

Racing To Win

Using Race Conditions to Build
Correct & Concurrent Software



Nathan Taylor | nathan.dijkstracula.net | @dijkstracula

All right, thanks so much for coming, everybody.

Racing To Win

Using Race Conditions to Build
Correct & Concurrent Software



Nathan Taylor | nathan.dijkstracula.net | @dijkstracula

All right, thanks so much for coming, everybody.



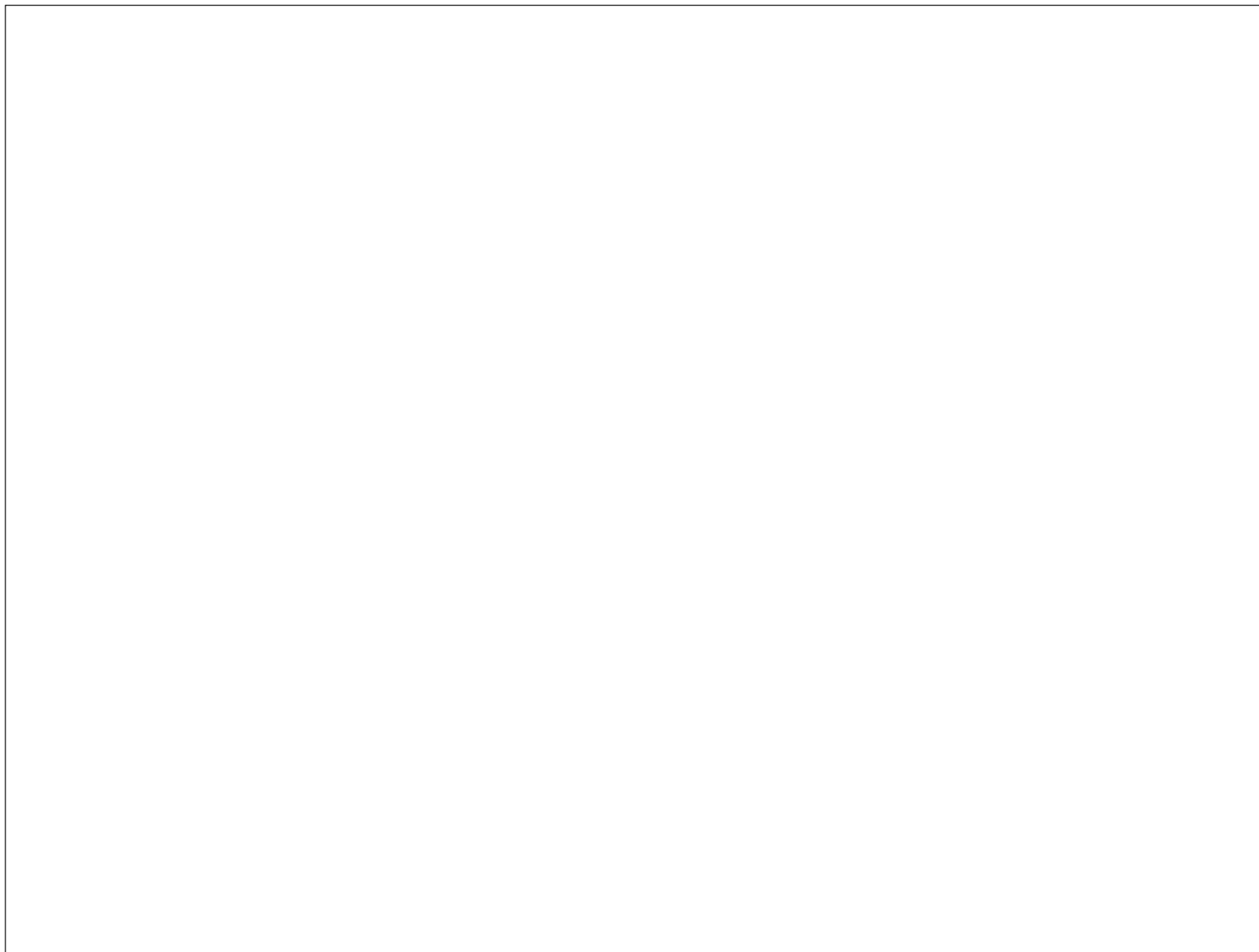
Hi, I'm **Nathan**.
(@dijkstracula)

My name is Nathan Taylor. As you may have guessed by now, I like bikes and old Atari computers. I'm a software developer based out of the bay area; as proof I offer a picture of me on the golden gate bridge. I've spent most of my career doing low-level systems work, at the hypervisor or OS or language runtime level, but right now I'm...



I'm an engineer at **fastly**.

working at a CDN called Fastly. The team that I'm on is responsible for the HTTP proxy that serves our customer's cached content to users, so all this layer 3 and layer 7 business is pretty far up the stack relative to where I usually am. Our software is a highly-concurrent multithreaded codebase. Given that, you can probably guess why a lot of the material I'm going to present is close to my heart.



This talk today is about <click> A problem that my team and I faced, <click> A thing we built that solves that problem <click> But it's really about the things we needed in our programming toolkit in order to implement that solution, given the problem requirements.

It's my hope that you'll be able to use these tools in *your* day-to-day when faced with similar challenges.

A problem

This talk today is about <click> A problem that my team and I faced, <click> A thing we built that solves that problem <click> But it's really about the things we needed in our programming toolkit in order to implement that solution, given the problem requirements.

It's my hope that you'll be able to use these tools in *your* day-to-day when faced with similar challenges.

A problem

A solution

This talk today is about <click> A problem that my team and I faced, <click> A thing we built that solves that problem <click> But it's really about the things we needed in our programming toolkit in order to implement that solution, given the problem requirements.

It's my hope that you'll be able to use these tools in *your* day-to-day when faced with similar challenges.

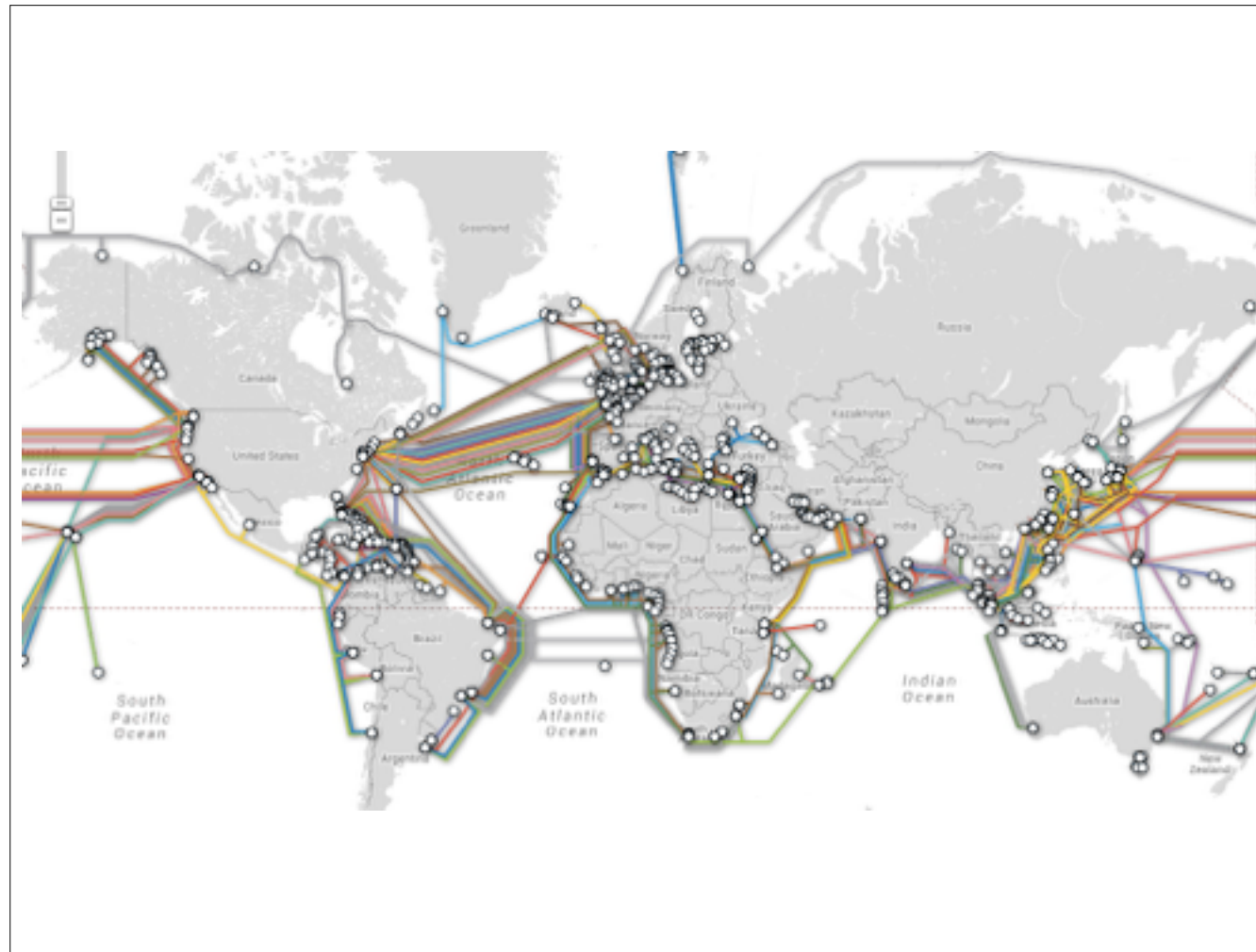
A problem



A solution

This talk today is about <click> A problem that my team and I faced, <click> A thing we built that solves that problem <click> But it's really about the things we needed in our programming toolkit in order to implement that solution, given the problem requirements.

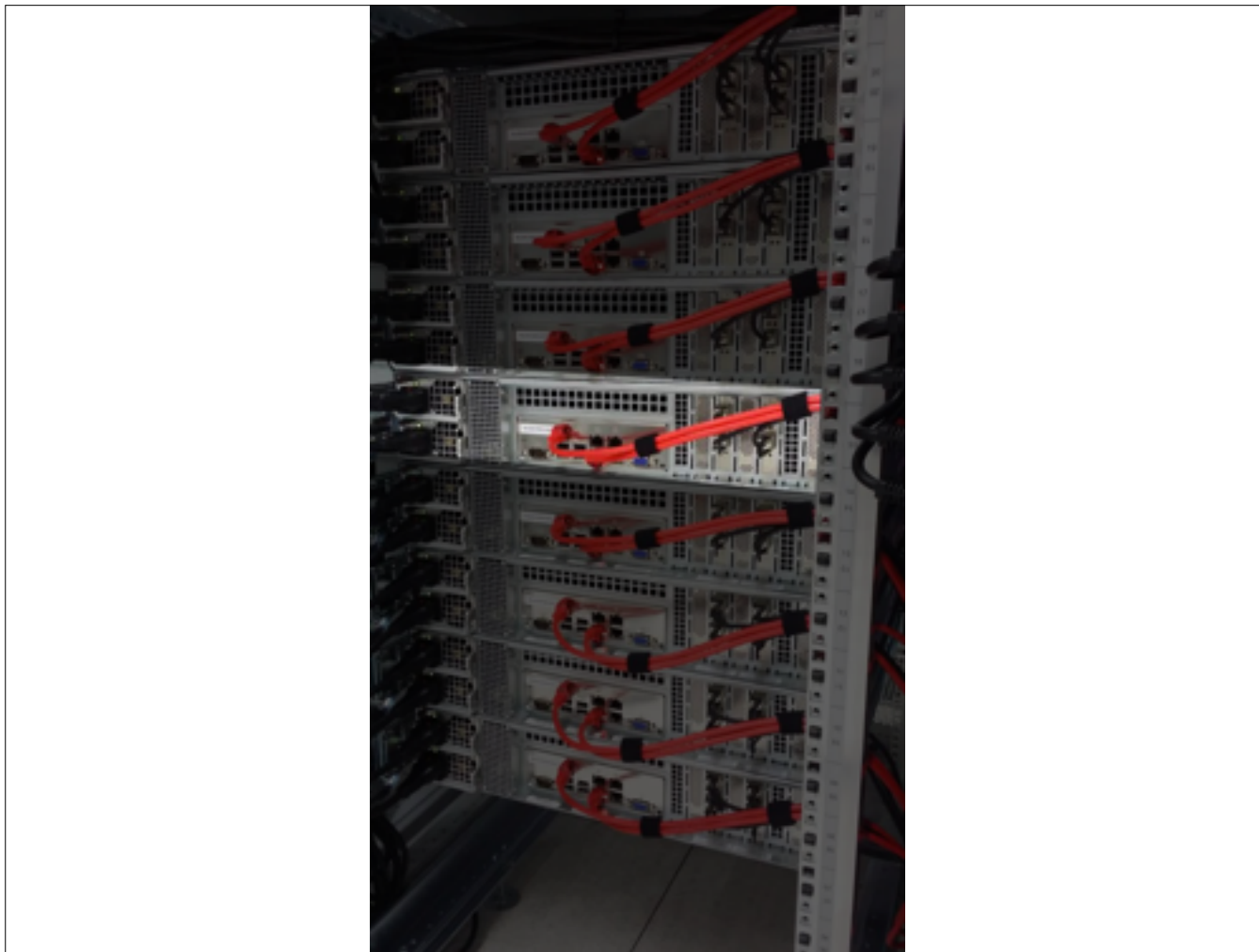
It's my hope that you'll be able to use these tools in *your* day-to-day when faced with similar challenges.



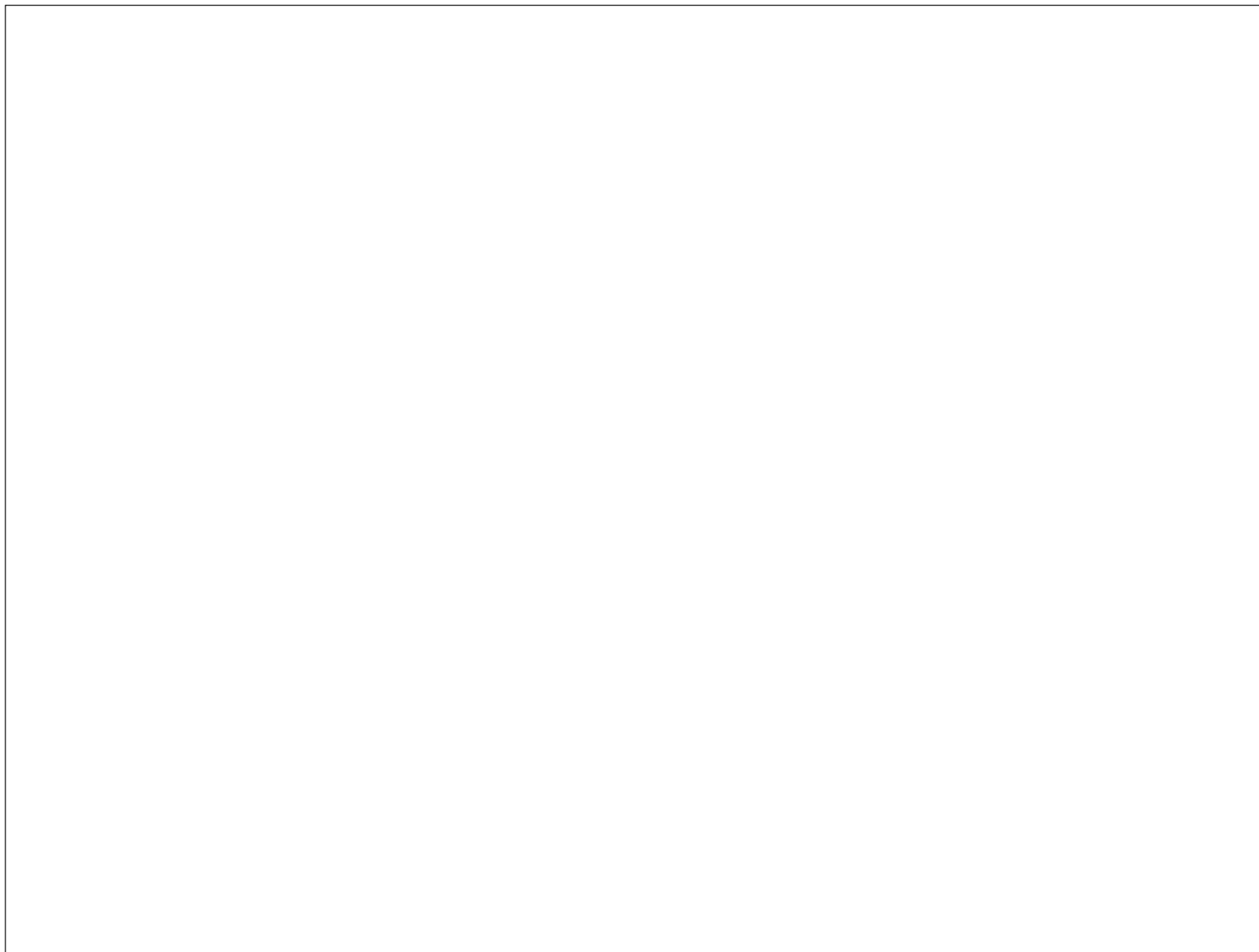
Often when folks from Fastly give presentations, the material is about, say, how a global network like ours is designed, or maybe it's about solving some hard distributed systems problem. For this talk, though, I'd like to zoom into



Not only within a Fastly server deployment, but

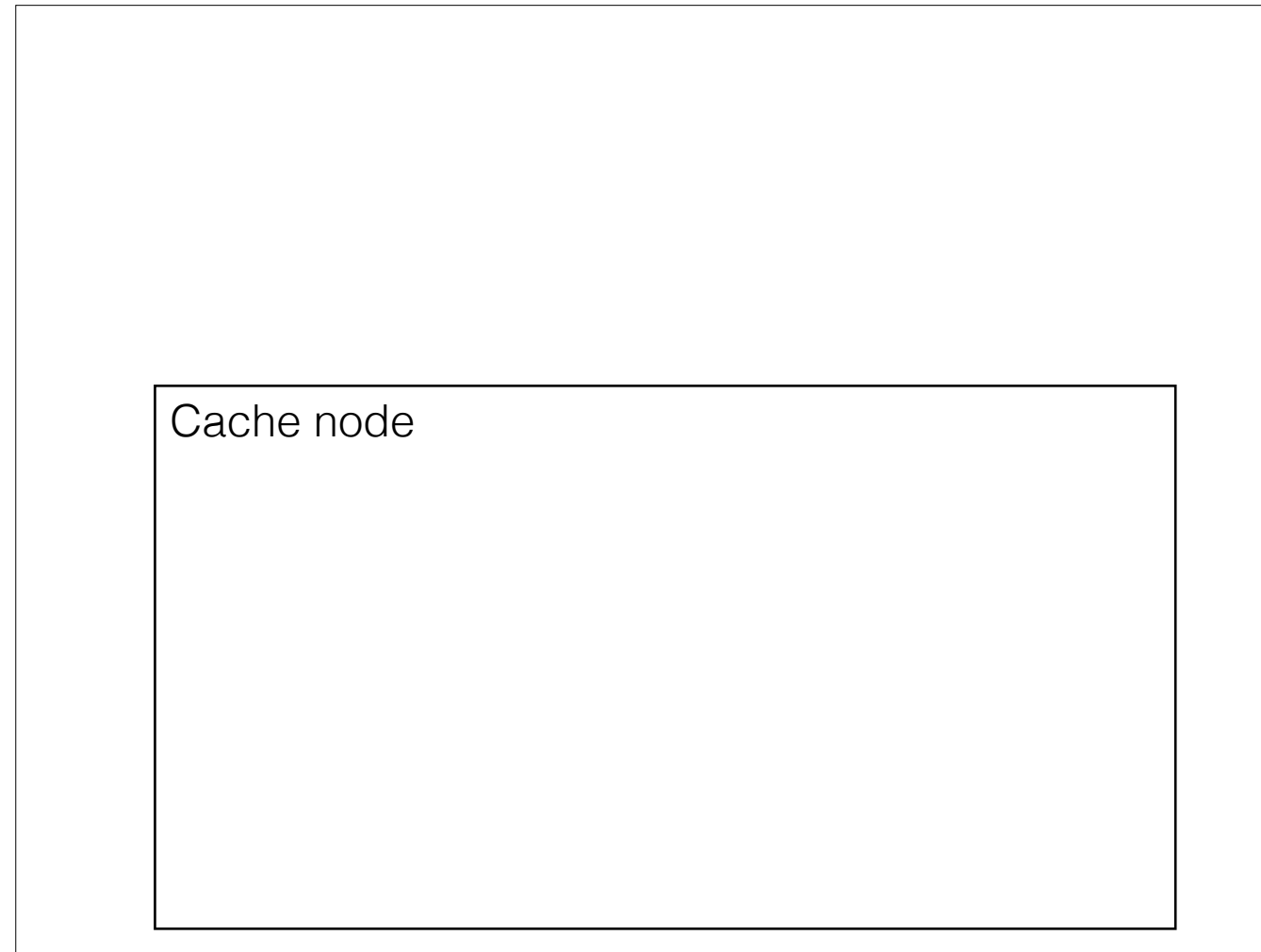


a single server. So, everything in this talk is in the context of a single piece of hardware.



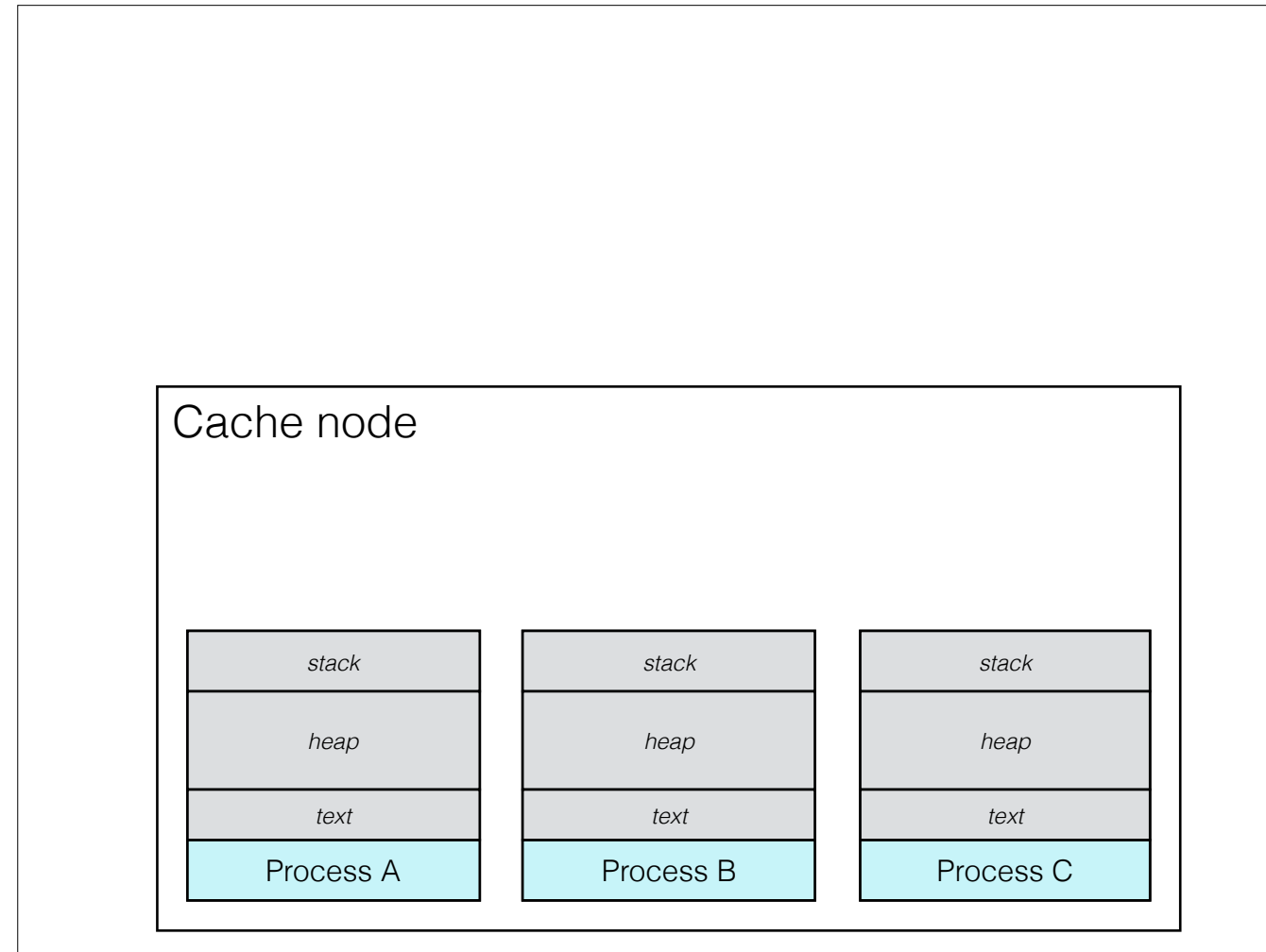
So given a <click> cache node, of course <click>, we have a collection of processes running atop it, and of course <click> each process can dynamically allocate and free memory within its address space.

The problem is that there's no general-purpose way in our current system for processes A, B, and C to allocate shared memory between themselves, such as, for instance, indices for cached content that lives on solid-state drives.



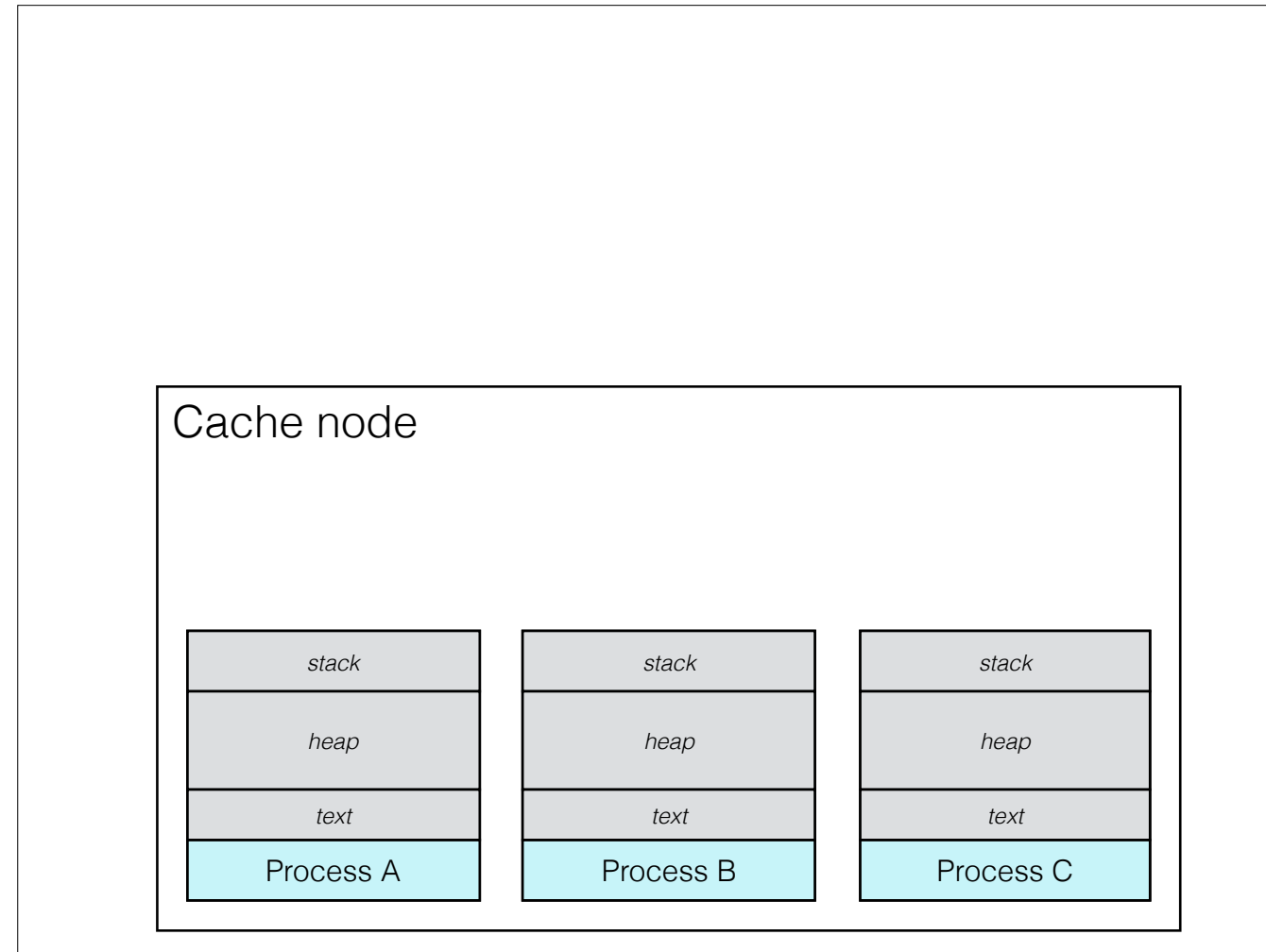
So given a <click> cache node, of course <click>, we have a collection of processes running atop it, and of course <click> each process can dynamically allocate and free memory within its address space.

The problem is that there's no general-purpose way in our current system for processes A, B, and C to allocate shared memory between themselves, such as, for instance, indices for cached content that lives on solid-state drives.



So given a <click> cache node, of course <click>, we have a collection of processes running atop it, and of course <click> each process can dynamically allocate and free memory within its address space.

The problem is that there’s no general-purpose way in our current system for processes A, B, and C to allocate shared memory between themselves, such as, for instance, indices for cached content that lives on solid-state drives.

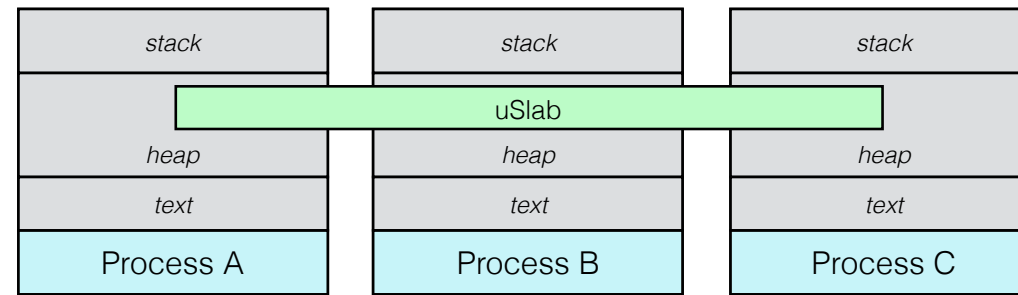


So given a <click> cache node, of course <click>, we have a collection of processes running atop it, and of course <click> each process can dynamically allocate and free memory within its address space.

The problem is that there’s no general-purpose way in our current system for processes A, B, and C to allocate shared memory between themselves, such as, for instance, indices for cached content that lives on solid-state drives.

A Persistent, Shared-State Memory Allocator

Cache node

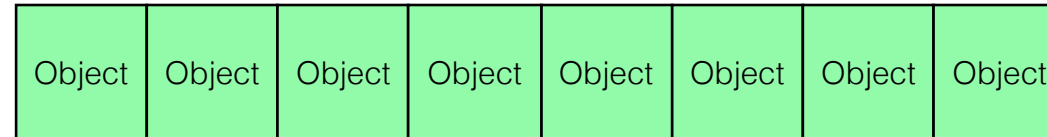


So, the thing we implemented was a persistent, shared-state memory allocator that processes can allocate from alongside their private heaps. How we did this in a scalable and correct way will be the running example through this talk.

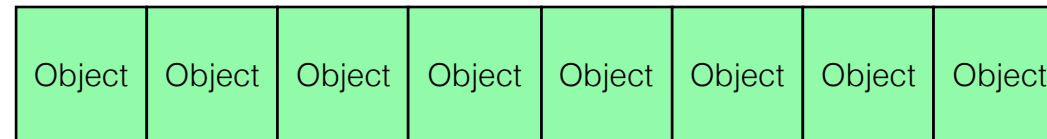
Slab allocation

The kind of allocator we implemented is called a Slab allocator. A slab is [a pool of shared memory](#), and the allocator returns fixed-sized objects from the pool.

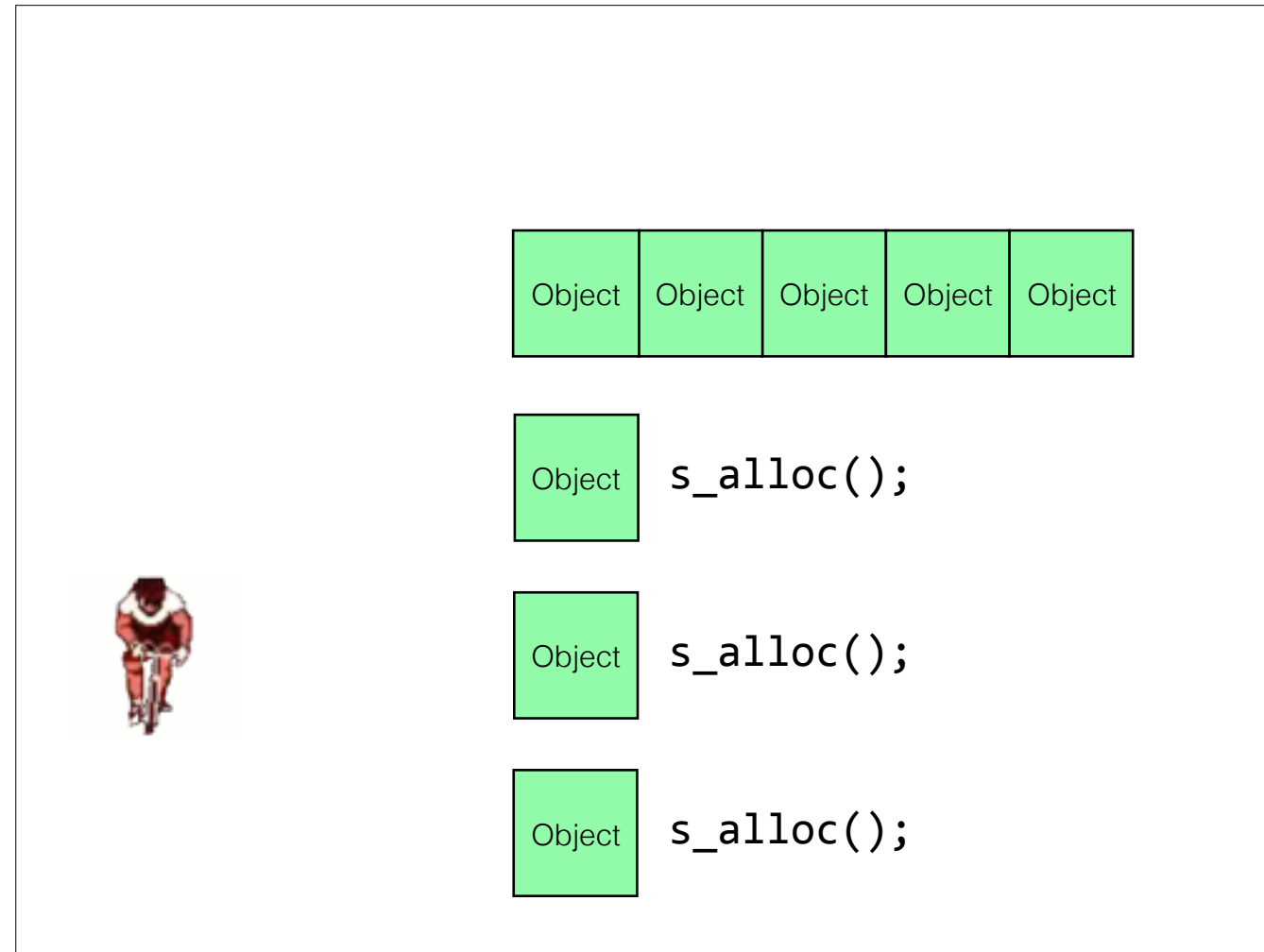
Slab allocation



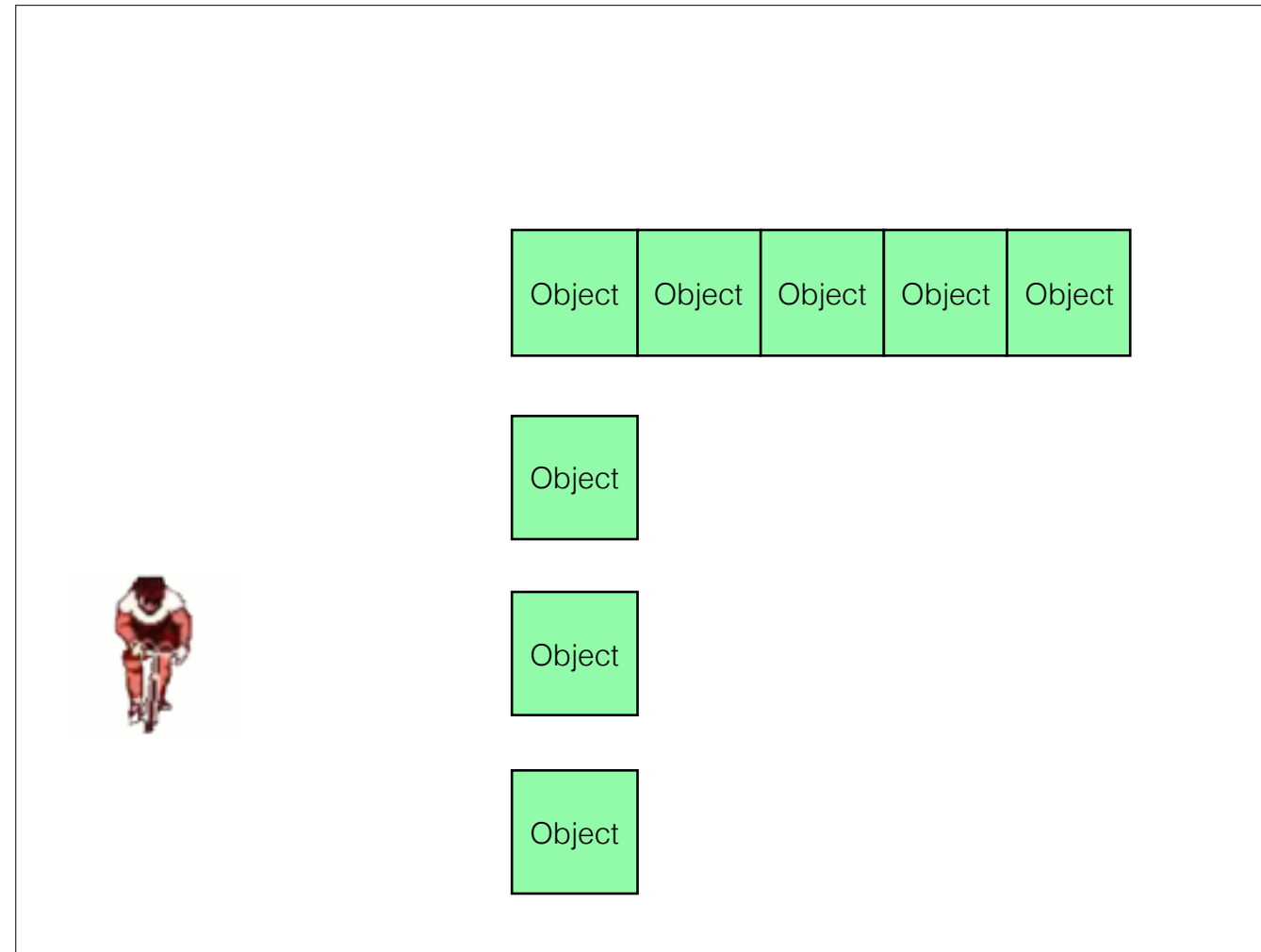
The kind of allocator we implemented is called a Slab allocator. A slab is [a pool of shared memory](#), and the allocator returns fixed-sized objects from the pool.



If we're one of those processes, <click> we can allocate an object by pulling some memory off a slab and returning it to the calling process,

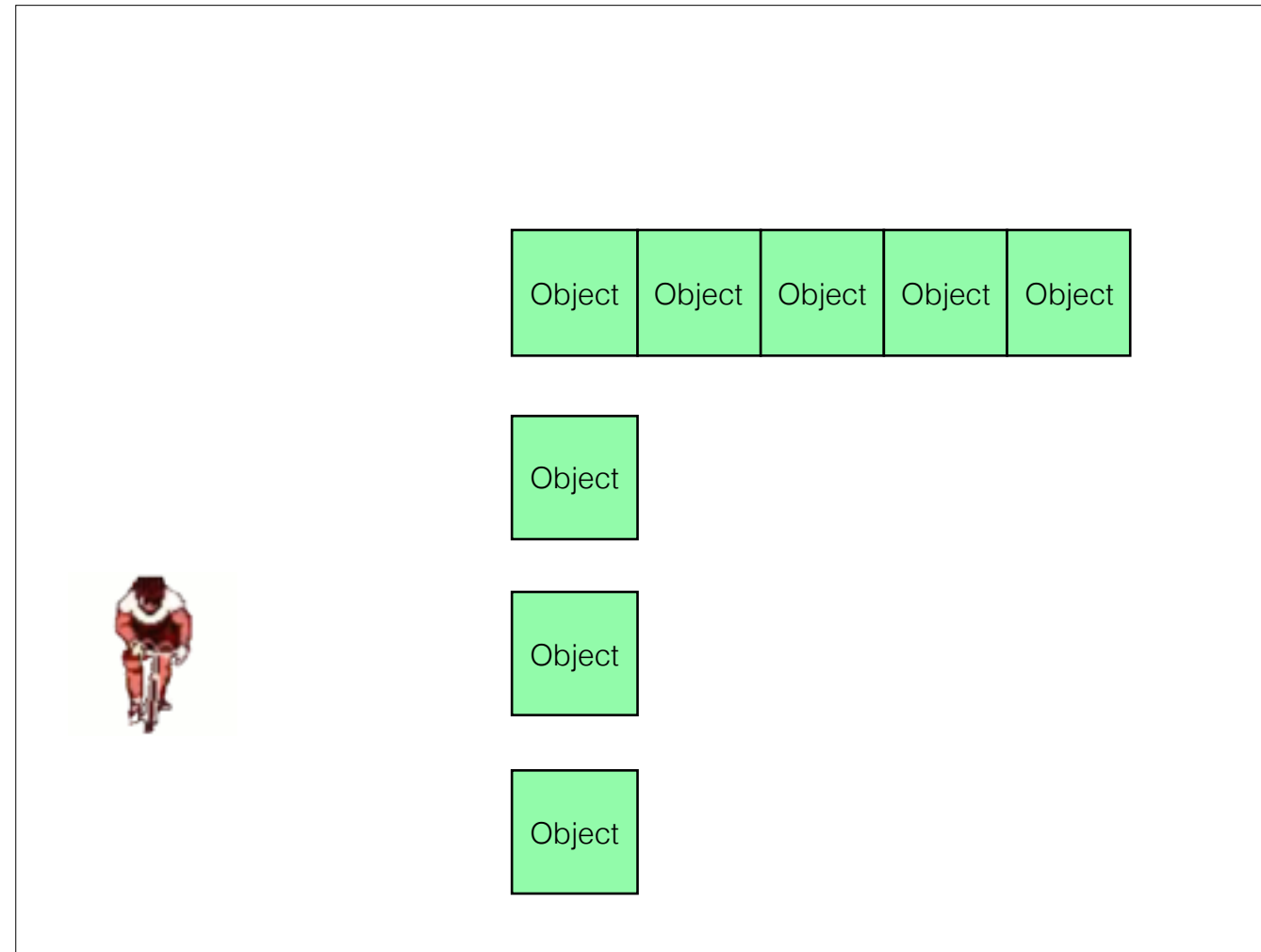


If we're one of those processes, <click> we can allocate an object by pulling some memory off a slab and returning it to the calling process,



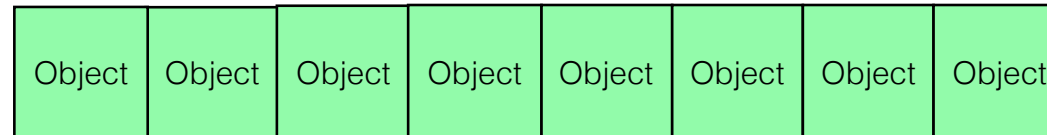
And conversely, we can free memory back to the allocator by, <click> conceptually, pushing objects back onto our slab.

The notion of a slab consisting of a sequence of unallocated elements is called a freelist, and as you may have noticed it's basically behaving exactly as a stack or other FILO data structure would.



And conversely, we can free memory back to the allocator by, <click> conceptually, pushing objects back onto our slab.

The notion of a slab consisting of a sequence of unallocated elements is called a freelist, and as you may have noticed it's basically behaving exactly as a stack or other FILO data structure would.



`s_free();`

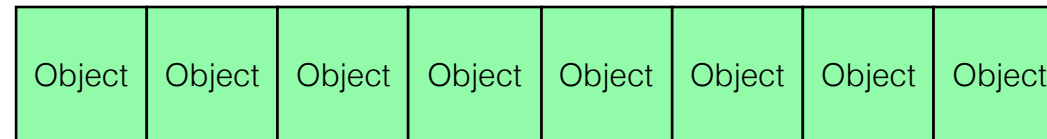


`s_free();`

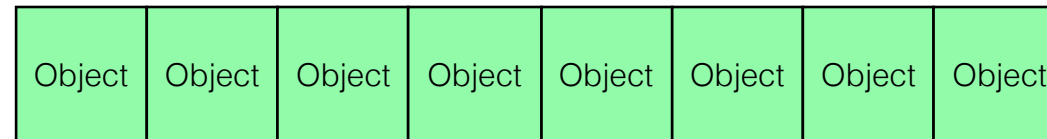
`s_free();`

And conversely, we can free memory back to the allocator by, <click> conceptually, pushing objects back onto our slab.

The notion of a slab consisting of a sequence of unallocated elements is called a freelist, and as you may have noticed it's basically behaving exactly as a stack or other FILO data structure would.



So far we've looked at a single process. What happens if we add some concurrency into the mix? What is the right way to reason about the allocator's behaviour in the presence of simultaneous, concurrent access to the freelist?



So far we've looked at a single process. What happens if we add some concurrency into the mix? What is the right way to reason about the allocator's behaviour in the presence of simultaneous, concurrent access to the freelist?

Allocation Protocol

- An **request to allocate** is followed by a response containing an object
 - A **request to free** is followed by a response after the supplied object has been released
-
- Allocation requests must not respond with an already-allocated object
 - A free request must not release an already-unallocated object

It's useful to talk about the behaviour of the allocator in terms of a protocol that consumes requests and produces responses. Our allocation protocol would have the two operations that we expect: object allocation and object freeing. I think we all agree that intuitively a response should come after a request but I'm going to go ahead and make that clear here. It's also useful to be able to say that the same object can't be allocated twice, nor can it be freed twice.

An Execution History

Once we have this protocol in place, we can start to talk about how multiple processes can make requests and receive responses over this protocol, by way of something called execution histories.

An execution history defines a ordering of requests and responses of some protocol over time. <click> So if a thread allocates and then frees an object like so, then <click> our execution history would consist of an allocation request and response, followed by a free request and response.

An Execution History

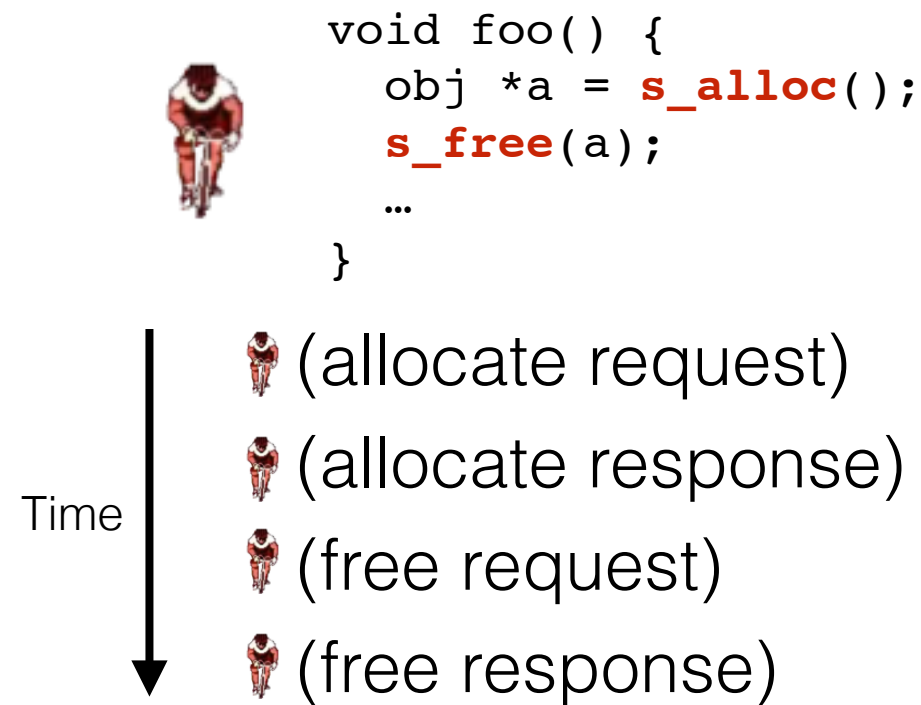


```
void foo() {  
    obj *a = s_alloc();  
    s_free(a);  
    ...  
}
```

Once we have this protocol in place, we can start to talk about how multiple processes can make requests and receive responses over this protocol, by way of something called execution histories.

An execution history defines a ordering of requests and responses of some protocol over time. <click> So if a thread allocates and then frees an object like so, then <click> our execution history would consist of an allocation request and response, followed by a free request and response.

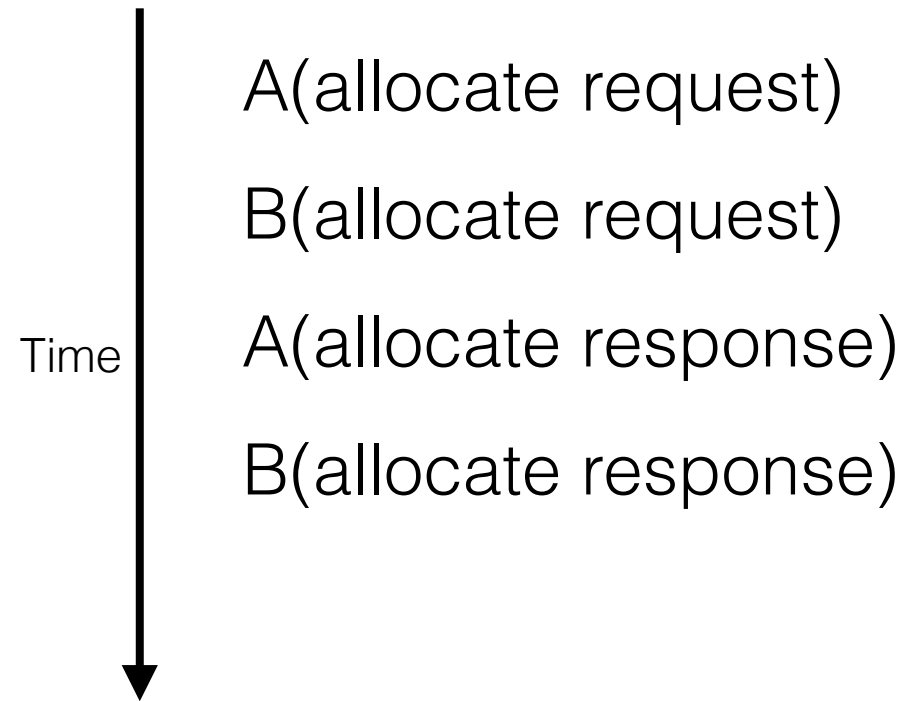
An Execution History



Once we have this protocol in place, we can start to talk about how multiple processes can make requests and receive responses over this protocol, by way of something called execution histories.

An execution history defines a ordering of requests and responses of some protocol over time. <click> So if a thread allocates and then frees an object like so, then <click> our execution history would consist of an allocation request and response, followed by a free request and response.

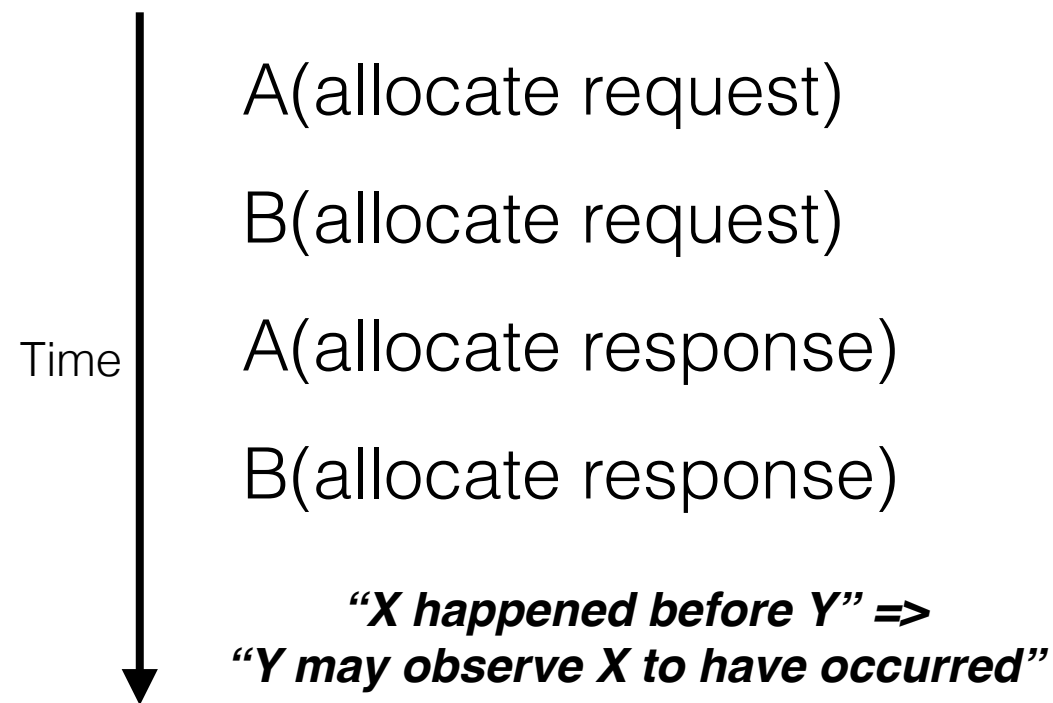
An Execution History



Execution histories become more interesting when more than one thread is in the mix. So in this example, thread A issued an allocation request before B, and it just so happens that A also received an allocation response before B did.

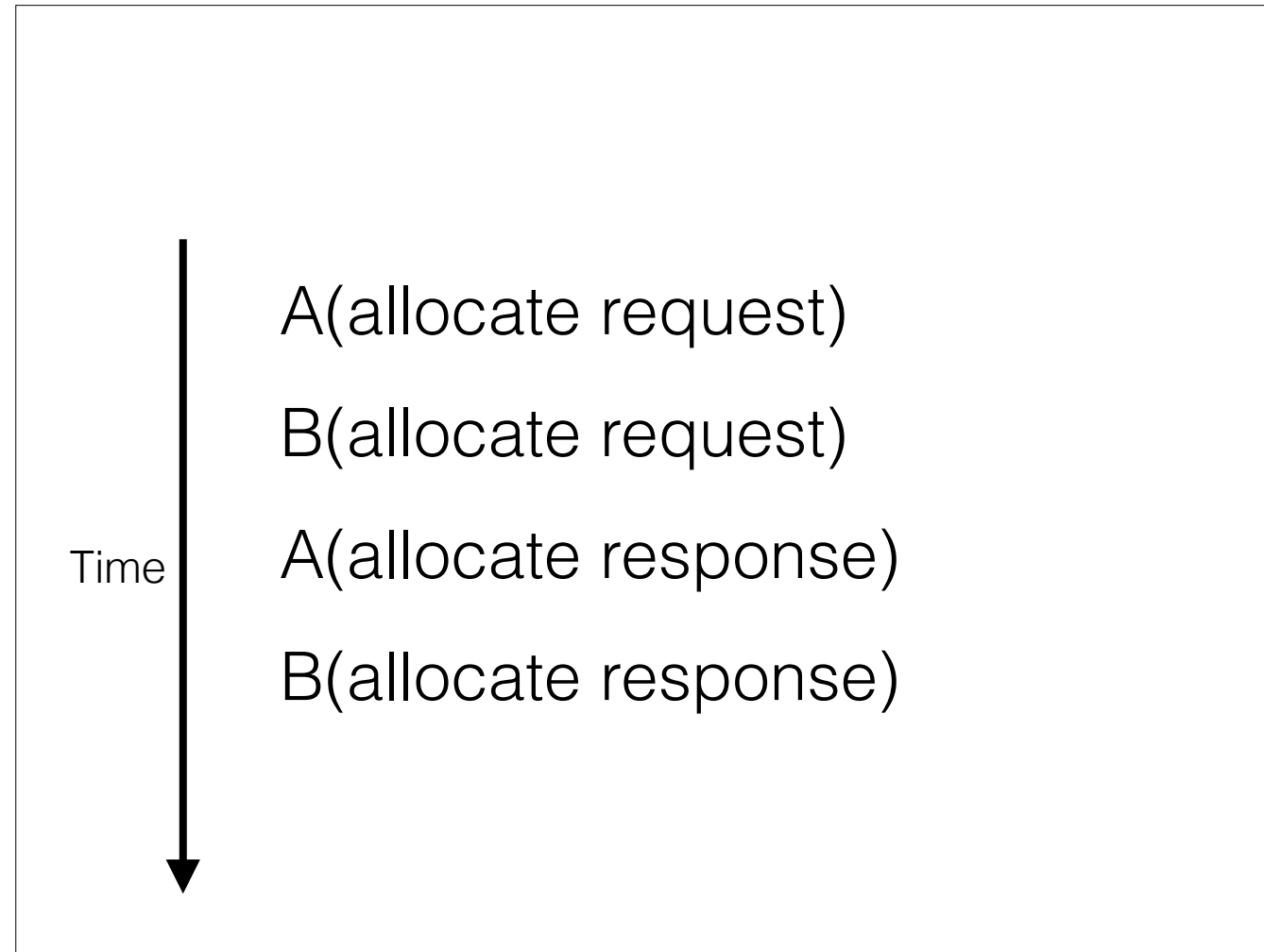
The notion of visibility is also crucial in an execution history that juggles multiple threads. The Y-axis implies some sort of “X happened before Y” ordering, but because all these operations are with respect to a shared data structure, <click> Y should be able to notice that X took effect. So in this case, B’s allocation response is dependent on the fact that A’s allocation response preceded it.

An Execution History

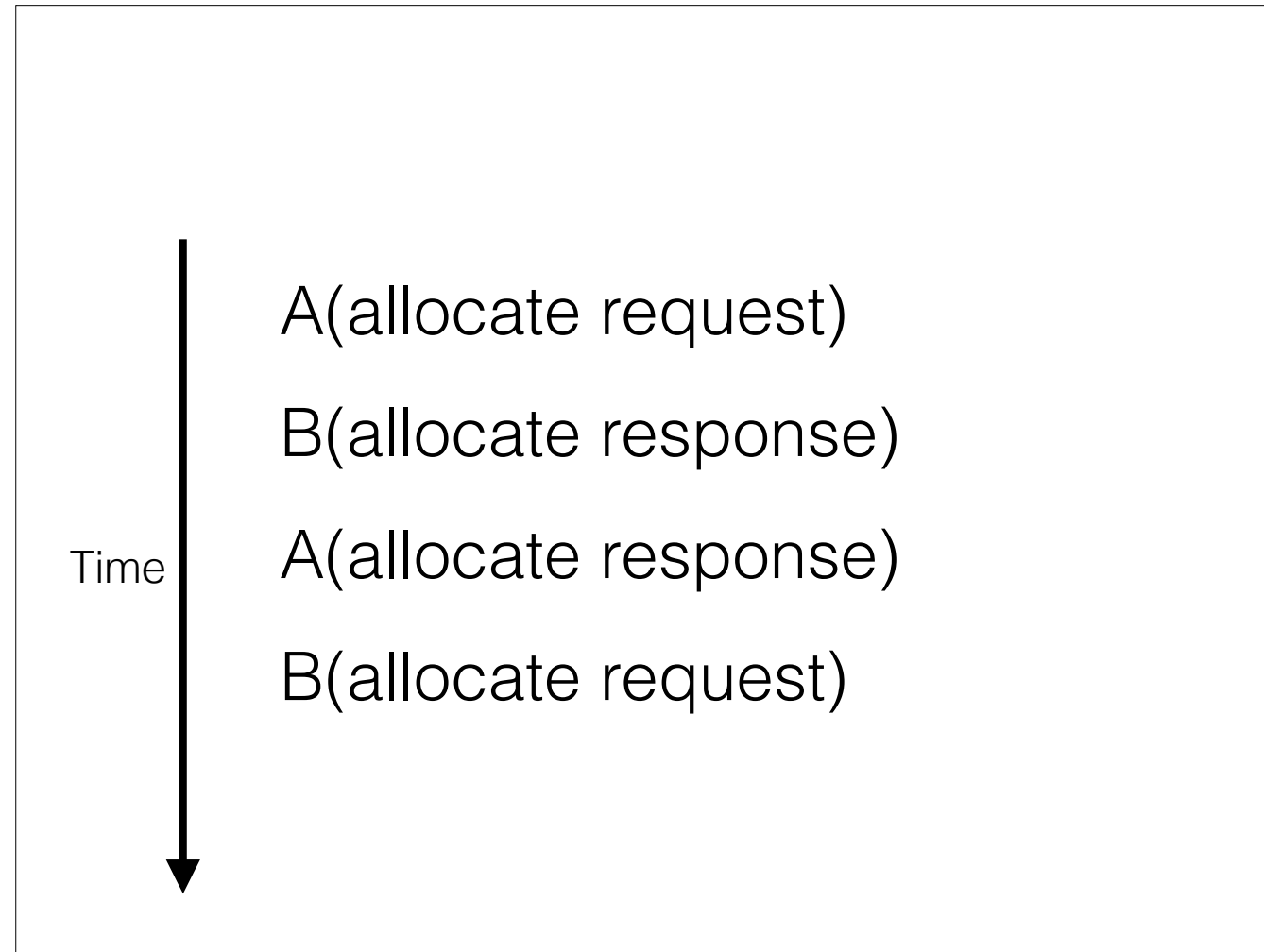


Execution histories become more interesting when more than one thread is in the mix. So in this example, thread A issued an allocation request before B, and it just so happens that A also received an allocation response before B did.

The notion of visibility is also crucial in an execution history that juggles multiple threads. The Y-axis implies some sort of "X happened before Y" ordering, but because all these operations are with respect to a shared data structure, <click> Y should be able to notice that X took effect. So in this case, B's allocation response is dependent on the fact that A's allocation response preceded it.

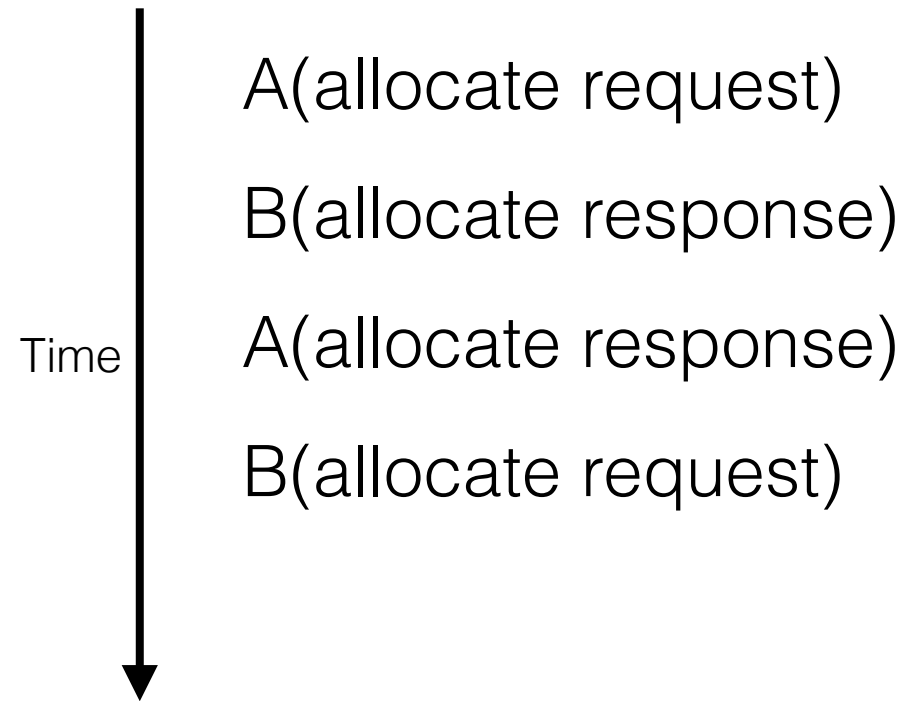


Hopefully it's not too hard to see that these four events in our execution history can be permuted in $4!$, 24, ways. However, the protocol we defined puts constraints on what constitutes a *valid* execution history with respect to our allocator. So <click> for instance we can't have a thread receive an allocation response before it asked for it, because that breaks causality and is mind bending. <click>. So we say that this has violated our protocol. <click> Let's put it back.

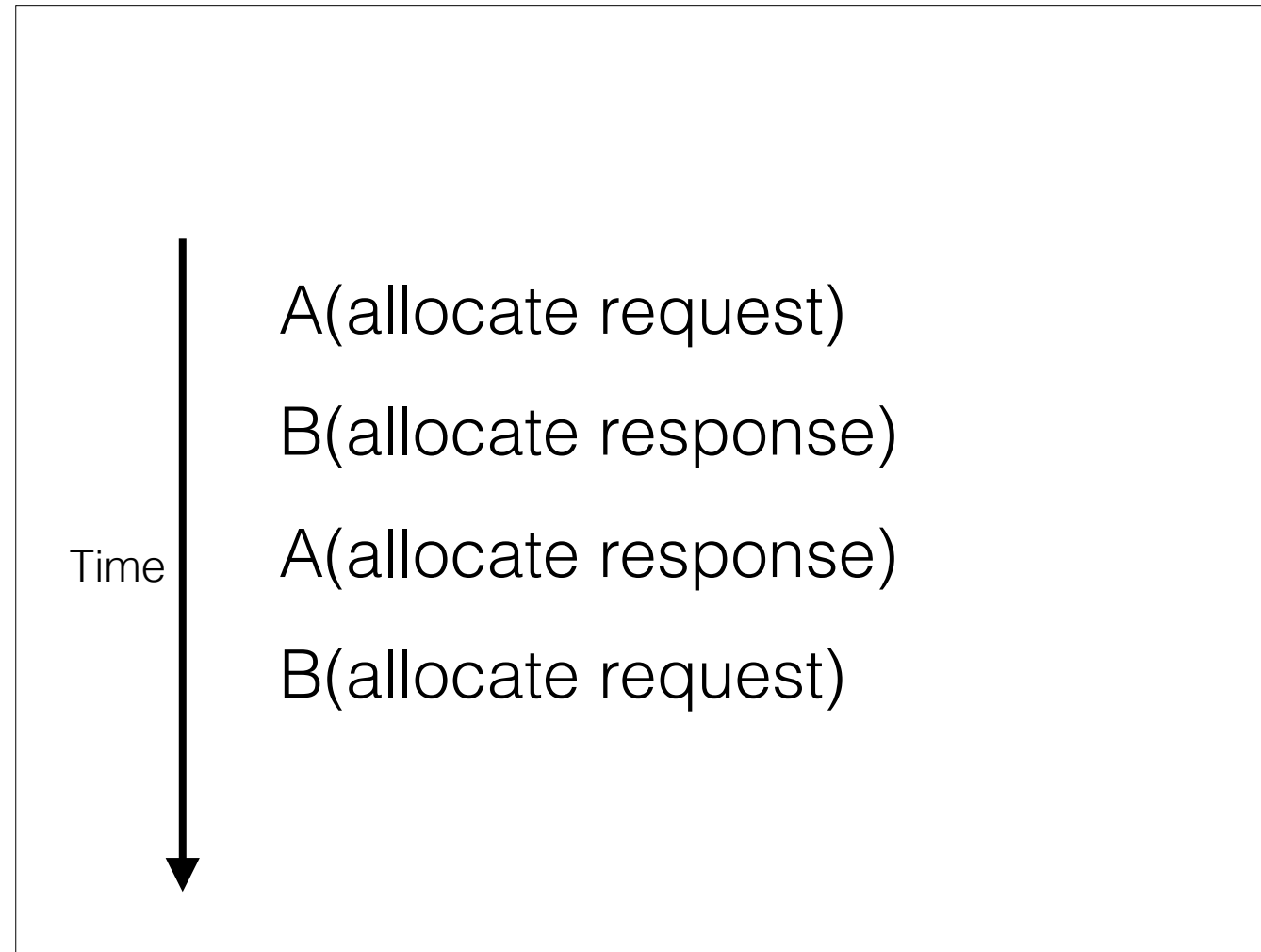


Hopefully it's not too hard to see that these four events in our execution history can be permuted in $4!$, 24, ways. However, the protocol we defined puts constraints on what constitutes a *valid* execution history with respect to our allocator. So <click> for instance we can't have a thread receive an allocation response before it asked for it, because that breaks causality and is mind bending. <click>. So we say that this has violated our protocol. <click> Let's put it back.

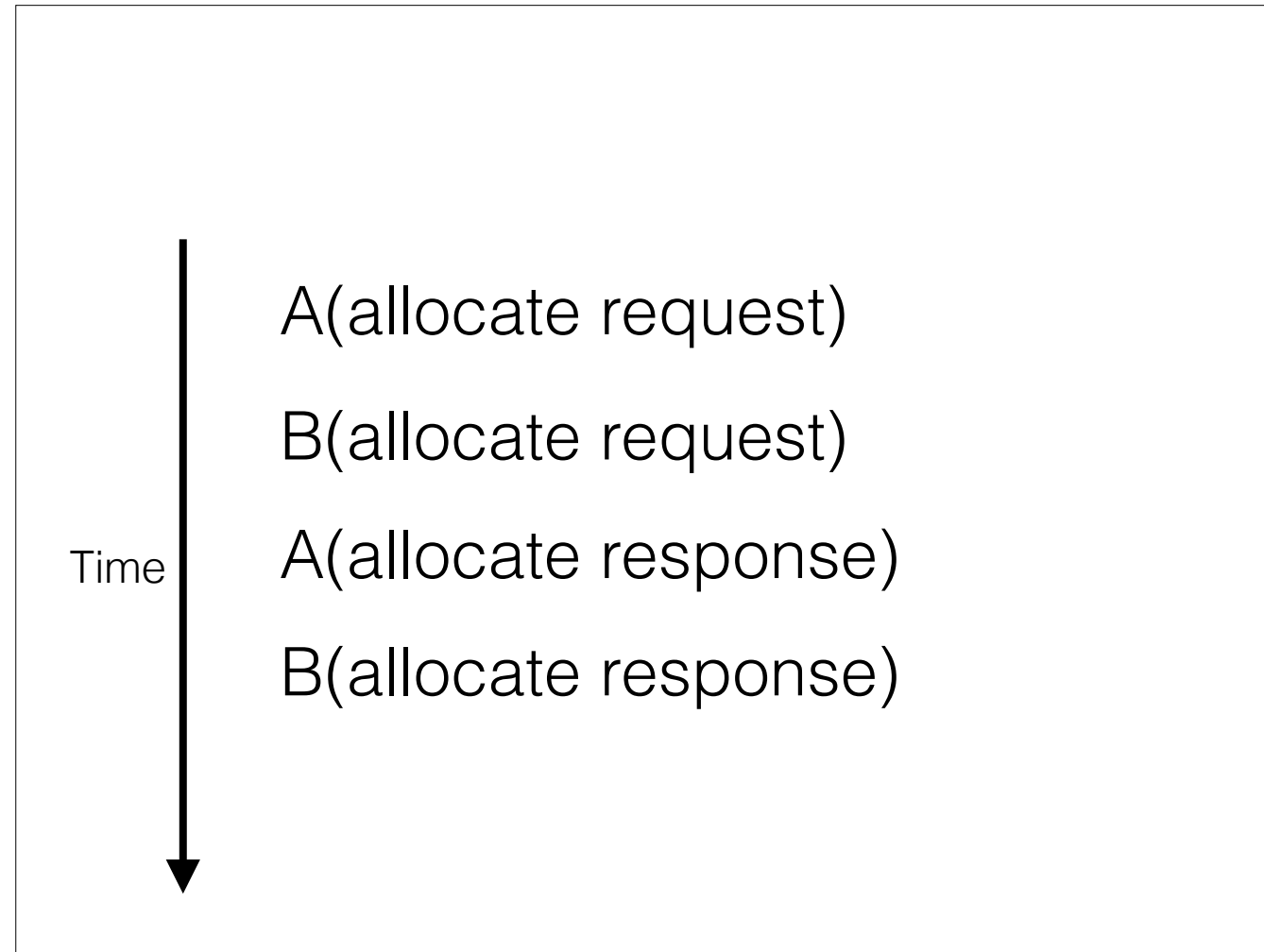
A protocol violation!



Hopefully it's not too hard to see that these four events in our execution history can be permuted in $4!$, 24, ways. However, the protocol we defined puts constraints on what constitutes a *valid* execution history with respect to our allocator. So <click> for instance we can't have a thread receive an allocation response before it asked for it, because that breaks causality and is mind bending. <click>. So we say that this has violated our protocol. <click> Let's put it back.



By putting constraints on our execution histories we're able to better reason about the correctness of our system. Now, let's think about ways that we can further constrain execution histories to ones where we can convince ourselves that our concurrent allocator will work correctly.



By putting constraints on our execution histories we're able to better reason about the correctness of our system. Now, let's think about ways that we can further constrain execution histories to ones where we can convince ourselves that our concurrent allocator will work correctly.

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

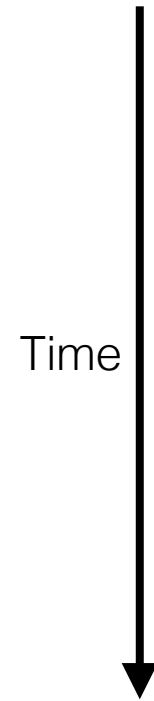
A history H is *sequential* if:

- (1) The first event of H is an invocation.
- (2) Each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by a matching invocation.

<http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf>

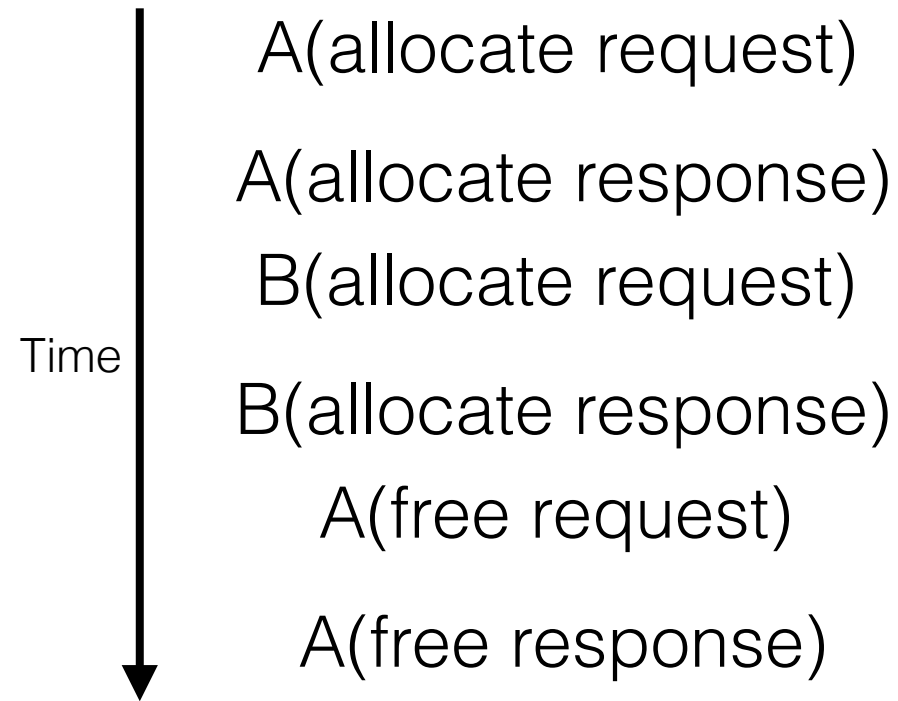
A sequential history is an execution history where any request is succeeded immediately by a response.

A Sequential History



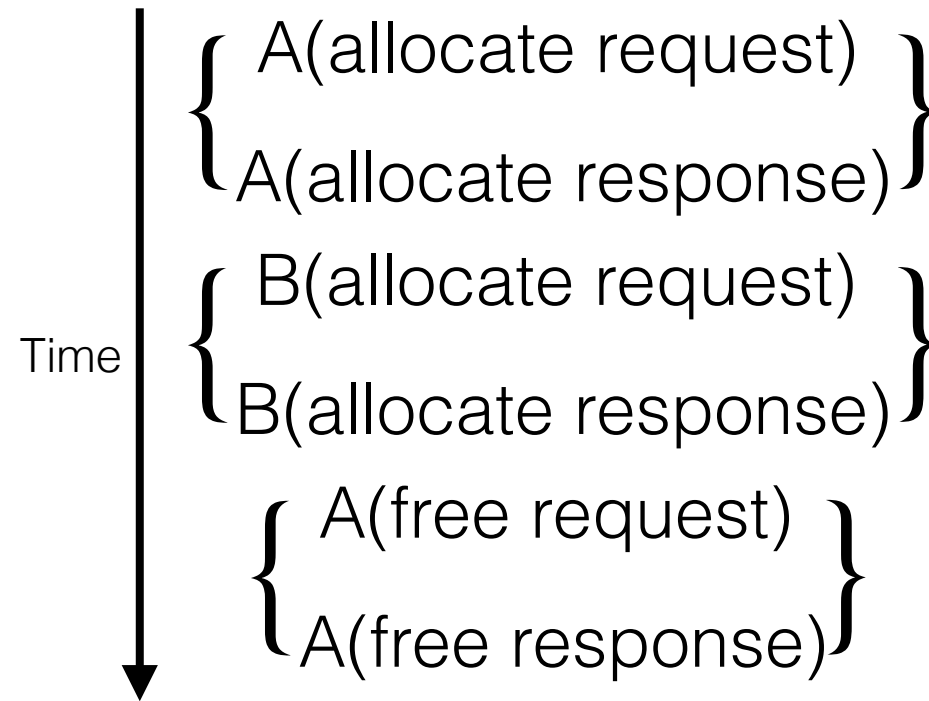
And here's an example of a sequential history. Every request is immediately followed by its response, which is <click> maybe easier to see if I group them like this. Our previous execution history is **not** a sequential history because the requests and responses were interleaved. So what this means for a sequentially-consistent allocator is that no thread can begin an allocation or a free until a previous thread's operation has succeeded.

A Sequential History



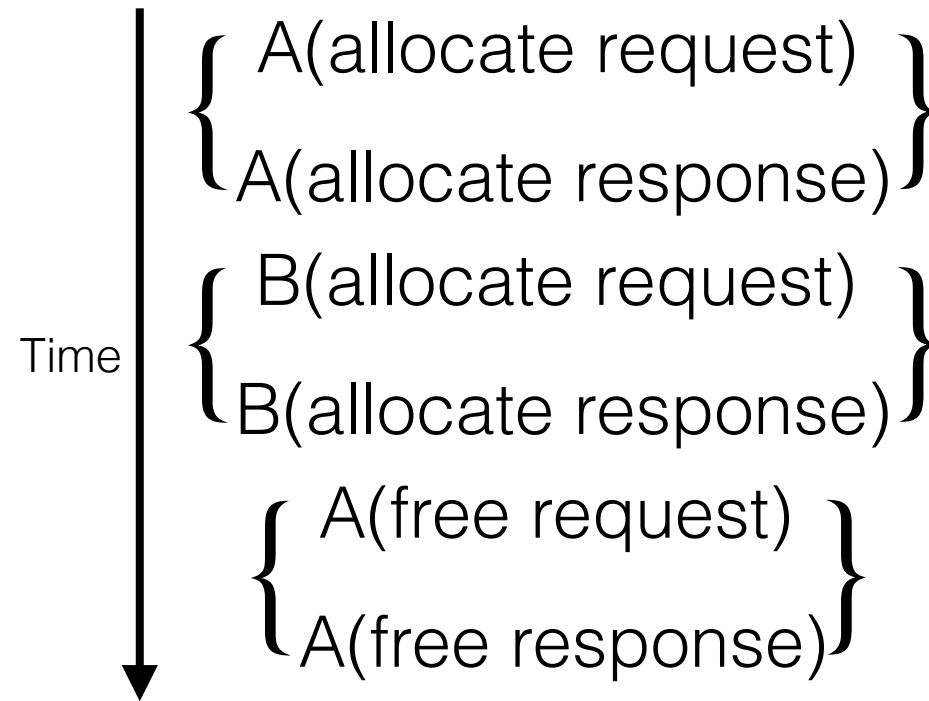
And here's an example of a sequential history. Every request is immediately followed by its response, which is <click> maybe easier to see if I group them like this. Our previous execution history is **not** a sequential history because the requests and responses were interleaved. So what this means for a sequentially-consistent allocator is that no thread can begin an allocation or a free until a previous thread's operation has succeeded.

A Sequential History



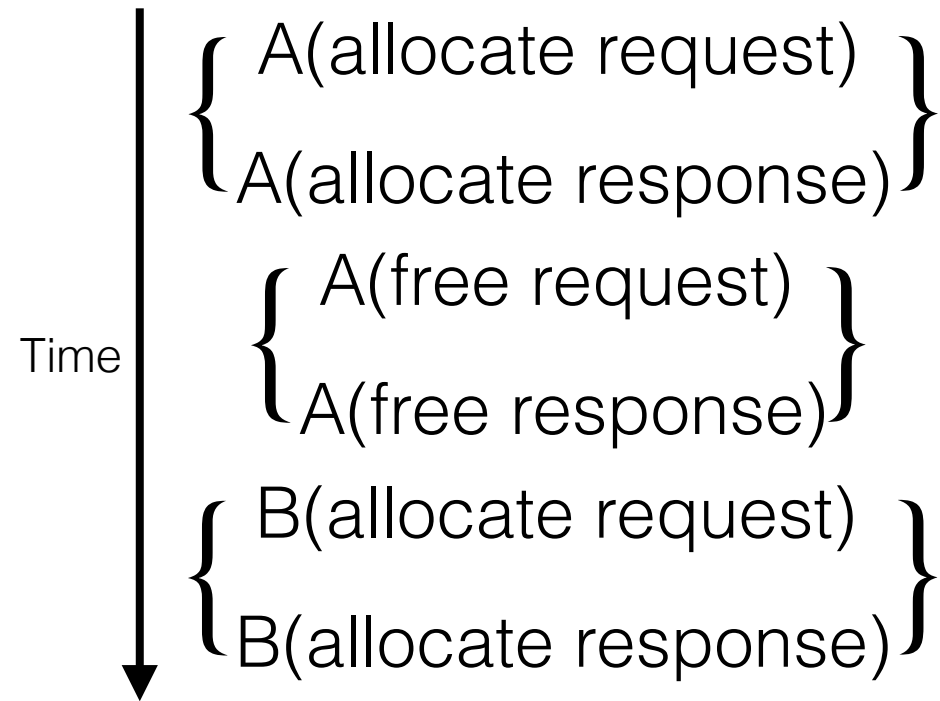
And here's an example of a sequential history. Every request is immediately followed by its response, which is <click> maybe easier to see if I group them like this. Our previous execution history is **not** a sequential history because the requests and responses were interleaved. So what this means for a sequentially-consistent allocator is that no thread can begin an allocation or a free until a previous thread's operation has succeeded.

A Sequential History



Notice that while we can't interleave requests and responses, <click> reordering *request/response pairs* still yields a valid sequential history. The notion of "what are we allowed to reorder and consider valid" is about to become super-critical so keep this in mind.

A Sequential History



Notice that while we can't interleave requests and responses, <click> reordering *request/response pairs* still yields a valid sequential history. The notion of "what are we allowed to reorder and consider valid" is about to become super-critical so keep this in mind.

```
obj *allocate(slab *s) {  
  
    obj *a = s->head;  
    if (a == NULL) return NULL;  
    s->head = a->next;  
  
    return a;  
}  
  
void free(slab *s, obj *o) {  
  
    o->next = s->head;  
    s->head = o;  
  
}
```

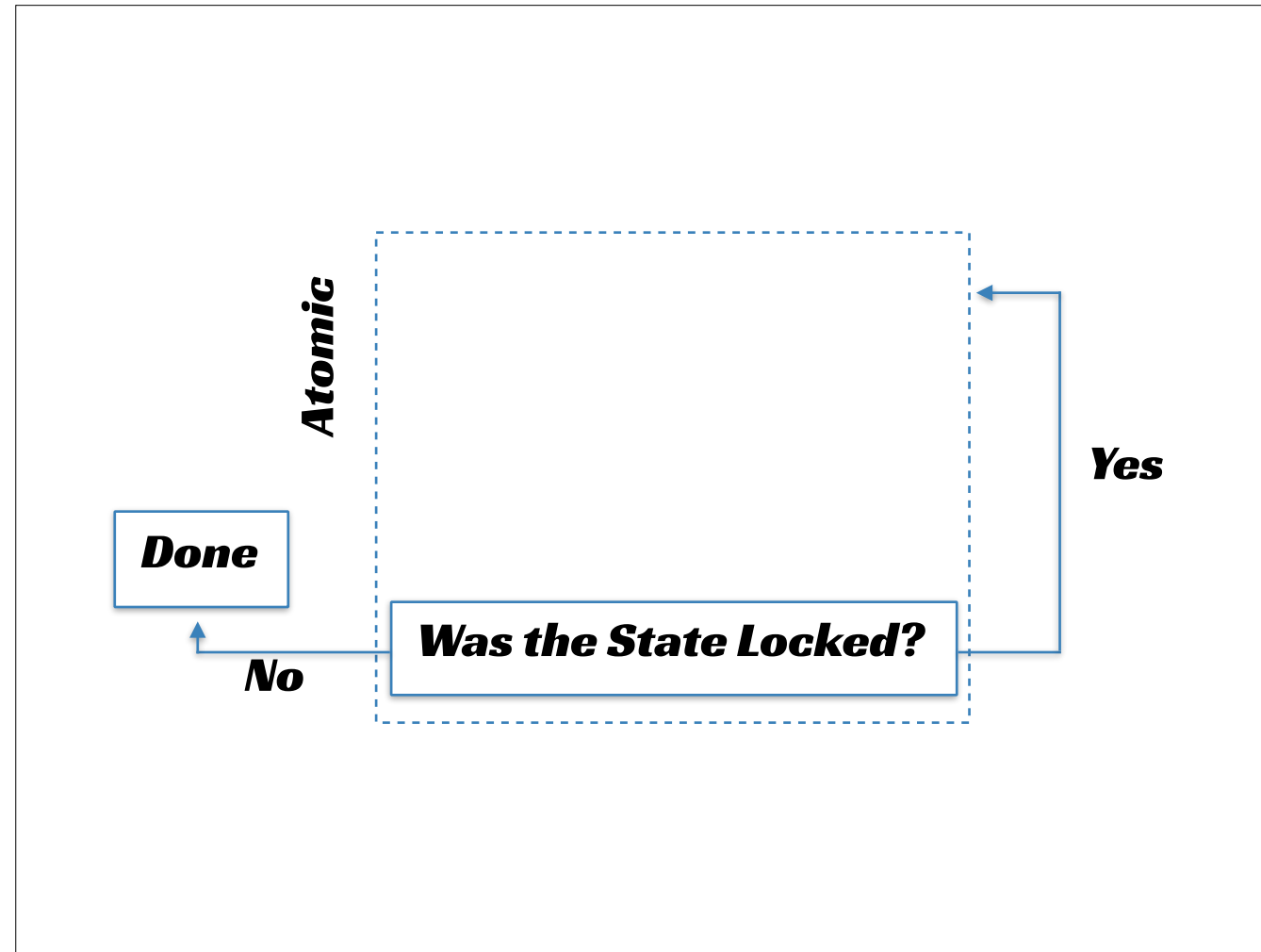
So if this is how we might implement our allocator, which right now just pops an element off the list to allocate and inserts it into the freelist to free, what do we have to add in order to guarantee sequential consistency?

```
obj *allocate(slab *s) {  
    lock(&allocator_lock);  
    obj *a = s->head;  
    if (a == NULL) return NULL;  
    s->head = a->next;  
    unlock(&allocator_lock);  
    return a;  
}  
  
void free(slab *s, obj *o) {  
    lock(&allocator_lock);  
    o->next = s->head;  
    s->head = o;  
    unlock(&allocator_lock);  
}
```

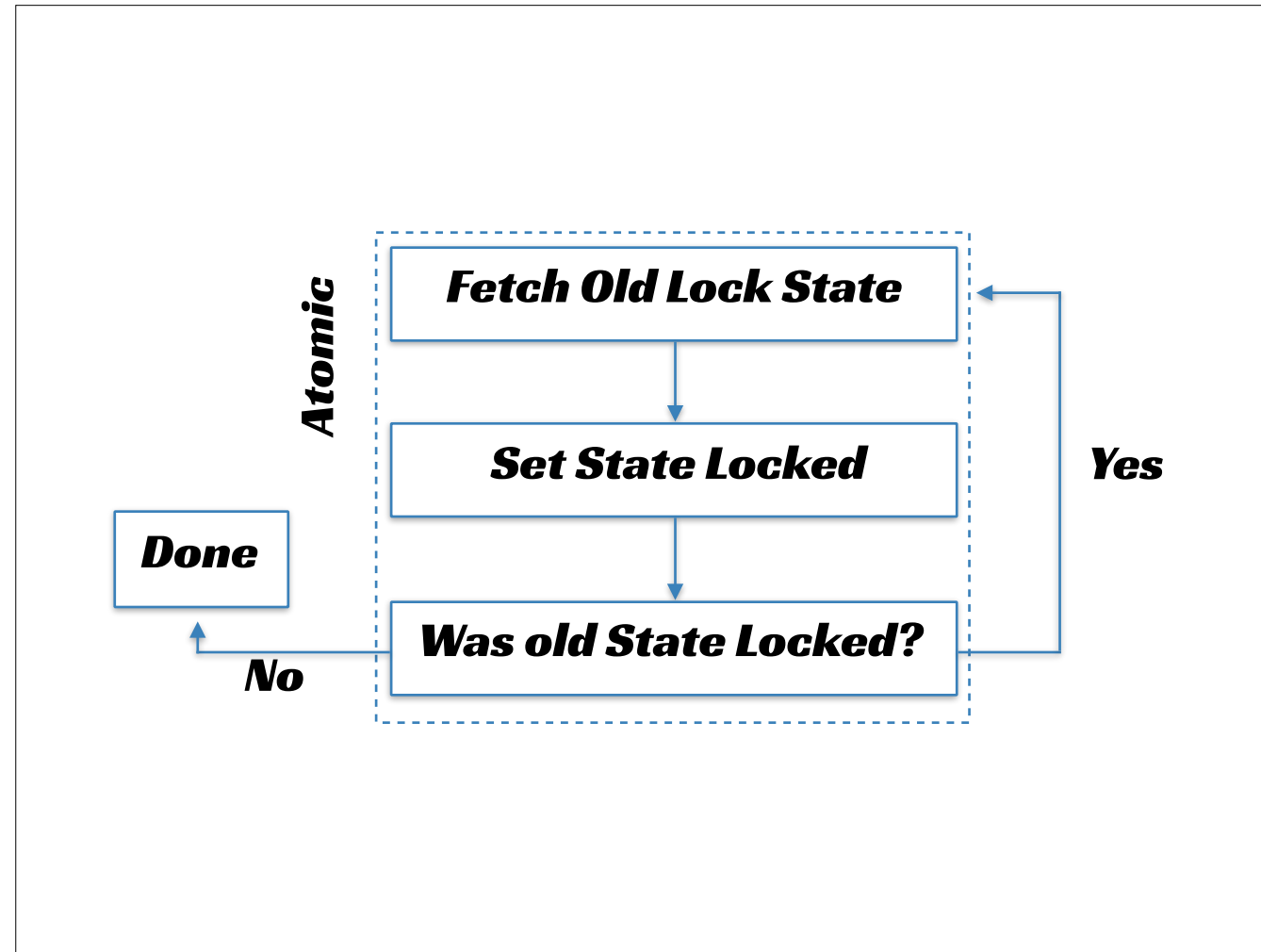
We can wrap all operations with a lock!

Lock-based synchronization is the canonical solution to sharing memory in multiprocessor systems. Locks are easy to reason about *because* they serialize: we only need to think about individual concurrent processes. Because they serialize access to their critical section, which is in this case pushing or popping the freelist, they provide a means for achieving a sequential history.

Okay, so that's cool, now let's look at how that lock might be implemented. There are many different ways of implementing a lock and production-caliber locks are usually a hybrid of various techniques, but the most instructive kind of lock for the purposes of this talk are *spinlocks*.

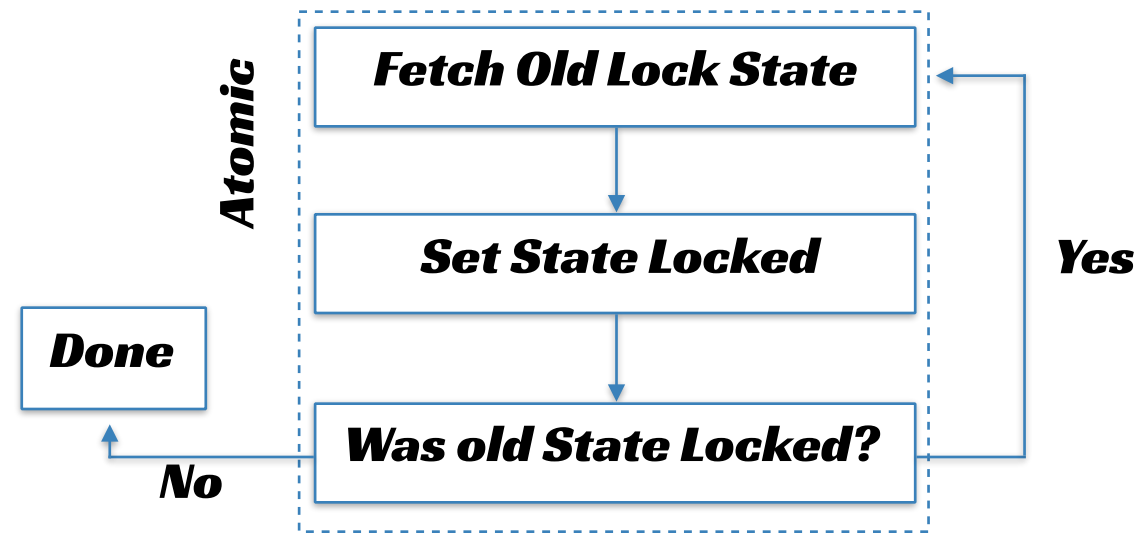


A spinlock loops (or spins) in an attempt to move the lock from an **unlocked** state to a **locked** one. If it's already locked, someone has control of the critical section and we have to keep spinning. If it's unlocked, though, we can set the state to "locked" and proceed into the critical section.



The way we do this is by first reading the lock's state, overwriting the state with "locked", and *then* asking "was the old value locked (e.g. held by someone else) or unlocked (eg. held by nobody). <click> This operation is implemented in hardware and is usually called Test And Set or Fetch and Set.

Test And Set Lock



The way we do this is by first reading the lock's state, overwriting the state with "locked", and *then* asking "was the old value locked (e.g. held by someone else) or unlocked (eg. held by nobody). <click> This operation is implemented in hardware and is usually called Test And Set or Fetch and Set.

Test And Set Unlock

Atomic

Set State Unlocked

Unlocking is simply a matter of atomically setting the state to unlocked.


```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    while (atomic_tas(m, LOCKED) == LOCKED)
        snooze();
}

void unlock(spinlock *m) {
    atomic_store(m, UNLOCKED);
}
```

*Many code examples
derived from Concurrency Kit
<http://concurrencykit.org>*

This is how an implementation might look in the C language. Everything looks correct, but can we define the correctness of this implementation more formally?

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

{ A(TAS request)
A(TAS response) }

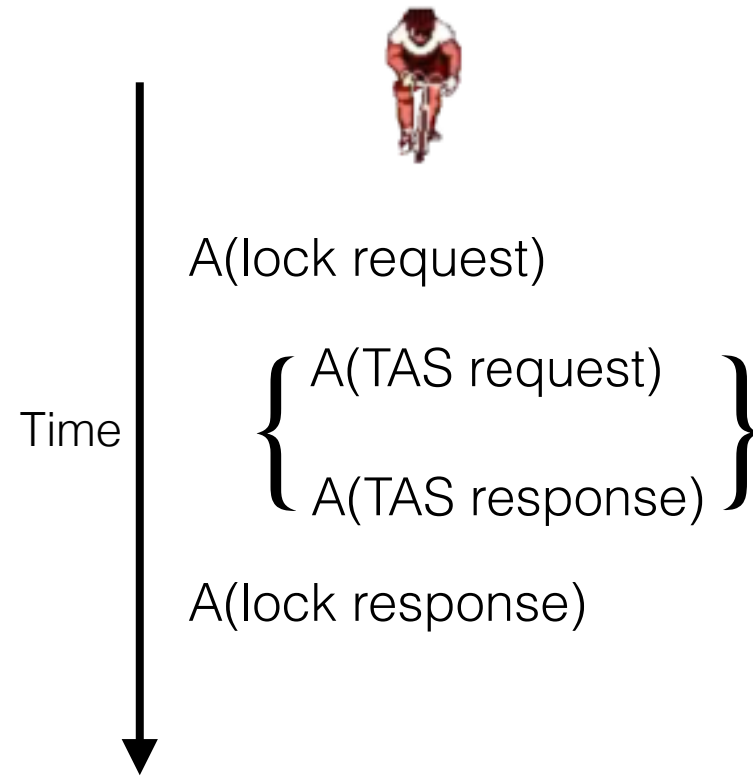
Because an atomic operation is, by definition, indivisible, any atomic operations to the `m` variable are sequentially consistent among themselves. Now, there are three things to notice here:

TAS is embedded in Lock

$$\left\{ \begin{array}{l} A(\text{TAS request}) \\ A(\text{TAS response}) \end{array} \right\}$$

First, that this test-and-set operation is really embedded in a larger operation in between the call to “lock” and the “lock” function returning.

TAS is embedded in Lock



First, that this test-and-set operation is really embedded in a larger operation in between the call to “lock” and the “lock” function returning.

TAS & Store can't be reordered



Time



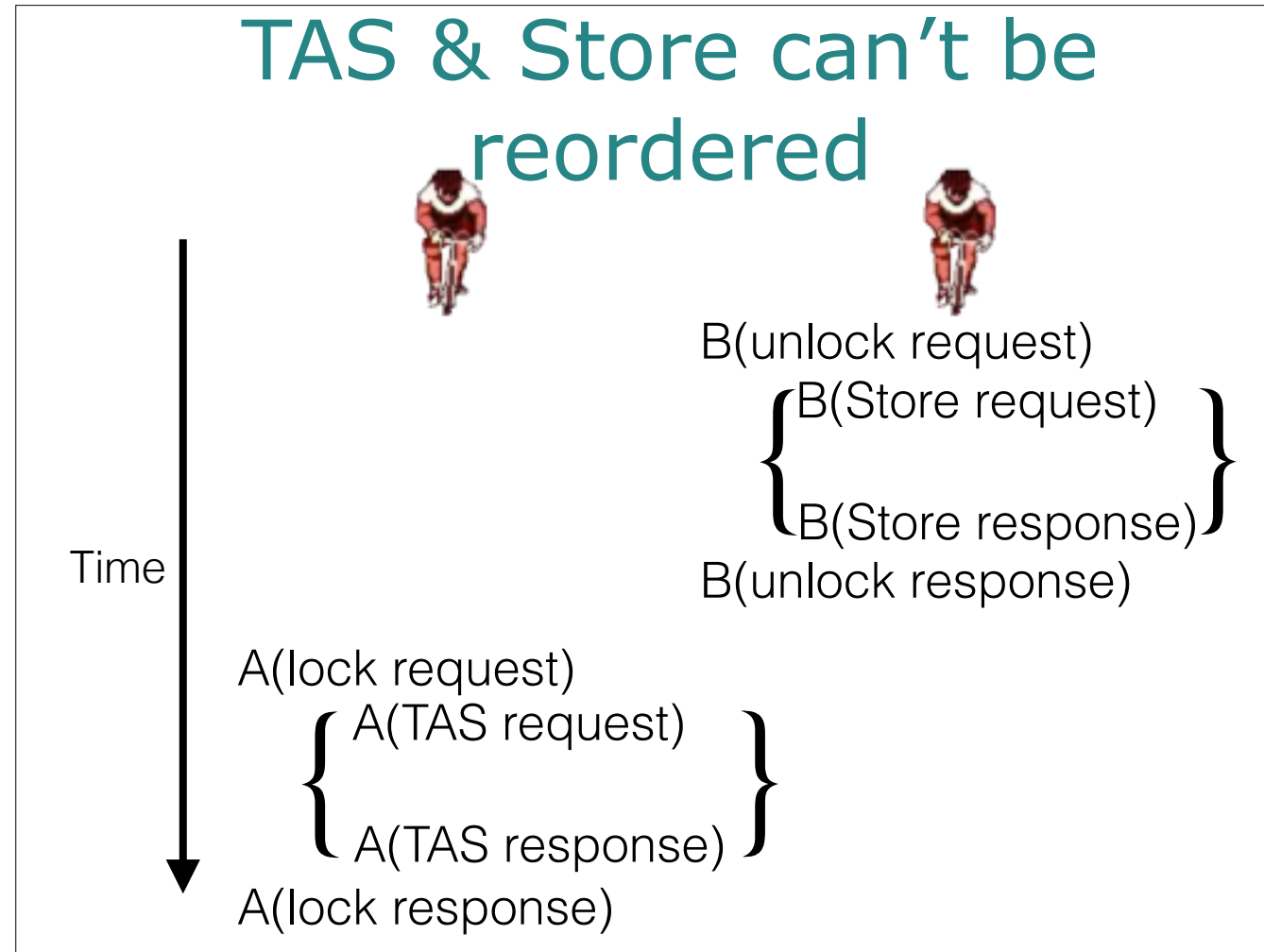
A(lock request)

{ A(TAS request)
A(TAS response) }

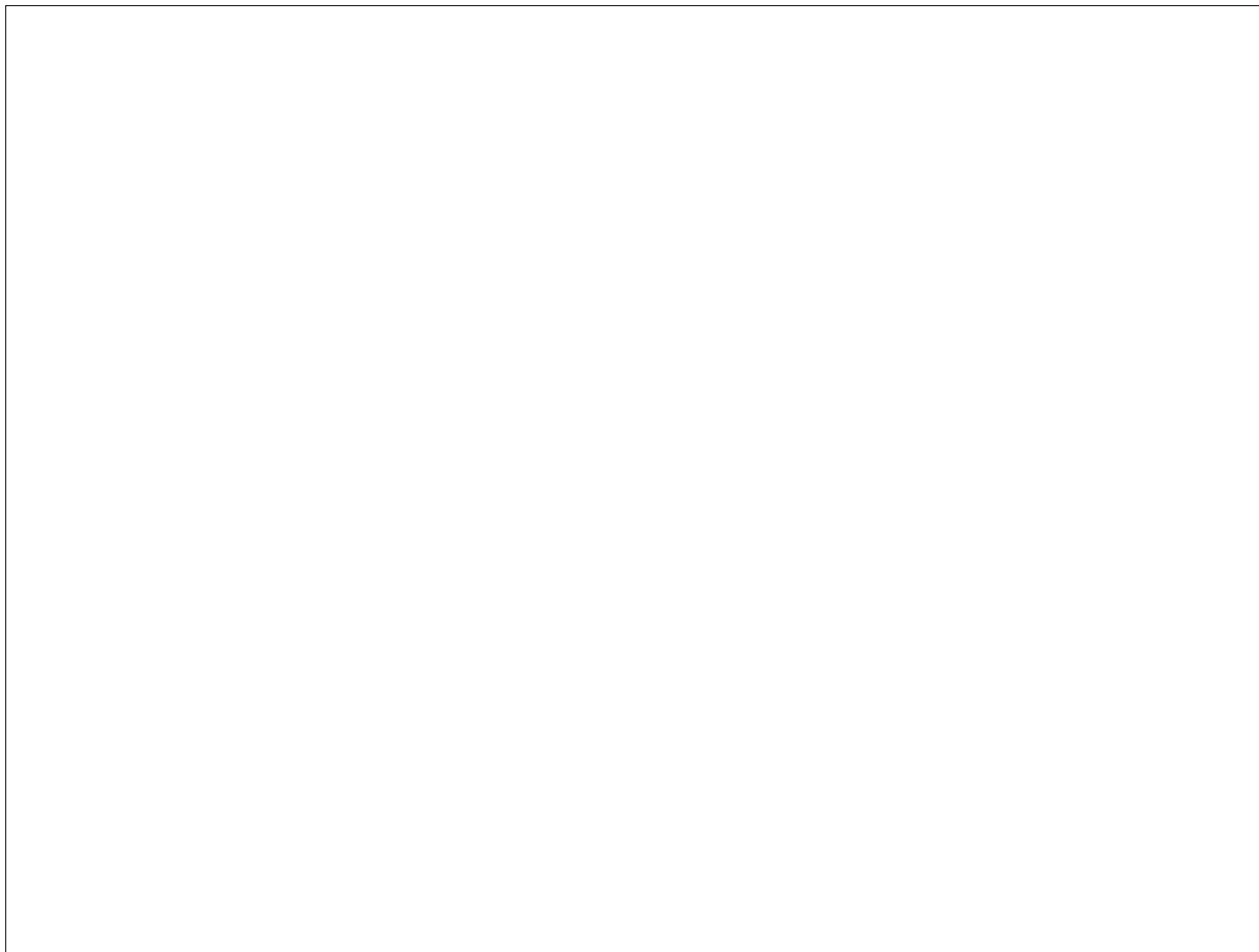
A(lock response)

Second, in the case of multiple threads, let's imagine has previously held the lock and is unlocks it before A tries to acquire it. Notice that we have a stronger constraint than sequential consistency here: we are not allowed to execute the atomic request/response pairs such that B's store follows A's test and set, because that would mean that A succeeded in taking the lock before B released it!

TAS & Store can't be reordered



Second, in the case of multiple threads, let's imagine has previously held the lock and is unlocks it before A tries to acquire it. Notice that we have a stronger constraint than sequential consistency here: we are not allowed to execute the atomic request/response pairs such that B's store follows A's test and set, because that would mean that A succeeded in taking the lock before B released it!



So if we've concluded that out of the set of all execution histories, there's a subset of those that satisfy the sequential consistency property, <click> it seems like we're moving towards a subset of **those** that satisfies an even stricter property.

All execution histories
 \supseteq
All sequentially-consistent
execution histories

So if we've concluded that out of the set of all execution histories, there's a subset of those that satisfy the sequential consistency property, <click> it seems like we're moving towards a subset of *those* that satisfies an even stricter property.

All execution histories

\supseteq

All sequentially-consistent
execution histories

\supseteq

All ???able execution
histories

So if we've concluded that out of the set of all execution histories, there's a subset of those that satisfy the sequential consistency property, <click> it seems like we're moving towards a subset of *those* that satisfies an even stricter property.

All execution histories

\supseteq

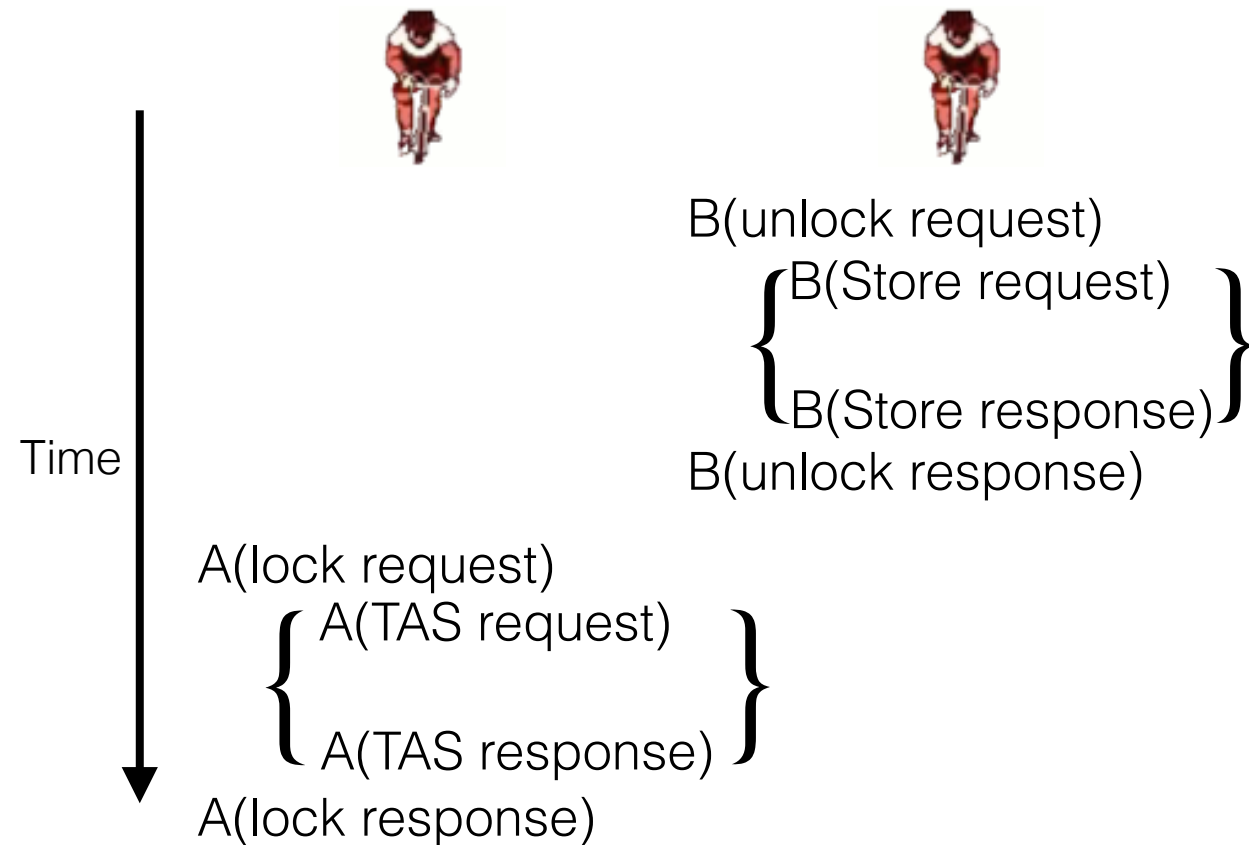
All sequentially-consistent
execution histories

\supseteq

All linearizable execution
histories

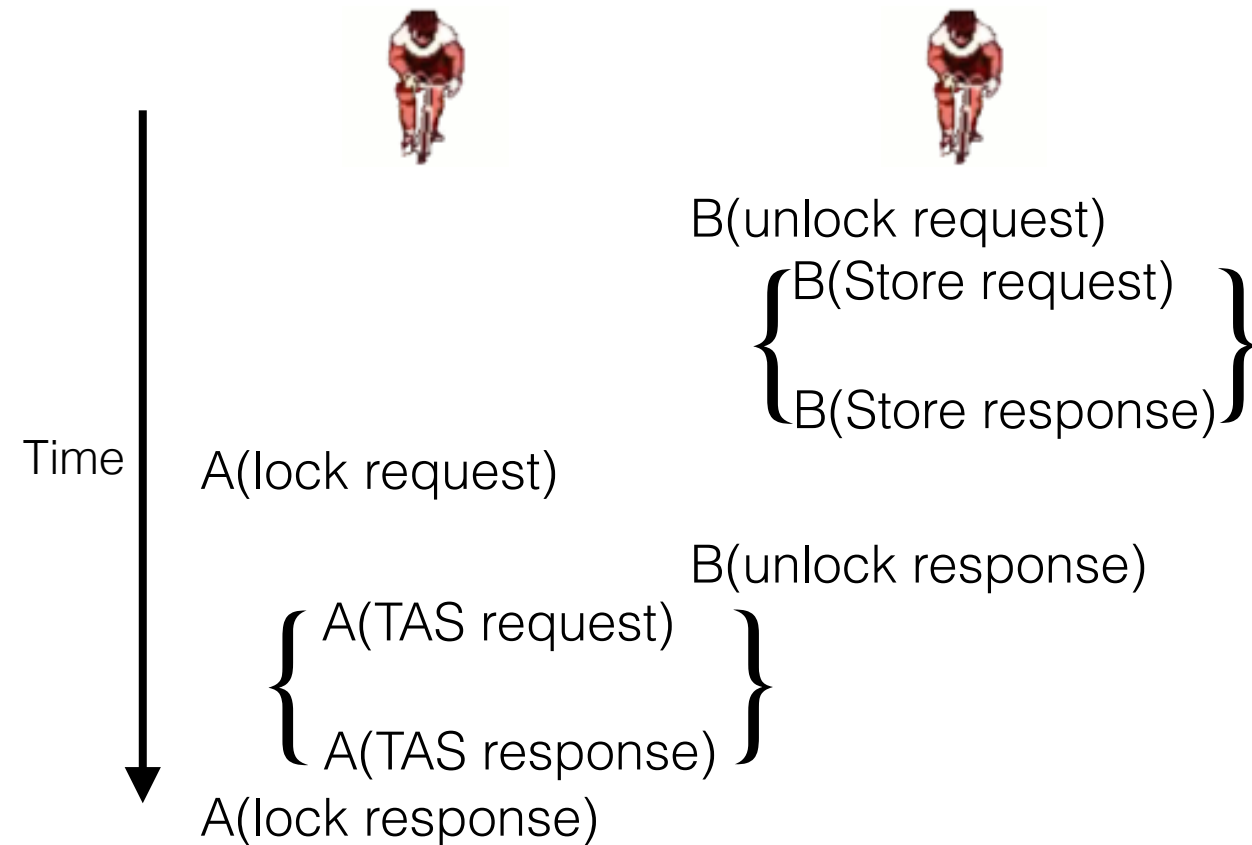
It happens to be that the term we're looking for is *linearizable*. A linearizable history is a sequential history, that additionally states that responses to requests must not be reordered with respect to each other.

Others can be reordered



Given this definition, we might think that there's only one valid execution history here. However, notice that if A enters the lock function in between B's store response and the unlock response, like so <click>, then we haven't broken the correctness of the protocol.

Others can be reordered



Given this definition, we might think that there's only one valid execution history here. However, notice that if A enters the lock function in between B's store response and the unlock response, like so <click>, then we haven't broken the correctness of the protocol.

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}  
  
void unlock(spinlock *m) {  
    atomic_store(m, UNLOCKED);  
}
```

Something is clearly special about these two operations.

Specifying Concurrent Program Modules

LESLIE LAMPORT
SRI International

3.4.2 *An Implementation.* In this specification, we have required that the queue behave as if adding or removing an element from it were atomic operations. This does not mean that the entire PUT and GET subroutines have to be implemented as single atomic actions. It does mean that when the PUT operation is adding an element to the queue, there must be some instant at which that element becomes visible to the GET subroutine, and similarly some instant at which the GET operation finishes removing the element from the queue. If there were not such an instant, then the GET subroutine might try to remove an element that the PUT subroutine had not finished putting in the queue, obtaining only part of the element.¹

© 1983 ACM 0164-0925/83/0400-0190 \$00.75

ACM Transactions on Programming Languages and Systems, Vol. 5, No. 2, April 1983, Pages 190-222.

<http://dl.acm.org/citation.cfm?id=69624.357207>

This is an excerpt from the paper by Lamport that is credited with coining the term “linearizable”. In the paper they are discussing enqueueing and dequeueing elements onto a concurrent queue. Lamport is telling us that it’s sufficient for there to be a single atomic *moment* in our functions where the action is “committed”, as opposed to insisting that the entire functions need to be atomic.

We call these moments linearization points.

```
obj *allocate(slab *s) {  
    lock(&allocator_lock);  
    obj *a = s->head;  
    if (a == NULL) return NULL;  
    s->head = a->next;  
    unlock(&allocator_lock);  
    return a;  
}
```

```
void free(slab *s, obj *o) {  
    lock(&allocator_lock);  
    o->next = s->head;  
    s->head = o;  
    unlock(&allocator_lock);  
}
```

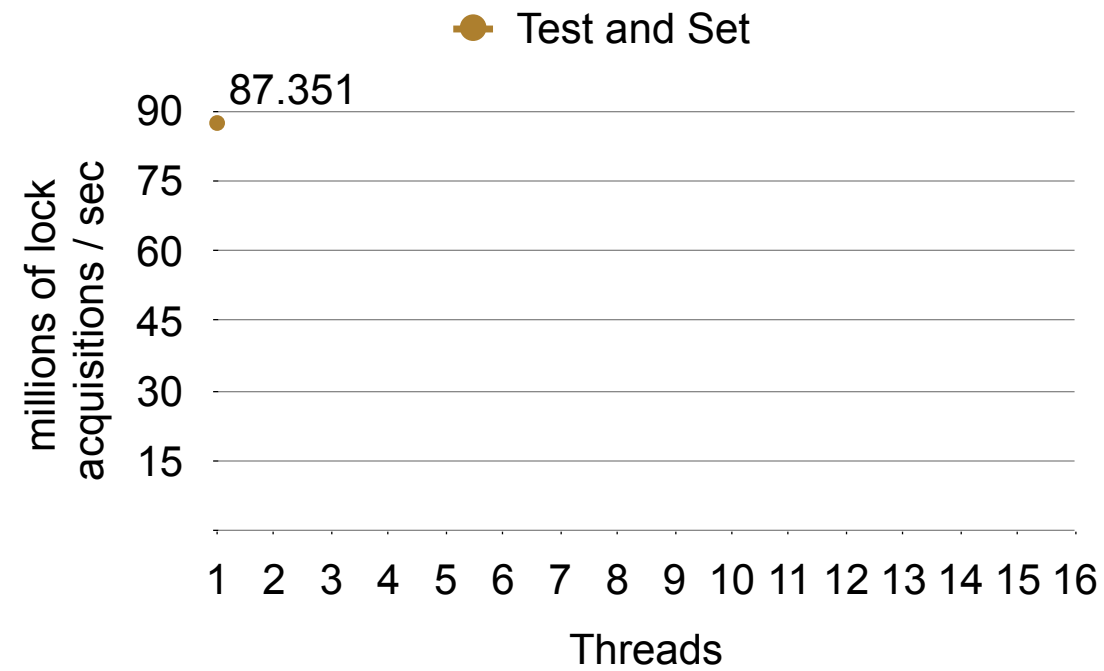
In functions that use a lock-based synchronization plan, those linearization moments would be the lock's critical sections.

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}  
  
void unlock(spinlock *m) {  
    atomic_store(m, UNLOCKED);  
}
```

And in the case of the lock itself, it's these hardware-provided atomic instructions.

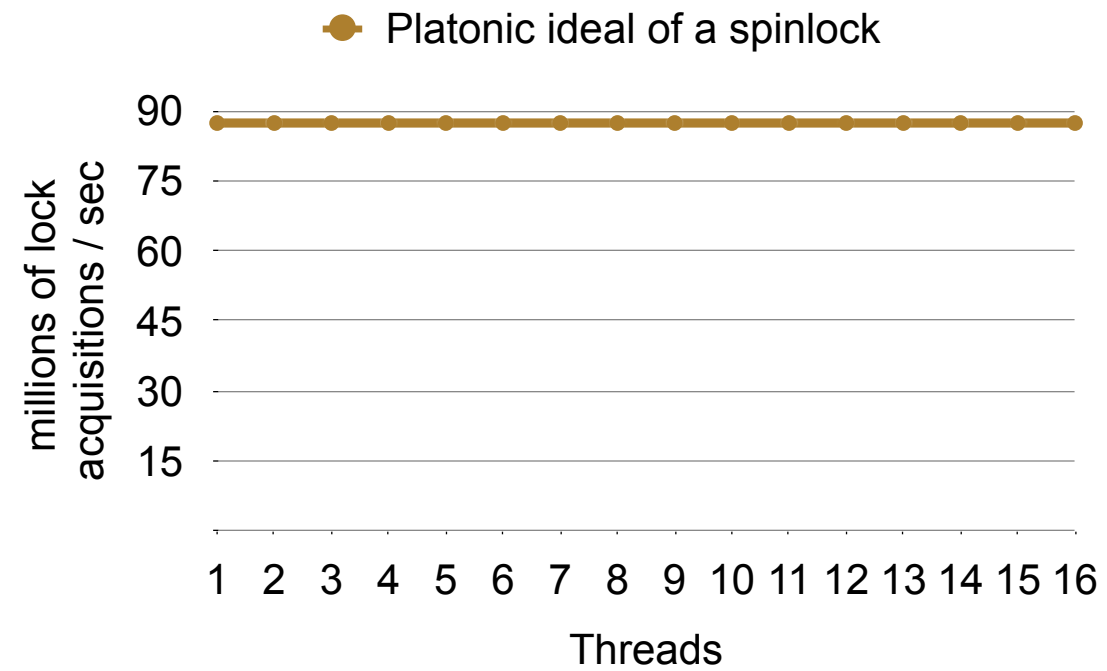
So, now that we've convinced ourselves that these linearization points enforce a correctness guarantee with respect to concurrent users for this lock implementation, another question we can ask ourselves is how well they actually perform.

Spinlock performance



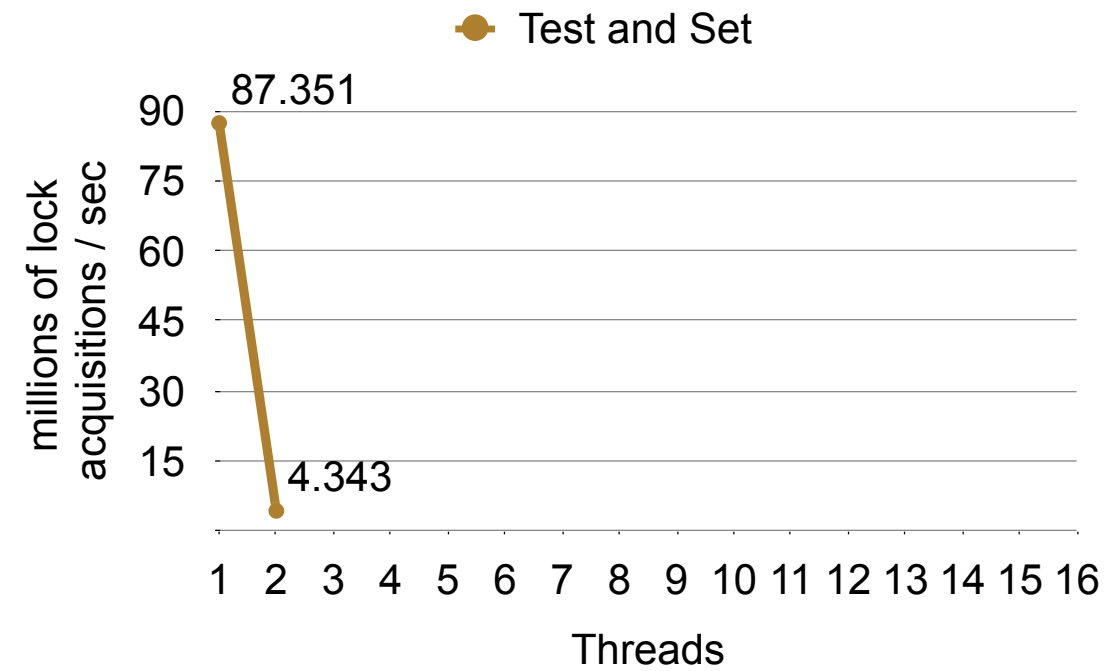
This graph shows lock total acquisitions on the Y axis across N threads represented on the X axis. We see that in the absence of contention we get great throughput - ninety million acquisitions per second corresponds to about 10 microseconds per lock acquisition. And, in a perfect world, However...

Spinlock performance



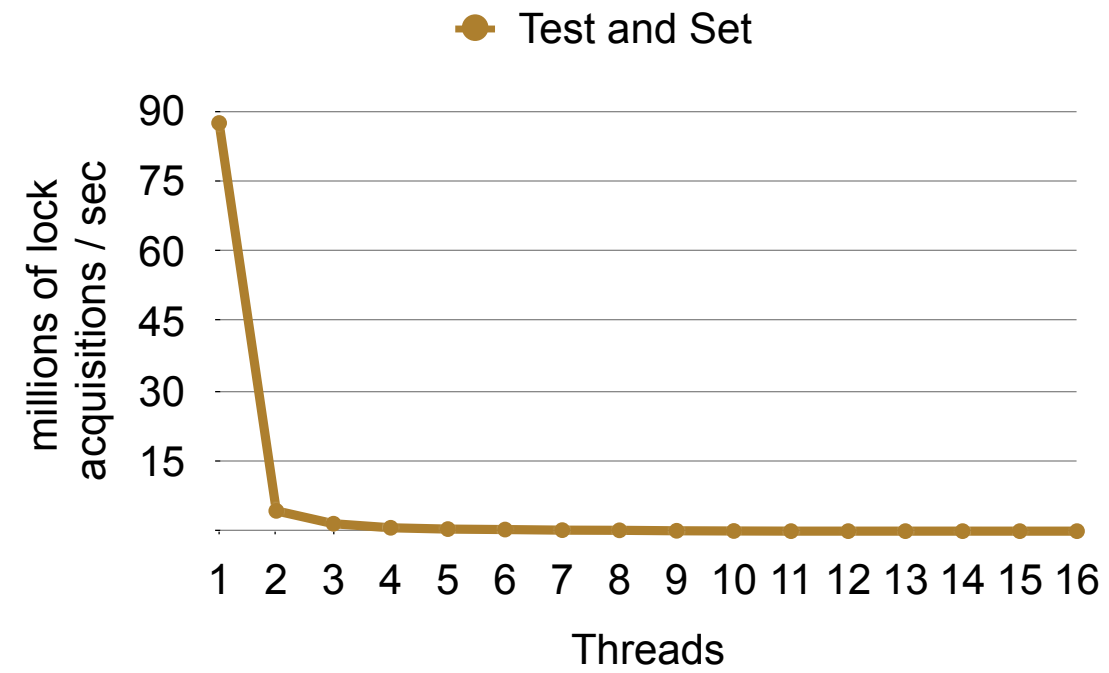
we would hope that the number of acquisitions we make is independent of the number of threads. However, ...

Spinlock performance



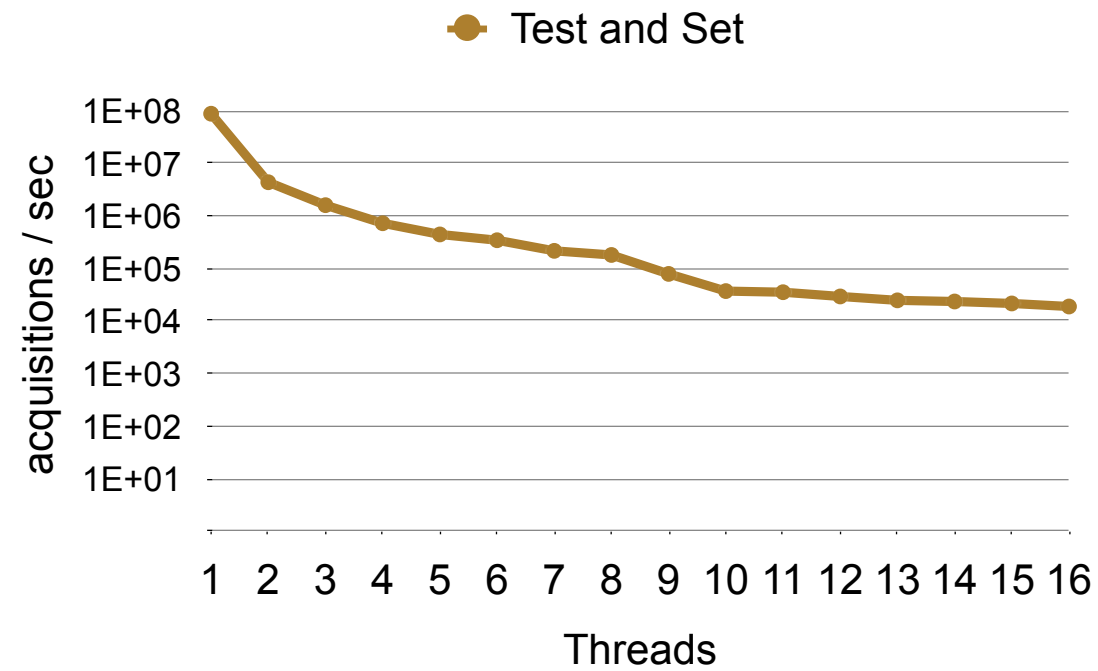
The moment we add a second thread, the number of acquisitions we can make drops by a factor of 20.

Spinlock performance



this really bottoms out as soon as we add more threads.

Spinlock performance



This logarithmic graph better illustrates exactly how bad it gets: even with 4 threads contending on the lock, our allocation throughput drops by two orders of magnitude. This effectively bounds the performance of our allocator, but more importantly for us, this demonstrates an inability to scale. Can we do better?

```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    while (atomic_tas(m, LOCKED) == LOCKED)
        snooze();
}
```

Recall that the test-and-set operation that serves as our linearization point also, well, sets the spinlock to locked. Every time through the while conditional we are forced to do an atomic read and write to memory, and if the lock is contended we're taking a spinlock that's set to LOCKED and, well, setting it to LOCKED over and over again. This smells like something we can improve on, because under high contention we're going to fail to get the lock most of the time

```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    while (atomic_tas(m, LOCKED) == LOCKED) {
        while (atomic_store(m) == LOCKED)
            snooze();
    }
}
```

Our lock now looks like this. If we fail the test-and-set let's try a cheaper read of the spinlock. Because we're not doing a write in the "already locked" loop, memory caches don't have to be invalidated and there's less bus contention.

The first time I saw a <click> test and test and set lock, I was completely, 100% convinced that it was incorrect. Isn't there a race condition right in front of us?

Technically, yes: A race condition is a system behavior where the output of the system is dependent on the sequence or timing of other uncontrollable events. Are races safe? They absolutely can be — they only become bugs when events occur out of intended order.

The reason why events won't occur out of intended order is because of the atomic_tas linearization point. This means that although multiple processes may read an unlocked state in the inner loop, we are still guaranteed that only one process is able to obtain the lock at any time. That behavior allows us to race many processes in the cheaper inner loop.

Does this new strategy help?

Test-and-Test-and-Set

```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    while (atomic_tas(m, LOCKED) == LOCKED) {
        while (atomic_store(m) == LOCKED)
            snooze();
    }
}
```

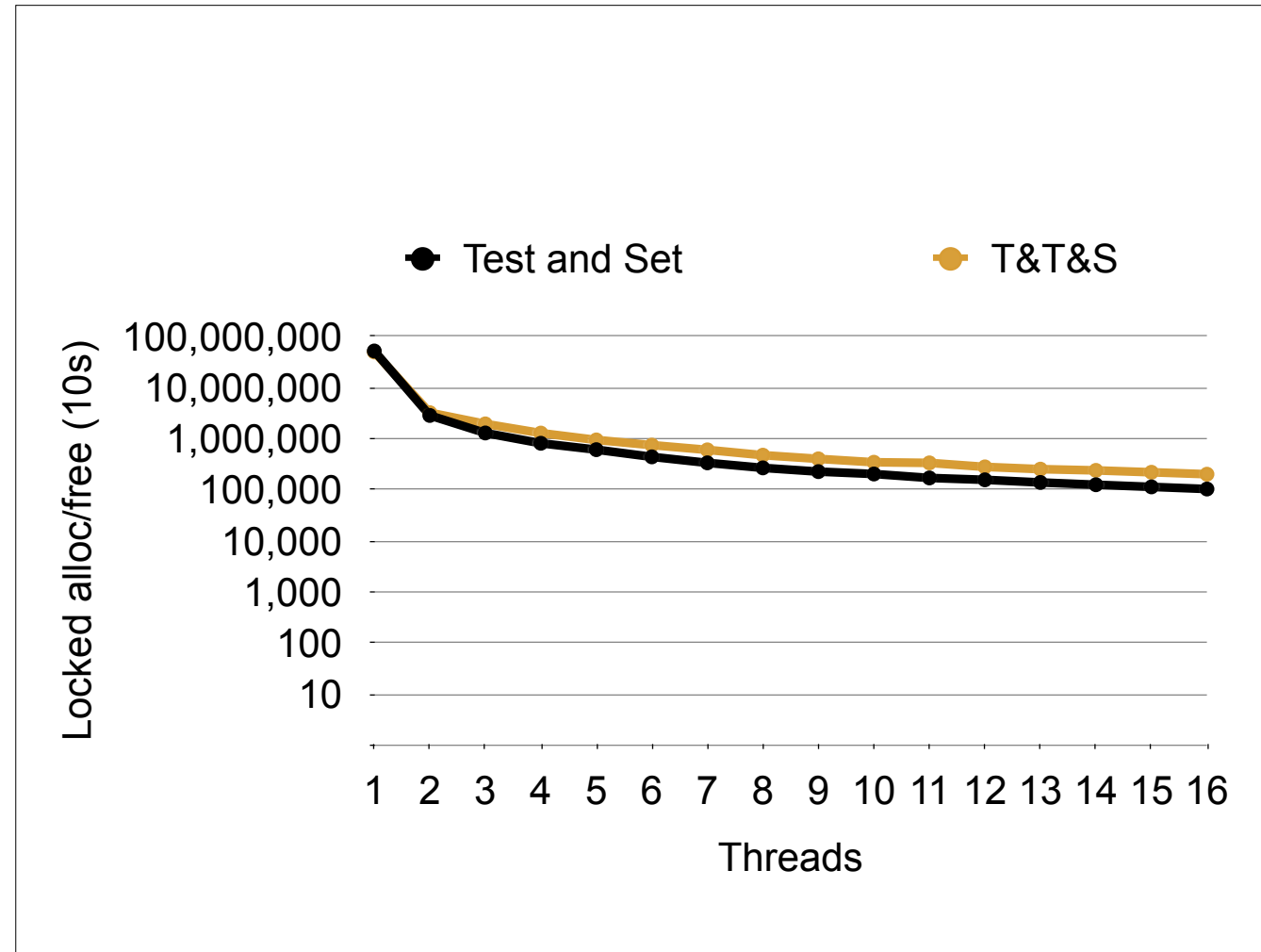
Our lock now looks like this. If we fail the test-and-set let's try a cheaper read of the spinlock. Because we're not doing a write in the "already locked" loop, memory caches don't have to be invalidated and there's less bus contention.

The first time I saw a <click> test and test and set lock, I was completely, 100% convinced that it was incorrect. Isn't there a race condition right in front of us?

Technically, yes: A race condition is a system behavior where the output of the system is dependent on the sequence or timing of other uncontrollable events. Are races safe? They absolutely can be — they only become bugs when events occur out of intended order.

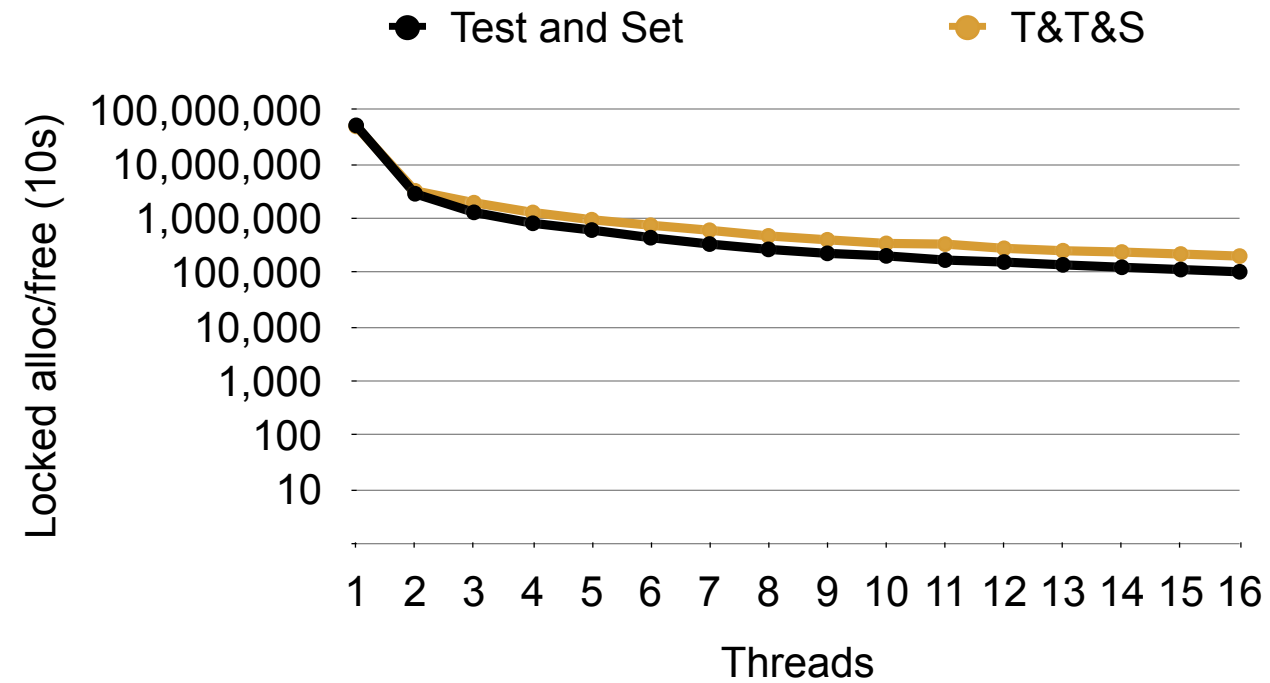
The reason why events won't occur out of intended order is because of the atomic_tas linearization point. This means that although multiple processes may read an unlocked state in the inner loop, we are still guaranteed that only one process is able to obtain the lock at any time. That behavior allows us to race many processes in the cheaper inner loop.

Does this new strategy help?



Well, sorta. The improvement isn't great, and more importantly, we're still not getting good, predictable throughput as contention increases.

Spinlock performance



Well, sorta. The improvement isn't great, and more importantly, we're still not getting good, predictable throughput as contention increases.

```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    unsigned long backoff, exp = 0;
    while (atomic_tas(m, LOCKED) == LOCKED) {
        for (i = 0; i < backoff; i++)
            snooze();
        backoff = (1ULL << exp++);
    }
}
```

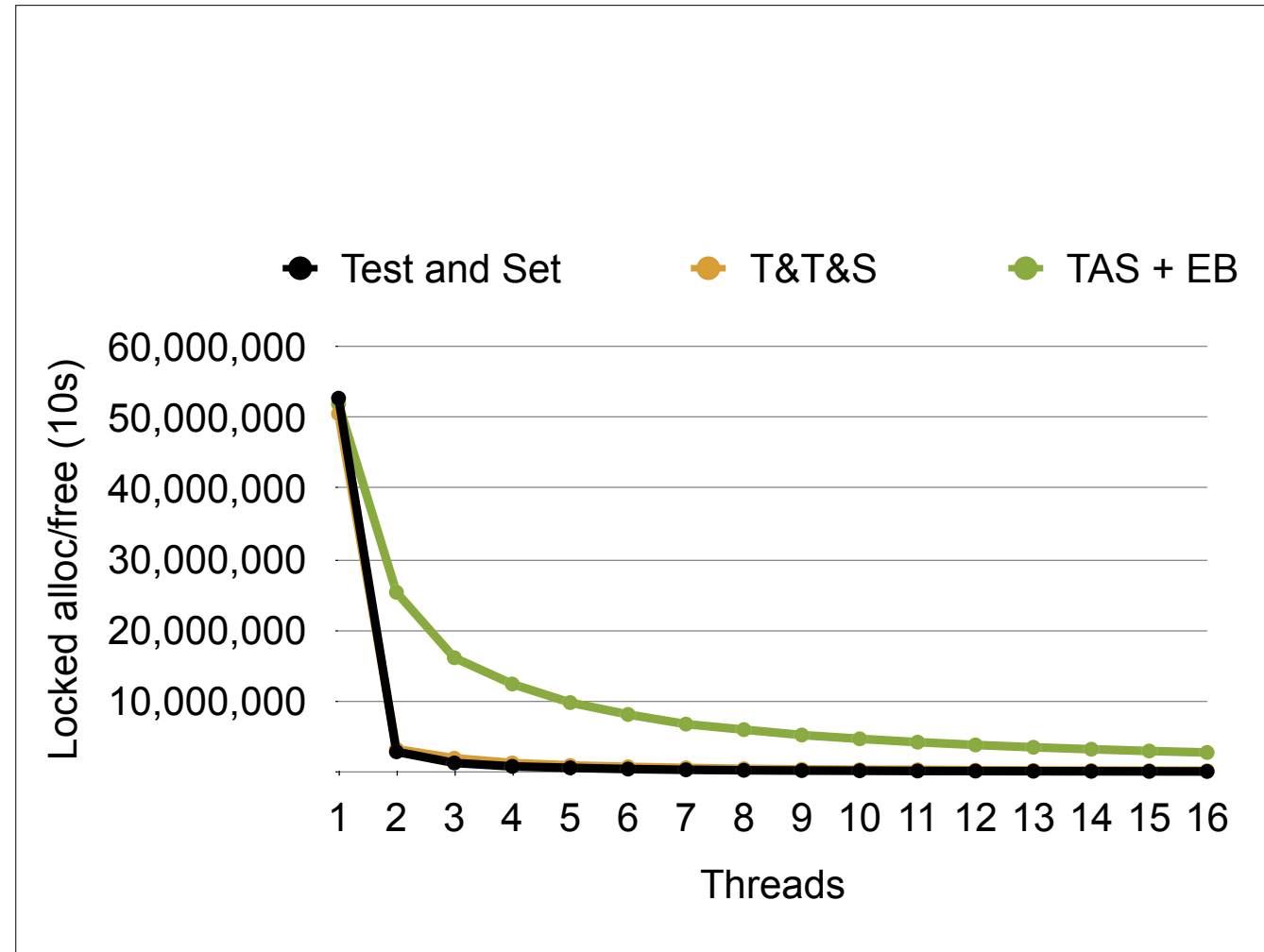
Well, we can add a bit of local state and implement a back off algorithm. This is still correct because it's exactly the same algorithm as our original TAS, but we don't run our atomic operation every loop iteration. This might seem problematic for a different reason; because we back off exponentially we could overshoot the amount we need to snooze for and end up spinning far longer than we need to, making things worse! Does this happen?

TAS + backoff

```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

void lock(spinlock *m) {
    unsigned long backoff, exp = 0;
    while (atomic_tas(m, LOCKED) == LOCKED) {
        for (i = 0; i < backoff; i++)
            snooze();
        backoff = (1ULL << exp++);
    }
}
```

Well, we can add a bit of local state and implement a back off algorithm. This is still correct because it's exactly the same algorithm as our original TAS, but we don't run our atomic operation every loop iteration. This might seem problematic for a different reason; because we back off exponentially we could overshoot the amount we need to snooze for and end up spinning far longer than we need to, making things worse! Does this happen?



So we get better throughput, but this kink in the graph right around $x = 2$ is still pretty depressing. We could probably keep going and trying every trick in the book, but it seems like we're only changing a coefficient on this exponential decay. In order to understand why, we have to make a quick diversion into something called a progress guarantee.

```
spinlock global_lock = UNLOCKED
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

Let's say we have two threads and some spinlock that's visible to both threads. The first thread

```
spinlock global_lock = UNLOCKED
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

Let's say we have two threads and some spinlock that's visible to both threads. The first thread


```
spinlock global_lock = UNLOCKED
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```


```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

enters the function


```
spinlock global_lock = UNLOCKED
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

enters the function


```
spinlock global_lock = LOCKED
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```


```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

swaps the state to locked, and exits. Now let's say

```
spinlock global_lock = LOCKED
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

swaps the state to locked, and exits. Now let's say

```
spinlock global_lock = LOCKED
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

The other thread comes in, and wants to grab the lock but of course it can't. It's progress is

```
spinlock global_lock = LOCKED
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

The other thread comes in, and wants to grab the lock but of course it can't. It's progress is

```
spinlock global_lock = LOCKED
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

halted at the Snooze call, and how long it has to wait is completely dependent on how long the top thread holds the lock for. The top thread is under no obligation to ever release the lock, so the other thread might be trying and failing and trying and failing forever - one can say that there's contention on the lock even though nobody else is trying to acquire it.

```
spinlock global_lock = LOCKED
```

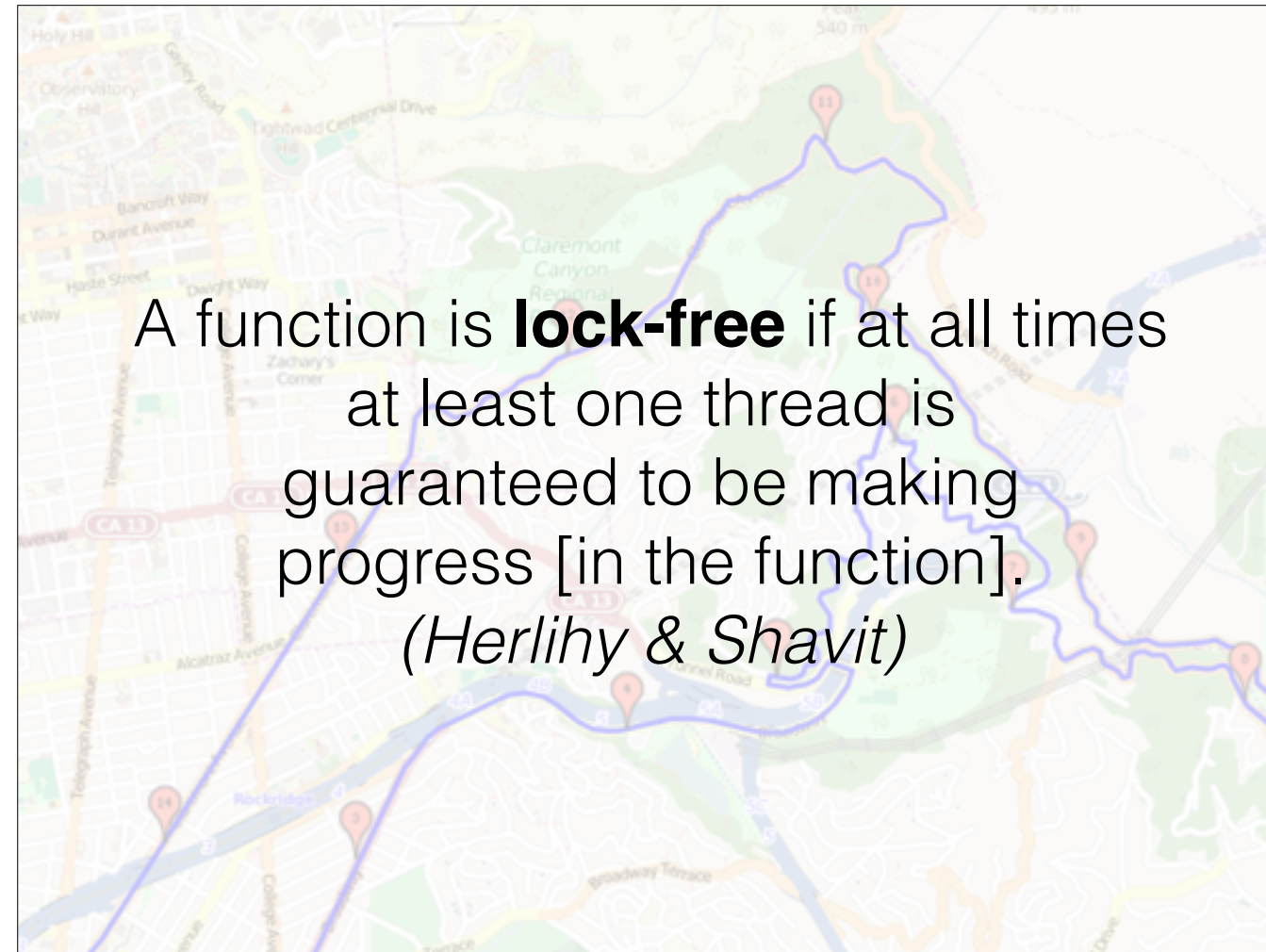


```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```



```
void lock(spinlock *m) {  
    while (atomic_tas(m, LOCKED) == LOCKED)  
        snooze();  
}
```

halted at the Snooze call, and how long it has to wait is completely dependent on how long the top thread holds the lock for. The top thread is under no obligation to ever release the lock, so the other thread might be trying and failing and trying and failing forever - one can say that there's contention on the lock even though nobody else is trying to acquire it.



A function is **lock-free** if at all times
at least one thread is
guaranteed to be making
progress [in the function].
(Herlihy & Shavit)

The top thread wasn't making progress in the lock function because, well, it's entered and exited the lock function! And the bottom function wasn't making progress in the lock function because it's not allowed to proceed until the lock is unlocked. So, if we define a lock-free function as one that at least one thread is guaranteed to be making progress in, the lock-free property does not hold here, because no thread is making progress in a locking or unlocking function.



This creates an inherent queueing effect where whole threads are stalled, that causes the immediate scalability degradation that we saw in our performance graphs. So, sadly it doesn't matter how fancy our locks are, we can do our best to limit this effect but there's nothing we can do to completely overcome it using this programming model.

```
// TODO: make this safe and scalable
obj *allocate(slab *s) {
    obj *a = s->head;
    if (a == NULL) return NULL;
    s->head = a->next;
    return a;
}

// TODO: make this safe and scalable
void free(slab *s, obj *o) {
    o->next = s->head;
    s->head = o;
}
```

We need to look for a different solution, maybe one that satisfies the lock-free criterion we just defined, so all threads in an allocate or free function should be able to make forward progress.

And maybe we can use some of the same techniques we used to build the lock around the allocator, to actually build the allocator itself!

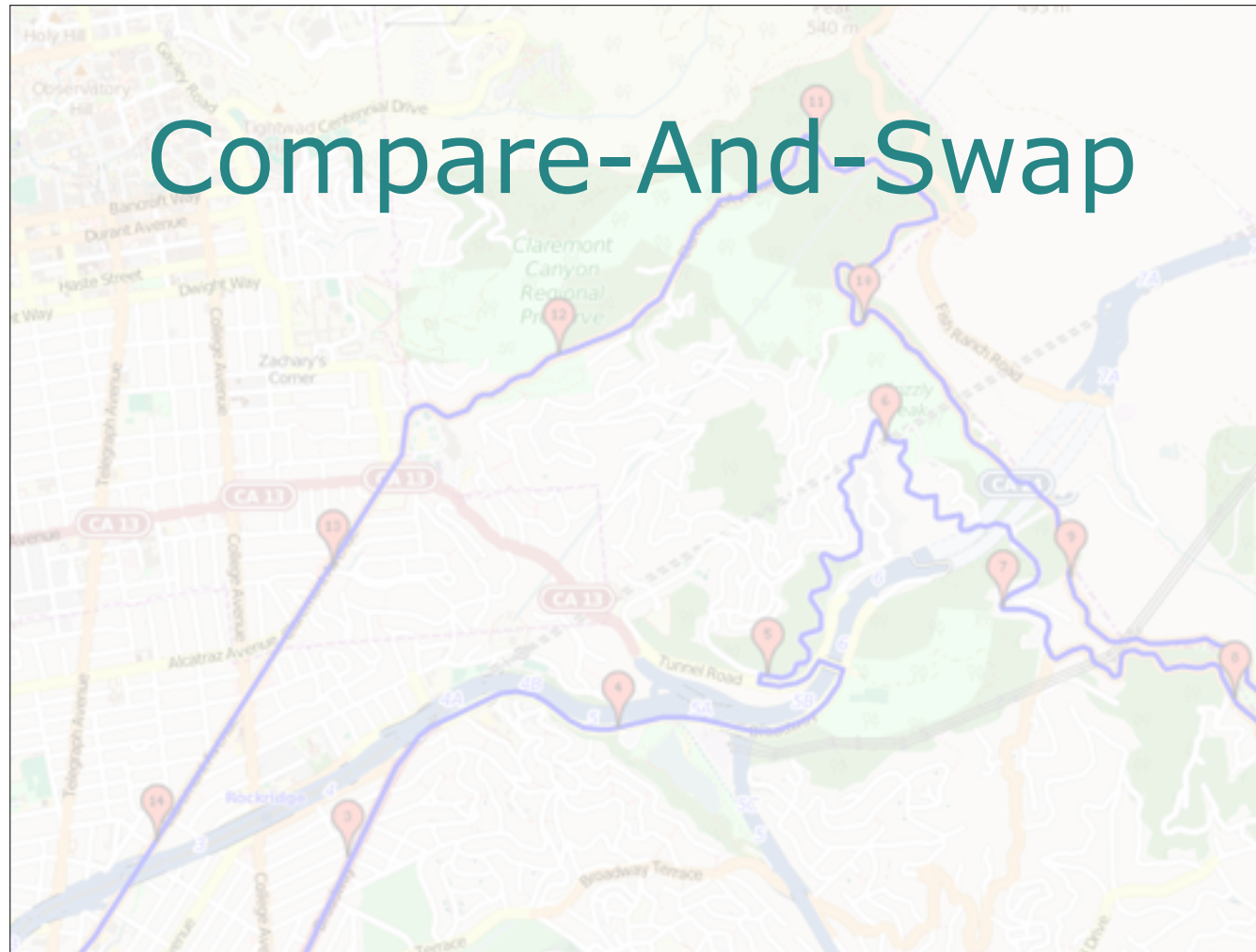
Non-Blocking Algorithms



Non-blocking algorithms and data structures are the world of “safe” and “scalable”. Unlike locks, they can make guarantees about system-wide progress under contention. How can we make our allocator non-blocking?

So, there are a few other atomic primitives that we haven’t talked about yet. One such primitive is


Compare-And-Swap

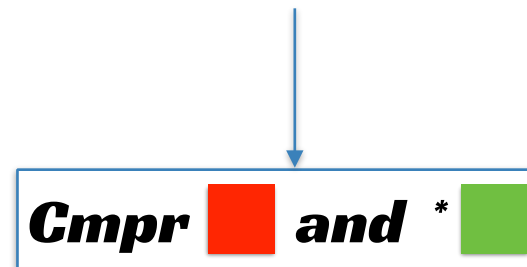


the Compare And Swap operation. I've always thought of this one as a generalization of test-and-set. How it works is that it

Compare-And-Swap

 Old value

 Destination
Address

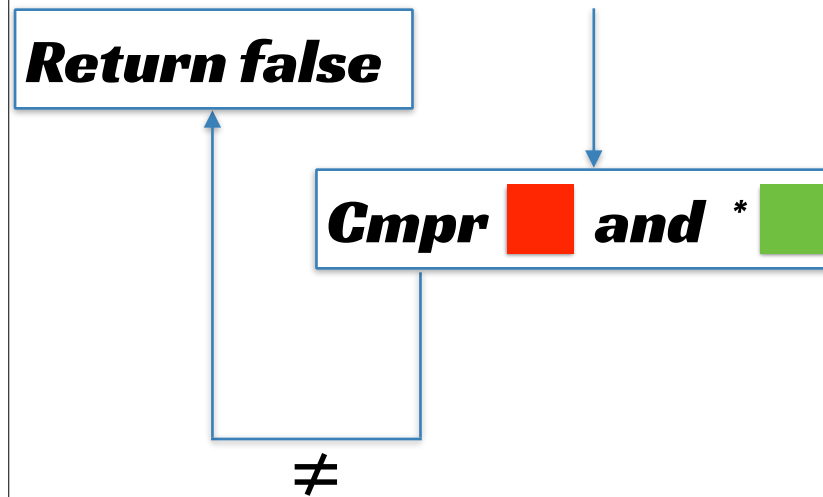


...is because it takes a value and and a pointer, dereferences the pointer, and compares those two values for equality.

Compare-And-Swap

Old value

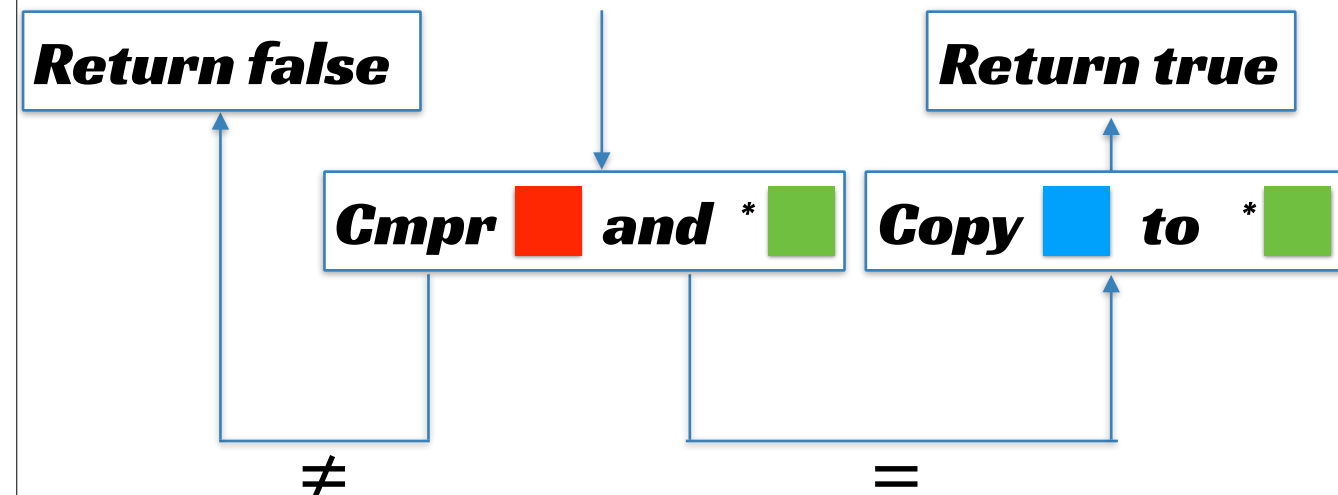
Destination
Address



If they differ, simply return false. However, if they are equal,

Compare-And-Swap

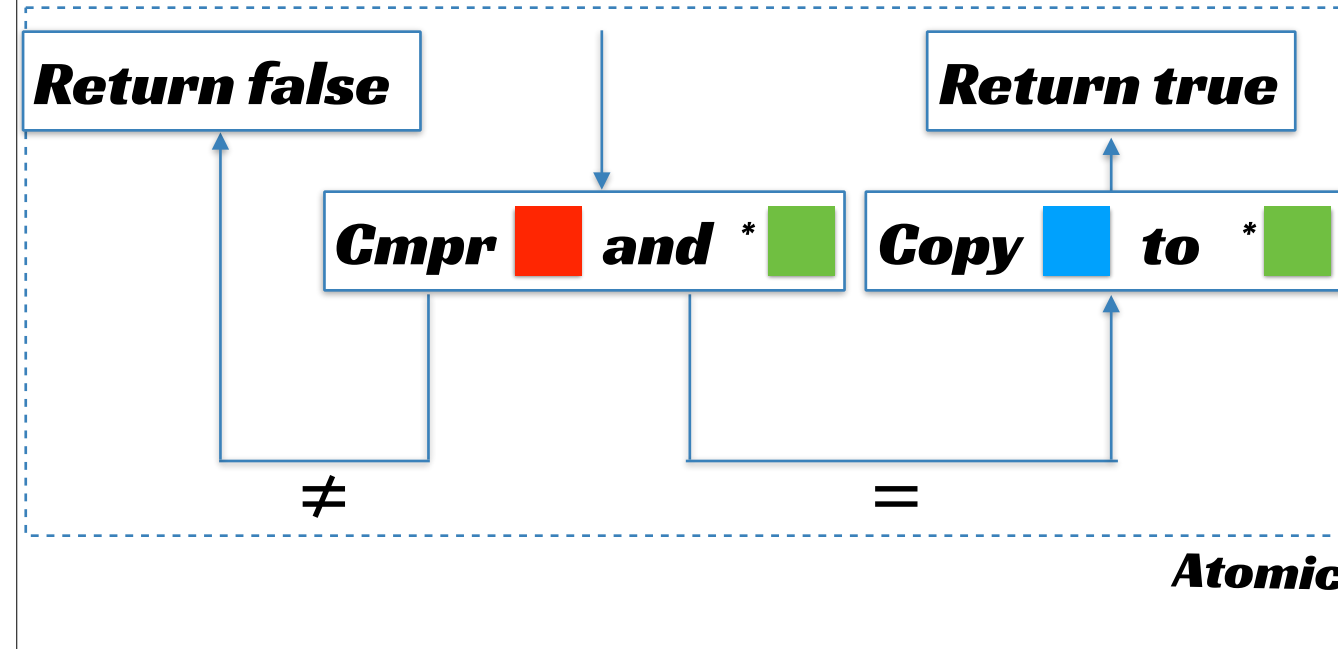
Old value New value Destination Address



then overwrite the contents of a given destination address with a new value, and then return true.

Compare-And-Swap

Old value New value Destination Address



And because this happens atomically, we can use this as another linearization point in our functions.

Let's look at a few simple examples of using a CAS.

Atomic `i = i+1;`

```
void atomic_inc(int *ptr) {  
    int i, i_plus_one;  
    do {  
        i = *ptr;  
        i_plus_one = i + 1;  
    } while (!cas(i, i_plus_one, ptr));  
}
```

This is about the simplest one I can think of. Let's use compare and swap to increment an integer, given a pointer to that integer. I should say that you'll often find on your own hardware a dedicated "atomic increment" instruction since it's such a common thing to do, but let's see how to do it was CAS anyway. We do it in three easy steps:

Atomic `i = i+1;`

```
void atomic_inc(int *ptr) {  
    int i, i_plus_one;  
    do {  
        i = *ptr;  
        i_plus_one = i + 1;  
    } while (!cas(i, i_plus_one, ptr));  
}
```

Read the value of the pointer into a local variable,

Atomic $i = i+1$;

```
void atomic_inc(int *ptr) {  
    int i, i_plus_one;  
    do {  
        i = *ptr;  
        i_plus_one = i + 1;  
    } while (!cas(i, i_plus_one, ptr));  
}
```

Do the difficult and complicated computation of taking a local variable and adding one to that variable and storing that value in another local variable,

Atomic `i = i+1;`

```
void atomic_inc(int *ptr) {  
    int i, i_plus_one;  
    do {  
        i = *ptr;  
        i_plus_one = i + 1;  
    } while (!cas(i, i_plus_one, ptr));  
}
```

and swap the pointer's value of `i` with `i + 1`.

Atomic `i = (i+1) % 32;`

```
void atomic_inc_mod_32(int *ptr) {  
    int i, new_i;  
    do {  
        i = *ptr;  
        new_i = i + 1;  
        new_i = new_i % 32;  
    } while (!cas(i, new_i, ptr));  
}
```

Here's a slightly more interesting thing to do that's actually not any more complicated in terms of making the operation atomic. Let's say we have a circular buffer with 32 elements, and a shared index into that buffer that we always want to wrap back around to 0 instead of walking off the end of the array.

Here, `new_i` is explicitly computed by two distinct operations but the overall structure looks the same.

TAS using CAS

```
void tas_loop(spinlock *m) {  
    do {  
        ;  
    } while (!cas(UNLOCKED, LOCKED, m));  
}
```

I said before that compare and swap is a more general version of test and set. I say this because you can use compare and swap to implement a test and set loop! Here we try to move the memory pointed to by ptr from an UNLOCKED state to a LOCKED state - if the CAS succeeds that means the previous value was UNLOCKED and we are now LOCKED. If it fails, that means we couldn't swap out an UNLOCKED value because it was different — that is, the lock was already set to LOCKED.

Read/Modify/Write

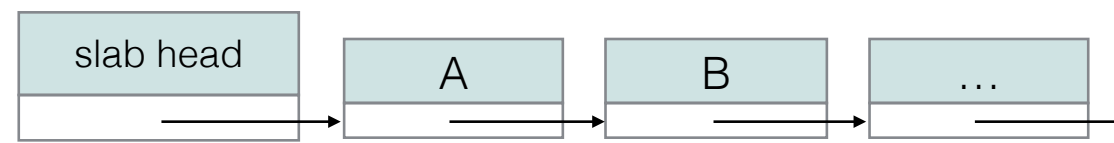
```
void atomic_inc_mod_32(int *ptr) {  
    int i, new_i;  
    do {  
        i = *ptr;                /* Read */  
        new_i = fancy_function(); /* Modify */  
    } while (!cas(i, new_i, ptr)); /* Write */  
}
```

Atomic operations that read the current state, modify that state in some way, and then attempt to write it back are often called Read/Modify/Write primitives.

Read/Modify/Write

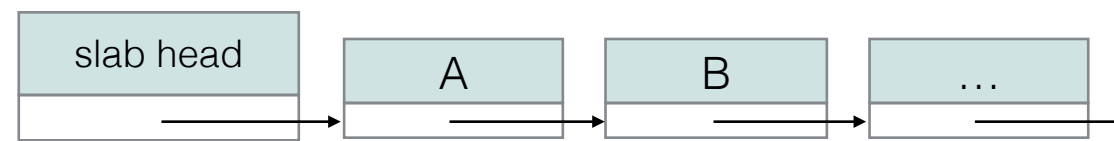
```
void atomic_inc_mod_32(int *ptr) {  
    int i, new_i;  
    do {  
        i = *ptr;                /* Read */  
        new_i = fancy_function(); /* Modify */  
    } while (!cas(i, new_i, ptr)); /* Write */  
                                   /* (or retry) */  
}
```

So far we've kind of glossed over what happens when the CAS returns false and we have to retry, by going back to the top of the loop. Let's talk about that.



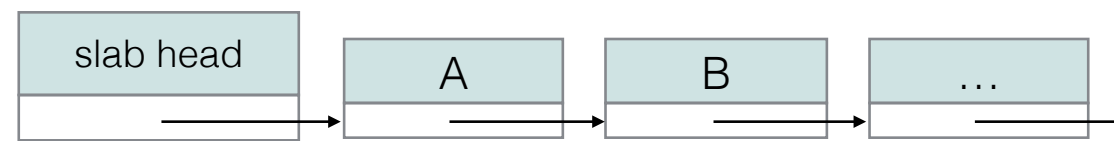
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head ));  
    return a;  
}
```

So, going back to the memory allocator, this is what the allocation function might look like in C.



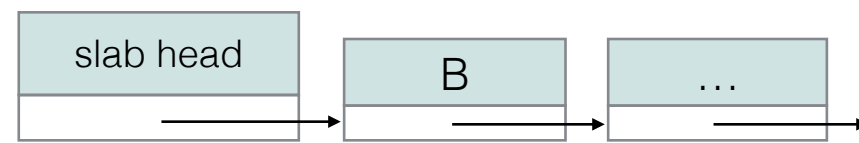
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head ));  
    return a;  
}
```

First, we read the state of our stack, namely the stack head, and its next element. So, variables A and B correspond to elements A and B in the diagram.

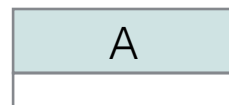


```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head ));  
    return a;  
}
```

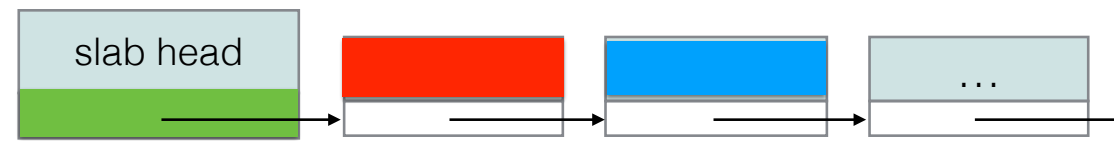
Next, we try to update the head of the stack, which is currently A, to B, using the CAS operation.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head ));  
    return a;  
}
```



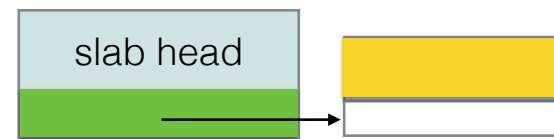
If we've succeeded, A has been popped off the list and its successor is now at the head of the list. But, what if the CAS returns false?



```
obj *allocate(slab *s) {
    obj *a, *b;
    do {
        a = s->head;
        if (a == NULL) return NULL;
        b = a->next;
    } while (!cas(a, b, &s->head));
    return a;
}
```

Cmpr ■ ***and*** * ■

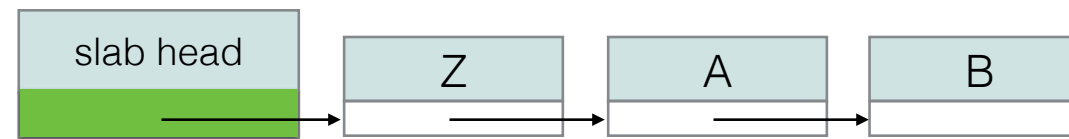
So if we go back to our color-coded explanation, if the CAS returns false then at some point between assigning a from s's head and the CAS,



```
obj *allocate(slab *s) {
    obj *a, *b;
    do {
        a = s->head;
        if (a == NULL) return NULL;
        b = a->next;
    } while (!cas(a, b, &s->head));
    return a;
}
```

Cmpr  **and** 

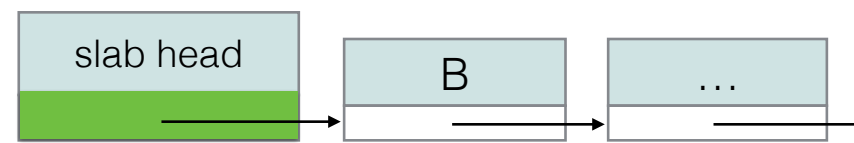
the head pointer was changed to a different value by another thread. That is this “yellow” element. Maybe a free happened and...



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```

Cmpr  ***and*** 

... A is now the second element in the list. Or, maybe another allocation happened,



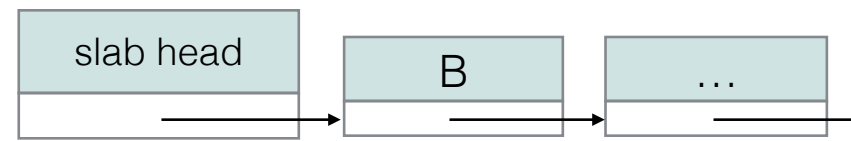
```
obj *allocate(slab *s) {
    obj *a, *b;
    do {
        a = s->head;
        if (a == NULL) return NULL;
        b = a->next;
    } while (!cas(a, b, &s->head));
    return a;
}
```

Cmpr ■ ***and*** ■

A got popped off, and B is now the head of the list (which is the thing we were trying to do!).

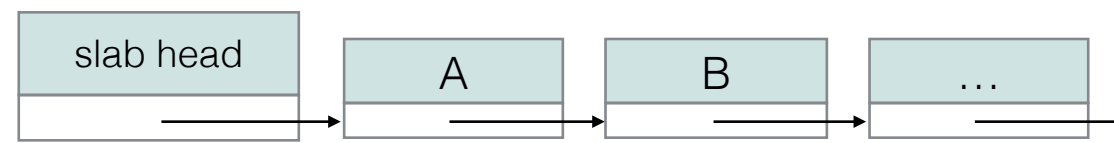
My friends, I think we have another race on our hands!

But, as before, because the CAS is a linearization point, we are able to detect that the state of the free pool changed from underneath us and so we loop back and re-read the state of the free list. But more significantly, because the CAS only returns false if it had changed between reading A and the CAS, even though we have to retry we know that some other thread succeeded. This makes this a lock-free implementation, in contrast to our implementation with the spinlock, where we had no choice to spin even though no one else was manipulating the lock object.



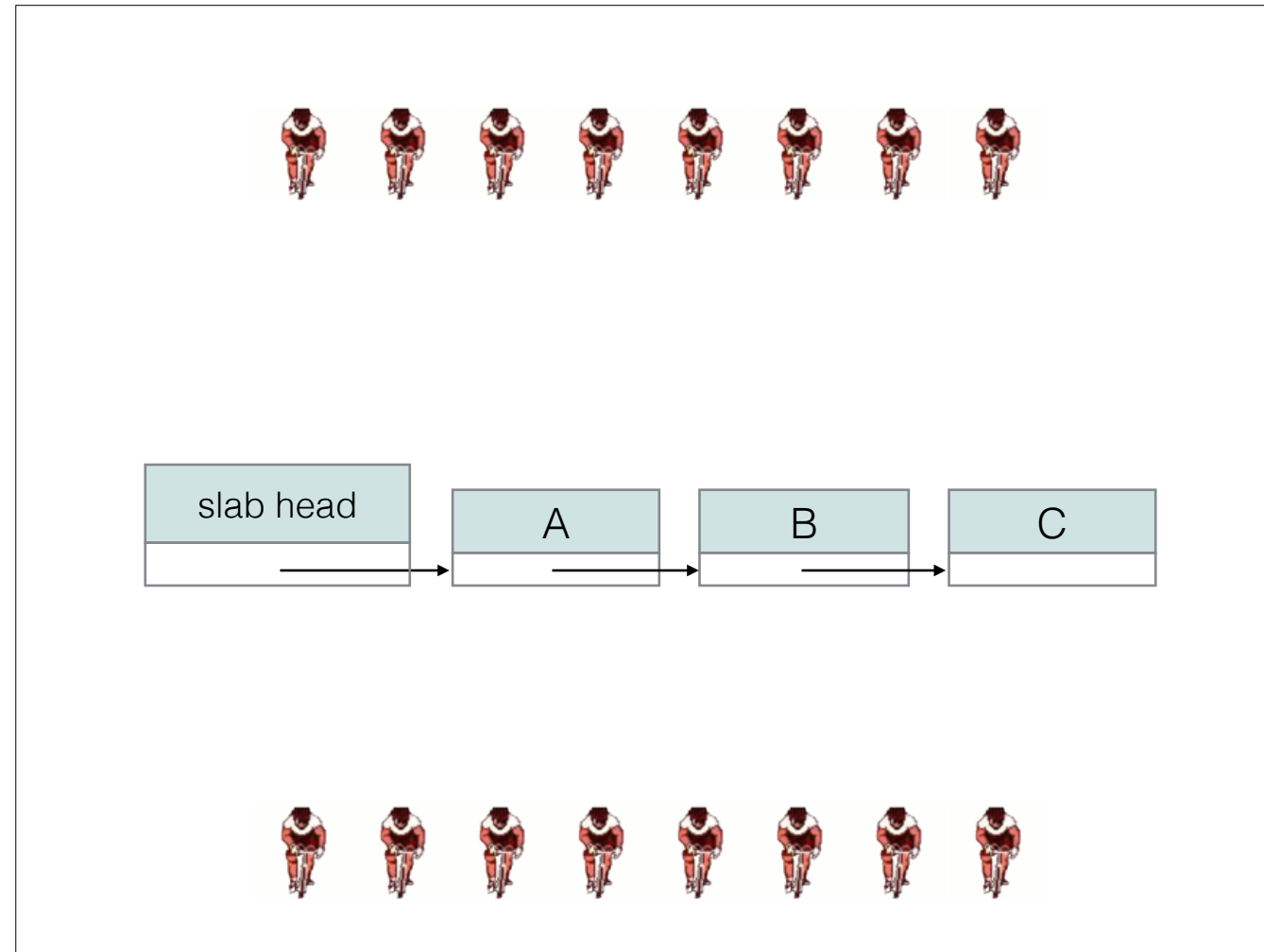
```
void free(slab *s, obj *o) {  
    do {  
        obj *t = s->head;  
        o->next = t;  
    } while (!cas(t, o, &s->head));  
}
```

Freeing follows a similar strategy. To free, we read the current stack head, assigning the current head to the next pointer of the object we are inserting back into our freelist...

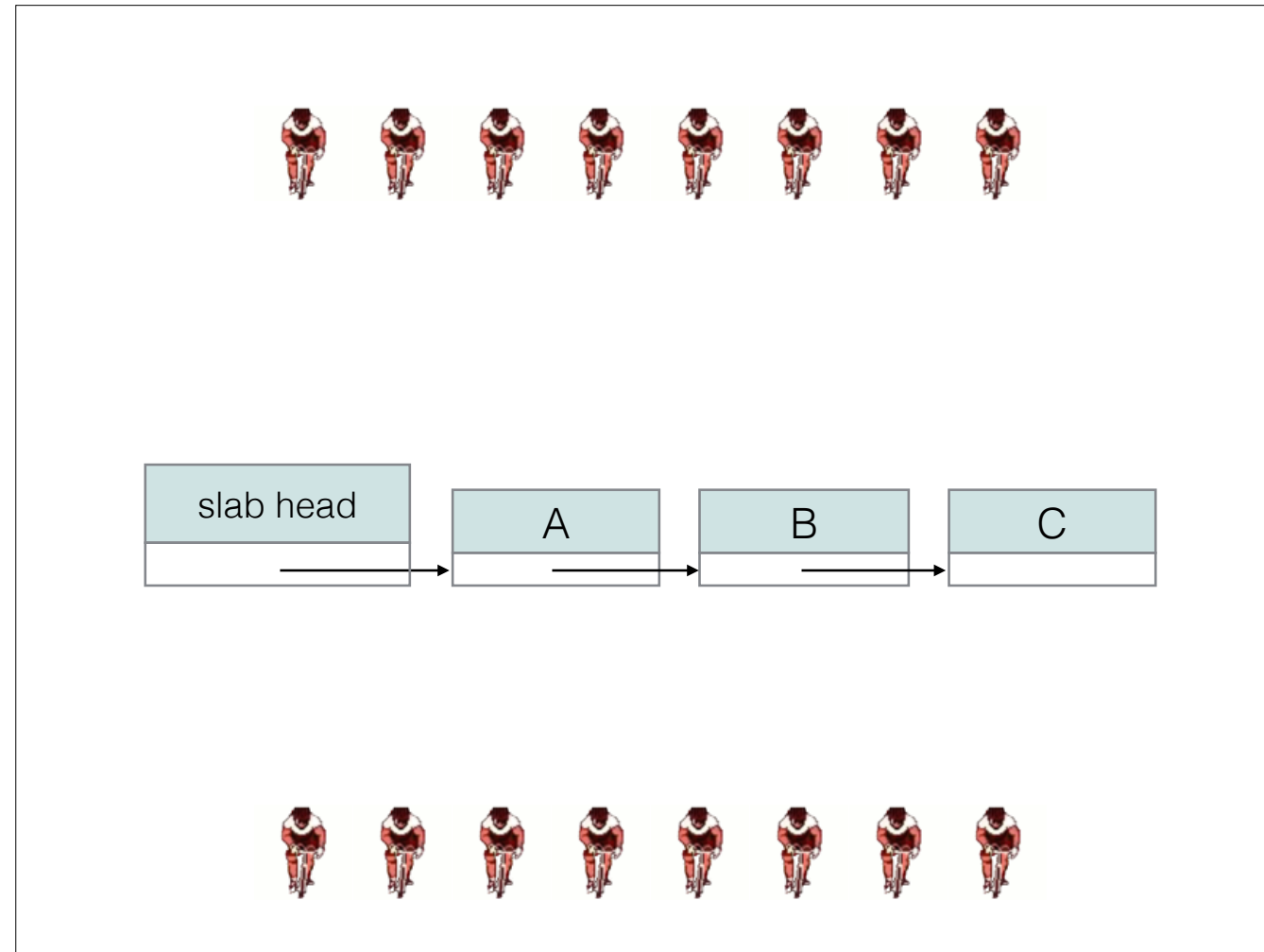


```
void free(slab *s, obj *o) {  
    do {  
        obj *t = s->head;  
        o->next = t;  
    } while (!cas(t, o, &s->head));  
}
```

And we swap our new object in. Again, if we fail, we simply refresh our head-of-list value for the T pointer and try again.



So this is great! We've defined allocate and free in terms of safe concurrent primitives, we can have tons of threads hammer this stack.



So this is great! We've defined allocate and free in terms of safe concurrent primitives, we can have tons of threads hammer this stack.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



So let's imagine that this thread is us, and we want to allocate an object.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



So let's imagine that this thread is us, and we want to allocate an object.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



Let's say we execute everything up to the CAS instruction: we inspect the head of the free list, get a pointer to its value, and also to the free list's next element.

Now remember that there are other threads operating on this data structure. So, it's totally possible for ...



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```

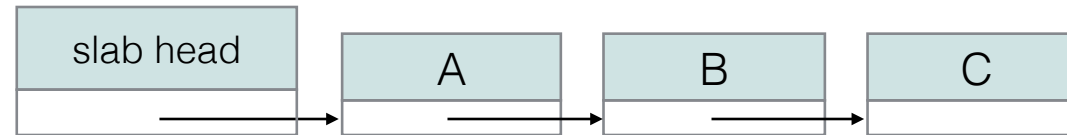


Let's say we execute everything up to the CAS instruction: we inspect the head of the free list, get a pointer to its value, and also to the free list's next element.

Now remember that there are other threads operating on this data structure. So, it's totally possible for ...



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```

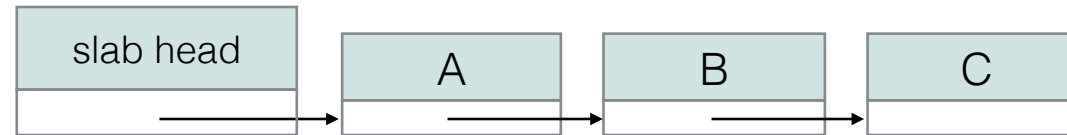


```
some_object = allocate(&shared_slab);
```

Another thread to come in and try to also allocate an object! And, because we haven't hit the linearization point yet, it can come in and



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```

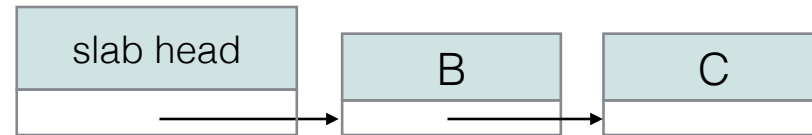


```
some_object = allocate(&shared_slab);
```

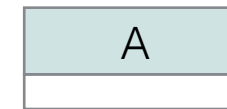
Another thread to come in and try to also allocate an object! And, because we haven't hit the linearization point yet, it can come in and



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



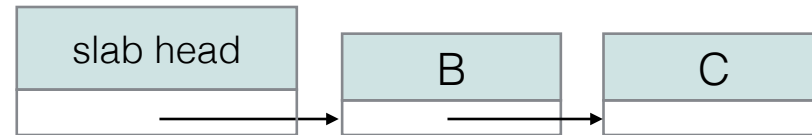
```
some_object = allocate(&shared_slab);
```



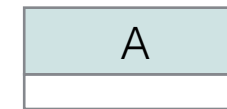
complete! That's ok, remember that the CAS will not let us swap A for B if it doesn't see A at the head of the list, so we haven't put the data structure in an inconsistent state. So, why don't we play this game again, and have another thread



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



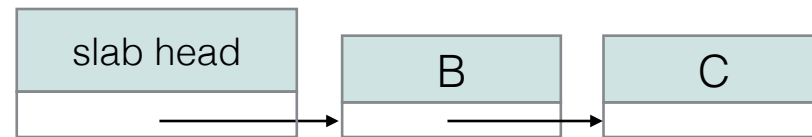
```
some_object = allocate(&shared_slab);
```



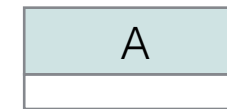
complete! That's ok, remember that the CAS will not let us swap A for B if it doesn't see A at the head of the list, so we haven't put the data structure in an inconsistent state. So, why don't we play this game again, and have another thread



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
some_object = allocate(&shared_slab);
```

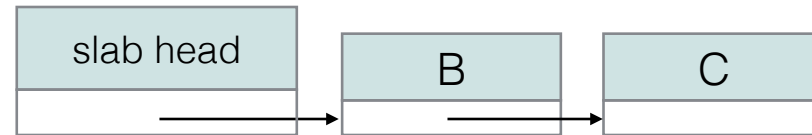


```
another_obj = allocate(&shared_slab);
```

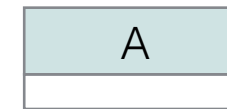
come in and do the same thing.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
some_object = allocate(&shared_slab);
```

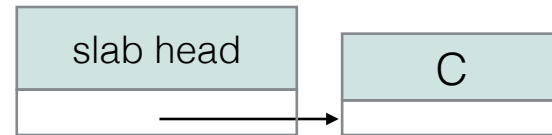


```
another_obj = allocate(&shared_slab);
```

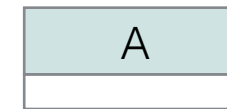
come in and do the same thing.



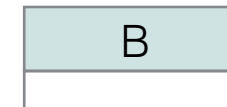
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



`some_object = allocate(&shared_slab);`



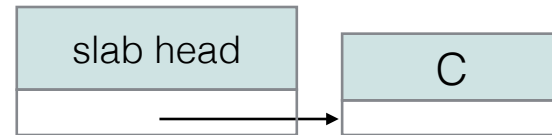
`another_obj = allocate(&shared_slab);`



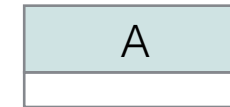
So this one now owns B. Meanwhile, let's say that



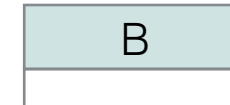
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



`some_object = allocate(&shared_slab);`



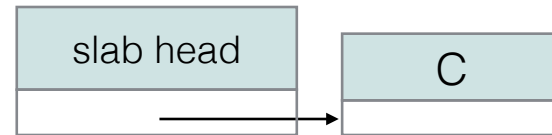
`another_obj = allocate(&shared_slab);`



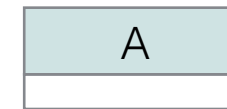
So this one now owns B. Meanwhile, let's say that



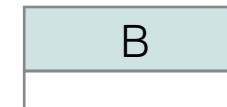
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



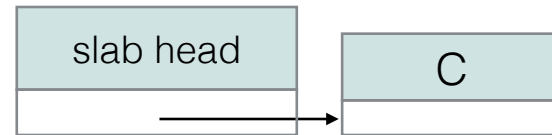
```
another_obj = allocate(&shared_slab);
```



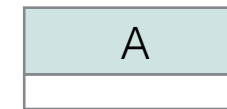
the thread that allocated A is done with the object and wants to free it back.



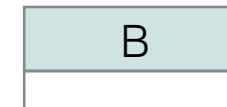
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



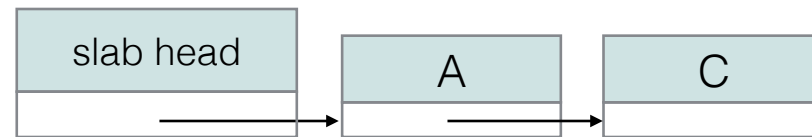
```
another_obj = allocate(&shared_slab);
```



the thread that allocated A is done with the object and wants to free it back.



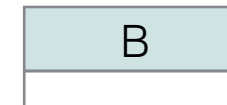
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



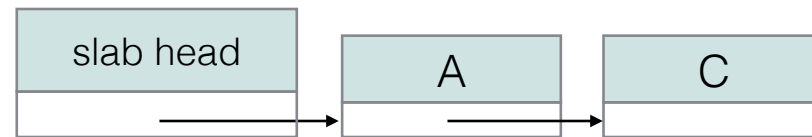
```
another_obj = allocate(&shared_slab);
```



Okay, done. Now here's where a problem can arise: Let's finally do our CAS operation.



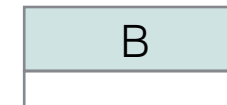
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



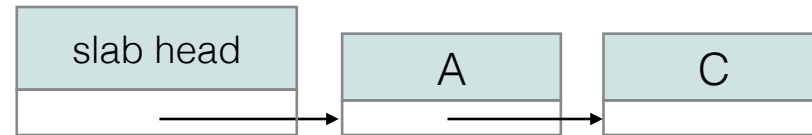
```
another_obj = allocate(&shared_slab);
```



Okay, done. Now here's where a problem can arise: Let's finally do our CAS operation.



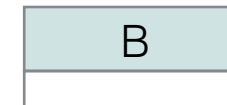
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



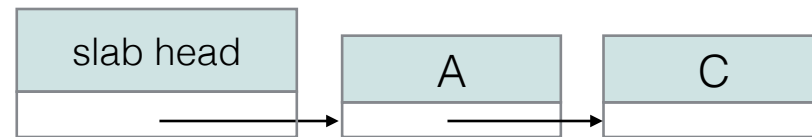
```
another_obj = allocate(&shared_slab);
```



Because A is now the head of the list again, as far as the CAS can tell we can safely swap A with its successor, which is now C, not B!



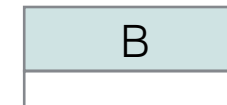
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



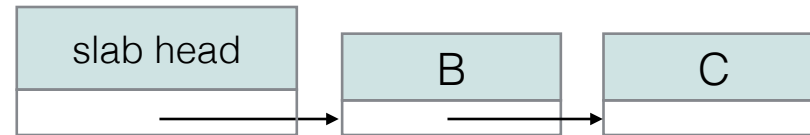
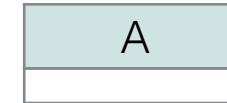
```
another_obj = allocate(&shared_slab);
```



Because A is now the head of the list again, as far as the CAS can tell we can safely swap A with its successor, which is now C, not B!



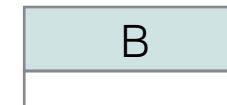
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



```
free(&shared_slab, some_object);
```



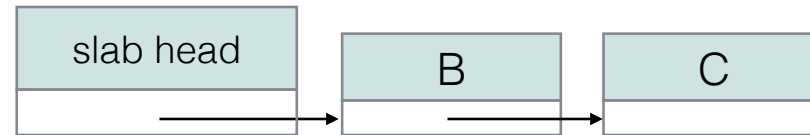
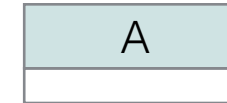
```
another_obj = allocate(&shared_slab);
```



So the CAS succeeds, but we've accidentally reinserted B into the stack. <click> So, now, B's memory is simultaneously seen to be freed and allocated. This is a classic concurrency bug called the ABA problem:



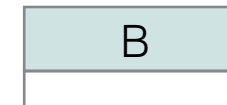
```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```



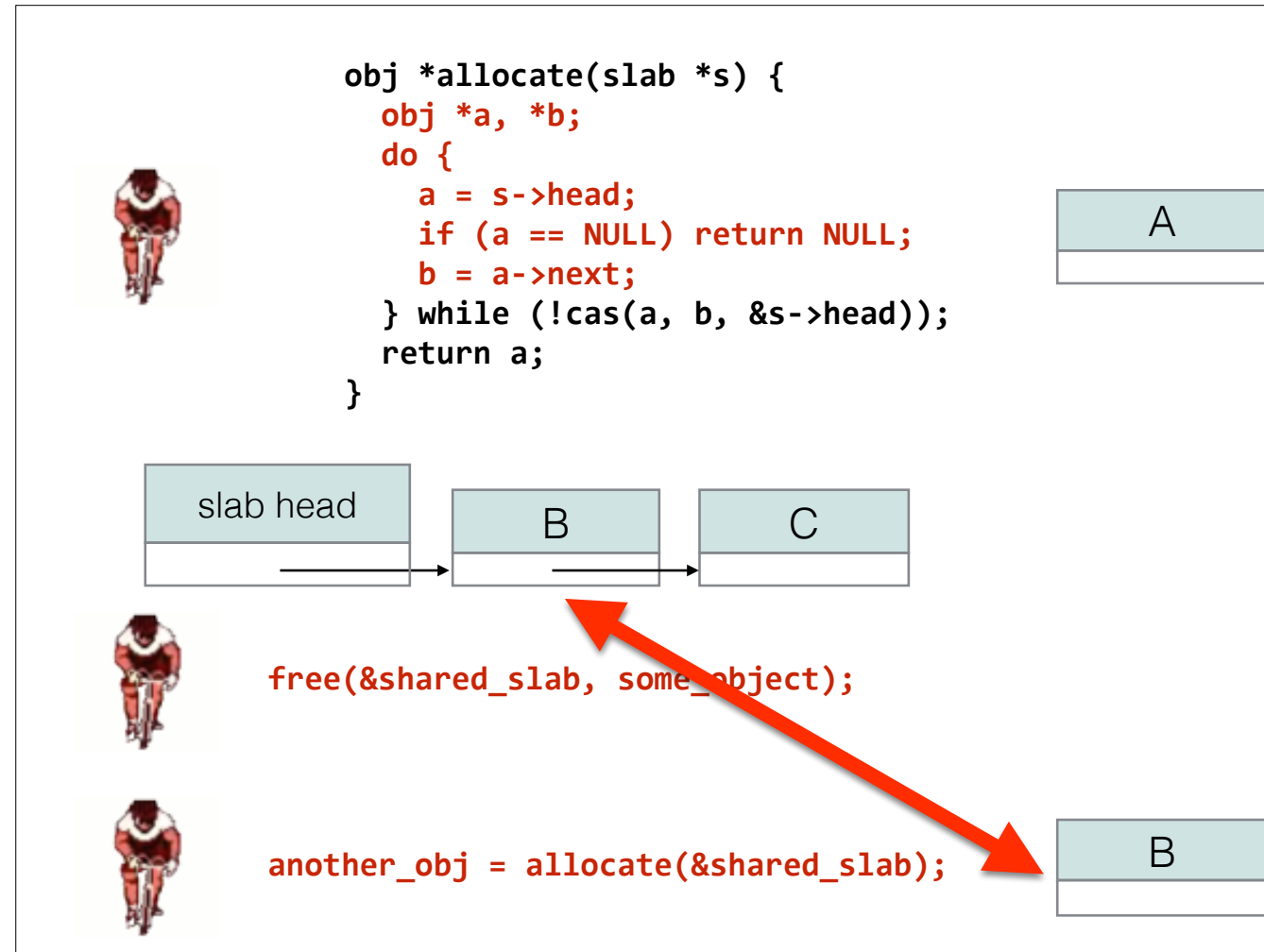
```
free(&shared_slab, some_object);
```



```
another_obj = allocate(&shared_slab);
```



So the CAS succeeds, but we've accidentally reinserted B into the stack. <click> So, now, B's memory is simultaneously seen to be freed and allocated. This is a classic concurrency bug called the ABA problem:



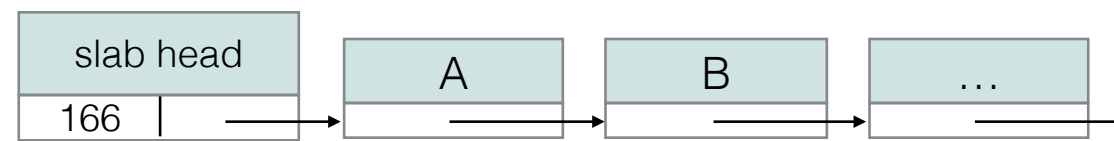
So the CAS succeeds, but we've accidentally reinserted B into the stack. <click> So, now, B's memory is simultaneously seen to be freed and allocated. This is a classic concurrency bug called the ABA problem:

The ABA Problem

“A reference about to be modified by a CAS changes from a to b and back to a again. As a result, the CAS succeeds even though its effect on the data structure has changed and no longer has the desired effect.” —Herlihy & Shavit, p. 235

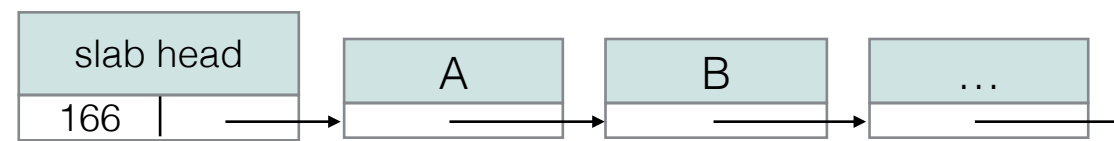
Note that this is only a problem because our workload requires support for multiple concurrent allocation calls. With a single allocator process operating concurrently with respect to any number of concurrent freeing processes, this problem can't occur. It is helpful to distinguish the requirements of your workload to determine whether you need to solve the ABA problem at all — solving it adds memory and time overhead.

But, for our purposes, it's something we have to solve.



```
obj *allocate(slab *s) {  
    obj *a, *b;  
    do {  
        a = s->head;  
        if (a == NULL) return NULL;  
        b = a->next;  
    } while (!cas(a, b, &s->head));  
    return a;  
}
```

A common approach is to encode some additional state with the structure. Introducing a generation counter, keeping track of the number of times we've allocated within this freelist, provides additional state: if we pop A, push B and push A, the generation counter would have changed inbetween popping A and pushing A again.



```
obj *allocate(slab *s) {
    slab orig, update;
    do {
        orig.gen = s.gen;
        orig.head = s.head;
        if (!orig.head) return NULL;
        update.gen = orig.gen + 1;
        update.head = orig.head->next;
    } while (!dcas(&orig, &update, s));
    return orig.head;
}
```

So rather than having local object pointers A and B we have temporary slab structures. We need that because we are using a new compare and swap operation. “dcas” is a double-wide CAS that lets us atomically swap two adjacent values, so let’s assume the head and the generation are adjacent in the slab structure definition. This solves the ABA problem because even though the A pointer would match in the CAS, the generation count would have changed.

<TODO: why gen before head>

```
free(slab *s, obj *o) {  
    do {  
        obj *t = s->head;  
        o->next = t;  
    } while (!cas(t, o, &s->head));  
}
```

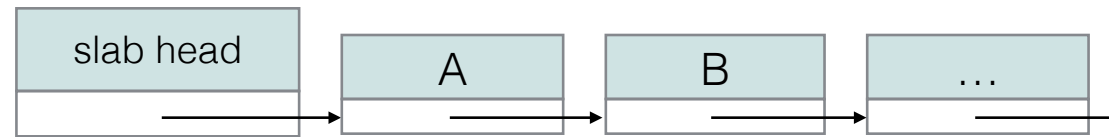
Freeing stays the same. A free operation *only* manipulates the head pointer, and any allocation operation necessarily manipulates the head pointer and its next value. There's no "B" element in this case, only an "A".

```
obj *allocate(slab *s) {  
    lock(&allocator_lock);  
    obj *a = s->head;  
    if (a == NULL) return NULL;  
    s->head = a->next;  
    unlock(&allocator_lock);  
    return a;  
}
```

```
void free(slab *s, obj *o) {  
    lock(&allocator_lock);  
    o->next = s->head;  
    s->head = o;  
    unlock(&allocator_lock);  
}
```

There's one final wrench that I would like to throw into the works, and the reason why we haven't needed to talk about it yet is that so far I've been silently assuming that we are all working on x86 hardware, so therefore I've been silently assuming that our concurrent programs will behave as they do on x86.

Let's look at the allocation and free implementation that uses spinlocks again. Given what I've told you, this is all correct. Mutual exclusion is guaranteed. But, given that I just said a sentence that begins with "given what I've told you, this is correct", there's clearly more that I have to tell you.

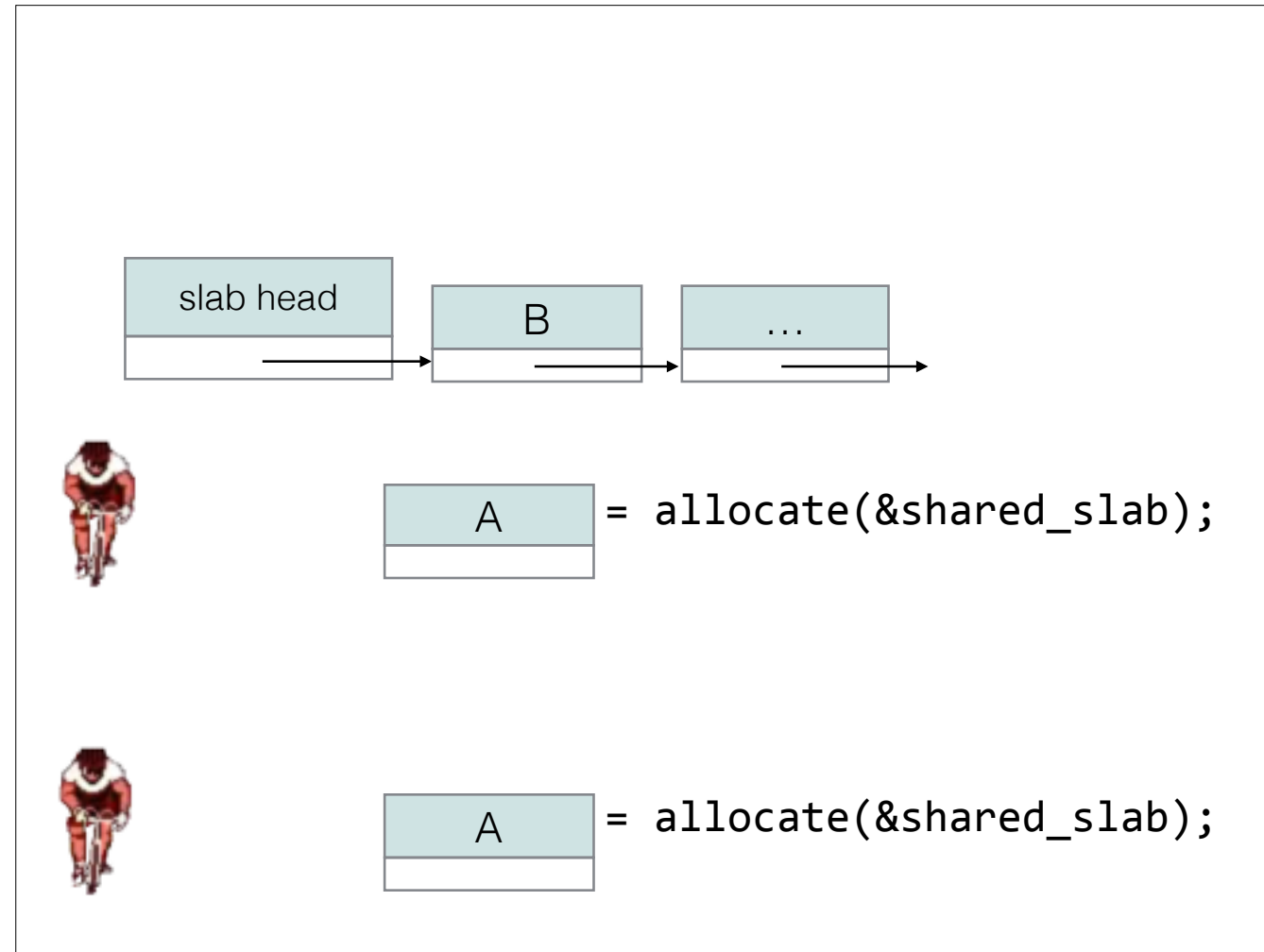


```
obj *o = allocate(&shared_slab);
```



```
obj *o = allocate(&shared_slab);
```

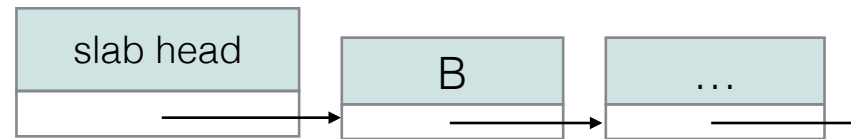
So given two threads, can it ever be the case that some ordering of operations that would cause



two threads to have the same block of memory returned to them? We should hope not! However...

In the event that you're running on different architectures, like ARM, or POWER, or ALPHA... the code as we've written may not work, <click> and this is because of the lack of a memory ordering primitive called a memory barrier.

Memory barriers



A = `allocate(&shared_slab);`



A = `allocate(&shared_slab);`

two threads to have the same block of memory returned to them? We should hope not! However...

In the event that you're running on different architectures, like ARM, or POWER, or ALPHA... the code as we've written may not work, <click> and this is because of the lack of a memory ordering primitive called a memory barrier.



```
lock(&allocator_lock);  
obj *a = s->head;  
...
```



```
lock(&allocator_lock);  
obj *a = s->head;  
...
```

Let's say these threads enter Allocate together. Well, what's happening under the hood as far as the CPU is concerned is a series of instructions, so what we actually have is



```
while (atomic_tas(m, LOCKED) == LOCKED)
    snooze();
obj *a = s->head;
...
```



```
while (atomic_tas(m, LOCKED) == LOCKED)
    snooze();
obj *a = s->head;
...
```

that test and set, and on success, reading the head of the list...

okay, I'm gonna do a thing, don't freak out. I believe in you, we're all Surge attendees, so I think it'll be ok to do this.

```

;;; IN lock()
lock_loop:
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?

    BEQ lock_done           ; Yes: we are all done
    BL snooze               ; No: Call snooze()...
    B lock_loop             ; ...then loop again
lock_done:
    B LR                    ; return

;;; IN allocate()
    LDR R0, [R1, 4]         ; a = s->head

```

So this is what some ARM assembly might look like. I say “might” because I’m a C programmer and not an optimizing ARM compiler, but for our purposes it’s sufficiently correct, I think. If you’re not used to staring at assembly, don’t worry, there are only two parts you need to know about:

```

;;; IN lock()
lock_loop:
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?

    BEQ lock_done           ; Yes: we are all done
    BL snooze               ; No: Call snooze()...
    B lock_loop             ; ...then loop again
lock_done:
    B LR                    ; return

;;; IN allocate()
    LDR R0, [R1, 4]         ; a = s->head

```

These three instructions, comprising of a load register exclusive, store register exclusive, and a comparison is our atomic test and set. If you're interested, on ARM and other RISCey architectures, we implement this using concurrent primitives that don't exist on x86 called load exclusive and conditional store.

```

;;; IN lock()
lock_loop:
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?
    BEQ lock_done          ; Yes: we are all done
    BL snooze              ; No: Call snooze()...
    B lock_loop            ; ...then loop again
lock_done:
    B LR                   ; return

;;; IN allocate()
    LDR R0, [R1, 4]         ; a = s->head

```

And this load down at the bottom is us back in the allocate function, reading the head of the slab after the lock function returns. So remember, this part has to be in the critical section.

| | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache/Pipeline? |
|----------------------|------------------------------|-------------------------------|--------------------------------|-------------------------------|---|--|----------------------------|--|
| Alpha | Y | Y | Y | Y | Y | Y | Y | Y |
| AMD64 | | | | Y | | | | |
| ARMv7-A/R | Y | Y | Y | Y | Y | Y | | Y |
| IA64 | Y | Y | Y | Y | Y | Y | | Y |
| (PA-RISC) | Y | Y | Y | Y | | | | |
| PA-RISC CPUs | | | | | | | | |
| POWER™ | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC RMO) | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC PSO) | | | Y | Y | | Y | | Y |
| SPARC TSO | | | | Y | | | | Y |
| x86 | | | | Y | | | | Y |
| (x86 OOSTore) | Y | Y | Y | Y | | | | Y |
| zSeries [®] | | | | Y | | | | Y |

Table C.5: Summary of Memory Ordering

McKenney , p. 504

Here's a table from Paul McKenzie parallel programming book - rows are different processor architectures and columns are the different kinds of memory instruction reordering s that the processor is allowed to do.

| | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache/Pipeline? |
|---------------|------------------------------|-------------------------------|--------------------------------|-------------------------------|---|--|----------------------------|--|
| Alpha | Y | Y | Y | Y | Y | Y | Y | Y |
| AMD64 | | | | Y | | | | |
| ARMv7-A/R | Y | Y | Y | Y | Y | Y | | Y |
| IA64 | Y | Y | Y | Y | Y | Y | | Y |
| (PA-RISC) | Y | Y | Y | Y | | | | |
| PA-RISC CPUs | | | | | | | | |
| POWER™ | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC RMO) | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC PSO) | | | Y | Y | | Y | | Y |
| SPARC TSO | | | | Y | | | | Y |
| x86 | | | | Y | | | | Y |
| (x86 OOSTore) | Y | Y | Y | Y | | | | Y |
| zSeries® | | | | Y | | | | Y |

Table C.5: Summary of Memory Ordering

McKenney , p. 504

So, x86 is pretty restrictive, the only thing it can do is move a store after an unrelated load. This can bite us in a few specific scenarios but in general this is pretty reasonable. we say that it has a strong memory model.

| | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache/Pipeline? |
|---------------|------------------------------|-------------------------------|--------------------------------|-------------------------------|---|--|----------------------------|--|
| Alpha | Y | Y | Y | Y | Y | Y | Y | Y |
| AMD64 | | | | Y | | | | |
| ARMv7-A/R | Y | Y | Y | Y | Y | Y | | Y |
| IA64 | Y | Y | Y | Y | Y | Y | | Y |
| (PA-RISC) | Y | Y | Y | Y | | | | |
| PA-RISC CPUs | | | | | | | | |
| POWER™ | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC RMO) | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC PSO) | | | Y | Y | | Y | | Y |
| SPARC TSO | | | | Y | | | | Y |
| x86 | | | | Y | | | | Y |
| (x86 OOSTore) | Y | Y | Y | Y | | | | Y |
| zSeries® | | | | Y | | | | Y |

Table C.5: Summary of Memory Ordering

McKenney , p. 504

Arm, however, is a lot more permissive in terms of what it's able to reorder. In particular, notice that

| | Loads Reordered After Loads? | Loads Reordered After Stores? | Stores Reordered After Stores? | Stores Reordered After Loads? | Atomic Instructions Reordered With Loads? | Atomic Instructions Reordered With Stores? | Dependent Loads Reordered? | Incoherent Instruction Cache/Pipeline? |
|---------------|------------------------------|-------------------------------|--------------------------------|-------------------------------|---|--|----------------------------|--|
| Alpha | Y | Y | Y | Y | Y | Y | Y | Y |
| AMD64 | | | | Y | | | | |
| ARMv7-A/R | Y | Y | Y | Y | Y | Y | | Y |
| IA64 | Y | Y | Y | Y | Y | Y | | Y |
| (PA-RISC) | Y | Y | Y | Y | | | | |
| PA-RISC CPUs | | | | | | | | |
| POWER™ | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC RMO) | Y | Y | Y | Y | Y | Y | | Y |
| (SPARC PSO) | | | Y | Y | | Y | | Y |
| SPARC TSO | | | | Y | | | | Y |
| x86 | | | | Y | | | | Y |
| (x86 OOSTore) | Y | Y | Y | Y | | | | Y |
| zSeries® | | | | Y | | | | Y |

Table C.5: Summary of Memory Ordering

McKenney , p. 504

Arm, however, is a lot more permissive in terms of what it's able to reorder. In particular, notice that ARM is allowed to move the order of unrelated load operations around, and can change the relative ordering of an atomic instruction and an unrelated load.

Home > Memory Type and Memory Ordering > Memory ordering

2.2. Memory ordering

The ARMv7-M and ARMv6-M architectures support a wide range of implementations, from low-end microcontrollers through to high-end, superscalar, *System on Chip* (SoC) designs. To do so, the architectures permit, and only mandate, a weakly-ordered memory model. This model defines the three memory types, each with different properties and ordering requirements.

To support high-end implementations, the architecture does not require the ordering of transactions between Normal and Strongly-ordered memory, and it does not mandate the ordering of load/store instructions with respect to either instruction prefetch or instruction execution.

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions. This is because:

- the processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence
- the processor can have multiple bus interfaces
- memory or devices in the memory map can be on different branches of an interconnect
- some memory accesses are buffered or speculative.

This isn't a bug or incorrect behavior, there's good reason to allow a processor to do this. The CPU can make just-in-time memory operation decisions that a static ahead-of-time compiler wouldn't be able to necessarily reason about. So, if we look at the ARM documentation site we see that the processor can reorder memory accesses to improve efficiency, providing that this does not affect the behaviour of the instruction sequence.

Well the problem is that

```

;;; IN lock()
lock_loop:
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?

    BEQ lock_done           ; Yes: we are all done
    BL snooze               ; No: Call snooze()...
    B lock_loop             ; ...then loop again
lock_done:
    B LR                    ; return

;;; IN allocate()
    LDR R0, [R1, 4]         ; a = s->head

```

An instruction sequence is all about how the program behaves in a single-threaded environment. So, from the processor's perspective, because the lock and the freelist are totally distinct from each other, there's nothing preventing them from

<click> flipping the order of these loads. And,

```

;;; IN lock()
lock_loop:

    LDR R0, [R1, 4]          ; a = s->head

    BEQ lock_done           ; Yes: we are all done
    BL  snooze              ; No: Call snooze()...
    B   lock_loop           ; ...then loop again
lock_done:
    B   LR                  ; return

;;; IN allocate()
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?

```

An instruction sequence is all about how the program behaves in a single-threaded environment. So, from the processor's perspective, because the lock and the freelist are totally distinct from each other, there's nothing preventing them from

<click> flipping the order of these loads. And,

Home > Memory Type and Memory Ordering > Memory ordering

2.2. Memory ordering

The ARMv7-M and ARMv6-M architectures support a wide range of implementations, from low-end microcontrollers through to high-end, superscalar, *System on Chip* (SoC) designs. To do so, the architectures permit, and only mandate, a weakly-ordered memory model. This model defines the three memory types, each with different properties and ordering requirements.

To support high-end implementations, the architecture does not require the ordering of transactions between Normal and Strongly-ordered memory, and it does not mandate the ordering of load/store instructions with respect to either instruction prefetch or instruction execution.

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions. This is because:

- the processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence
- the processor can have multiple bus interfaces
- memory or devices in the memory map can be on different branches of an interconnect
- some memory accesses are buffered or speculative.

because some memory accesses can be done speculatively, these reordering can even happen if there's a conditional in between!



```
obj *a = s->head;  
lock(&allocator_lock);  
...
```



```
obj *a = s->head;  
lock(&allocator_lock);  
...
```

So, from our perspective, what we observe to have happened was this. This is clearly not what we want - reordering has pulled the read of the head pointer out of the critical section.



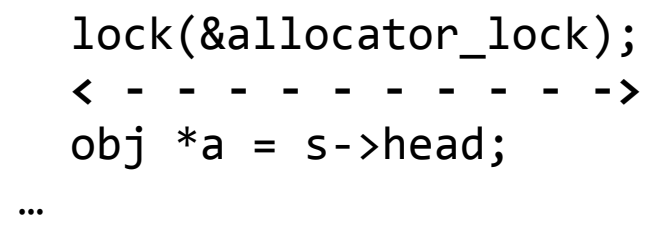
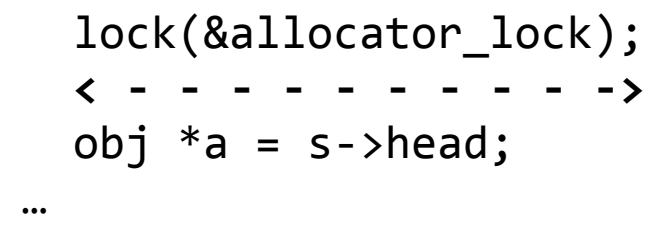
```
lock(&allocator_lock);  
obj *a = s->head;  
...
```



```
lock(&allocator_lock);  
obj *a = s->head;  
...
```

So how can we get ourselves out of this mess?

What we need to do is to



Draw a line in the sand. Any memory operations above this line can be reordered by the CPU, but only up to this line. They may not be moved below.

Cortex-M3 Devices Generic User Guide

Home > The Cortex-M3 Instruction Set > Miscellaneous instructions > DMB

3.10.3. DMB

Data Memory Barrier.

Syntax

`DMB{cond}`

where:

`cond`

Is an optional condition code, see [Conditional execution](#).

Operation

`DMB` acts as a data memory barrier. It ensures that all explicit memory accesses that appear, in program order, before the `DMB` instruction are completed before any explicit memory accesses that appear, in program order, after the `DMB` instruction. `DMB` does not affect the ordering or execution of instructions that do not access memory.

Condition flags

This instruction does not change the flags.

Examples

```
DMB ; Data Memory Barrier
```

As it turns out, there's such an instruction to draw such a line on ARM! Actually, on most processors there's something like this. It's called the "data memory barrier", and the description is exactly what we said. All memory accesses before the DMB instruction are completed before any memory accesses after the DMB instruction.

```

;;; IN lock()
lock_loop:
    LDREX R5, [m]           ; TAS: fetch. . .
    STREXEQ R5, LOCKED, [m] ; TAS: . . . and set
    CMPEQ R5, #0            ; Did we succeed?
    BEQ lock_done           ; Yes: we are all done
    BL snooze               ; No: Call snooze()...
    B lock_loop             ; ...then loop again
lock_done:
    DMB                    ; Ensure all previous reads
                           ; have been completed
    B LR                   ; return

;;; IN unlock()
    MOV R0, UNLOCKED
    DMB                    ; Ensure all previous reads have
                           ; been completed
    STR R0, LR

```

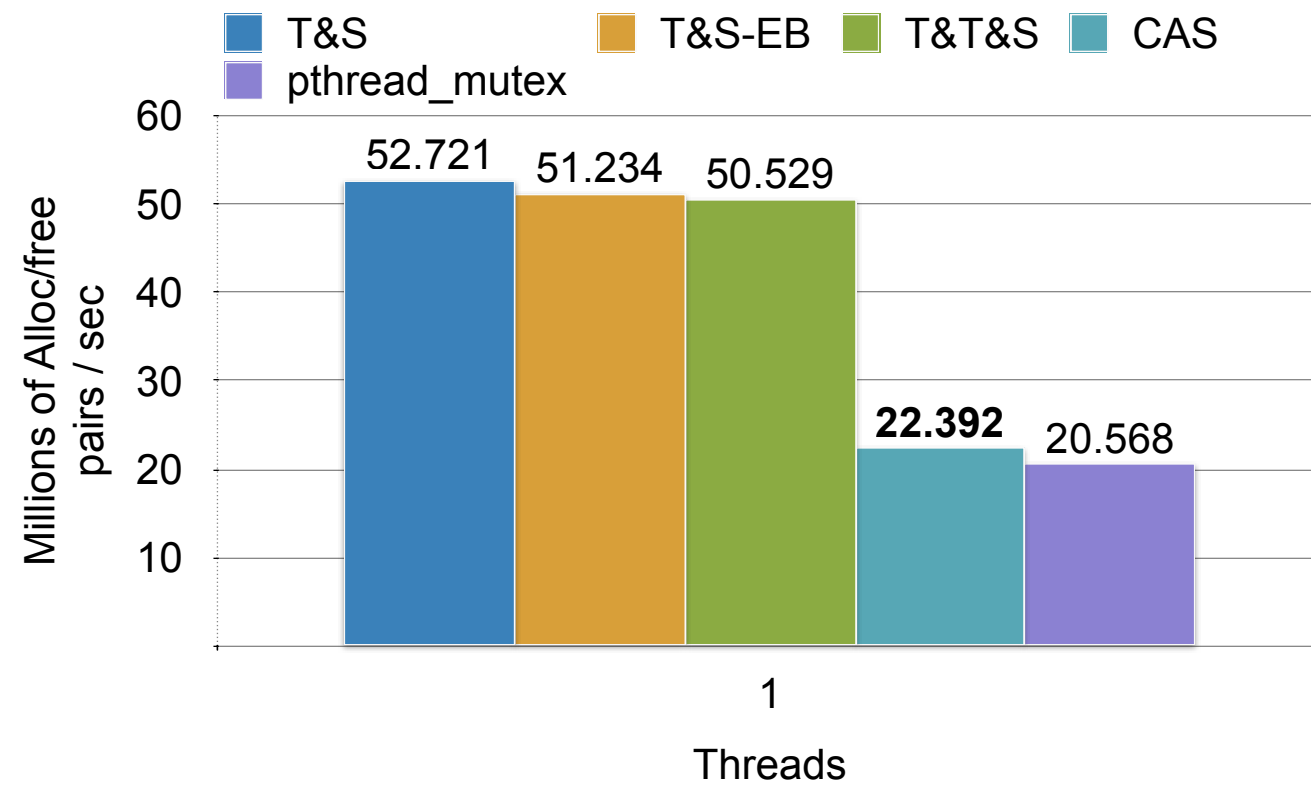
So by adding memory barriers into our lock and unlock functions, we tell the CPU that there's an inter-thread ordering dependency that it's not allowed to violate. <TODO explain better I am tired>

Allocator performance

```
nathan~$ cat /proc/cpuinfo | grep "physical.*0" | wc -l
16
nathan~$ cat /proc/cpuinfo | grep "model name" | uniq
model name : Intel(R) Xeon(R) CPU E5-2690 0 @ 2.90GHz
```

Now let's show a few micro benchmarks of our allocator, where we tried to find the max throughput for the collector - how many allocation/free pairs can we do per second in a given run?. For reference, these tests were run on one of Fastly's cache nodes — not while serving traffic, don't worry — a 16 core Xeon.

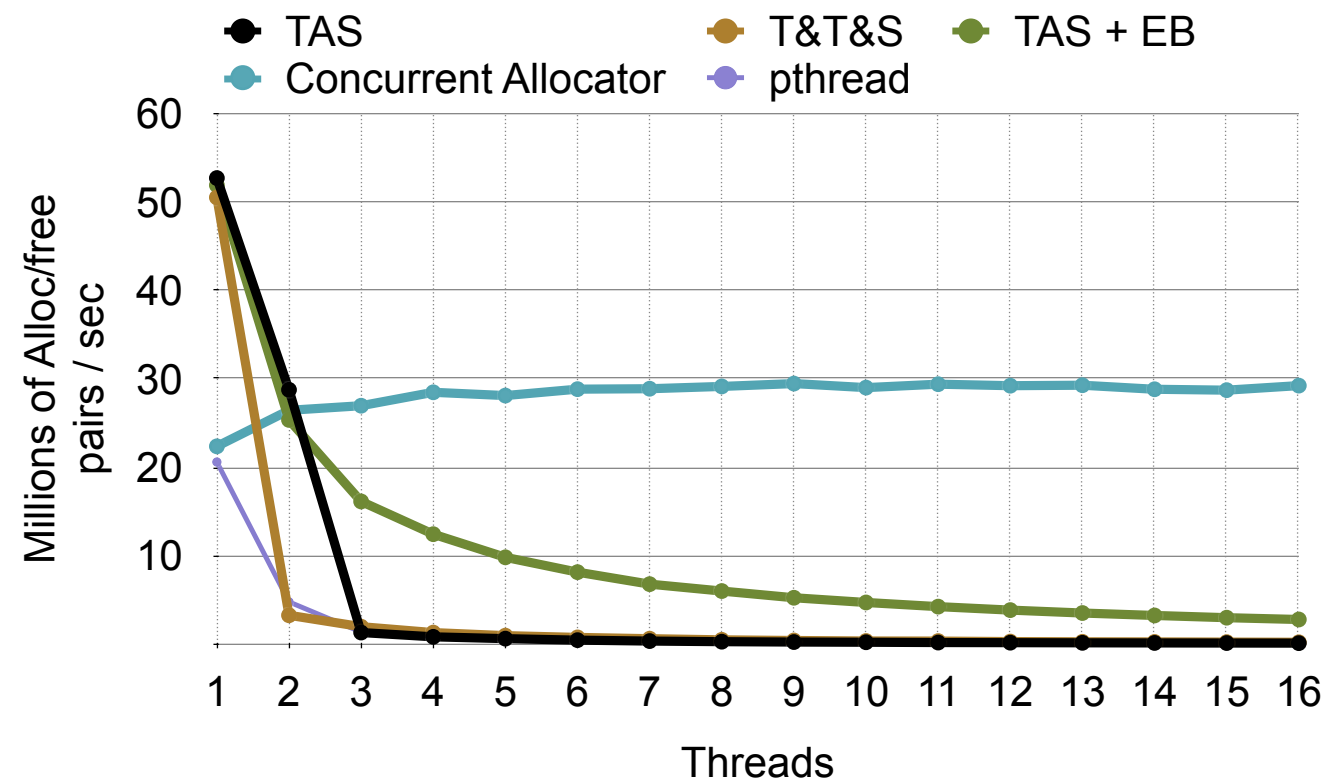
Allocator Throughput



In this test we guarantee proper access to the free list using either the test and test and set lock, the tas with exponential backoff, a pthread_mutex, and our CAS operation.

So when the allocator runs sequentially, with only one thread, it doesn't look like we do very well. We perform about as well as with locking with pthread mutex, but the test and set locks smoke us! It seems like maybe the CAS instruction, which is logically more complicated than a lock's test and set, might be more trouble than it's worth.

Allocator Throughput



But, when we scale up to a maximum of 16 cores, we find that the concurrent allocator using CAS remains reasonably stable with its throughput as the number of cores increase. Indeed, notice a slight uptick in throughput as the number of cores increase, from 22 million and change to close to 29 and change. In other words, from going from one core to two we are able to do (marginally) more things!

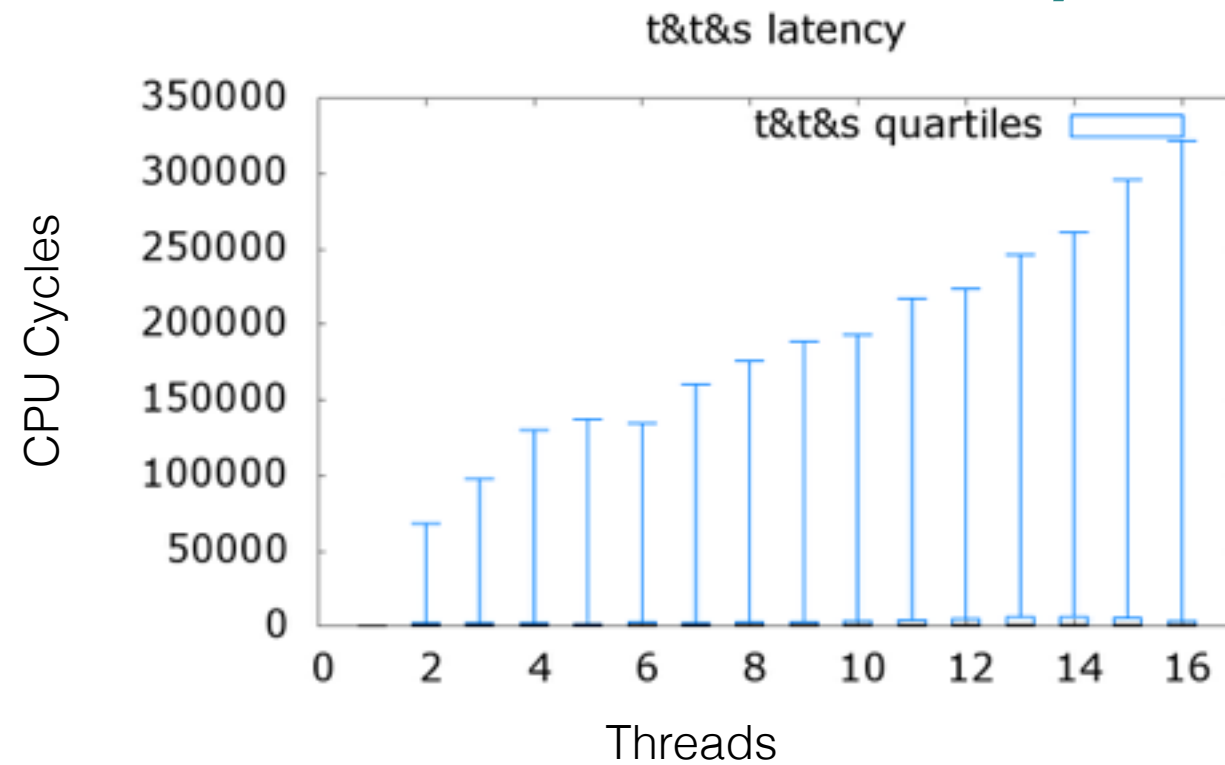
This is the sort of thing that we want to see with a lock-free algorithm: even if a CAS fails for us it's always succeeding for someone else, so our throughput doesn't go down the drain as the number of people trying to modify the freelist increases.

I do want to point out that we're being unfair to pthread_mutex here. As we all know when we build production-grade software there are goals apart from pure performance, and the pthread library has lots of great features like OS interoperability, bookkeeping debugging data, and so on, that the spin locks don't. So, don't take this graph as an apples-to-apples comparison or leave with the opinion that you should roll your own replacement, we've included it here out of interest's sake.

Allocator latency

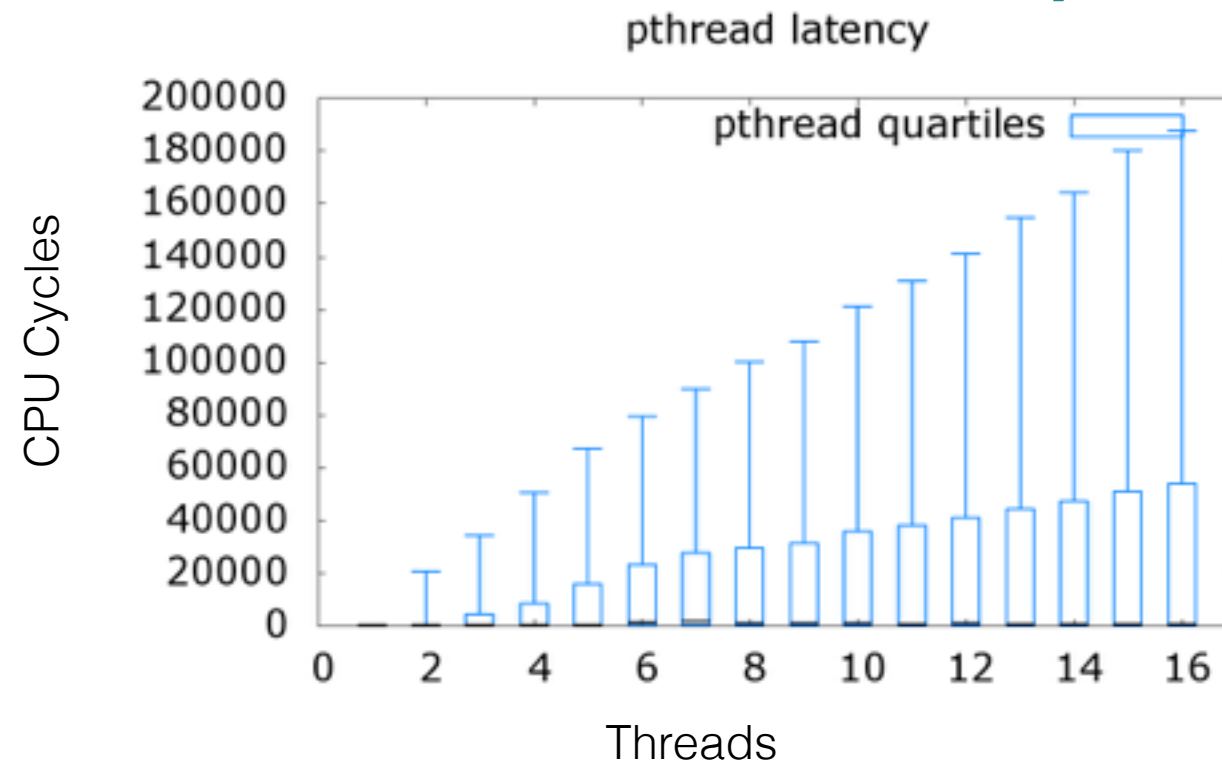
Next let me show you some latency distributions. Even though throughput is an important thing to know, we'd also like to know something about how much variation to expect for an operation to succeed. Because locks don't provide guarantees of progress, we expect a high amount of jitter in how long it takes to acquire a lock. These numbers are plotted with the top 1% of samples removed and show how long (in CPU cycles) it took to acquire a lock, sampled 1 million times.

Allocator latency



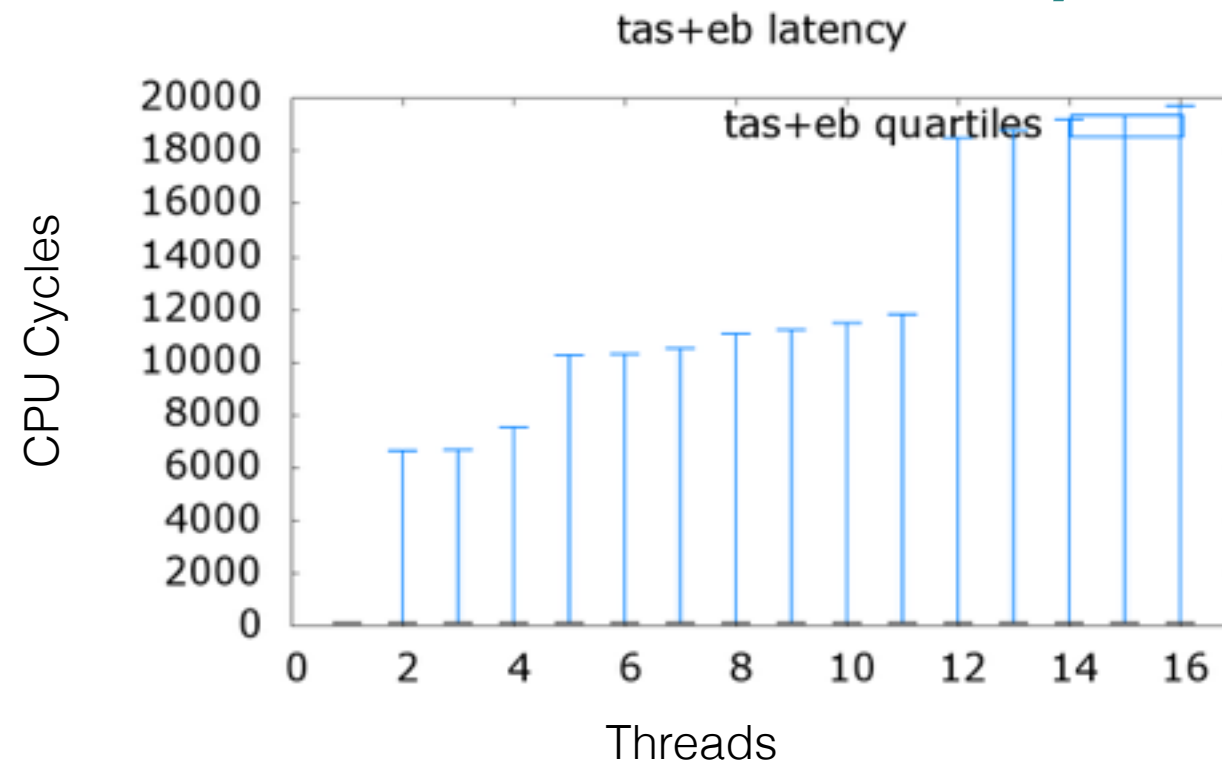
Even though this is nominally a box-and-whiskers plot the boxes are all but invisible. On 16 cores we see upwards of 300,000 cycles for an allocation using test and test and set

Allocator latency



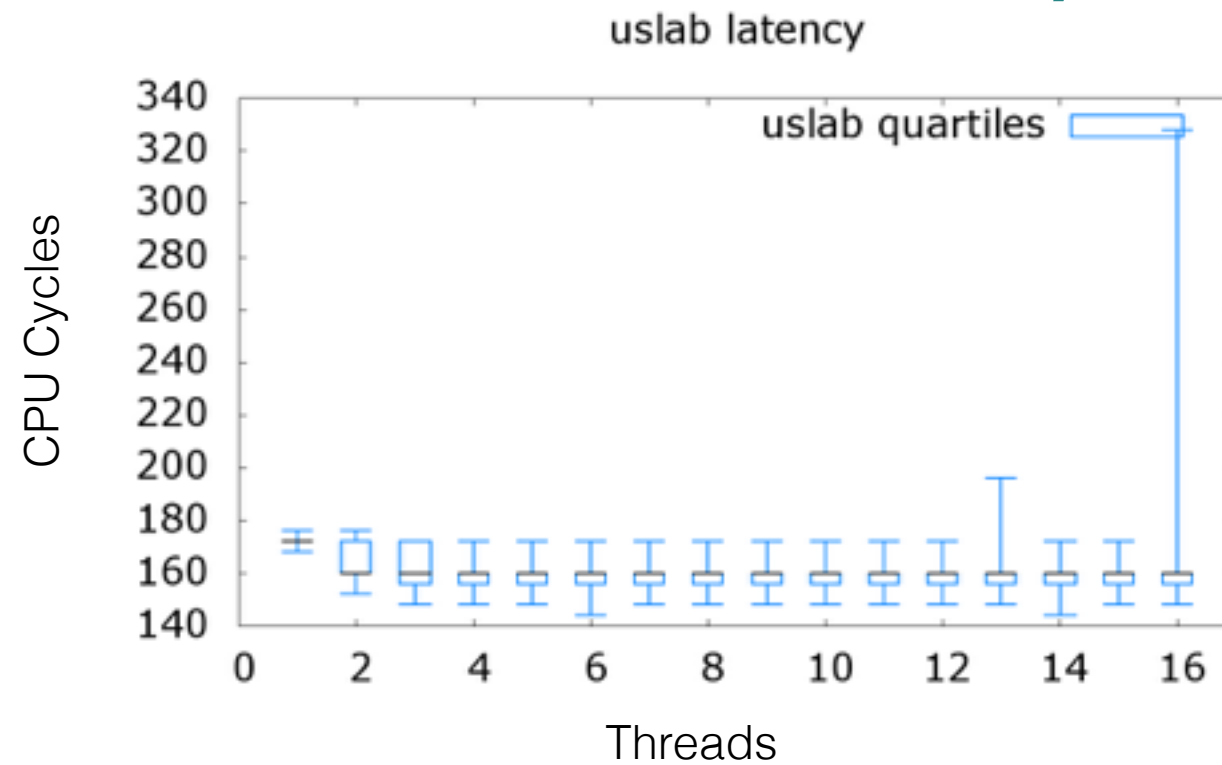
The linux pthread interface is a great general purpose interface, and the taller boxes suggest a more even latency distribution, but we still see significant variability between our bottom and top quartiles, and further jitter to our max value. And our Y scale is still on the same order of magnitude.

Allocator latency

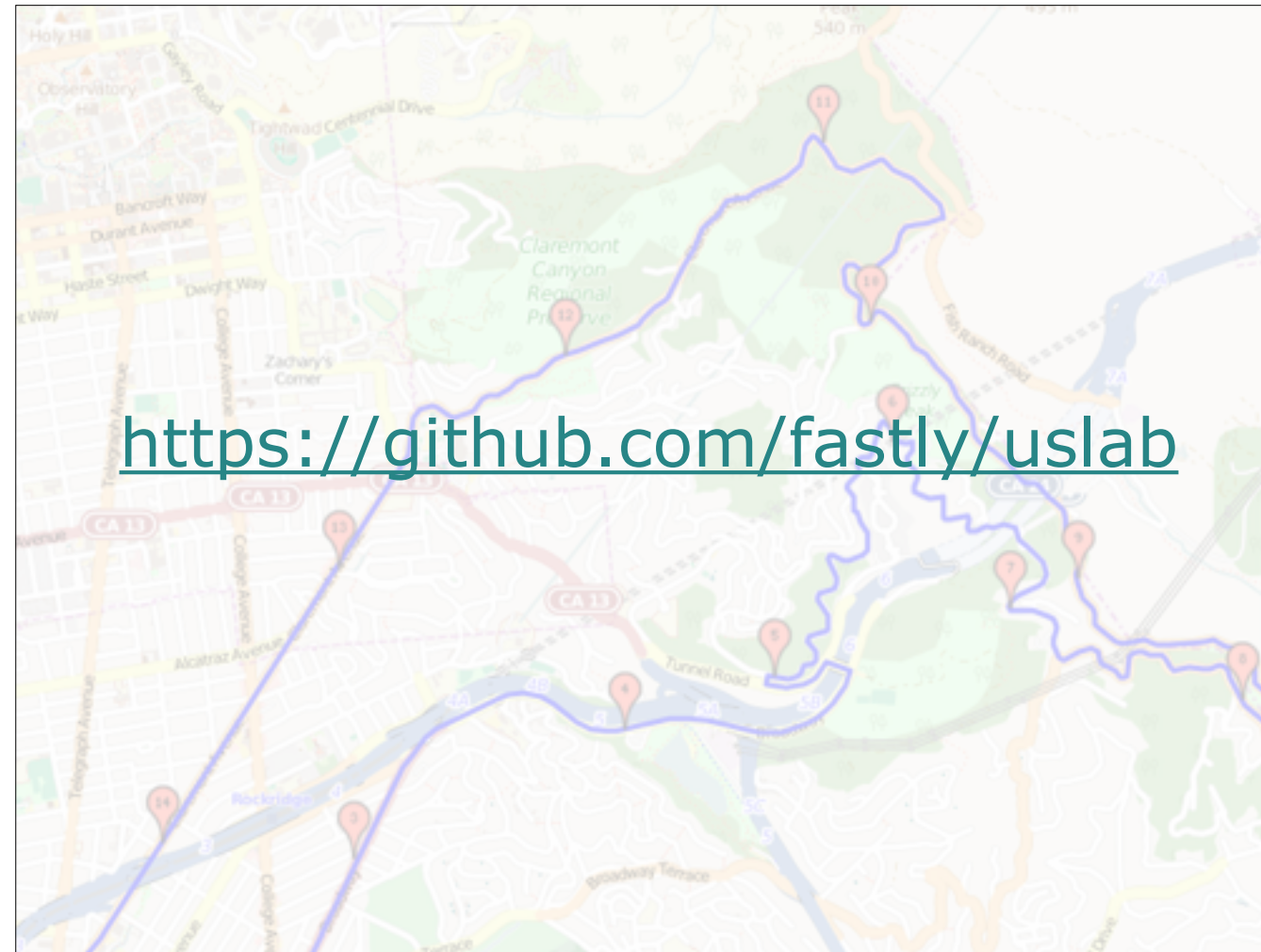


Adding exponential backoff, as we saw from earlier graphs, performs better in the worst case - notice the Y axis dropped by an order of magnitude, but we still have an incredible variance in latency.

Allocator latency



And here's the thing. Our compare and swap interface behaves extremely predictably even under high concurrent load. Our Y scale has reduced by two orders of magnitude and we see very little deviation from our average latency.



<https://github.com/fastly/uslab>

I feel pretty strongly that having reproducible results is important when giving these kinds of talks, so we've put the allocator and benchmark runners up on Github if you're inclined to take a look and play around with them yourself.

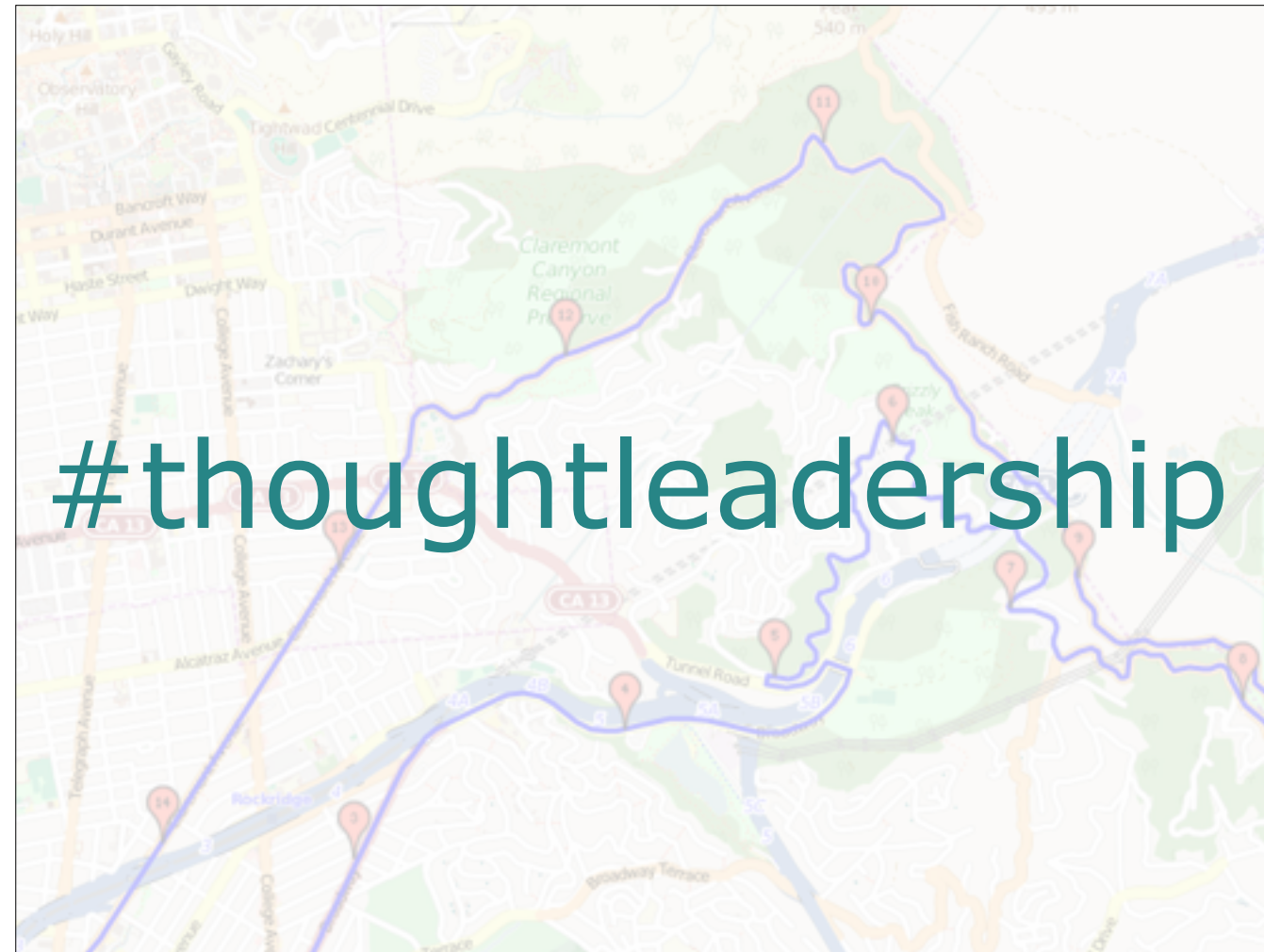
The lzf so short, the CAS so longe to lerne

- Cache coherency and NUMA architecture
- Transactional memory

We can only scratch the surface of scalable multiprocessor programming in this talk, I'm afraid. But, there's so much more that there is to say if we want the full story.

<click> For instance, I purposefully tried to not bring cache coherency and the memory hierarchy into the mix, but if we want a detailed picture of what was happening with the poor lock performance, we'd need to understand how main memory backs processor caches, and how CPUs synchronize with each other in order to maintain a globally-consistent view of memory.

<click>We saw an example of a compare and set operation operating over two words of memory, so a good question might be whether we can extend this to an arbitrarily-large region of memory. It turns out that much work in a field called transactional memory might prove to be the solution to this: it's very much inspired by the notion of a database transaction: we still still operate concurrent on a shared region of memory but can detect when multiple threads have written different values into different parts of memory, and can roll those operations back to a safe state. Doing this efficiently in hardware is very much ongoing work.



So, let's summarize what we've talked about and maybe entertain a few broader thoughts on the subject.

When is a race



a safe race?

Safely racing processes is not only common when addressing throughput and latency issues in software, it is a crucial part of the process. We've seen how racing waiters on spinlocks can help improve throughput and how our concurrent allocator races processes against the state of the freelist head. We also saw how bolting mutual exclusion on top of a sequential stack — that is, abstracting a single-threaded data structure behind concurrency primitives — yielded a correct but slow implementation, and how scalability was achieved through implementing the stack primitives themselves in terms of concurrent primitives.

The most important theme of the day, though, was that of linearization and linearization points. Without the property of linearization, all of these concurrent stack operations would fail. The linearization points created by using atomic primitives like TAS and CAS can help us prevent race conditions from becoming bugs in our concurrent software. And in general this is how non-blocking data structures work: snapshot state, create a “next good state” and attempt to atomically commit the new state.



Now it might be that you're not in the business of having to write your own memory allocator. It might be the case that your software environment, say, if you're running atop the JVM, doesn't even give you the option to manage your memory. It might even be the case that you're never faced with the sort of performance problems that a scalable memory allocator would help you solve in the first place.

<click> While all that might be true, let me put forth a few reasons why the material in this talk should be of a broader appeal than memory management library authors.

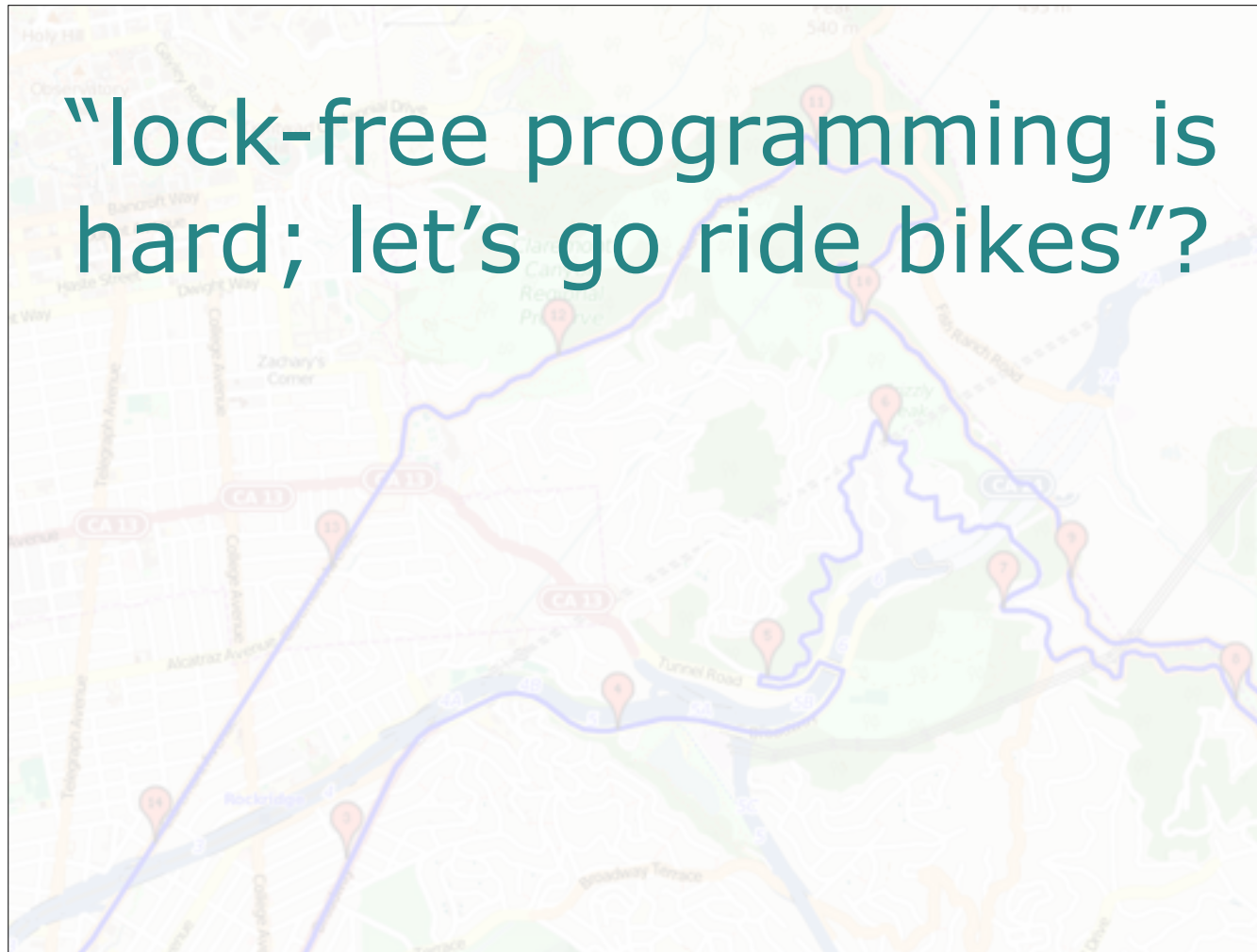
<click> The most important reason is that in my opinion, writing high performance high-level code is harder than writing high performance low-level code because you have to understand how all the intermediate layers of your stack perform and interact and are implemented. Now that you've made it through this talk, you could probably take a guess now about how Java's AtomicInteger class or Clojure's swap functions are implemented, and might have a better grasp about how they could be used in your own software.

Further, higher-level languages like Java and now even C++ have concurrency guarantees that may or may not be stricter than what the hardware provides. Is your code behaving correctly because it conforms to the Java memory model or the x86 strict memory model? If it's actually the latter and you switch platforms to ARM, will it still run correctly?

<click> In case you missed the memo, your computer is actually a distributed system in the small. There's a ton of overlap between what we needed to know about execution histories, different kinds of consistency guarantees and so forth, in order to start working in a concurrent world, and all of this *and more* is needed to start working in the distributed world.

<click> The final point is entirely subjective, but I think it's interesting and fun. Hopefully you might now, as well!

“lock-free programming is hard; let’s go ride bikes”?



Now it might be that you’re not in the business of having to write your own memory allocator. It might be the case that your software environment, say, if you’re running atop the JVM, doesn’t even give you the option to manage your memory. It might even be the case that you’re never faced with the sort of performance problems that a scalable memory allocator would help you solve in the first place.

<click> While all that might be true, let me put forth a few reasons why the material in this talk should be of a broader appeal than memory management library authors.

<click> The most important reason is that in my opinion, writing high performance high-level code is harder than writing high performance low-level code because you have to understand how all the intermediate layers of your stack perform and interact and are implemented. Now that you’ve made it through this talk, you could probably take a guess now about how Java’s AtomicInteger class or Clojure’s swap functions are implemented, and might have a better grasp about how they could be used in your own software.

Further, higher-level languages like Java and now even C++ have concurrency guarantees that may or may not be stricter than what the hardware provides. Is your code behaving correctly because it conforms to the Java memory model or the x86 strict memory model? If it’s actually the latter and you switch platforms to ARM, will it still run correctly?

<click> In case you missed the memo, your computer is actually a distributed system in the small. There’s a ton of overlap between what we needed to know about execution histories, different kinds of consistency guarantees and so forth, in order to start working in a concurrent world, and all of this *and more* is needed to start working in the distributed world.

<click> The final point is entirely subjective, but I think it’s interesting and fun. Hopefully you might now, as well!



“lock-free programming is hard; let’s go ride bikes”?

- high-level performance necessitates an understanding of low level performance

Now it might be that you’re not in the business of having to write your own memory allocator. It might be the case that your software environment, say, if you’re running atop the JVM, doesn’t even give you the option to manage your memory. It might even be the case that you’re never faced with the sort of performance problems that a scalable memory allocator would help you solve in the first place.

<click> While all that might be true, let me put forth a few reasons why the material in this talk should be of a broader appeal than memory management library authors.

<click> The most important reason is that in my opinion, writing high performance high-level code is harder than writing high performance low-level code because you have to understand how all the intermediate layers of your stack perform and interact and are implemented. Now that you’ve made it through this talk, you could probably take a guess now about how Java’s AtomicInteger class or Clojure’s swap functions are implemented, and might have a better grasp about how they could be used in your own software.

Further, higher-level languages like Java and now even C++ have concurrency guarantees that may or may not be stricter than what the hardware provides. Is your code behaving correctly because it conforms to the Java memory model or the x86 strict memory model? If it’s actually the latter and you switch platforms to ARM, will it still run correctly?

<click> In case you missed the memo, your computer is actually a distributed system in the small. There’s a ton of overlap between what we needed to know about execution histories, different kinds of consistency guarantees and so forth, in order to start working in a concurrent world, and all of this *and more* is needed to start working in the distributed world.

<click> The final point is entirely subjective, but I think it’s interesting and fun. Hopefully you might now, as well!



“lock-free programming is hard; let’s go ride bikes”?

- high-level performance necessitates an understanding of low level performance
- your computer is a distributed system

Now it might be that you’re not in the business of having to write your own memory allocator. It might be the case that your software environment, say, if you’re running atop the JVM, doesn’t even give you the option to manage your memory. It might even be the case that you’re never faced with the sort of performance problems that a scalable memory allocator would help you solve in the first place.

<click> While all that might be true, let me put forth a few reasons why the material in this talk should be of a broader appeal than memory management library authors.

<click> The most important reason is that in my opinion, writing high performance high-level code is harder than writing high performance low-level code because you have to understand how all the intermediate layers of your stack perform and interact and are implemented. Now that you’ve made it through this talk, you could probably take a guess now about how Java’s AtomicInteger class or Clojure’s swap functions are implemented, and might have a better grasp about how they could be used in your own software.

Further, higher-level languages like Java and now even C++ have concurrency guarantees that may or may not be stricter than what the hardware provides. Is your code behaving correctly because it conforms to the Java memory model or the x86 strict memory model? If it’s actually the latter and you switch platforms to ARM, will it still run correctly?

<click> In case you missed the memo, your computer is a actually a distributed system in the small. There’s a ton of overlap between what we needed to know about execution histories, different kinds of consistency guarantees and so forth, in order to start working in a concurrent world, and all of this *and more* is needed to start working in the distributed world.

<click> The final point is entirely subjective, but I think it’s interesting and fun. Hopefully you might now, as well!



“lock-free programming is hard; let’s go ride bikes”?

- high-level performance necessitates an understanding of low level performance
- your computer is a distributed system
- (optional third answer: it’s real neato)

Now it might be that you’re not in the business of having to write your own memory allocator. It might be the case that your software environment, say, if you’re running atop the JVM, doesn’t even give you the option to manage your memory. It might even be the case that you’re never faced with the sort of performance problems that a scalable memory allocator would help you solve in the first place.

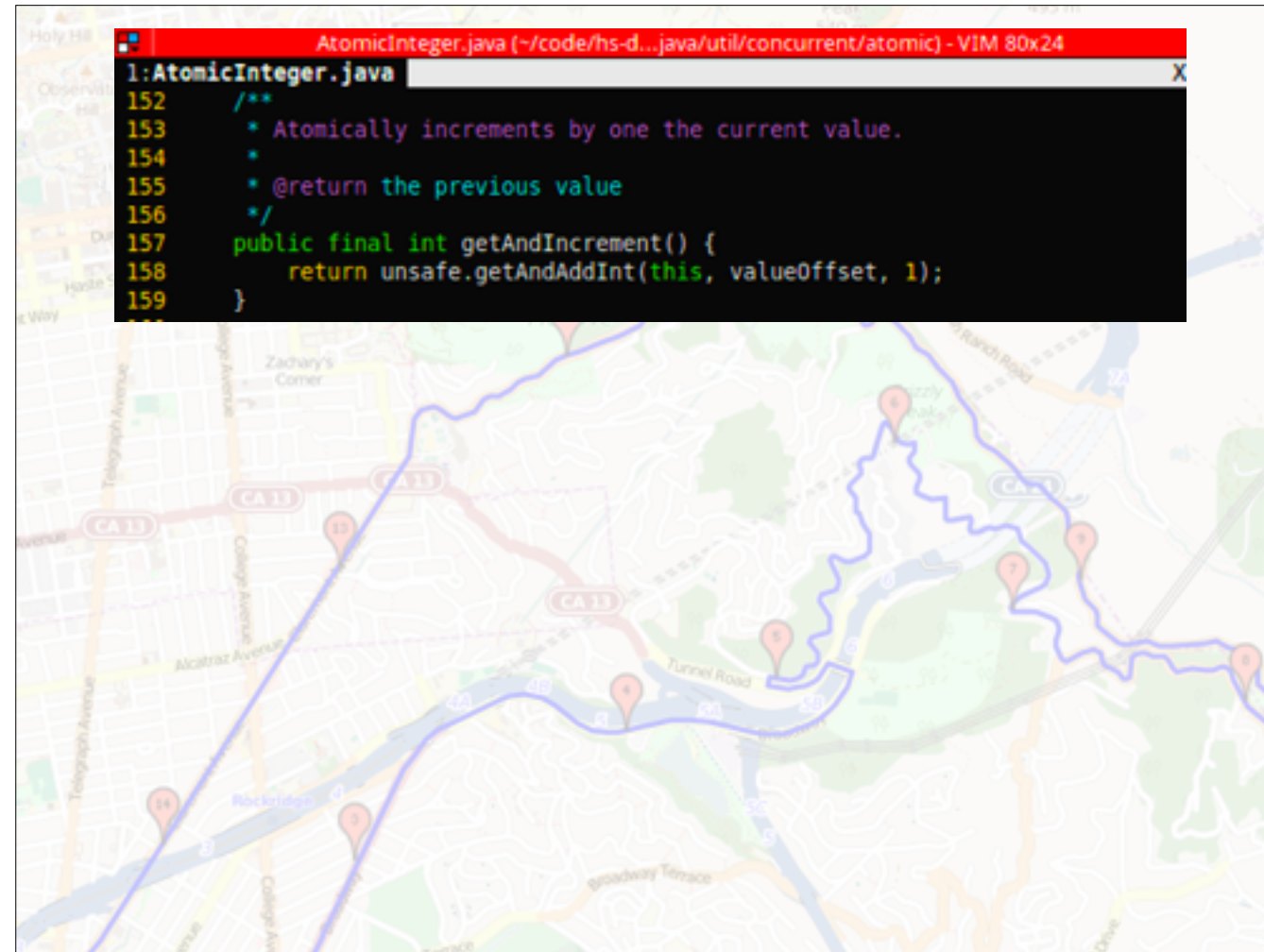
<click> While all that might be true, let me put forth a few reasons why the material in this talk should be of a broader appeal than memory management library authors.

<click> The most important reason is that in my opinion, writing high performance high-level code is harder than writing high performance low-level code because you have to understand how all the intermediate layers of your stack perform and interact and are implemented. Now that you’ve made it through this talk, you could probably take a guess now about how Java’s AtomicInteger class or Clojure’s swap functions are implemented, and might have a better grasp about how they could be used in your own software.

Further, higher-level languages like Java and now even C++ have concurrency guarantees that may or may not be stricter than what the hardware provides. Is your code behaving correctly because it conforms to the Java memory model or the x86 strict memory model? If it’s actually the latter and you switch platforms to ARM, will it still run correctly?

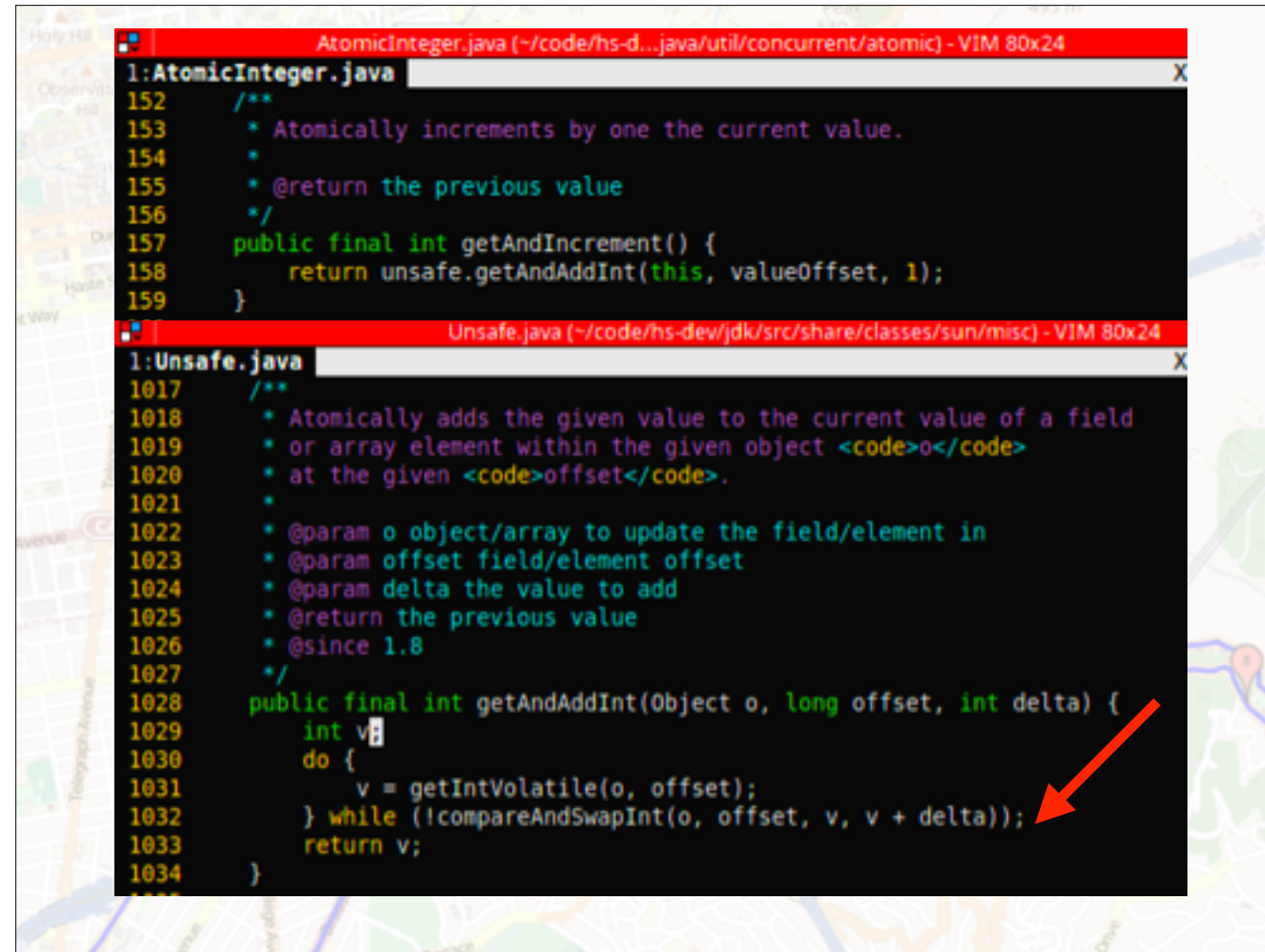
<click> In case you missed the memo, your computer is a actually a distributed system in the small. There’s a ton of overlap between what we needed to know about execution histories, different kinds of consistency guarantees and so forth, in order to start working in a concurrent world, and all of this *and more* is needed to start working in the distributed world.

<click> The final point is entirely subjective, but I think it’s interesting and fun. Hopefully you might now, as well!



Forgive me, I couldn't resist doing this. Once a language runtime hacker, always a language runtime hacker, I guess.

So if on the last slide you took a guess about AtomicInteger's implementation, we can look at the JDK source to see how AtomicInteger is implemented. If you didn't make a guess, you have three seconds to come up with one. <click> It's indeed a compare and swap in a loop.

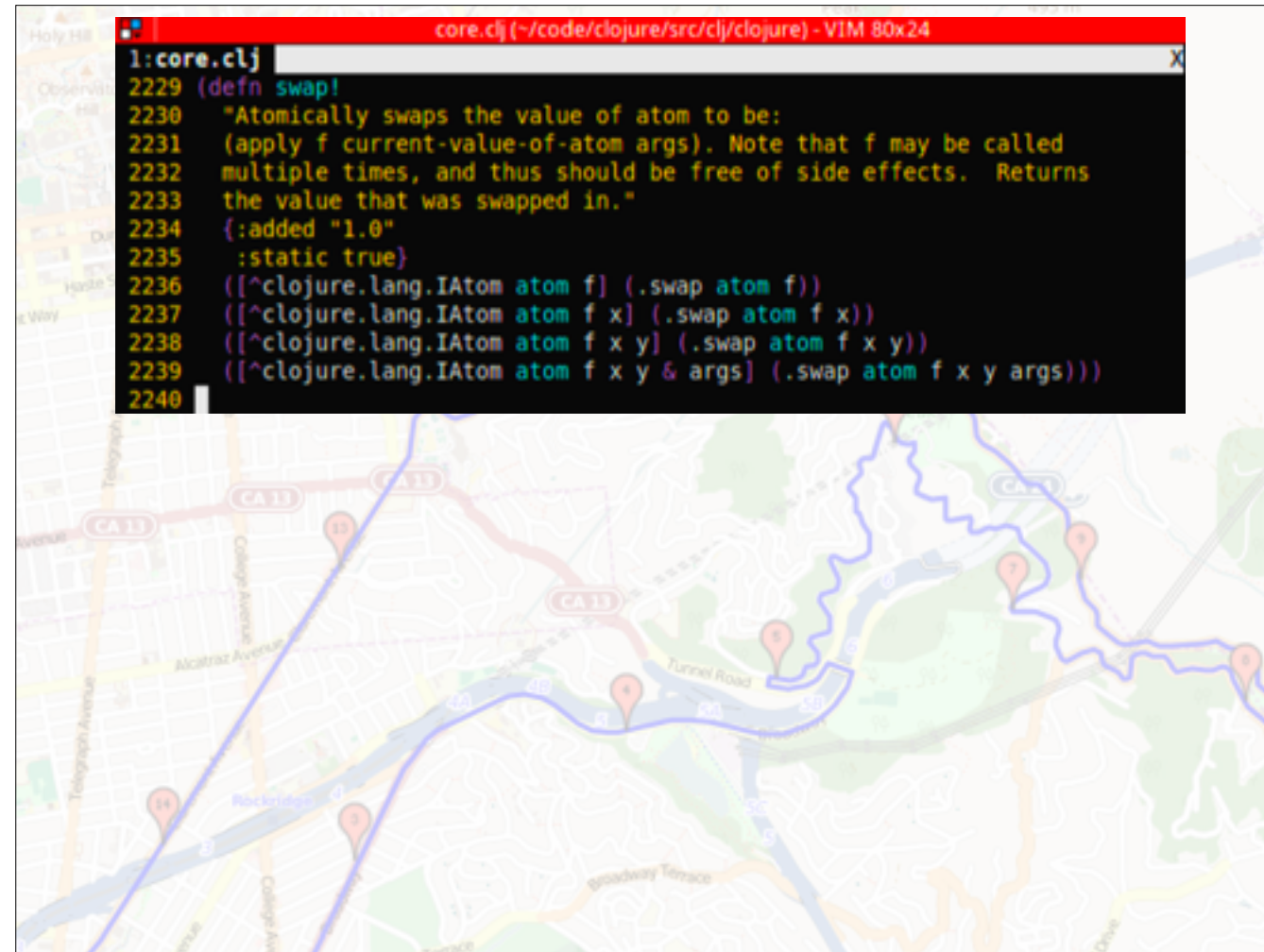


```
AtomicInteger.java (~/.code/hs-d...java/util/concurrent/atomic) - VIM 80x24
1:AtomicInteger.java
152  /**
153   * Atomically increments by one the current value.
154   *
155   * @return the previous value
156   */
157  public final int getAndIncrement() {
158      return unsafe.getAndAddInt(this, valueOffset, 1);
159  }

Unsafe.java (~/.code/hs-dev/jdk/src/share/classes/sun/misc) - VIM 80x24
1:Unsafe.java
1017 /**
1018  * Atomically adds the given value to the current value of a field
1019  * or array element within the given object <code>o</code>
1020  * at the given <code>offset</code>.
1021  *
1022  * @param o object/array to update the field/element in
1023  * @param offset field/element offset
1024  * @param delta the value to add
1025  * @return the previous value
1026  * @since 1.8
1027  */
1028 public final int getAndAddInt(Object o, long offset, int delta) {
1029     int v;
1030     do {
1031         v = getIntVolatile(o, offset);
1032     } while (!compareAndSwapInt(o, offset, v, v + delta));
1033     return v;
1034 }
```

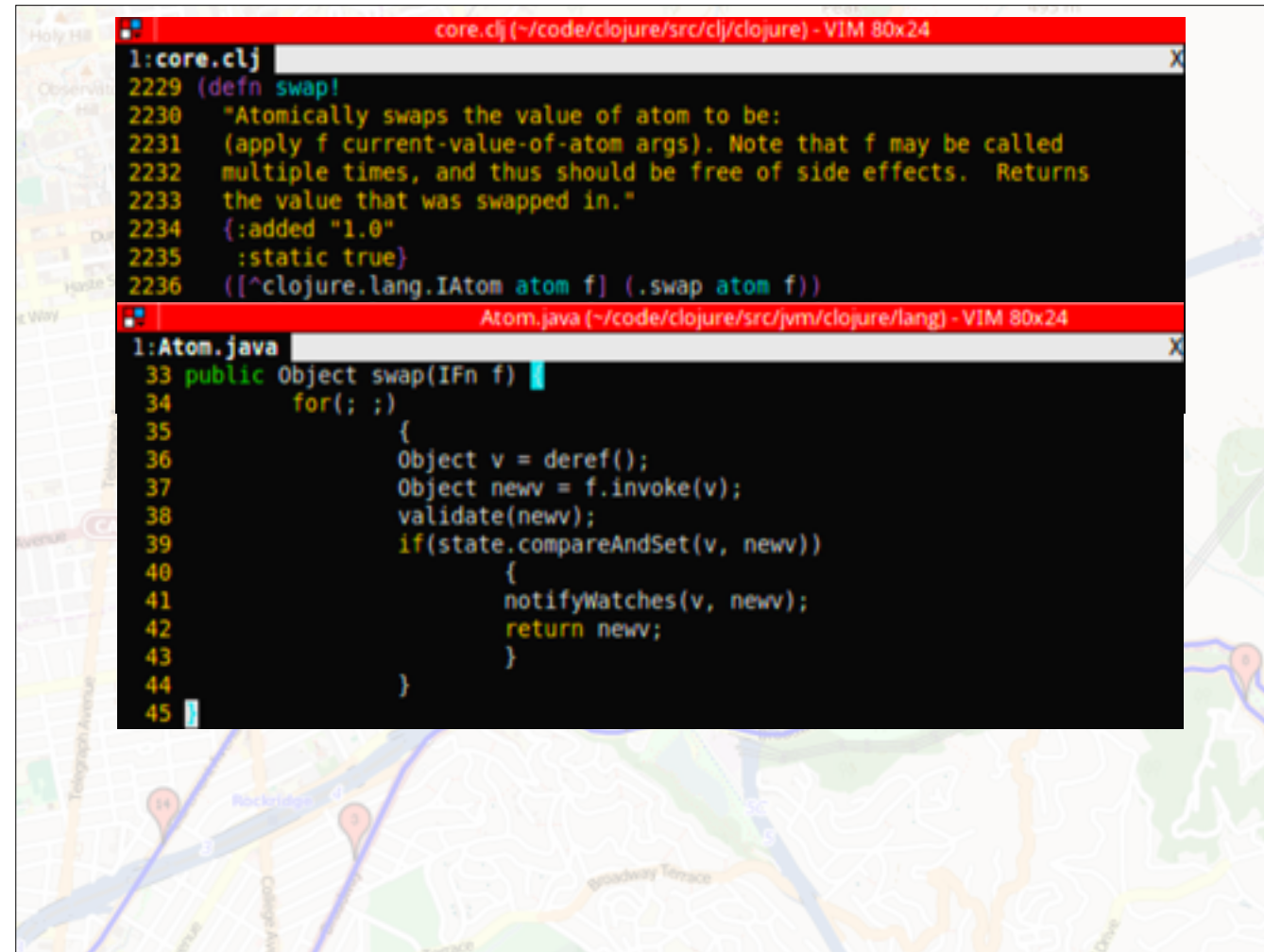
Forgive me, I couldn't resist doing this. Once a language runtime hacker, always a language runtime hacker, I guess.

So if on the last slide you took a guess about AtomicInteger's implementation, we can look at the JDK source to see how AtomicInteger is implemented. If you didn't make a guess, you have three seconds to come up with one. <click> It's indeed a compare and swap in a loop.



And now, what about Clojure's atoms. [This one's](#) nested a level deeper but [hey](#) look a compare and swap again!

Code splunking how Java locks are implemented is an exercise to the listener! Actually Java locks are very interesting in that (at least on OpenJDK under Linux) they wrap pthreads, but offer some interesting features.



And now, what about Clojure's atoms. <click> This one's nested a level deeper but <click>hey look a compare and swap again!

Code splunking how Java locks are implemented is an exercise to the listener! Actually Java locks are very interesting in that (at least on OpenJDK under Linux) they wrap pthreads, but offer some interesting features.

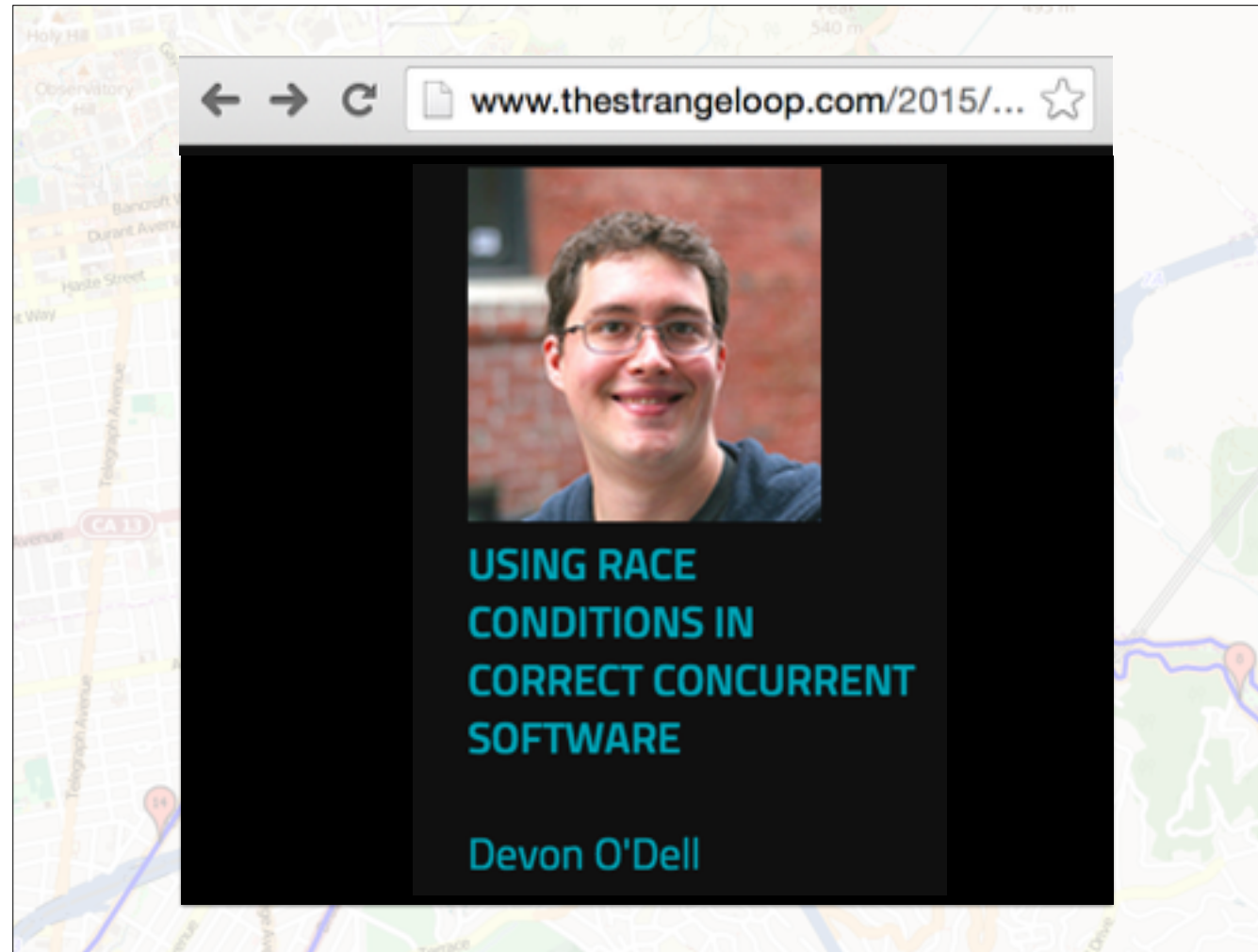

```
core.clj (~/.code/clojure/src/clj/clojure) - VIM 80x24
2229 (defn swap!
2230   "Atomically swaps the value of atom to be:
2231   (apply f current-value-of-atom args). Note that f may be called
2232   multiple times, and thus should be free of side effects. Returns
2233   the value that was swapped in."
2234   (:added "1.0"
2235    :static true)
2236   ([^clojure.lang.IAtom atom f] (.swap atom f)))

Atom.java (~/.code/clojure/src/jvm/clojure/lang) - VIM 80x24
33 public Object swap(IFn f) {
34     for(;; ;){
35         {
36             Object v = deref();
37             Object newv = f.invoke(v);
38             validate(newv);
39             if(state.compareAndSet(v, newv))
40                 {
41                     notifyWatches(v, newv);
42                     return newv;
43                 }
44     }
45 }

AtomicReference.java (~/.code/hs...java/util/concurrent/atomic) - VIM 80x24
115 public final boolean compareAndSet(V expect, V update) {
116     return unsafe.compareAndSwapObject(this, valueOffset, expect,
117     update);
117 }
```

And now, what about Clojure's atoms. <click> This one's nested a level deeper but <click>hey look a compare and swap again!

Code splunking how Java locks are implemented is an exercise to the listener! Actually Java locks are very interesting in that (at least on OpenJDK under Linux) they wrap pthreads, but offer some interesting features.



Lastly, I should say that these slides were made together with my colleague Devon O'Dell, who is giving a talk on the same material at Strangeloop this week. So, if you found this material interesting you should without question check out his talk, as I'm sure it'll be interesting and informative as well.

Thanks



Come see us at the **fastly** booth!

credits, code, and additional material at
<https://github.com/dijkstracula/Surge2015/>

Nathan Taylor | nathan.dijkstracula.net | @dijkstracula