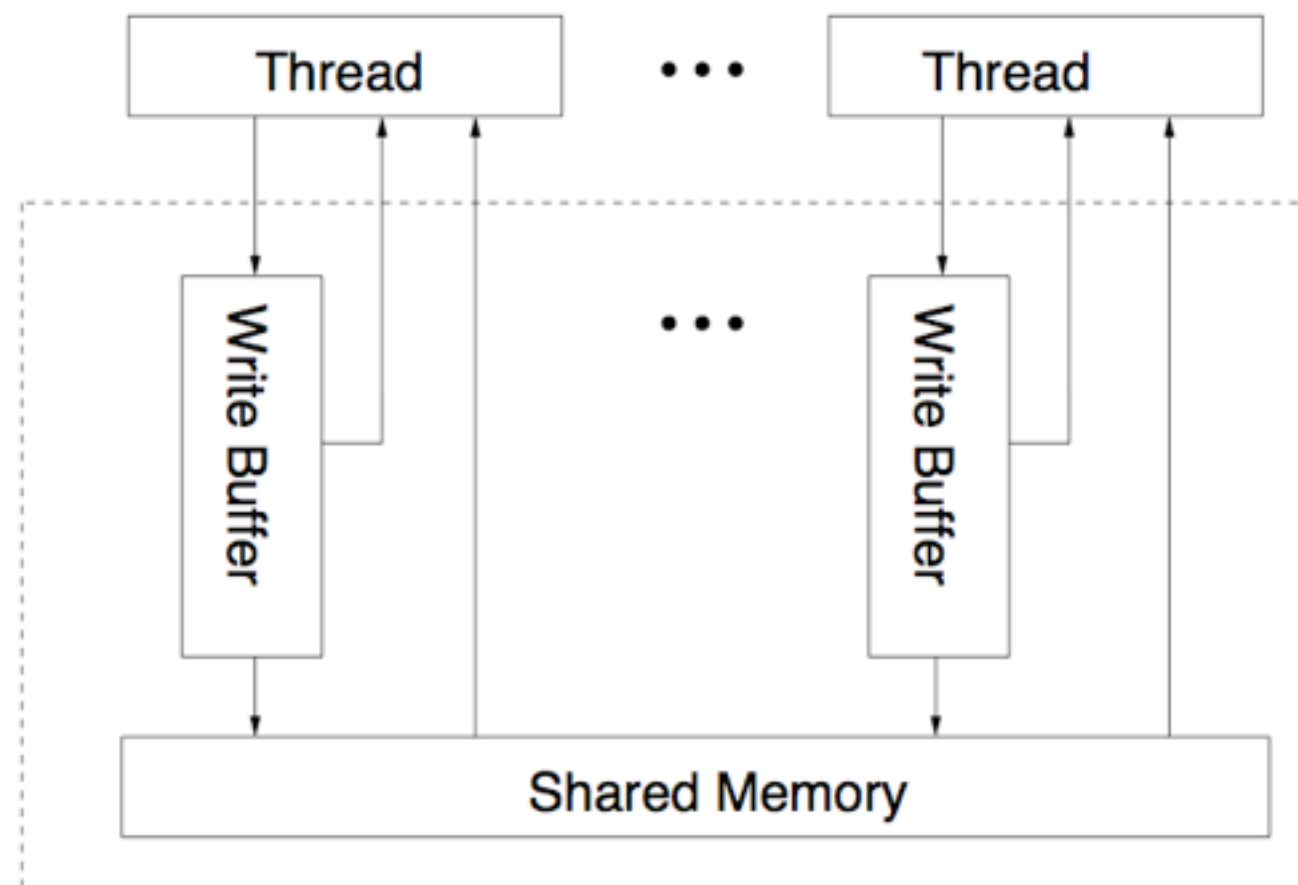


Litmus test 2.4: intra-processor forwarding is allowed

Thread 0	Thread 1
$\text{MOV } [x] \leftarrow 1$ $\text{MOV EAX} \leftarrow [x] \text{ (1)}$ $\text{MOV EBX} \leftarrow [y] \text{ (0)}$	$\text{MOV } [y] \leftarrow 1$ $\text{MOV ECX} \leftarrow [y] \text{ (1)}$ $\text{MOV EDX} \leftarrow [x] \text{ (0)}$
Allowed final state: $0:\text{EAX} = 1 \wedge 0:\text{EBX} = 0 \wedge$ $1:\text{ECX} = 1 \wedge 1:\text{EDX} = 1$	

Litmus test 2.4: intra-processor forwarding is allowed

Thread 0	Thread 1
$\text{MOV } [x] \leftarrow 1$ $\text{MOV EAX} \leftarrow [x] \text{ (1)}$ $\text{MOV EBX} \leftarrow [y] \text{ (0)}$	$\text{MOV } [y] \leftarrow 1$ $\text{MOV ECX} \leftarrow [y] \text{ (1)}$ $\text{MOV EDX} \leftarrow [x] \text{ (0)}$
Allowed final state: $0:\text{EAX} = 1 \wedge 0:\text{EBX} = 0 \wedge$ $1:\text{ECX} = 1 \wedge 1:\text{EDX} = 1$	

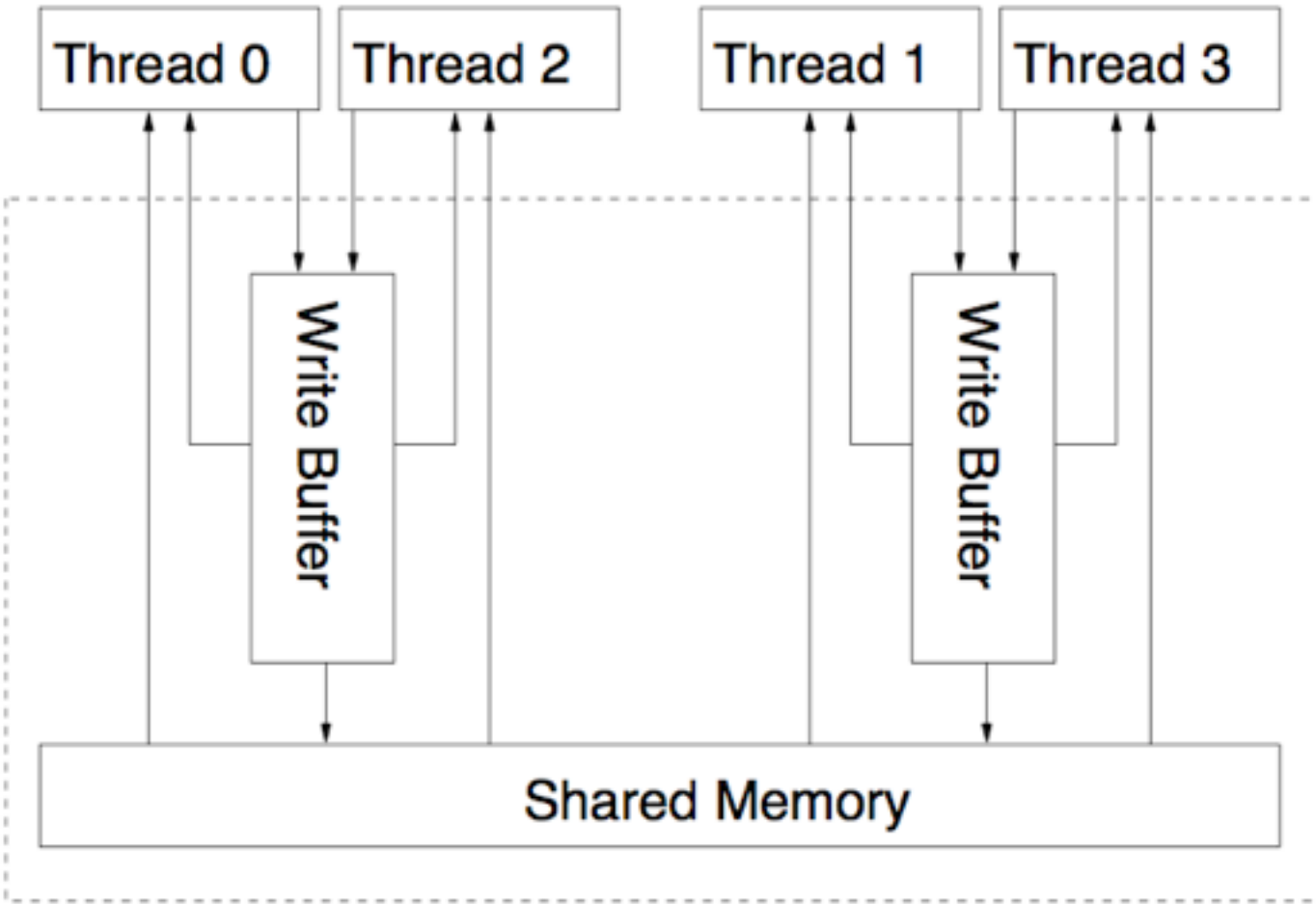


Thread 0	Thread 1	Thread 2	Thread 3
MOV [x] ← 1	MOV [y] ← 1	MOV EAX ← [x] MOV EBX ← [y]	MOV ECX ← [y] MOV EDX ← [x]
		(1) (0)	(1) (0)
Final state: 2:EAX = 1 ∧ 2:EBX = 0 ∧ 3:ECX = 1 ∧ 3:EDX = 0			

Thread 0	Thread 1	Thread 2	Thread 3
MOV [x] ← 1	MOV [y] ← 1	MOV EAX ← [x] (1)	MOV ECX ← [y] (1)
		MOV EBX ← [y] (0)	MOV EDX ← [x] (0)
Final state: 2:EAX = 1 ∧ 2:EBX = 0 ∧ 3:ECX = 1 ∧ 3:EDX = 0			

Microarchitecturally plausible?

Yes, with e.g. shared store buffers.



Ambiguity

P1-P4: ... may be reordered with ...

P5: Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation.

Thread 0	Thread 1	Thread 2
MOV [x] ← 1	MOV EAX ← [x] (1) MOV [y] ← 1	MOV EBX ← [y] (1) MOV ECX ← [x] (0)
Forbidden final state: $1:EAX = 1 \wedge 2:EBX = 1 \wedge 2:ECX = 0$		

Ambiguity

P1-P4: ... may be reordered with ...

*P5: In
i.e. stores
consis*

*es —
r*

Ambiguity:
when are two stores casually related?

Thread 0	Thread 1	Thread 2
MOV [x] ← 1	MOV EAX ← [x] (1) MOV [y] ← 1	MOV EBX ← [y] (1) MOV ECX ← [x] (0)
Forbidden final state: 1:EAX = 1 ∧ 2:EBX = 1 ∧ 2:ECX = 0		

Unsoundness

Example from Paul Loewenstein:

Thread 0	Thread 1
$[x] \leftarrow 1$ $EAX \leftarrow [x] \text{ (1)}$ $EBX \leftarrow [y] \text{ (0)}$	$[y] \leftarrow 2$ $[x] \leftarrow 2$
$0:EAX = 1 \wedge 0:EBX = 0 \wedge x = 1$	

Observed on real hardware, but not allowed by the ‘principles’:

- “Stores are not reordered with other stores”
- “Stores to the same location have a total order”

Unsoundness

Example from Paul Loewenstein:

The Intel White Paper specification
is unsound

(and our POPL x86-CC paper too)

Observe

- “Store to the same location have a total order”
- “Stores to the same location have a total order”

Intel 64/IA32 and AMD64, Nov. 2008 - now

SDM rev 29-31.

- Not unsound in the previous sense
- Explicitly exclude IRIW, so not weak in that sense. New principle:

Any two stores are seen in a consistent order by processors other than those performing the stores.

But... still ambiguous, and the view by those processors is left entirely unspecified!

Intel 64/IA32 and AMD64, Nov. 2008 - now

SDM rev 29-31.

- Not unsound in the previous sense
- Explicitly exclude IRIW, so not weak in that sense. New principle:

Any two stores are seen in a consistent order by processors other than those performing the stores.

But... still ambiguous, and the view by those processors is left entirely unspecified!

Thread 0	Thread 1
MOV [x] ← 1	MOV [x] ← 2
MOV EAX ← [x] (2)	MOV EBX ← [x] (1)
0:EAX = 2 ∧ 1:EBX = 1	

Power: much more relaxed than x86

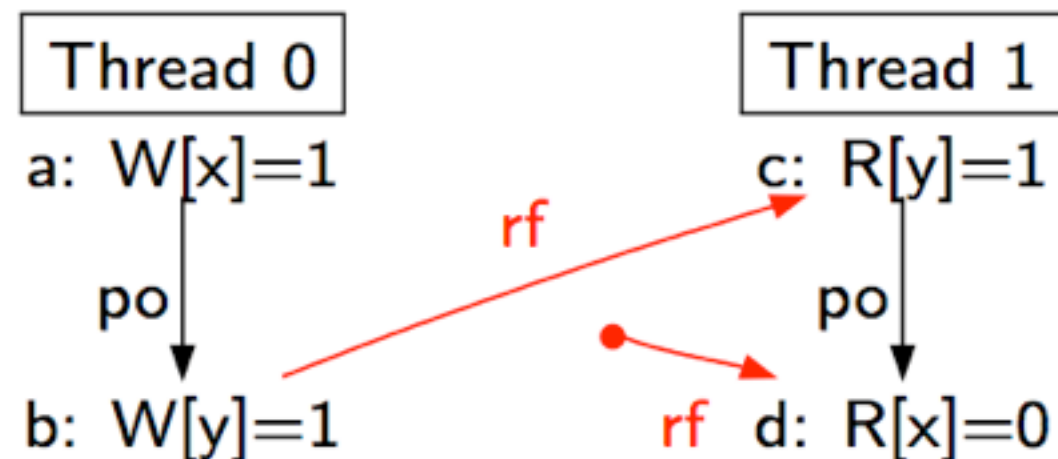
Thread 0	Thread 1
<code>x = 1</code> <code>y = 1</code>	<code>while (y==0) {};</code> <code>r = x</code>

Observable behaviour: `r = 0`

Power: much more relaxed than x86

Thread 0	Thread 1
$x = 1$ $y = 1$	<code>while (y==0) {};</code> $r = x$

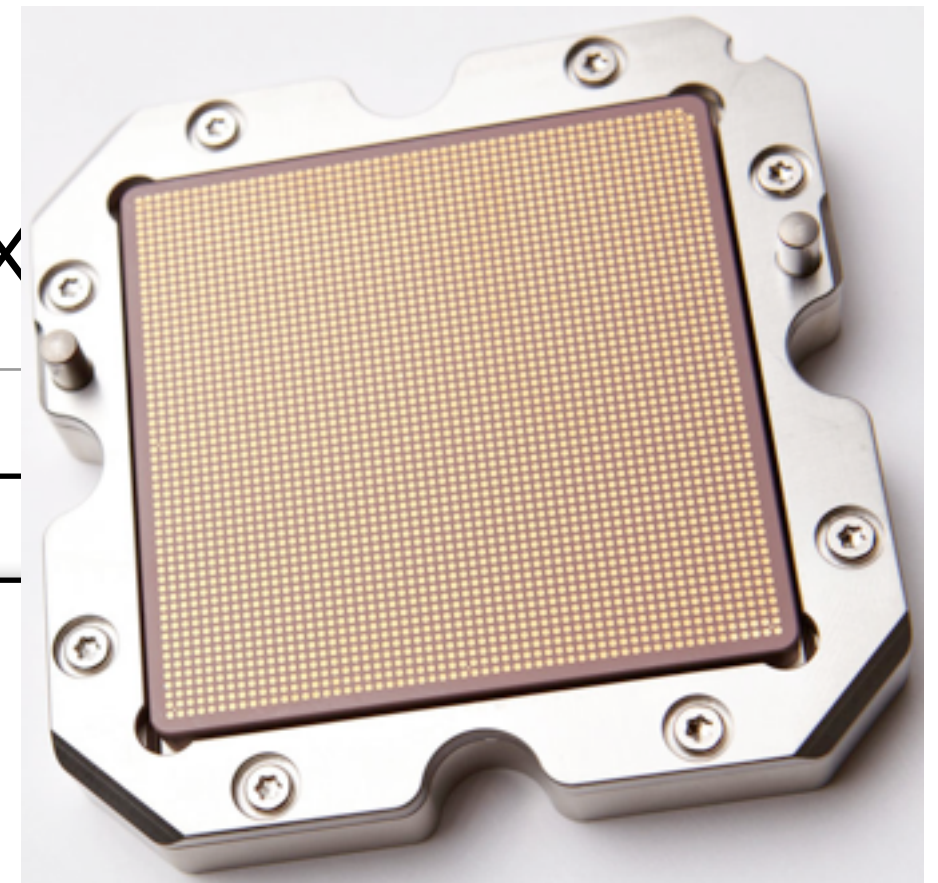
Observable behaviour: $r = 0$



Forbidden on SC and x86-TSO

Allowed and observed on Power

Power: much more relaxed than x



Thread 0

Thread

Let's see...

Thread

a: $W[x]$

po
↓

b: $W[y]$

SO
power

Power ISA 2.06 and ARM v7



Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to compute dependencies and to define the semantics of barriers.

Power ISA 2.06 and ARM v7



Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to compute dependencies and to define the semantics of barriers.

The definition of performed refers to an hypothetical store by P2.

A memory model should define if a particular execution is allowed. It is awkward to make a definition that **explicitly quantifies over all hypothetical variant executions.**

Power ISA 2.06 and ARM v7



Key concept: actions being performed.

A load by a processor (P1) is performed with respect to any

"all that horrible horribly incomprehensible and confusing [...] text that no-one can parse or reason with — not even the people who wrote it"

— Anonymous Processor Architect, 2011

A memory model should define if a particular execution is allowed.

It is awkward to make a definition that **explicitly quantifies over all hypothetical variant executions.**

Why all these problems?

Recall that vendor architectures are:

- loose specifications
- claimed to cover a wide range of past and future processor implementations.

Architectures should:

- reveal enough for effective programming;
- without unduly constraining future processor design.

Why all these problems?

Recall that vendor **architectures** are:

- loose
- claim
- imple

Archite

- reve
- with

There is a big tension between these,
with internal politics and inertia.

Compounded by the informal-prose specification style.



Way out? Create *rigorous* memory models

- Unambiguous

mathematical language

- Sound w.r.t. experience

rigorous testing of the model against the hardware

- Consistent with what we know of vendor intentions

interaction with hardware developers

The ARM / IBM POWER memory model formalisation

(expressed in LEM, compiled to HOL / Isabelle / Coq)

```
let write_reaching_coherence_point_action m s w =  
  let writes_past_coherence_point' =  
    s.writes_past_coherence_point union {w} in  
  let coherence' = s.coherence union  
    { (w,wother) | forall (wother IN (writes_not_past_coherence s)) |  
      (not (wother = w)) && (wother.w_addr = w.w_addr) } in  
  <| s with coherence = coherence';  
    writes_past_coherence_point = writes_past_coherence_point' |>  
  
let sem_of_instruction i ist =  
  match i with  
  | Padd set rD rA rB -> op3regs Add set rD rA rB ist  
  | Pandi rD rA simm -> op2regi And SetCR0 rD rA (intToV simm) ist  
end
```

Executing the specifications

Make the model accessible to programmers

Given a litmus test, compute the model-allowed executions:

- *operational*: search of abstract matching LTS
- *axiomatic*: enumerate all candidates, filter by axioms

Lem $\xrightarrow{\text{Lem}}$ OCaml $\xrightarrow{\text{search algorithm}}$ OCaml $\xrightarrow{\text{js_of_ocaml}}$ JavaScript

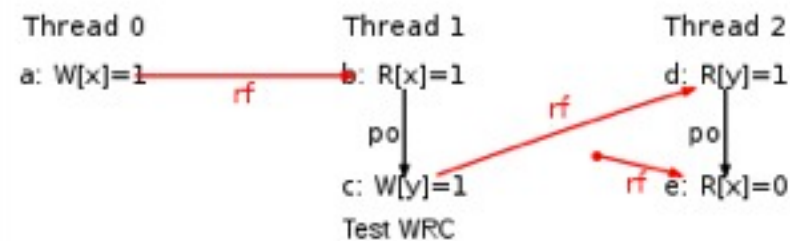
Examples tomorrow!

Testing the specifications

1. Systematically generate litmus tests out of the spec
2. Test them on real hardware and compare with the model

Test WRC

[Run WRC in model, using ppcmem](#)



```
PPC WRC
"Rfe PodRW Rfe PodRR Fre"
Cycle=Rfe PodRW Rfe PodRR Fre
{
0:r2=x;
1:r2=x; 1:r4=y;
2:r2=y; 2:r4=x;
}
P0      P1      P2
li r1,1  lwz r1,0(r2)  lwz r1,0(r2)
stw r1,0(r2)  li r3,1  lwz r3,0(r4)
          stw r3,0(r4)

exists
(1:r1=1 /\ 2:r1=1 /\ 2:r3=0)
```

WRC: Write to Read Causality

		Model	PowerG5	Power6	Power7
WRC	Allow	=	Ok, 44k/2.5G	Ok, 1.2M/13G	Ok, 25M/104G
WRC+data+addr	Allow	=	No, 0/3.3G	Ok, 705k/13G	Ok, 166k/105G
			Allow unseen		
WRC+syncs	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/157G
WRC+sync+addr	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/157G
WRC+lwsync+addr	Forbid	=	Ok, 0/3.3G	Ok, 0/17G	Ok, 0/137G
WRC+data+sync	Allow	=	No, 0/3.3G	Ok, 176k/13G	Ok, 75k/105G
			Allow unseen		
WRC+addr+ctrl	Allow	=	Ok, 43k/1.3G	Ok, 313k/4.3G	Ok, 4.5M/24G
WRC+addr+ctrlisync	Allow	=	No, 0/2.1G	Ok, 402k/4.3G	Ok, 69k/25G
			Allow unseen		
WRC+addr+isync	Allow	=	No, 0/2.1G	Ok, 403k/4.3G	Ok, 49k/25G
			Allow unseen		

Testing the specifications

- 1. Systematically generate litmus tests out of the spec
- 2. Test them on real hardware and compare with the model

Rigorous testing and interaction with hardware architects to validate the formalisation of the memory models

Test W

Run WR

Thread 0
a: W[x]=1

```
PPC WRC
"Rfe Po
Cycle=Rfe PodRW Rfe PodRR PFe
{
0:r2=x;
1:r2=x; 1:r4=y;
2:r2=y; 2:r4=x;
}
P0      P1      P2
li r1,1  lwz r1,0(r2) lwz r1,0(r2)
stw r1,0(r2) li r3,1  lwz r3,0(r4)
          stw r3,0(r4)
exists
(1:r1=1 /\ 2:r1=1 /\ 2:r3=0)
```

WRC+data+sync	Allow	=	No, 0/3.3G	Ok, 176k/13G	Ok, 75k/105G
			Allow unseen		
WRC+addr+ctrl	Allow	=	Ok, 43k/1.3G	Ok, 313k/4.3G	Ok, 4.5M/24G
WRC+addr+ctrlisync	Allow	=	No, 0/2.1G	Ok, 402k/4.3G	Ok, 69k/25G
			Allow unseen		
WRC+addr+isync	Allow	=	No, 0/2.1G	Ok, 403k/4.3G	Ok, 49k/25G
			Allow unseen		

Power7
25M/104G
66k/105G
0k, 0/157G
0k, 0/157G
0k, 0/137G

Key interfaces

Low-level software

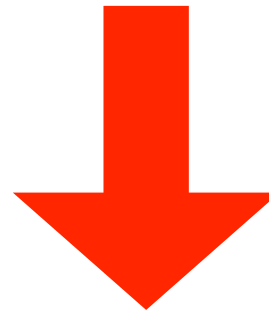
Applications

architectures

language definitions

Hardware

Low-level software



These key interfaces are necessarily loose specifications.

Informal prose is a terrible way to express loose specifications: ambiguous, untestable, and usually wrong.

Architectures and language definitions should be mathematically rigorous, clarifying precisely just how loose one wants them to be.

(common misconception: *precise* = *tight*?)



Concurrent programming
is hard!

1st year, Introduction to programming

Concurrent programming
is hard!

Concurrent programming
is hard!



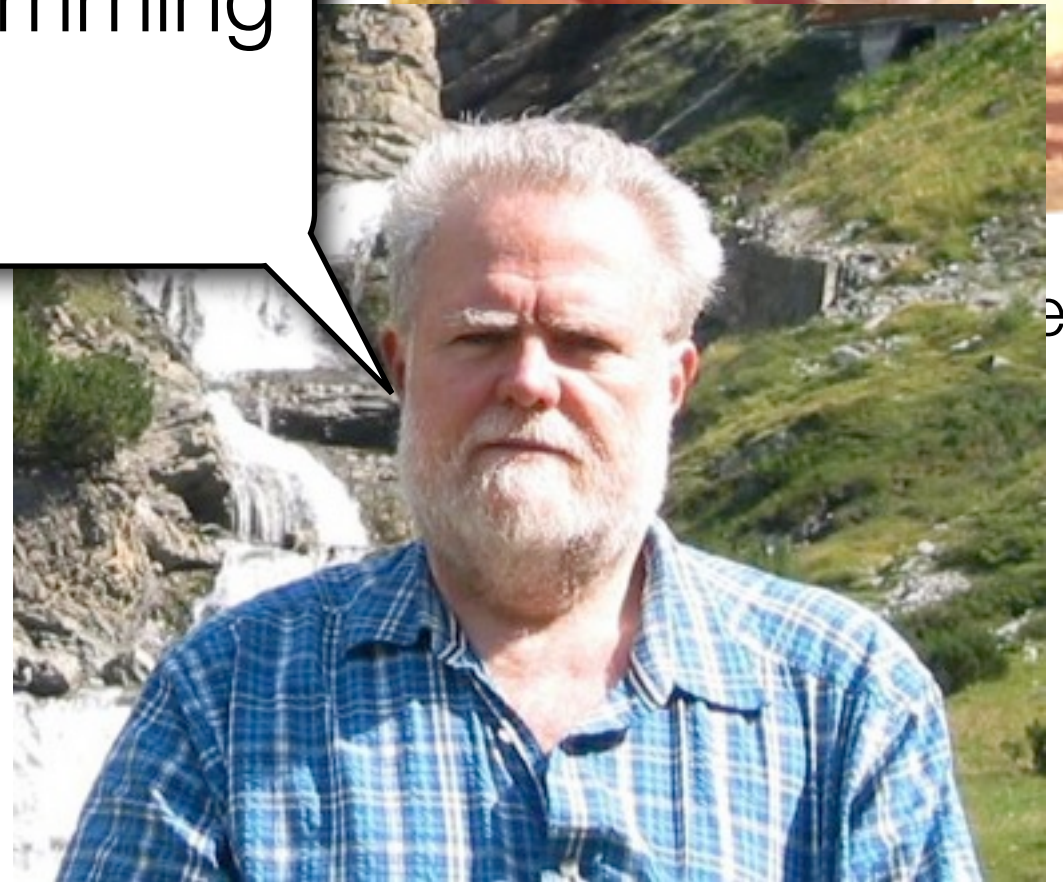
2nd year, Operating systems

1st year, Introduction to programming

Concurrent programming
is hard!

Concurrent programming

Concurrent programming
is hard!



1st year, Introduction to programming

4th year, Advanced programming languages

Concurrent programming
is hard!



DEA, Concurrency

Concurrent programming

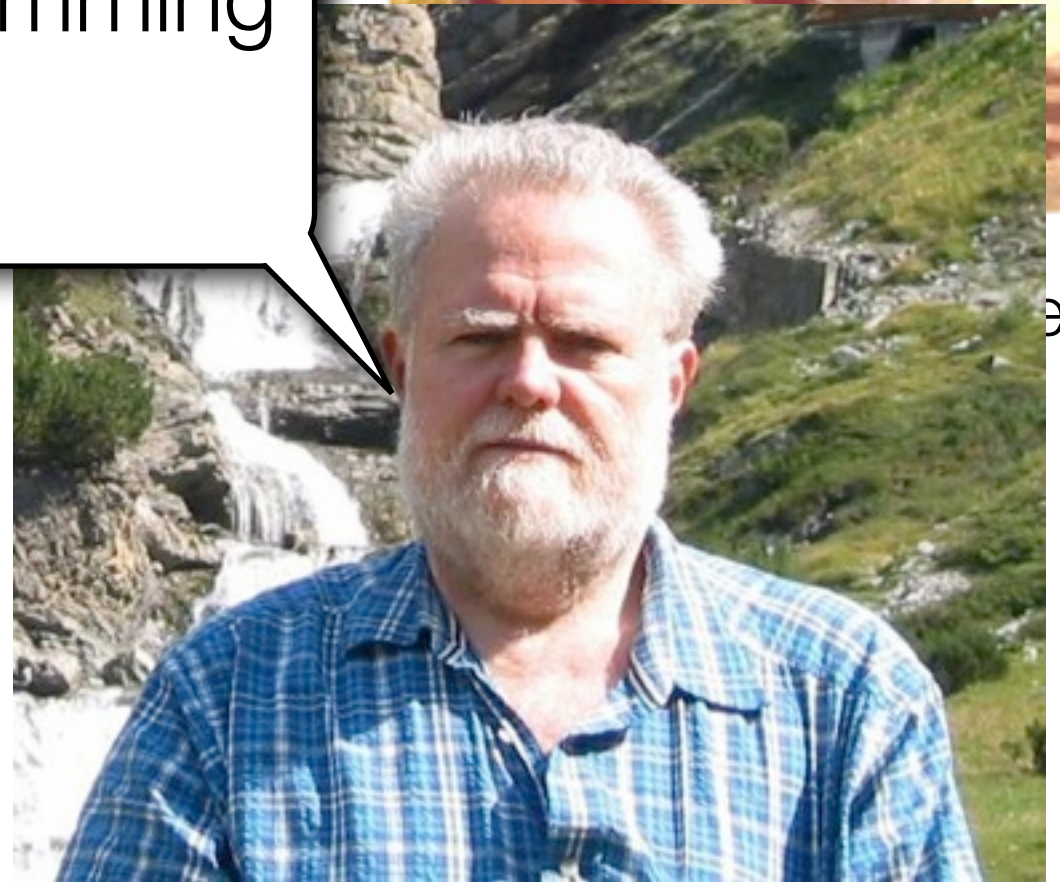
Concurrent programming
is hard!



1st year, Introduction to programming



ems



4th year, Advanced programming languages

Concurrent programming
is hard!

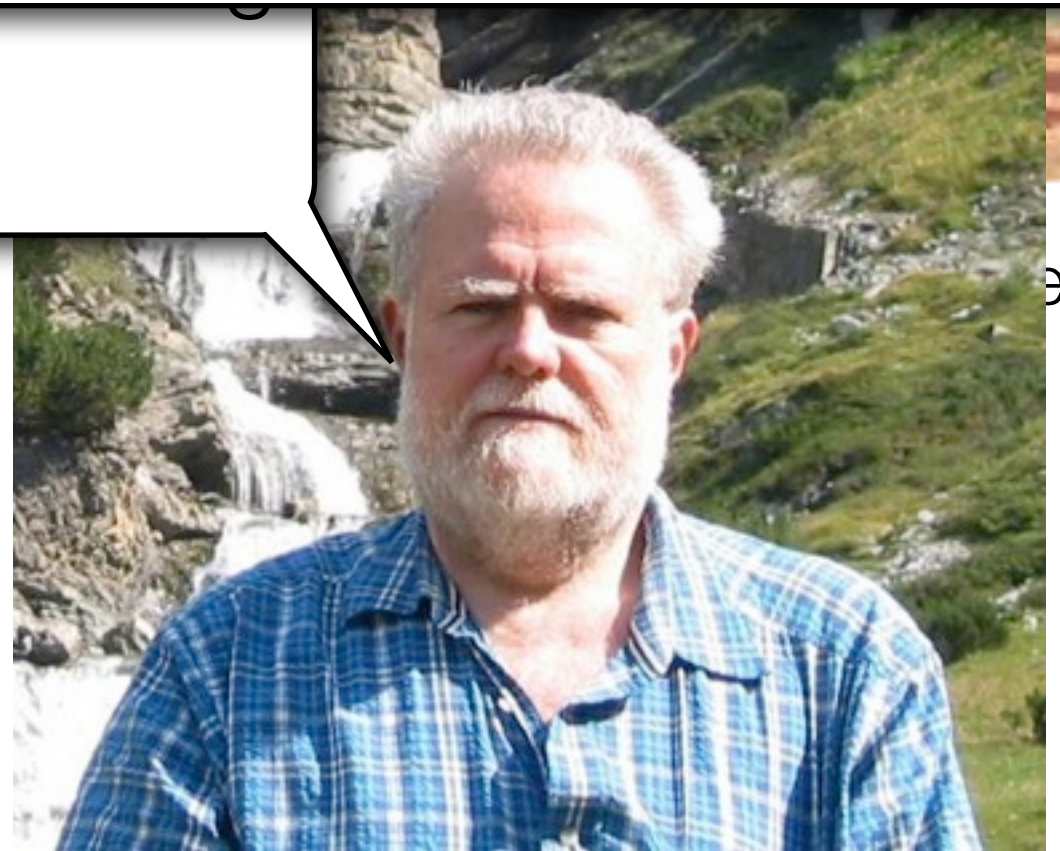
Concurrent programming is *even harder* than
what I was taught at university!

DLA, Concurrency

is hard!



1st year, Introduction to programming



4th year, Advanced programming languages

Concurrent programming
is hard!

Concurrent programming is *even harder* than
what I was taught at university!

We can't ignore it anymore:

we'll see that precise semantics, formal methods,
appropriate language design, clever algorithms,
are needed to put concurrent programming on solid basis.



Lunch now at ENS canteen
(follow Filippo)

x86: this afternoon



ARM/Power: tomorrow