Racing To Win

Using Race Conditions to Build Correct & Concurrent Software

Nathan Taylor | nathan.dijkstracula.net | @dijkstracula

Racing To Win

Using Race Conditions to Build Correct & Concurrent Software

Nathan Taylor | nathan.dijkstracula.net | @dijkstracula

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

I’m an engineer at fastly.

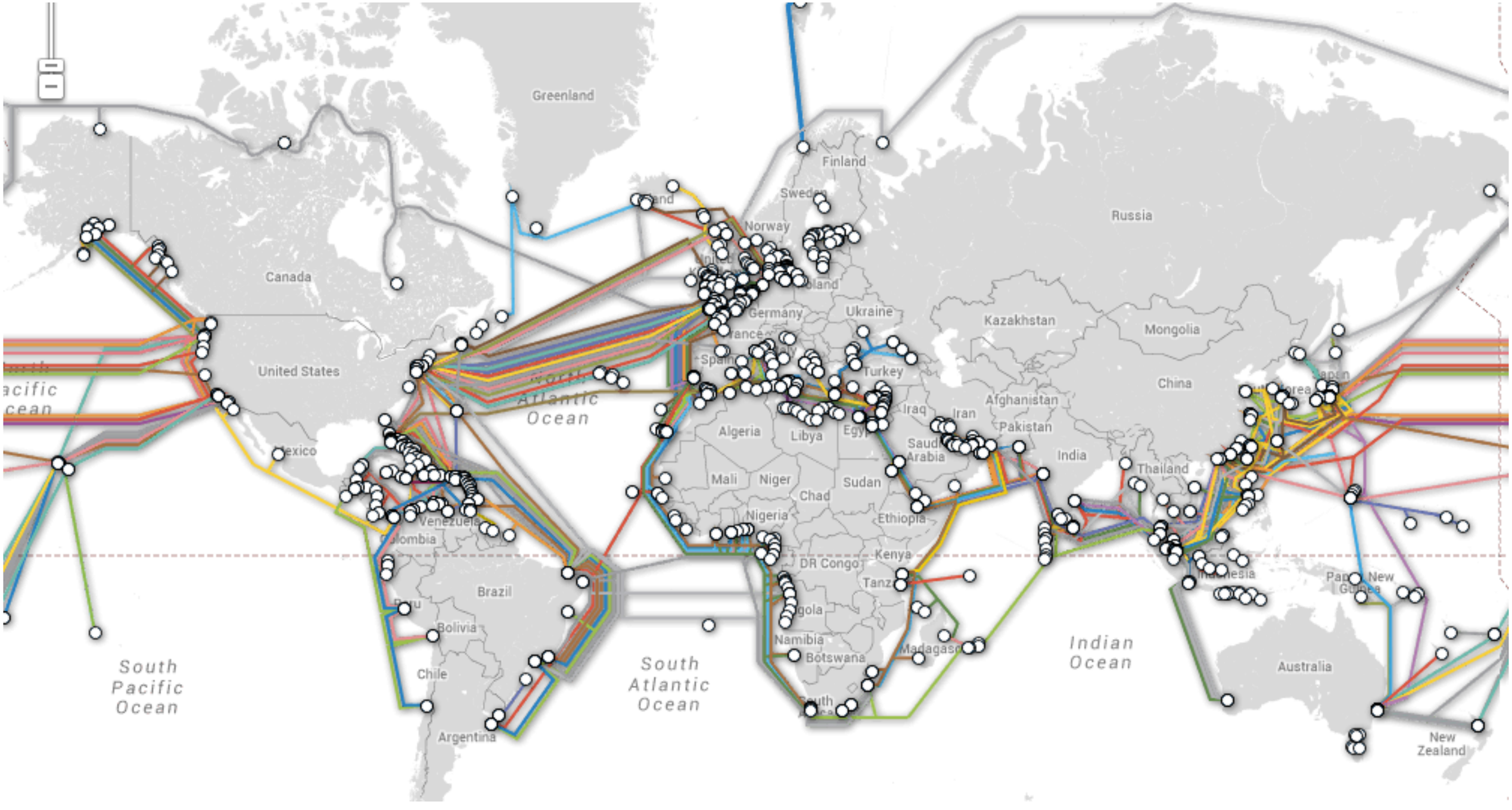
A problem

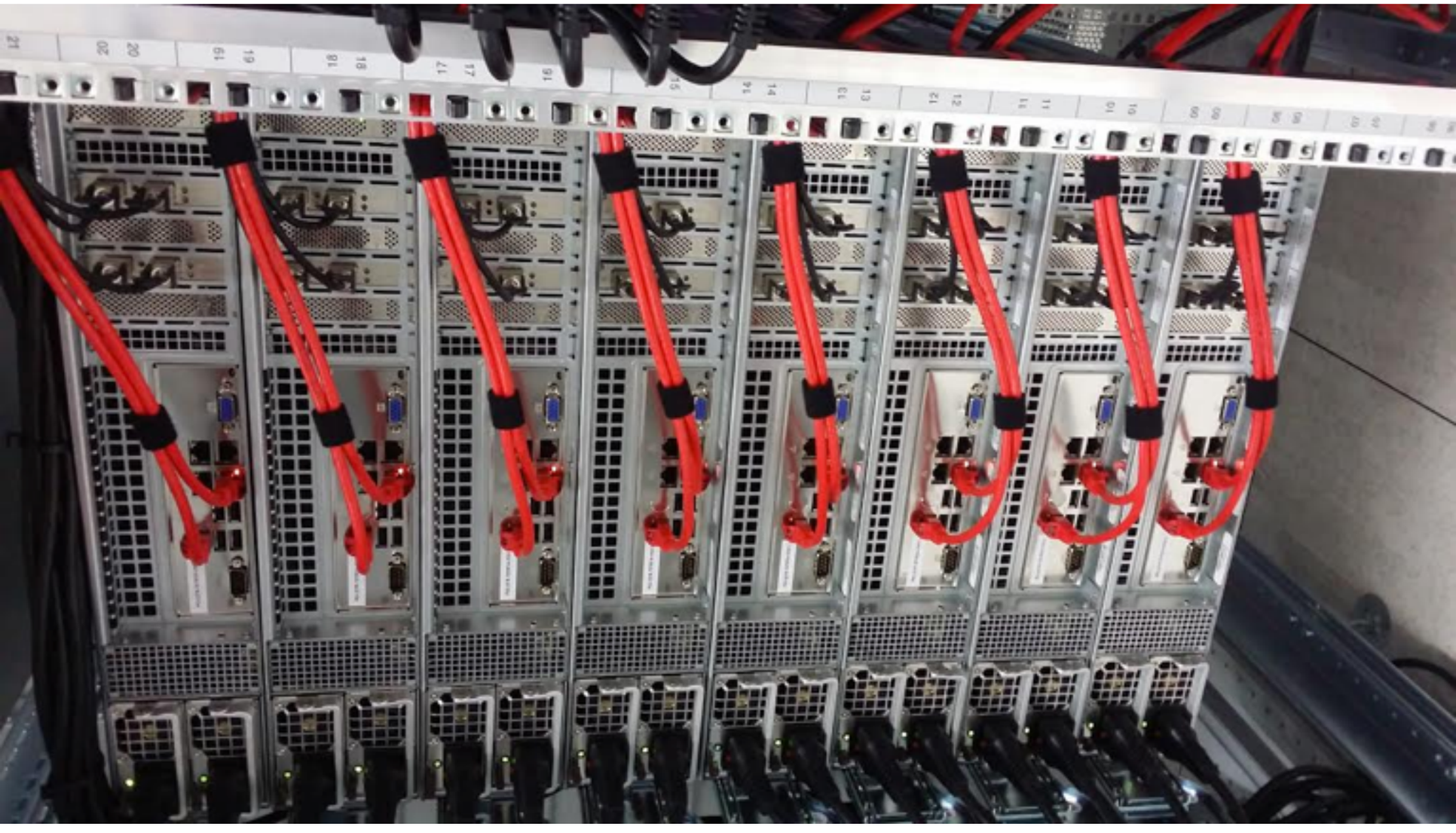
A problem

A solution

A problem

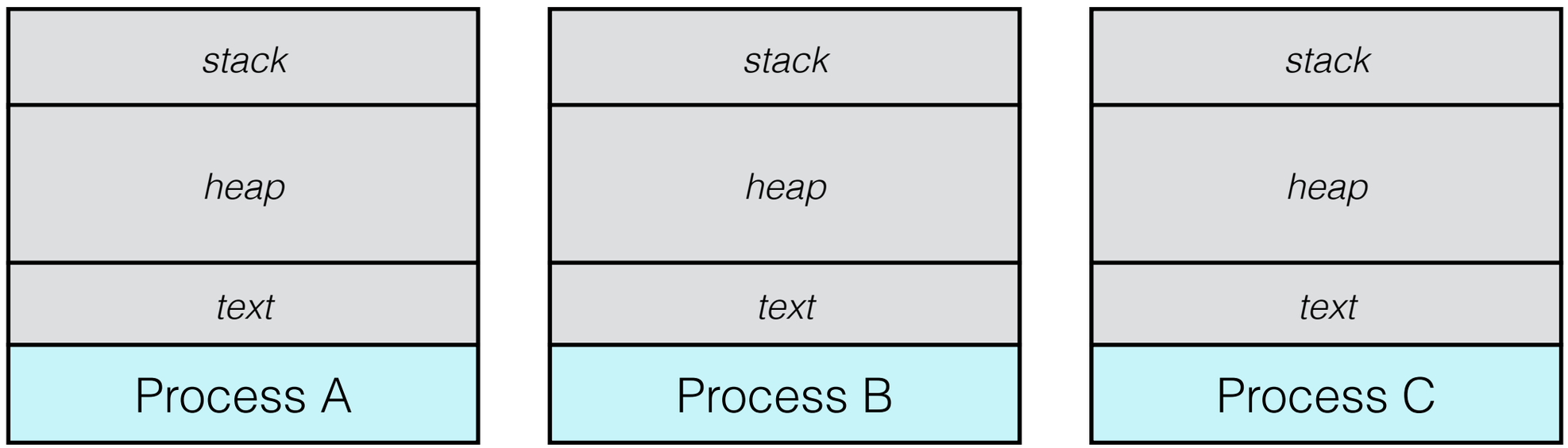
A solution


Greenland
Canada
United States
Mexico
Venezuela
Colombia
Brazil
Peru
Bolivia
Chile
Argentina
North Atlantic Ocean
South Pacific Ocean
South Atlantic Ocean
Indian Ocean
Finland
Sweden
Norway
Russia
Germany
France
Spain
Ukraine
Turkey
Kazakhstan
Mongolia
China
Japan
Korea
Afghanistan
Iraq
Iran
Pakistan
India
Thailand
Saudi Arabia
Algeria
Libya
Egypt
Mali
Niger
Chad
Sudan
Nigeria
Ethiopia
Kenya
DR Congo
Tanzania
Angola
Namibia
Botswana
Madagascar
Indonesia
Papua New Guinea
Australia
New Zealand

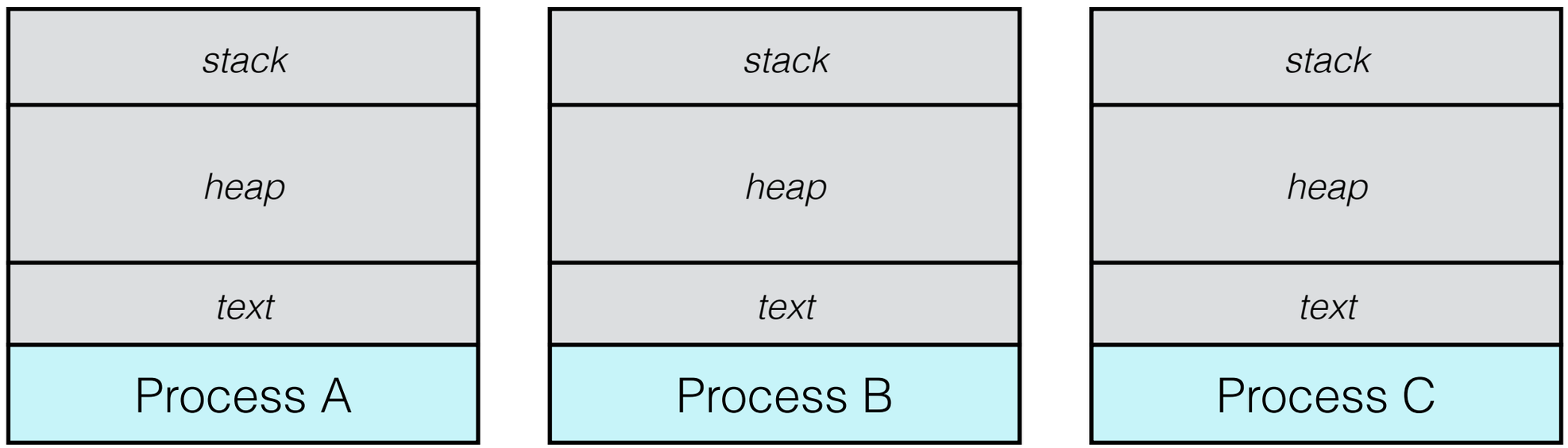


Cache node

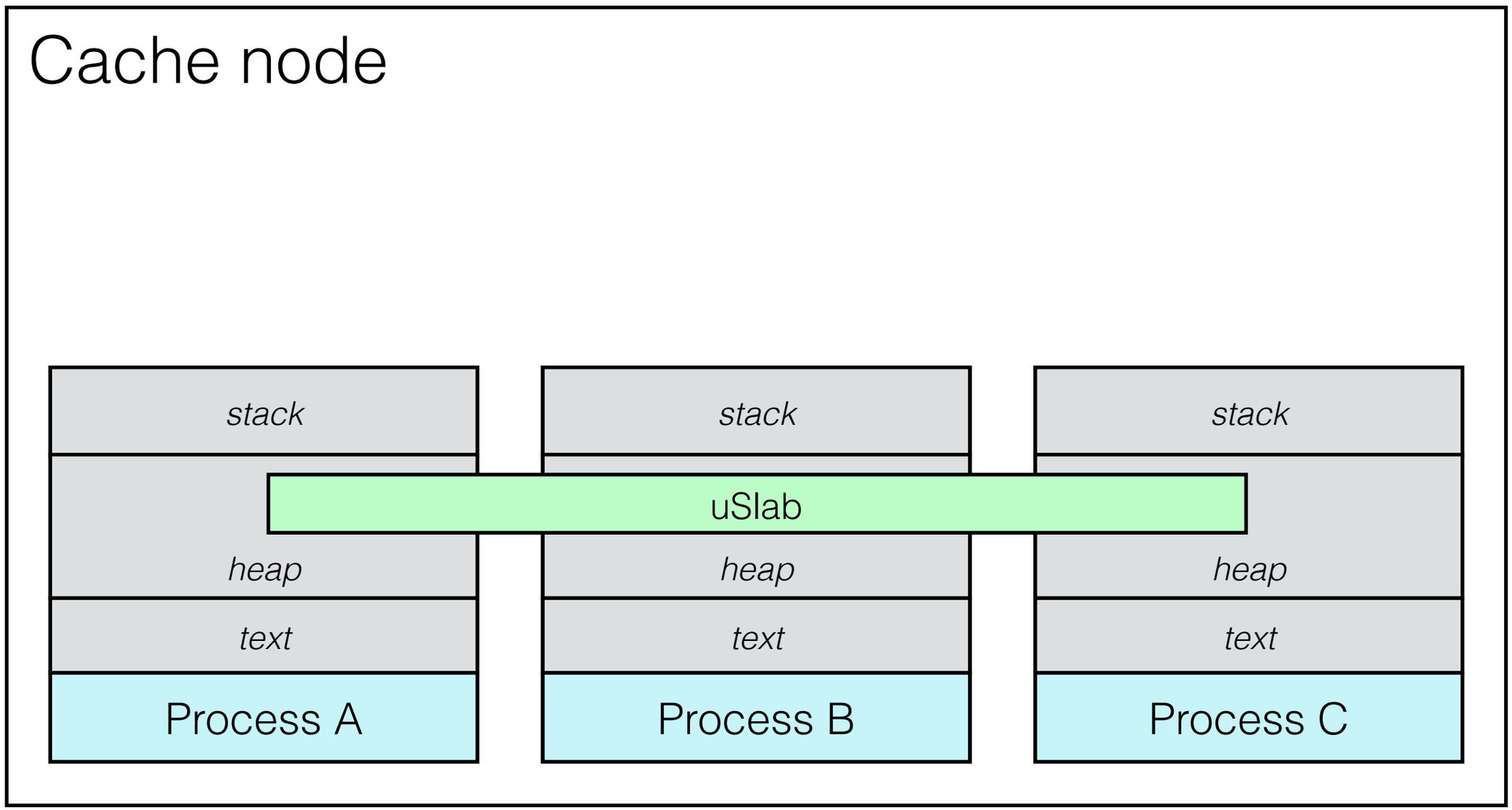
Cache node


stack
heap
text
Process A
stack
heap
text
Process B
stack
heap
text
Process C

Cache node

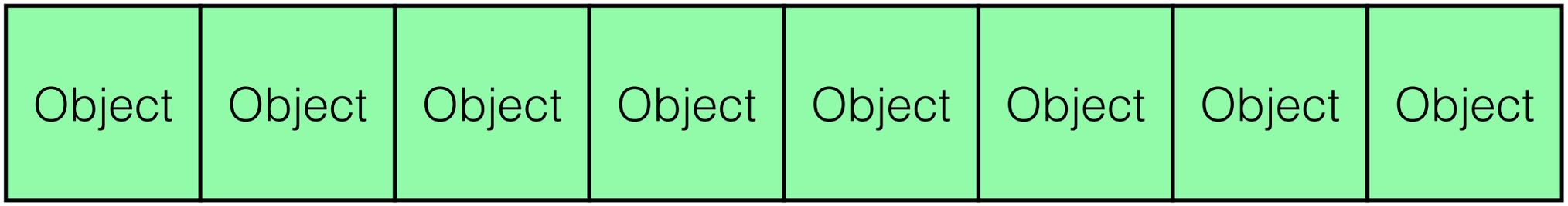

stack
heap
text
Process A
stack
heap
text
Process B
stack
heap
text
Process C

A Persistent, Shared-State Memory Allocator

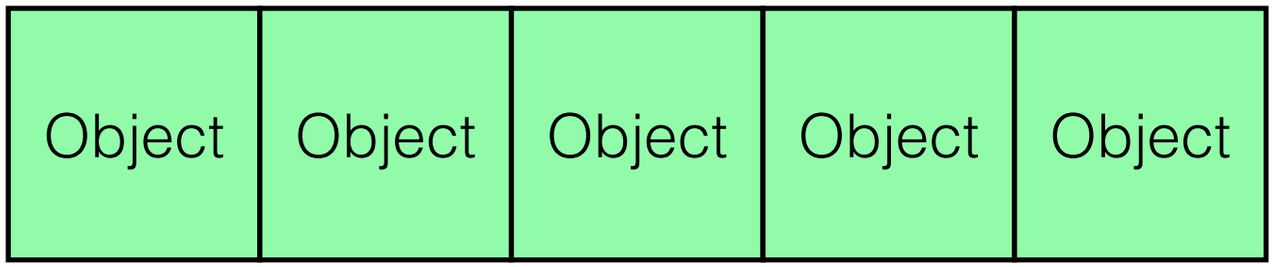


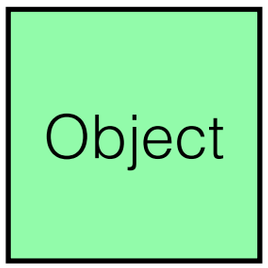
Slab allocation

Slab allocation

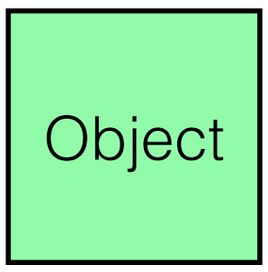


Object	Object	Object	Object	Object	Object	Object	Object

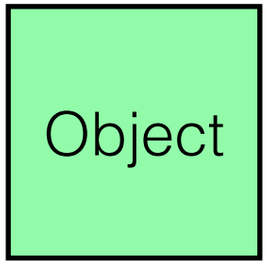





```
s_alloc();
```




```
s_alloc();
```




```
s_alloc();
```

Object | Object | Object | Object | Object

Object

Object

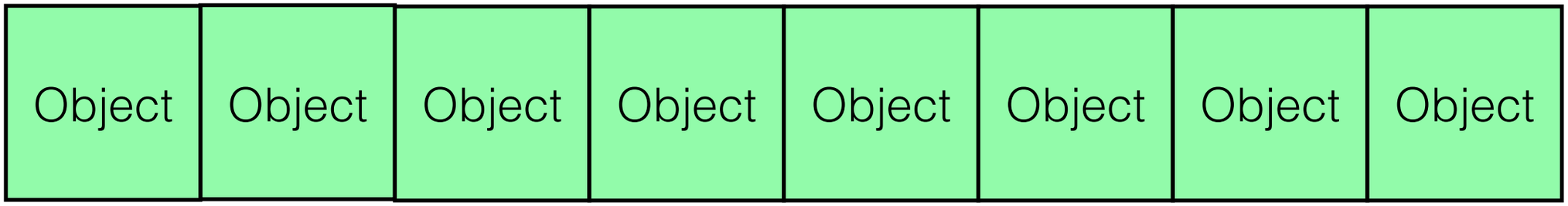
Object

Object | Object | Object | Object | Object

Object

Object

Object




```
s_free(        );
```

```
s_free(        );

s_free(        );
```

Object	Object	Object	Object	Object	Object	Object	Object

Object	Object	Object	Object	Object	Object	Object	Object

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

An Execution History

An Execution History

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

An Execution History

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

Time

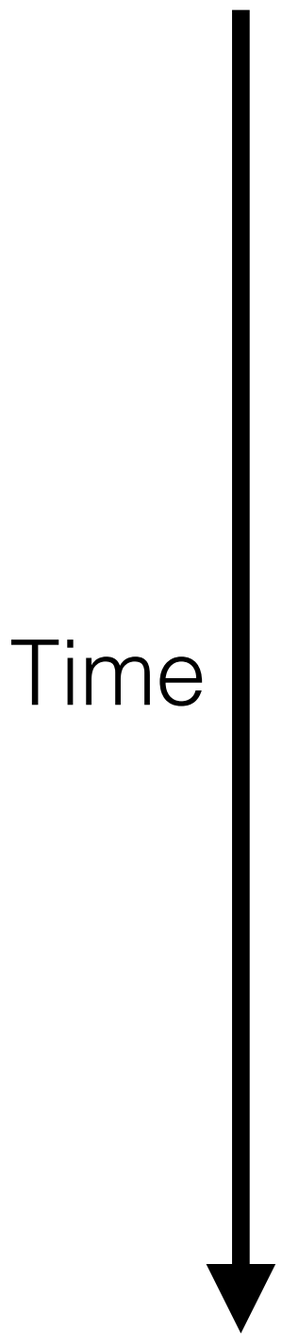
 (allocate request)

 (allocate response)

 (free request)

 (free response)

An Execution History



A(allocate request)

B(allocate request)

A(allocate response)

B(allocate response)

An Execution History

Time

A(allocate request)

B(allocate request)

A(allocate response)

B(allocate response)

"X happened before Y" =>
"Y may observe X to have occurred"

A(allocate request)

B(allocate request)

A(allocate response)

B(allocate response)

Time

A(allocate request)

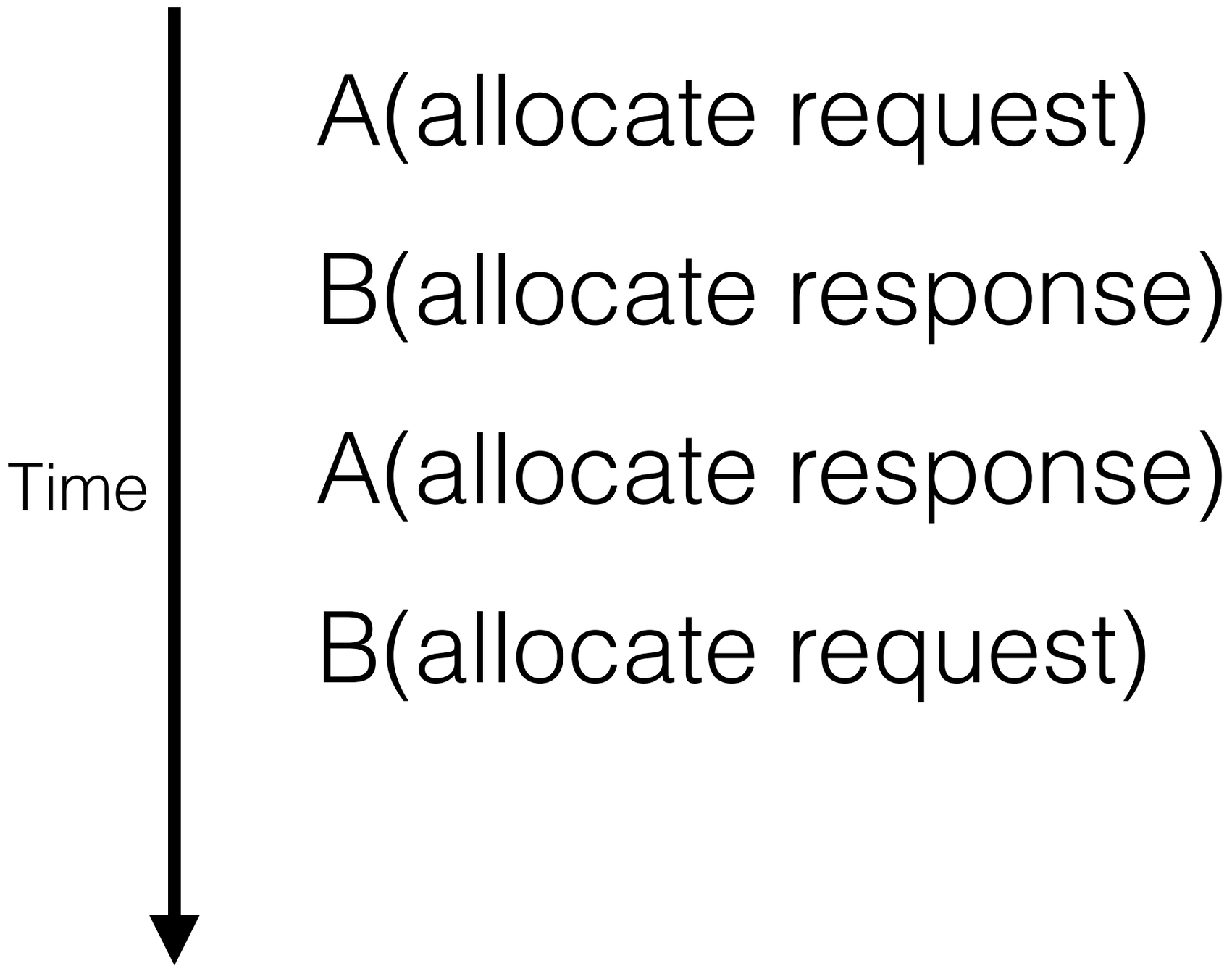
B(allocate response)

A(allocate response)

B(allocate request)

Time

A protocol violation!



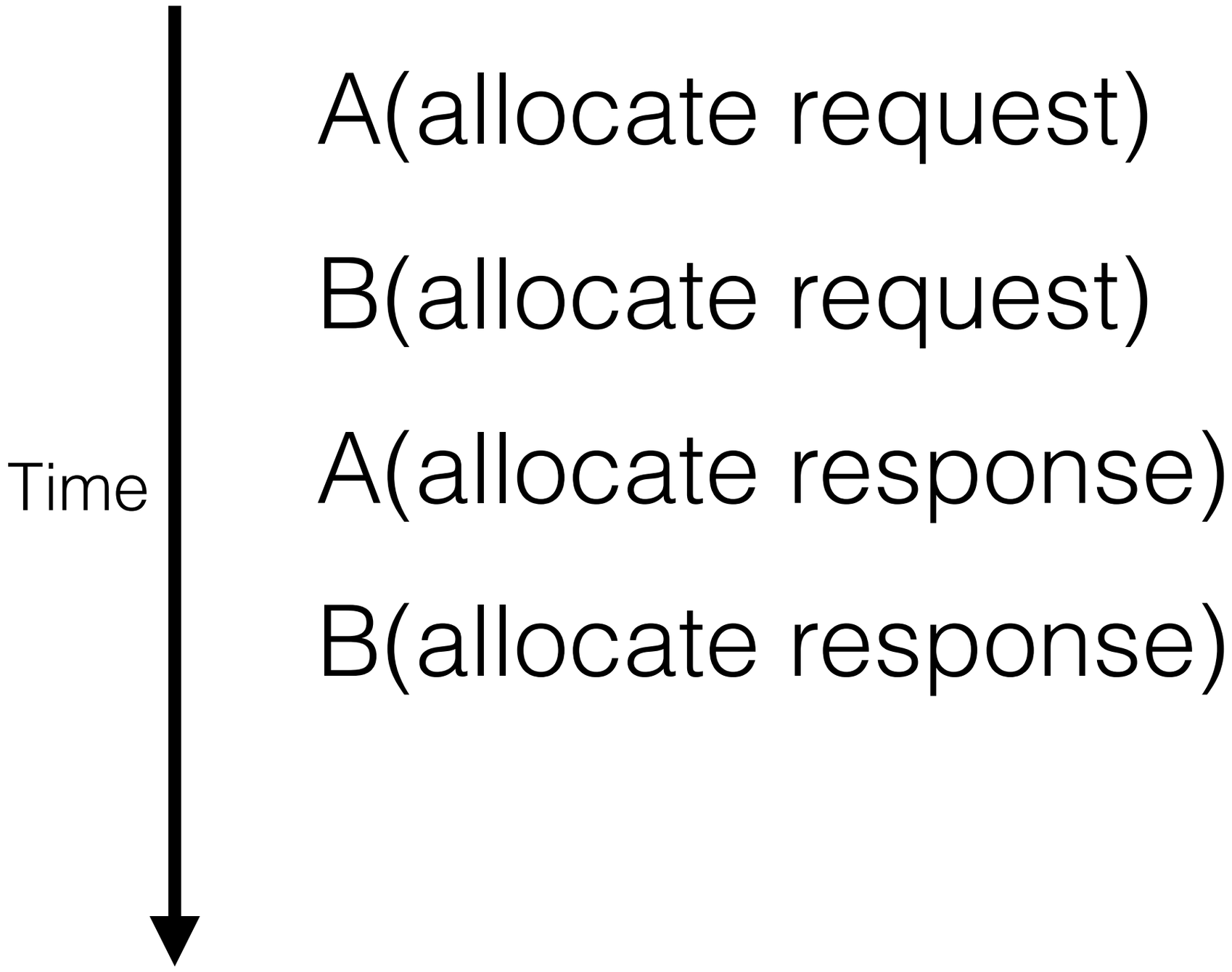
A(allocate request)

B(allocate response)

Time

A(allocate response)

B(allocate request)


A(allocate request)
B(allocate request)
A(allocate response)
B(allocate response)
Time

Linearizability: A Correctness Condition for Concurrent Objects

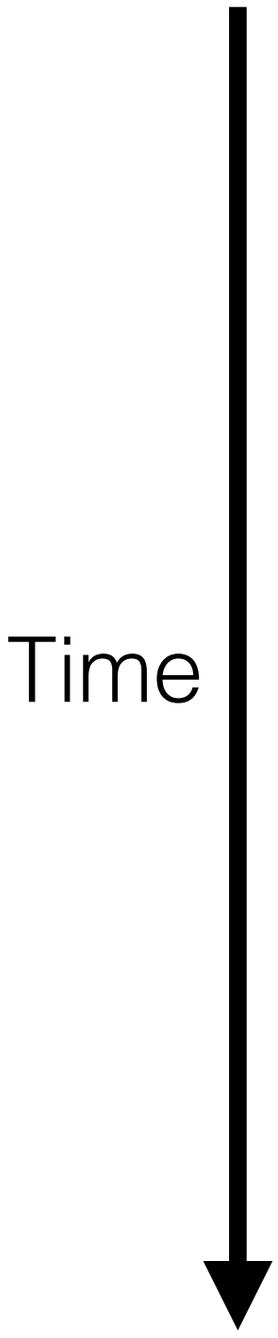
MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A history H is *sequential* if:

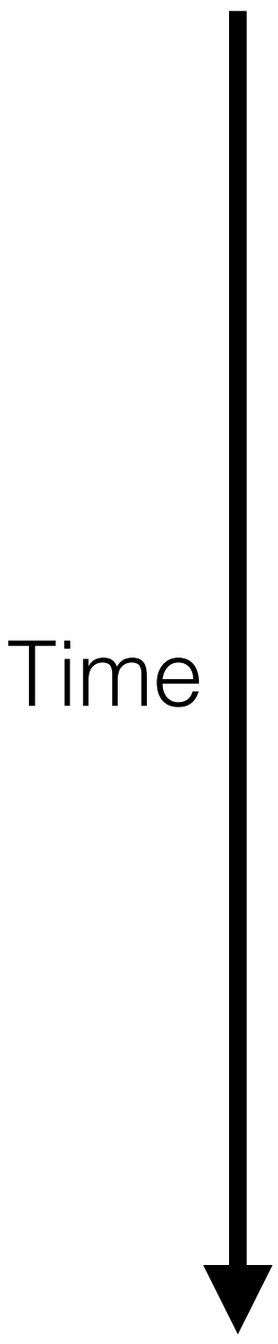
(1) The first event of H is an invocation.

(2) Each invocation, except possibly the last, is immediately followed by a matching response. Each response is immediately followed by a matching invocation.

A Sequential History



A Sequential History



A(allocate request)

A(allocate response)

B(allocate request)

B(allocate response)

A(free request)

A(free response)

A Sequential History

Time

{ A(allocate request)
A(allocate response) }

{ B(allocate request)
B(allocate response) }

{ A(free request)
A(free response) }

A Sequential History

Time

{ A(allocate request)
A(allocate response) }

{ B(allocate request)
B(allocate response) }

{ A(free request)
A(free response) }

A Sequential History

Time

{ A(allocate request)
A(allocate response) }

{ A(free request)
A(free response) }

{ B(allocate request)
B(allocate response) }

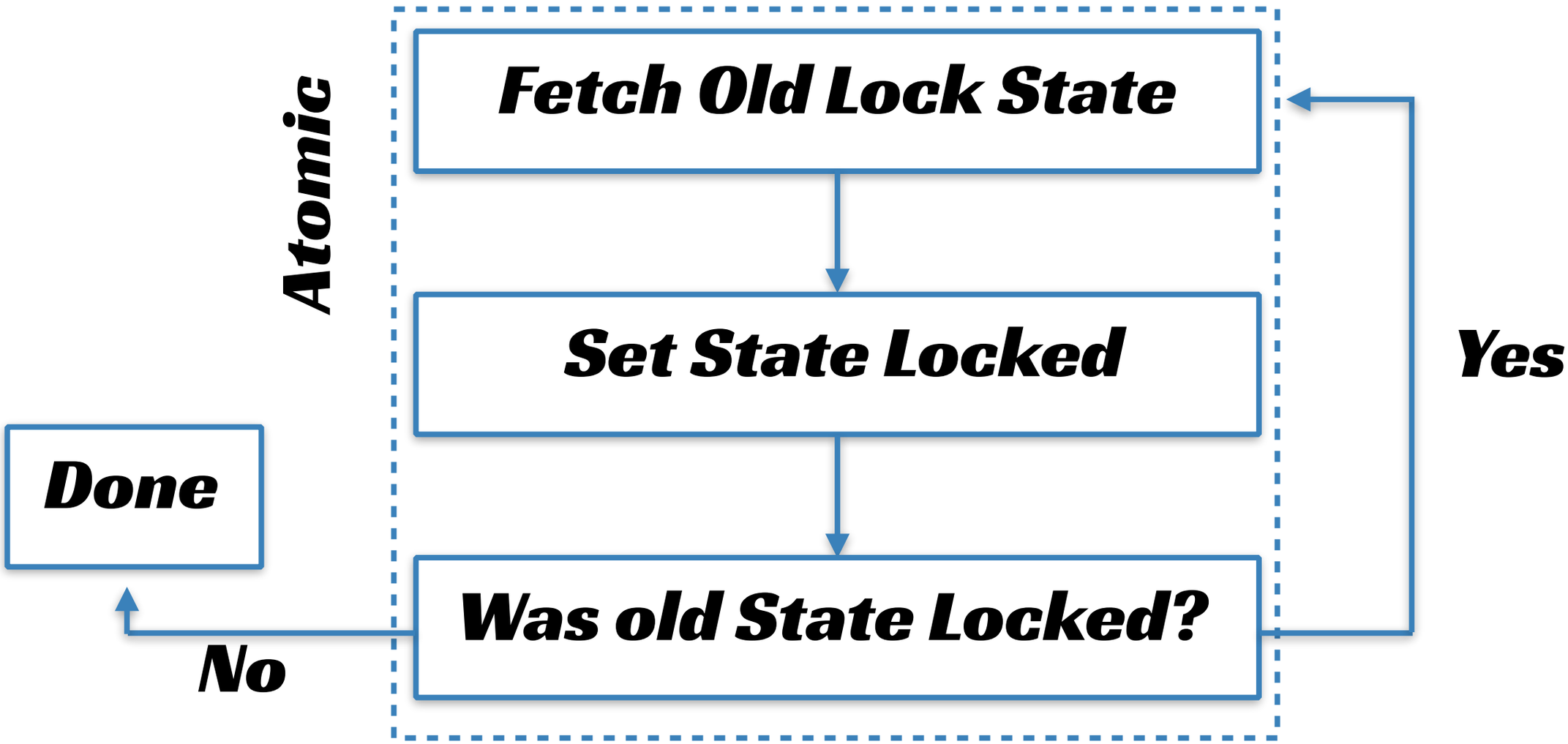
```
obj *allocate(slab *s) {

  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;

}
```

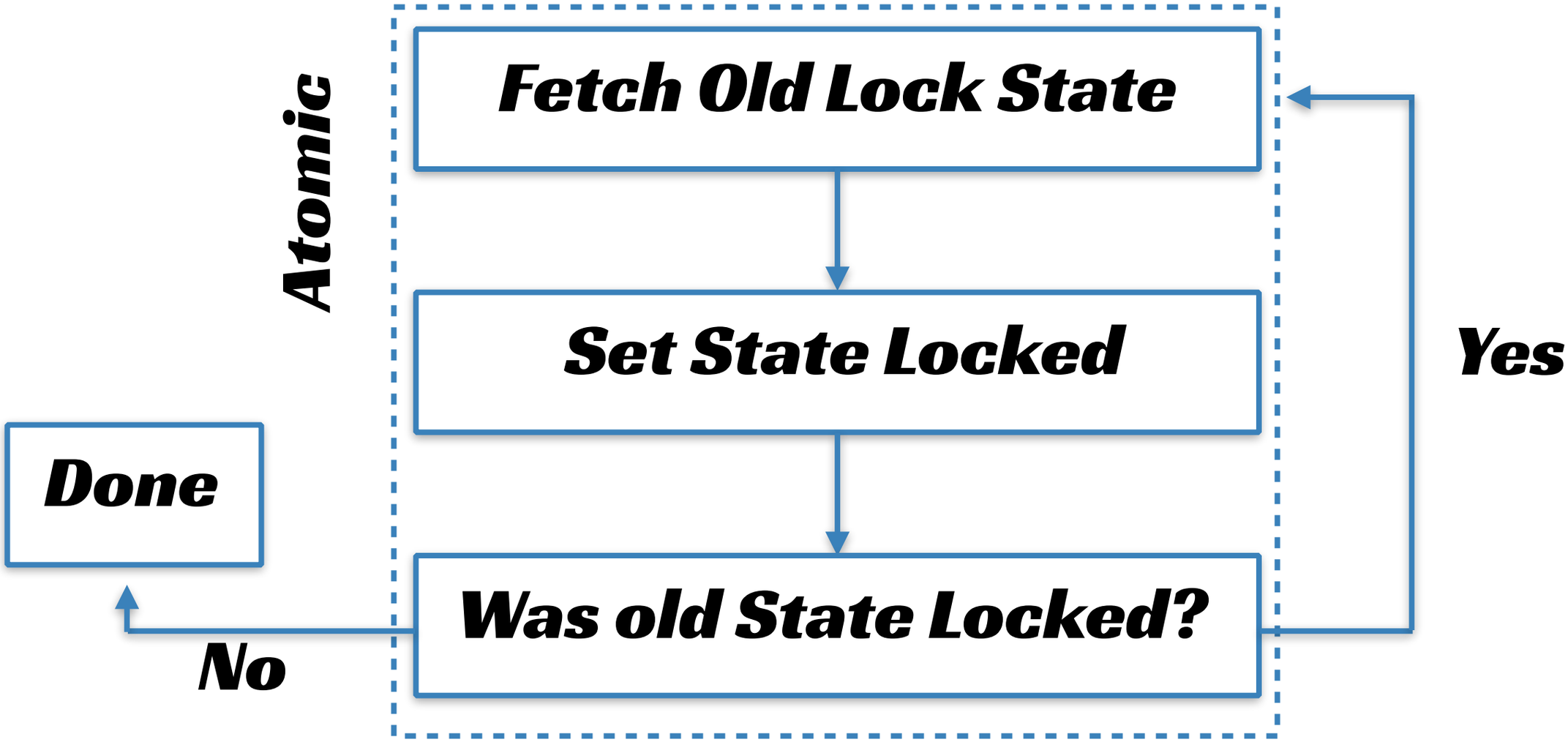
```
obj *allocate(slab *s) {
  lock(&allocator_lock);
  obj *a = s->head;
  if (a == NULL) return NULL;
  s->head = a->next;
  unlock(&allocator_lock);
  return a;
}
```

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

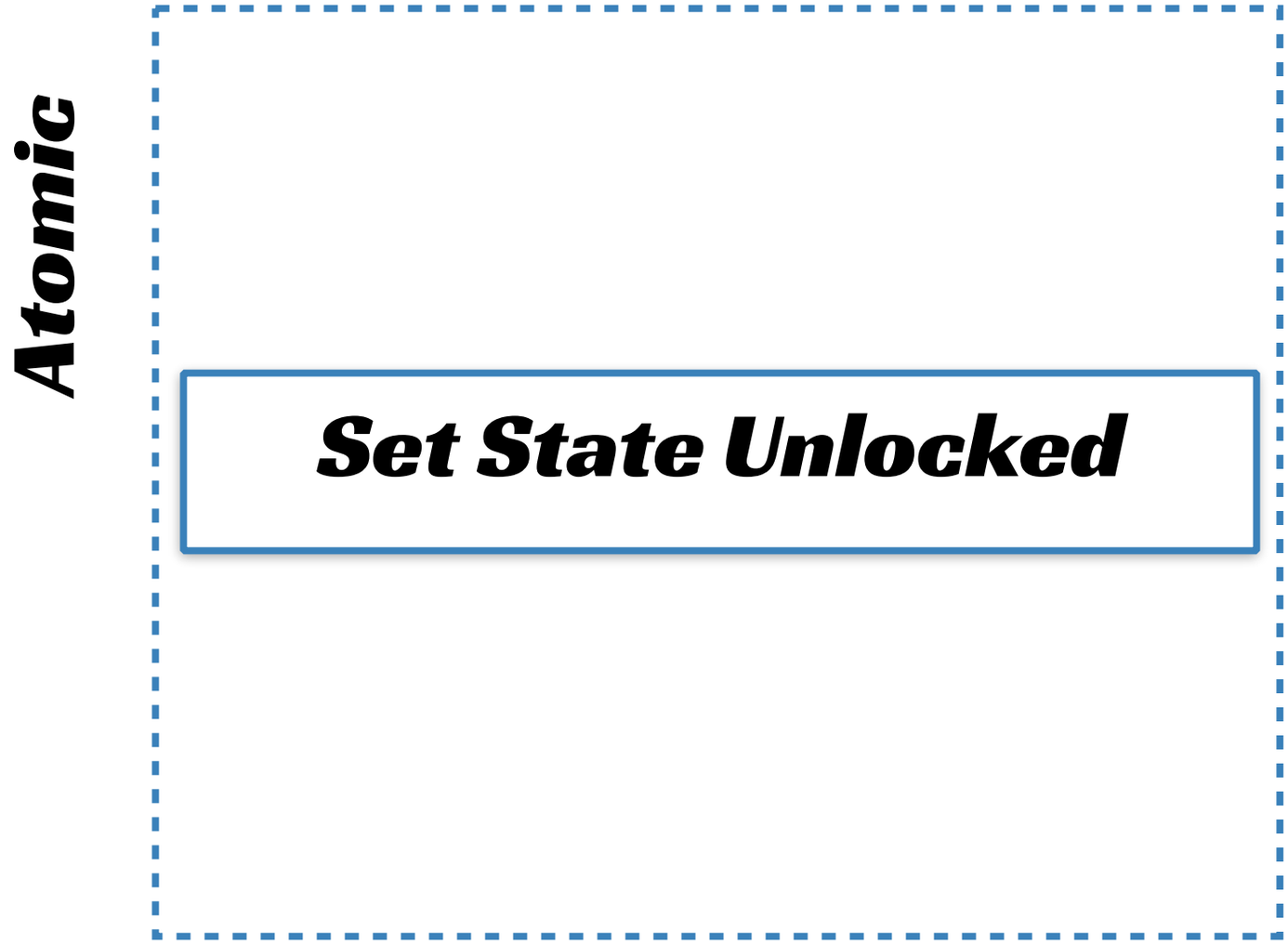

Atomic
Yes
Done
Was the State Locked?
No


Atomic
Fetch Old Lock State
Set State Locked
Was old State Locked?
Yes
No
Done

Test And Set Lock



Test And Set Unlock



```
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

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

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

TAS is embedded in Lock

{ A(TAS request)

A(TAS response) }

TAS is embedded in Lock

A(lock request)

Time

{ A(TAS request)

A(TAS response) }

A(lock response)

TAS & Store can’t be reordered

Time

A(lock request)

{ A(TAS request)

A(TAS response) }

A(lock response)

TAS & Store can't be reordered

Time

B(unlock request)

{ B(Store request)

B(Store response) }

B(unlock response)

A(lock request)

{ A(TAS request)

A(TAS response) }

A(lock response)

All execution histories

$\supseteq$

All sequentially-consistent execution histories

All execution histories

$\supseteq$

All sequentially-consistent execution histories

$\supseteq$

All ???able execution histories

All execution histories

$\supseteq$

All sequentially-consistent execution histories

$\supseteq$

All linearizable execution histories

Others can be reordered

Time

B(unlock request)

{ B(Store request) }

{ B(Store response) }

B(unlock response)

A(lock request)

{ A(TAS request) }

{ A(TAS response) }

A(lock response)

Others can be reordered

Time

B(unlock request)

{ B(Store request)

B(Store response) }

A(lock request)

B(unlock response)

{ A(TAS request)

A(TAS response) }

A(lock response)

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

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

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]

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

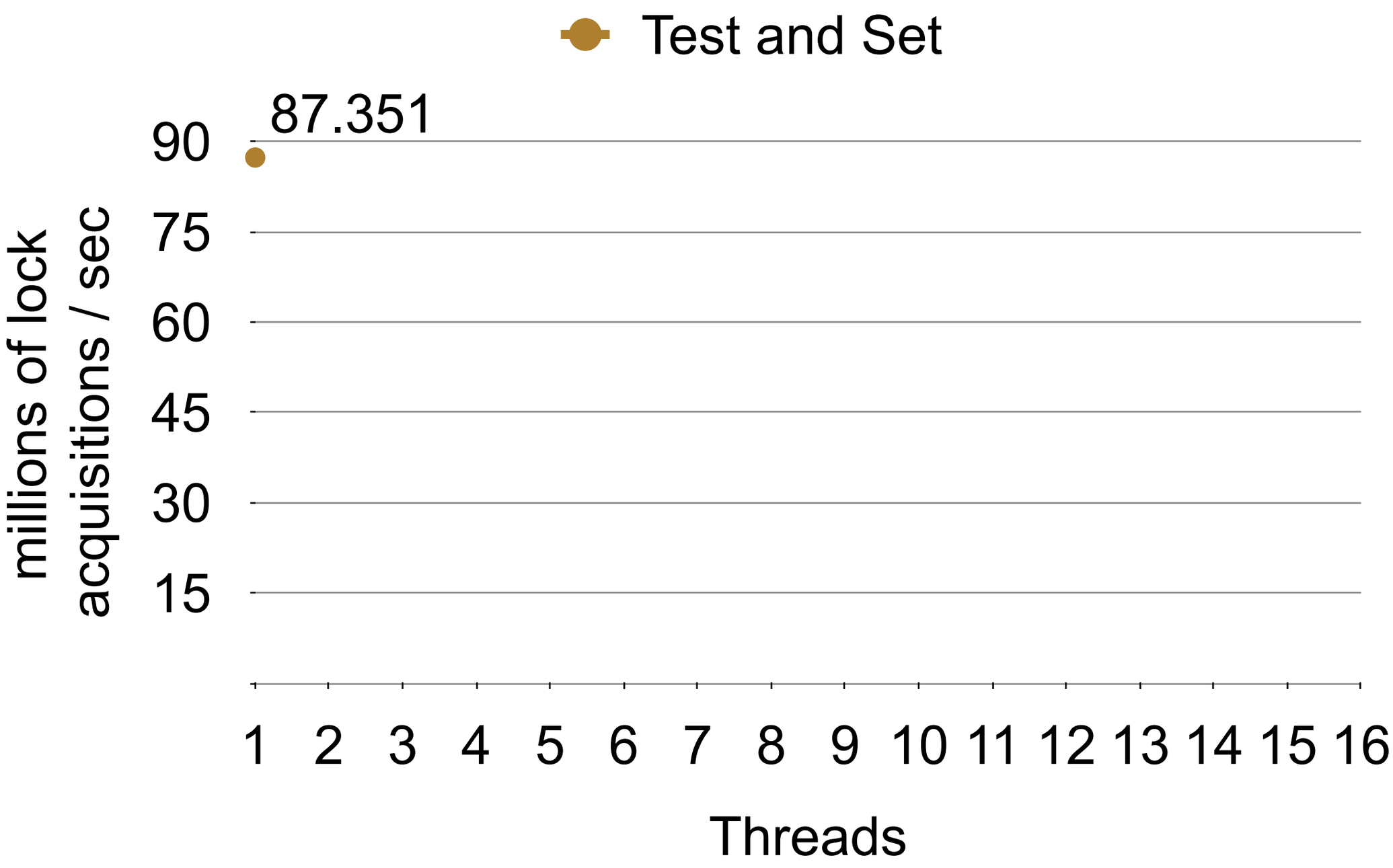
```
obj *allocate(slab *s) {
  lock(&allocator_lock);
  obj *a = s->head;
  if (a == NULL) return NULL;
  s->head = a->next;
  unlock(&allocator_lock);
  return a;
}
```

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

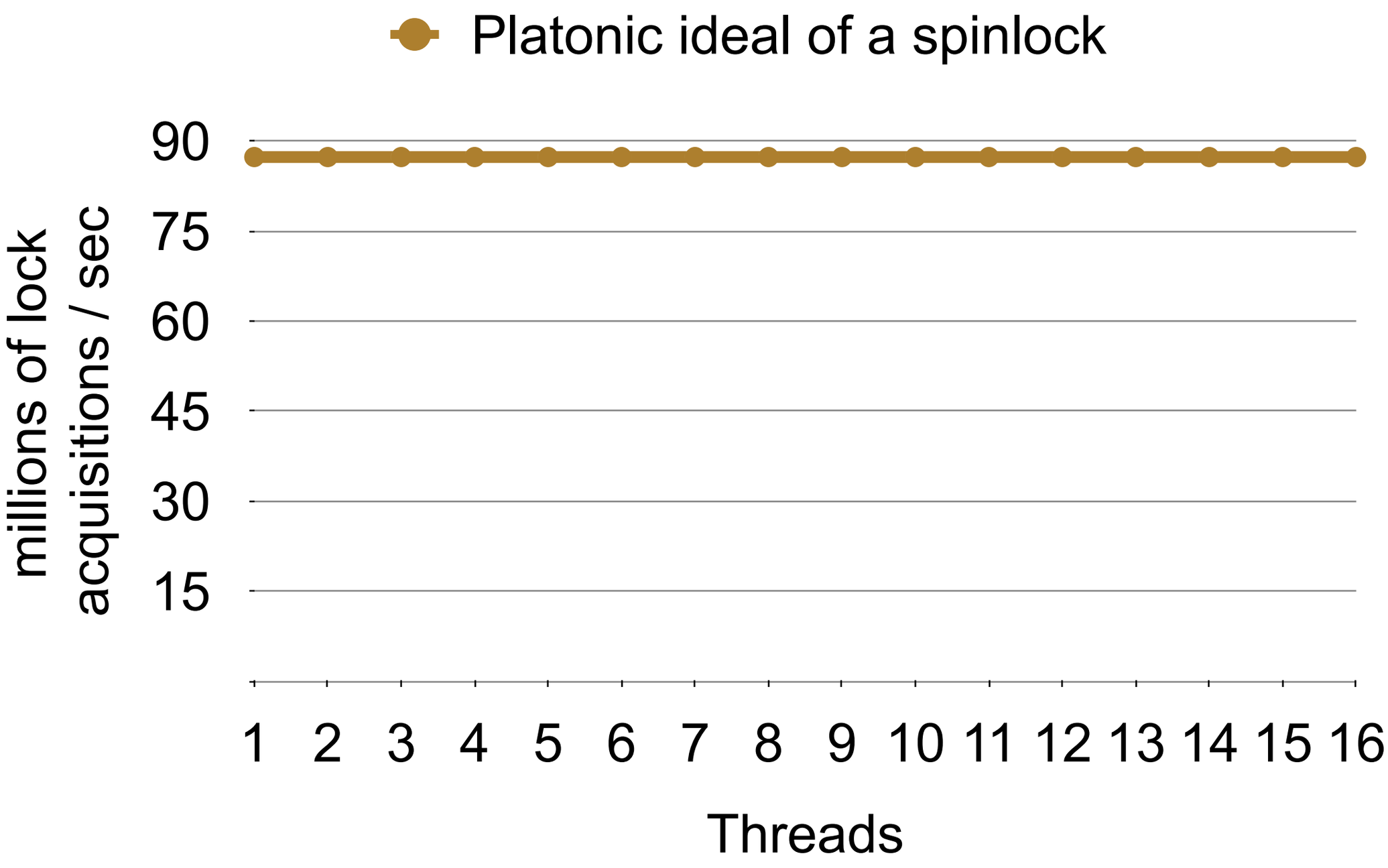
```
void lock(spinlock *m) {
  while (atomic_tas(m, LOCKED) == LOCKED)
    snooze();
}
```

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

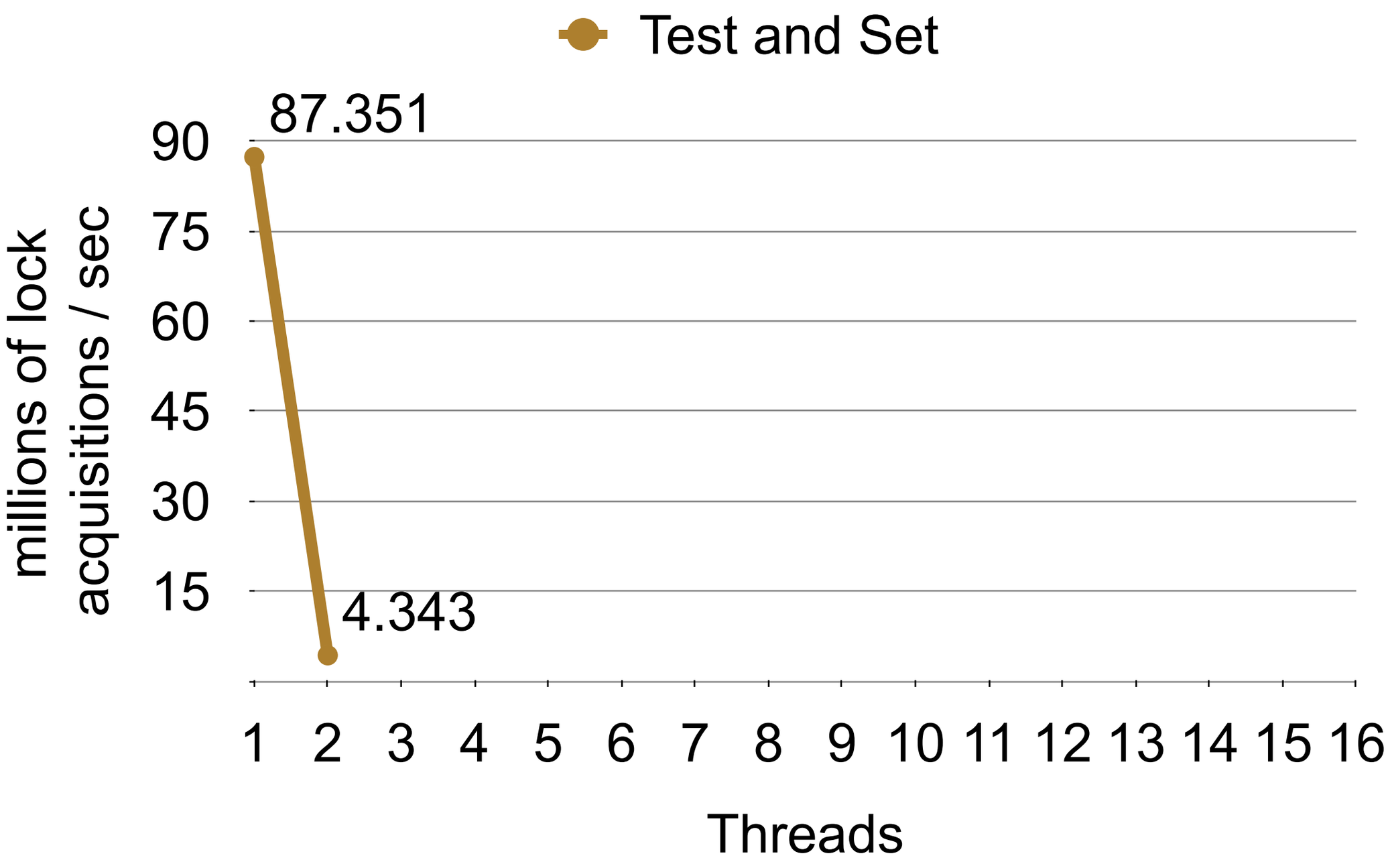
Spinlock performance



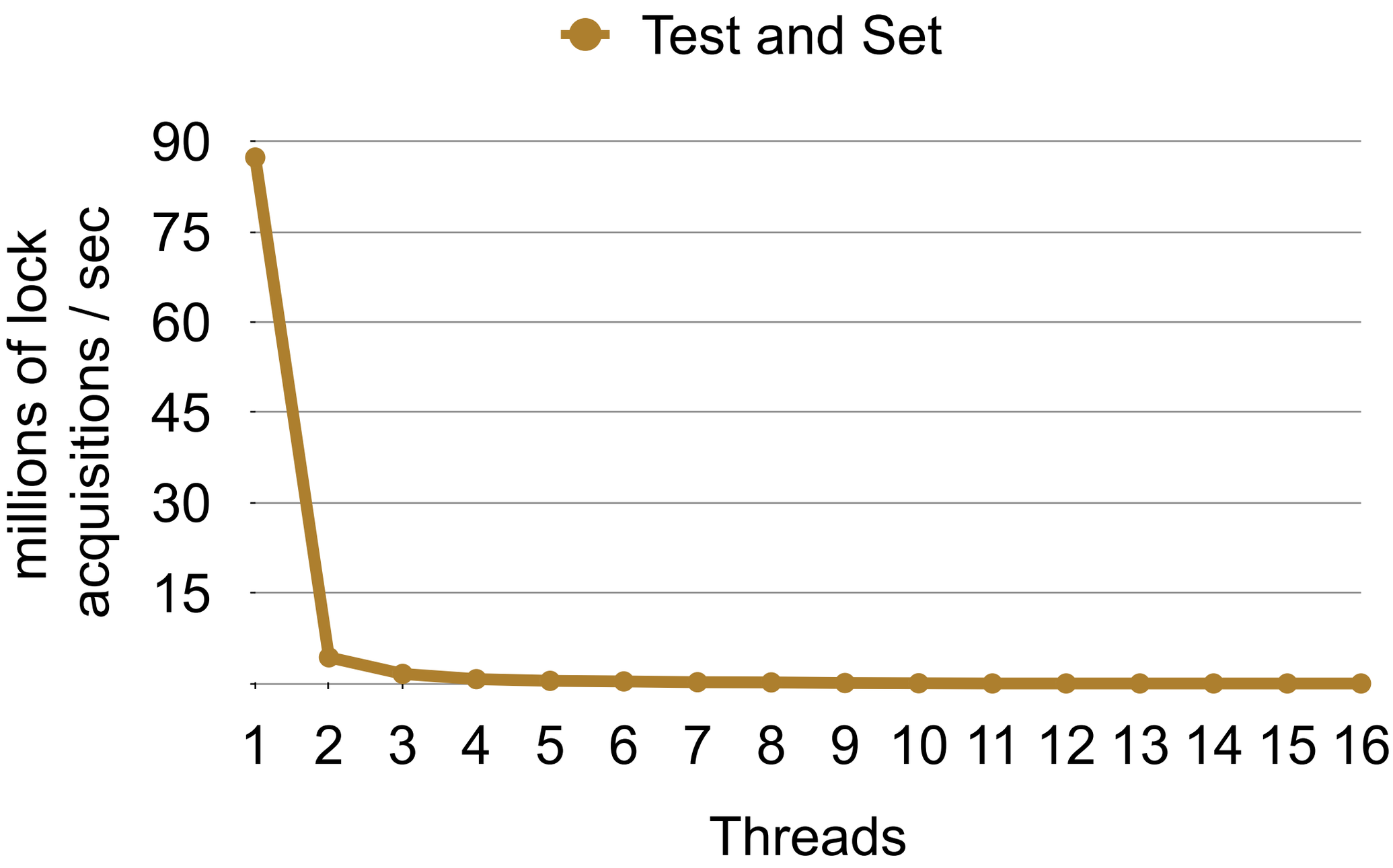
Spinlock performance



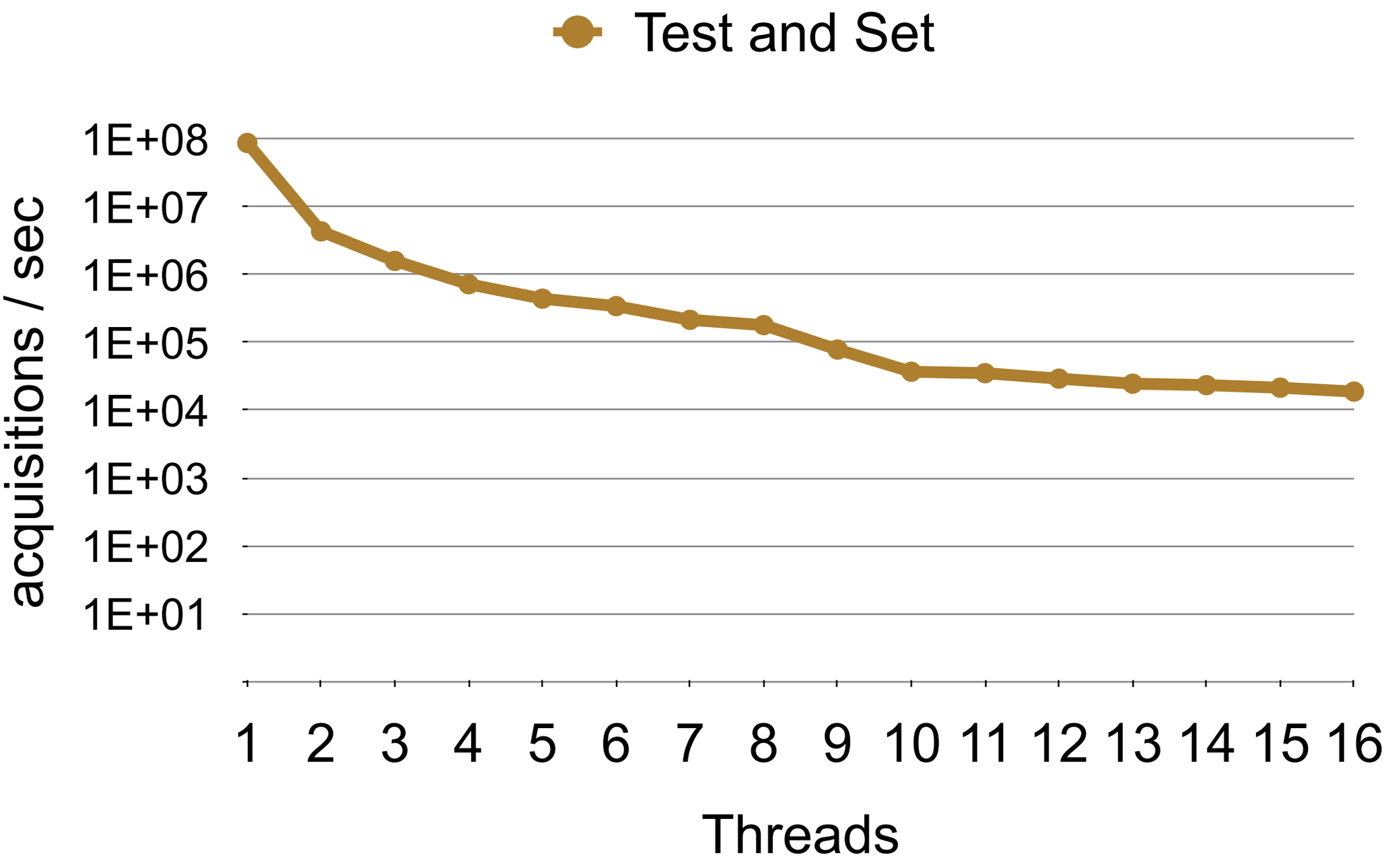
Spinlock performance



Spinlock performance



Spinlock performance

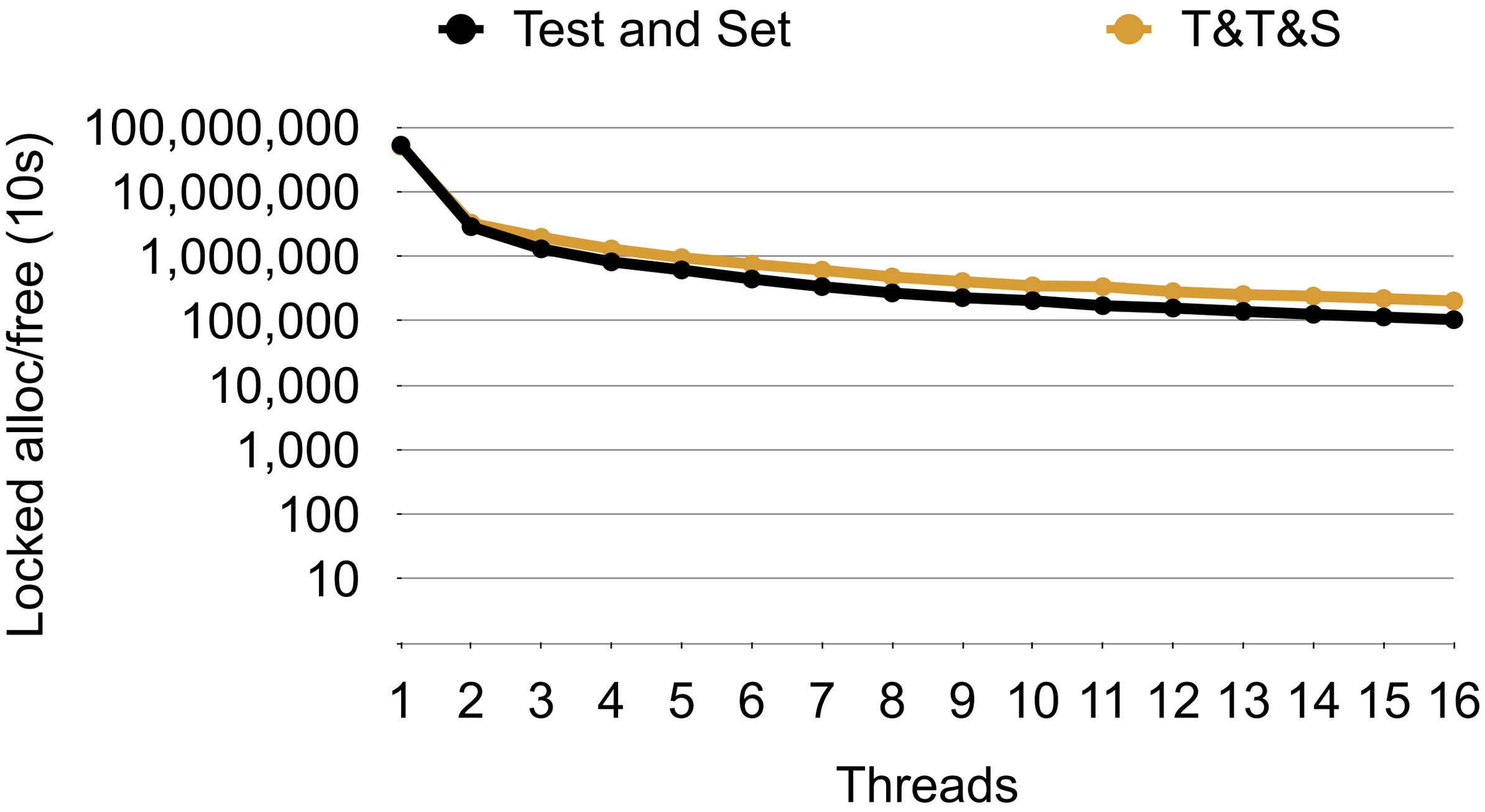


```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

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

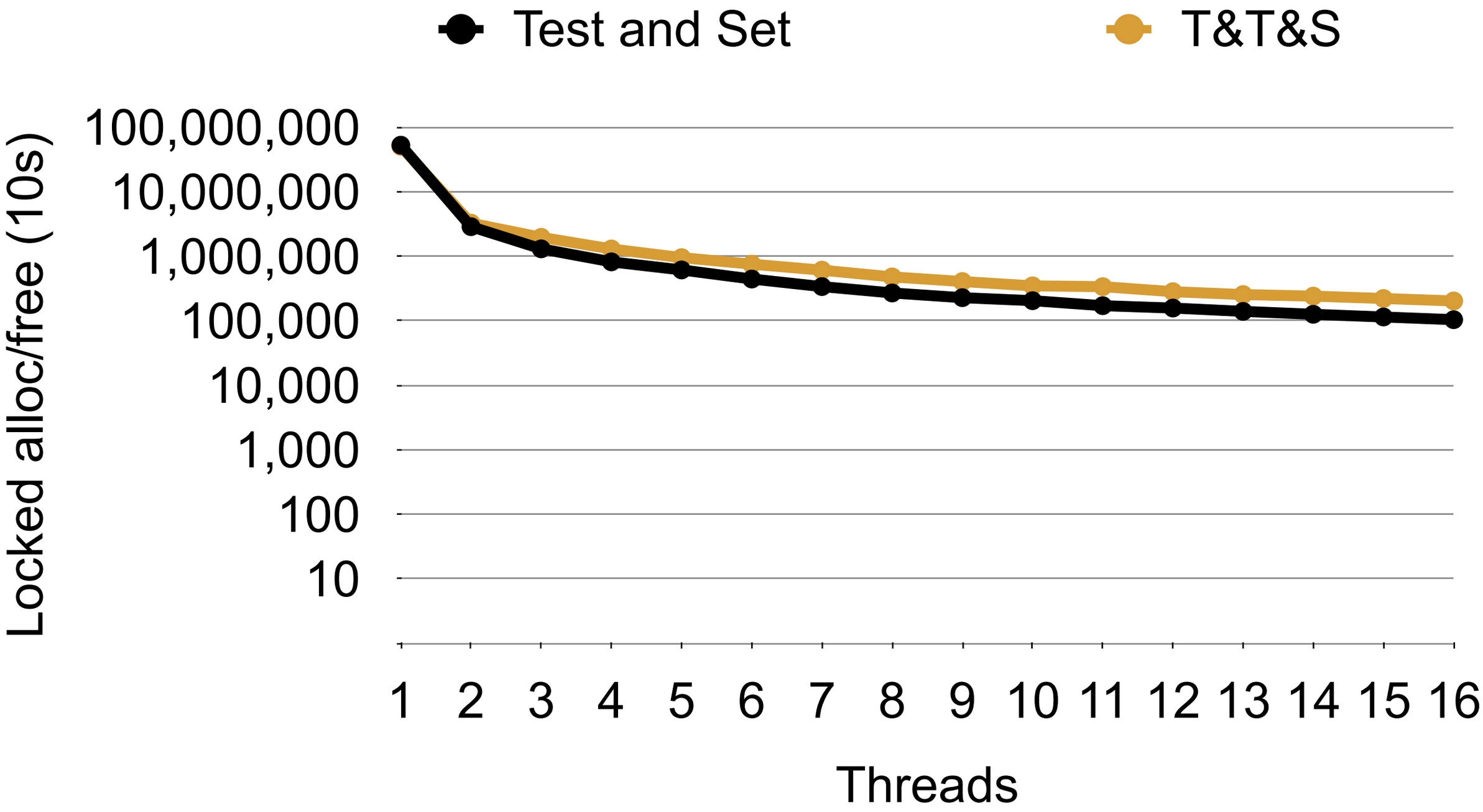
```
typedef spinlock int;
#define LOCKED 1
#define UNLOCKED 0

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

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();
   }
}
```


Test and Set
T&T&S
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10
Locked alloc/free (10s)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Threads

Spinlock performance



```
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++);
  }
}
```

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++);
  }
}
```

Test and Set
T&T&S
TAS + EB
60,000,000
50,000,000
40,000,000
30,000,000
20,000,000
10,000,000
Locked alloc/free (10s)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Threads

```
 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();
}
```

```
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();
}
```

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();
}
```

```
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();
}
```

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();
}
```

```
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();
}
```

```
 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();
}
```

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();
}
```

```
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();
}
```

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();
}
```

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

(Herlihy & Shavit)

```
// 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;
}
```

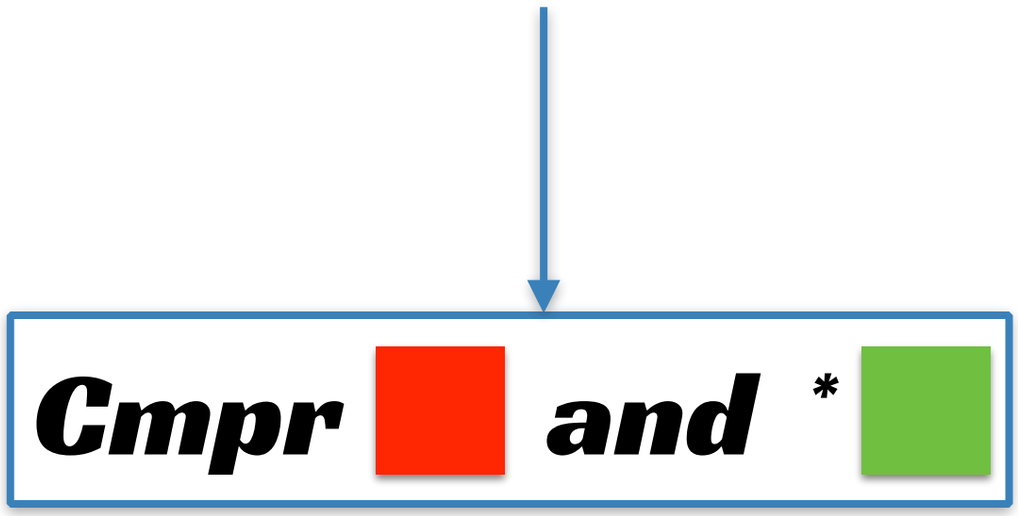
Non-Blocking Algorithms

Compare-And-Swap

Compare-And-Swap

Old value

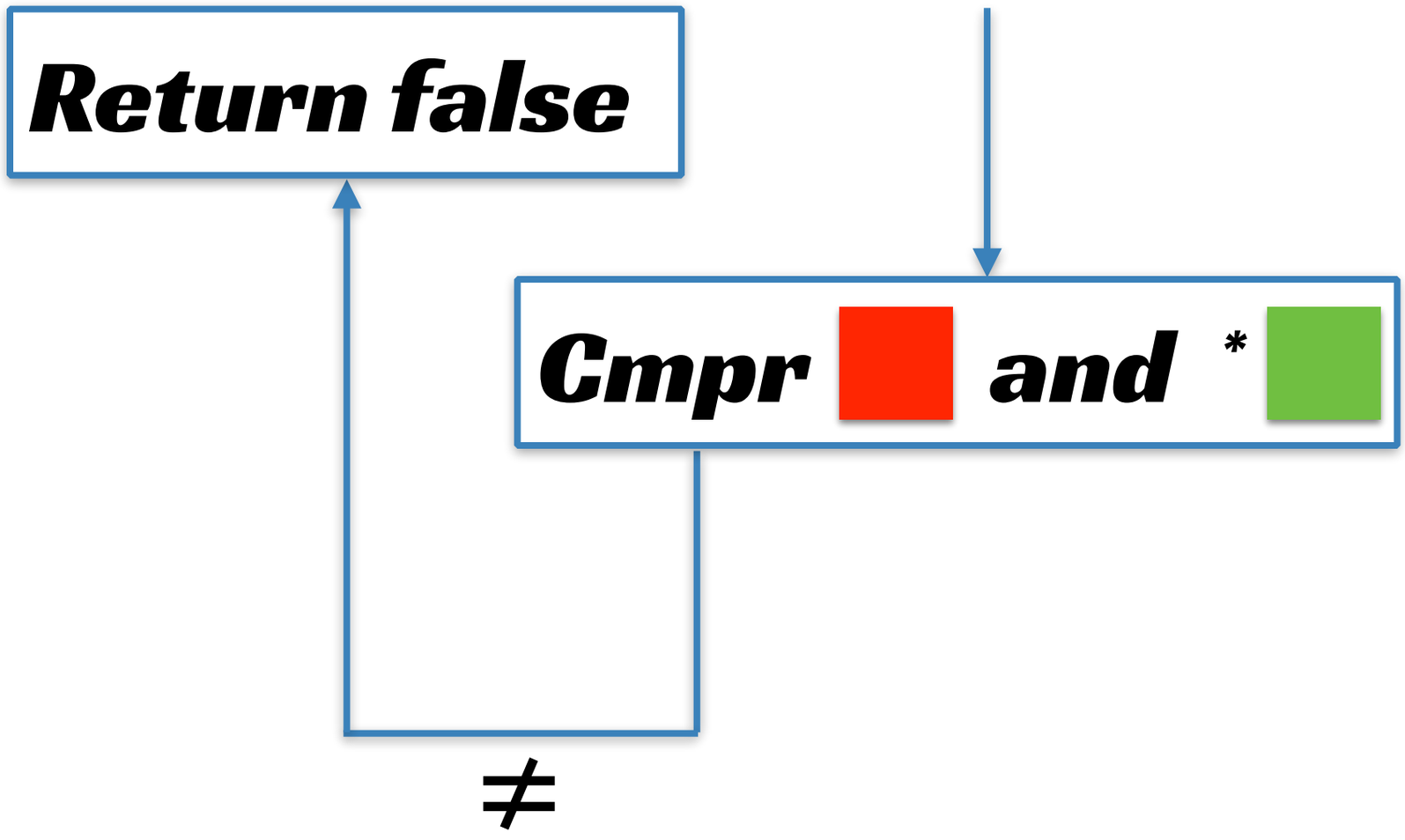
Destination Address



Compare-And-Swap

Old value

Destination Address



Compare-And-Swap

Old value

New value

Destination Address

Return false

Return true

Cmpr ***and*** * ***Copy*** ***to*** *

≠

=

Compare-And-Swap

Old value

New value

Destination Address

Return false

Return true

Cmpr **and** *

Copy **to** *

≠

=

Atomic

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));

}
```

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));

}
```

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));

}
```

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));

}
```

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));
}
```

TAS using CAS

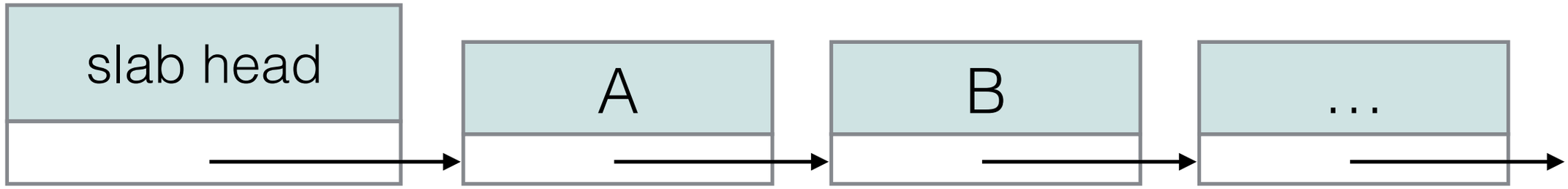
```
void tas_loop(spinlock *m) {
   do {
      ;
   } while (!cas(UNLOCKED, LOCKED, m));
}
```

Read/Modify/Write

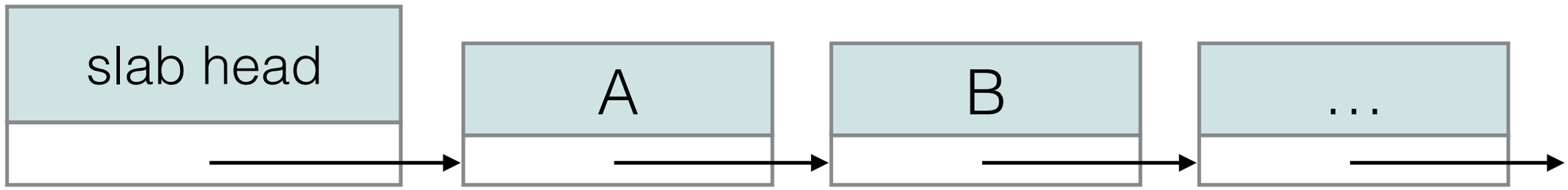
```
void atomic_inc_mod_32(int *ptr) {
  int i, new_i;
  do {
    i = *ptr;                          /* Read */
    new_i = fancy_function();    /* Modify */
  } while (!cas(i, new_i, ptr)); /* Write */

}
```

Read/Modify/Write

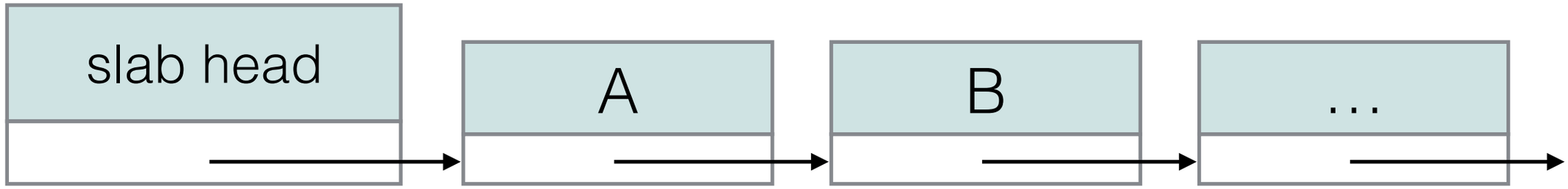
```
void atomic_inc_mod_32(int *ptr) {
  int i, new_i;
  do {
    i = *ptr;                                  /* Read */
    new_i = fancy_function();                  /* Modify */
  } while (!cas(i, new_i, ptr));               /* Write */
                                               /* (or retry) */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



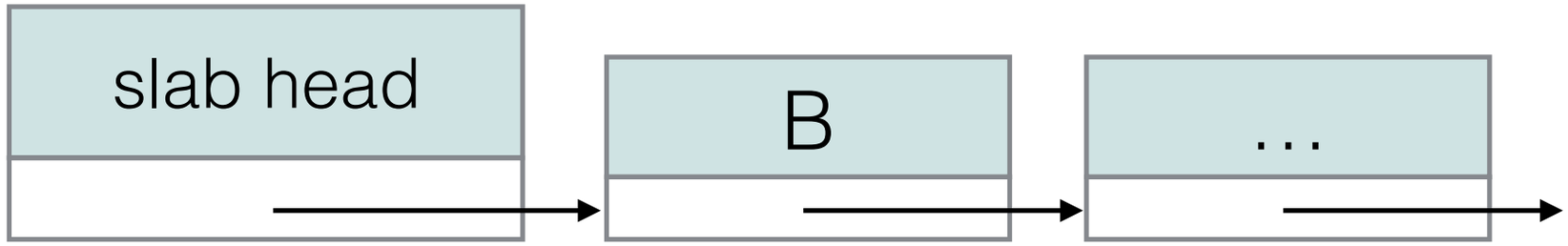

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head ));
  return a;
}
```



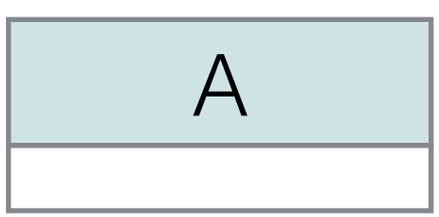

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head ));
  return a;
}
```

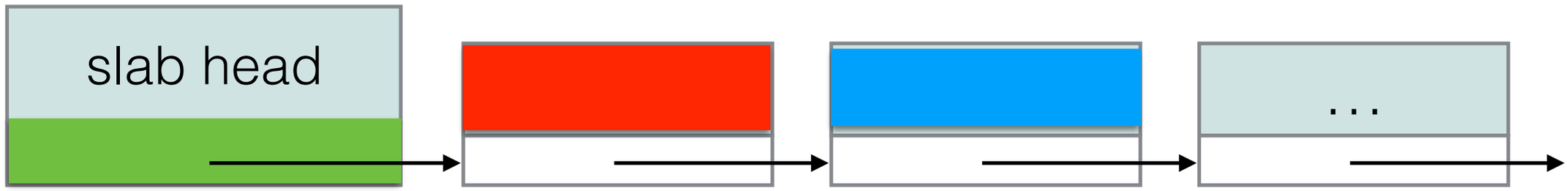



```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head ));
  return a;
}
```



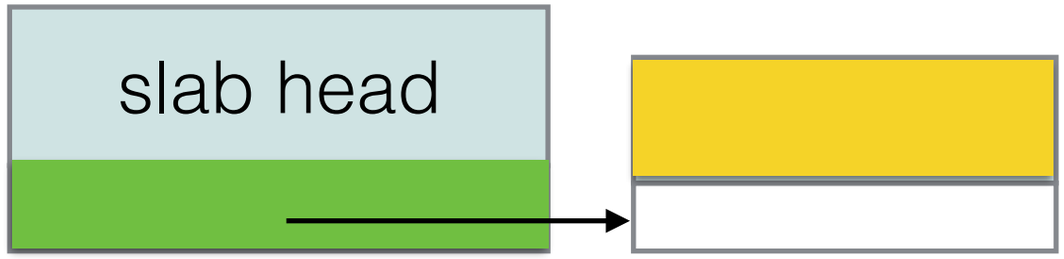

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head ));
  return a;
}
```





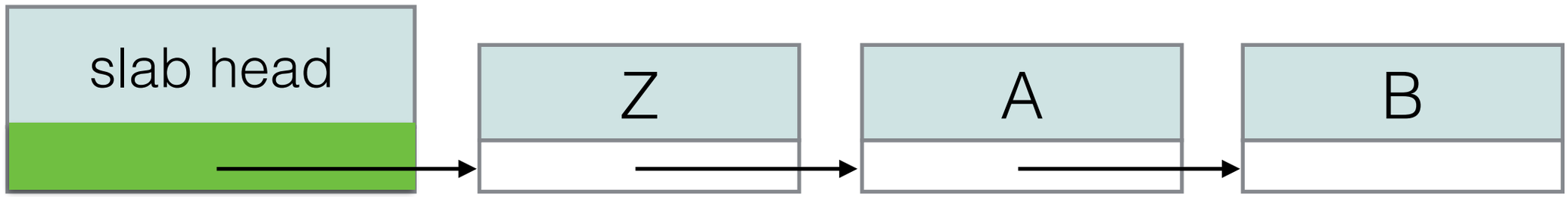

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

***Cmpr* and ***



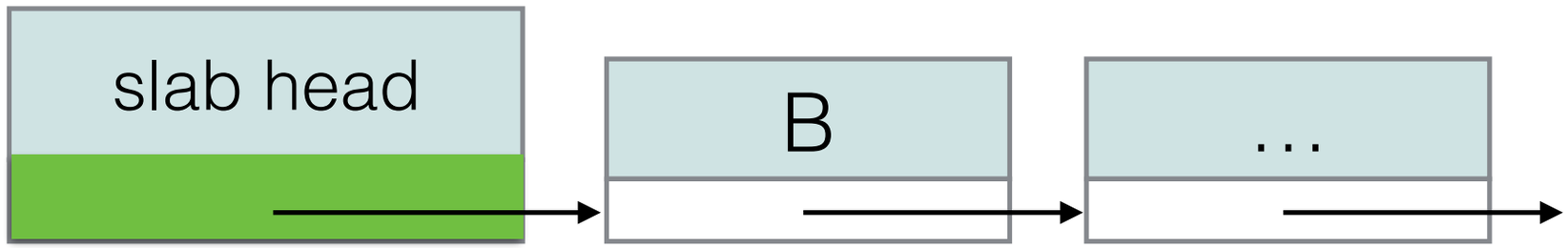

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a,b, &s->head ));
  return a;
}
```

Cmpr and



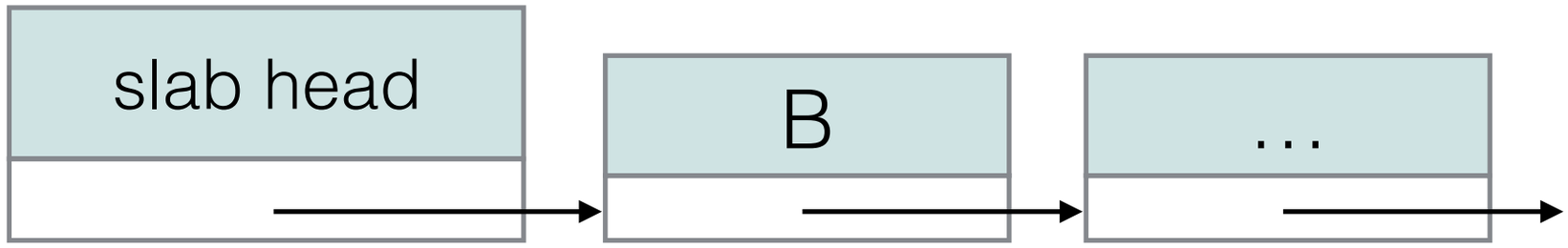

```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

***Cmpr* and**

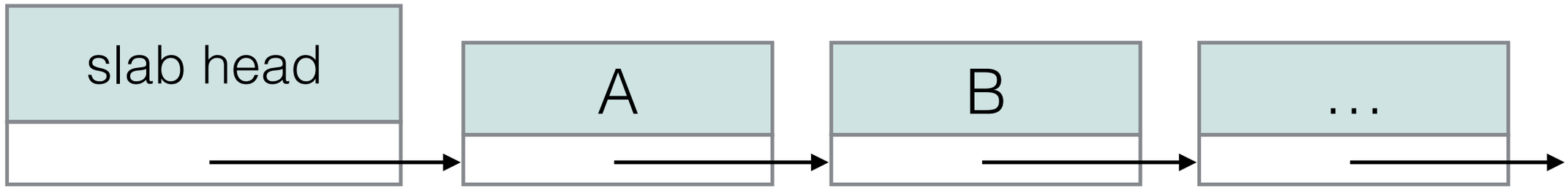



```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
     a = s->head;
     if (a == NULL) return NULL;
     b = a->next;
  } while (!cas( a , b , &s->head ));
  return a;
}
```

Cmpr ■ ***and*** ■




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

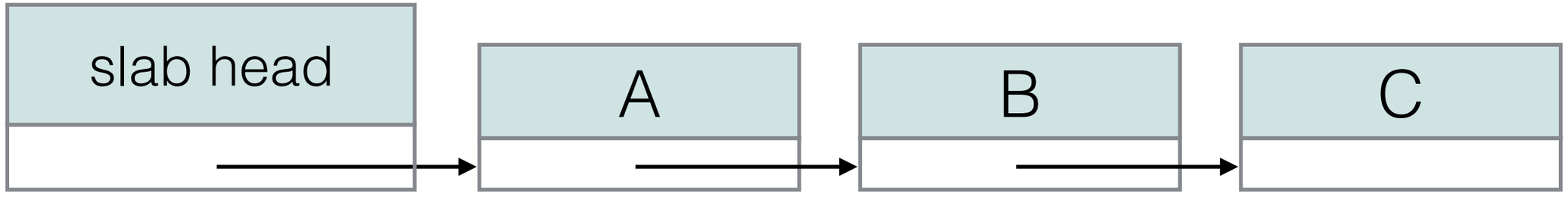



```
void free(slab *s, obj *o) {
    do {
        obj *t = s->head;
        o->next = t;
    } while (!cas(t, o, &s->head));
}
```

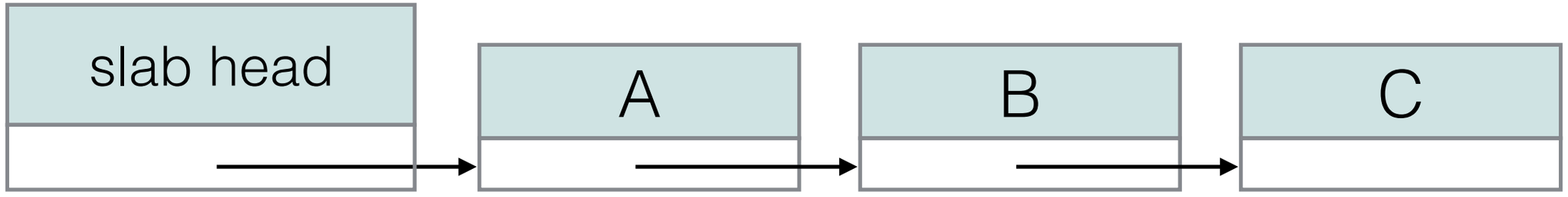

slab head
A
B
C


slab head
A
B
C

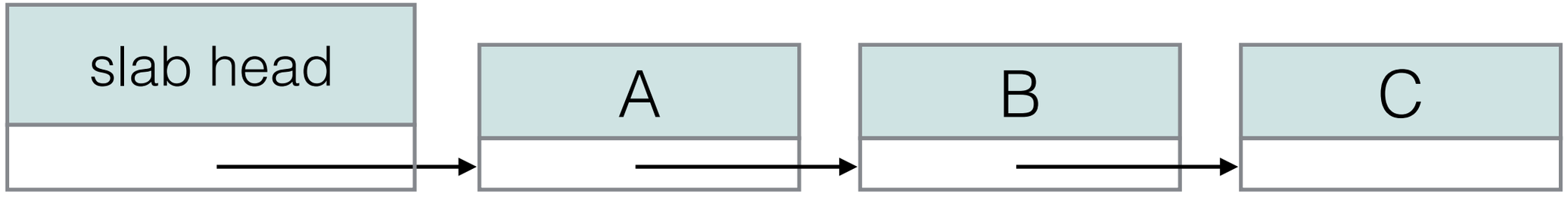
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



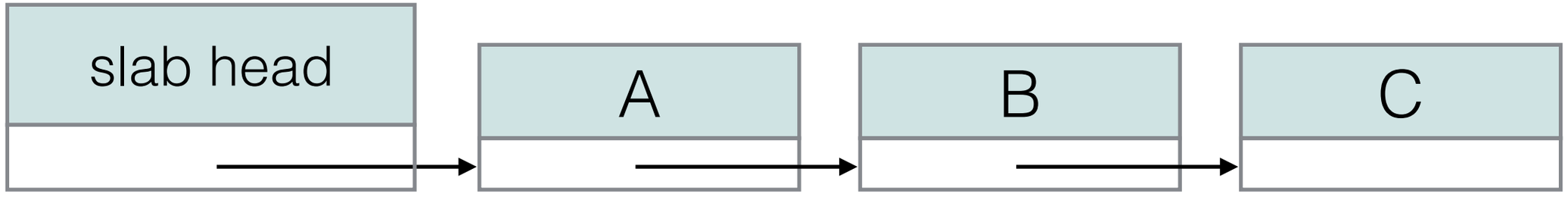
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



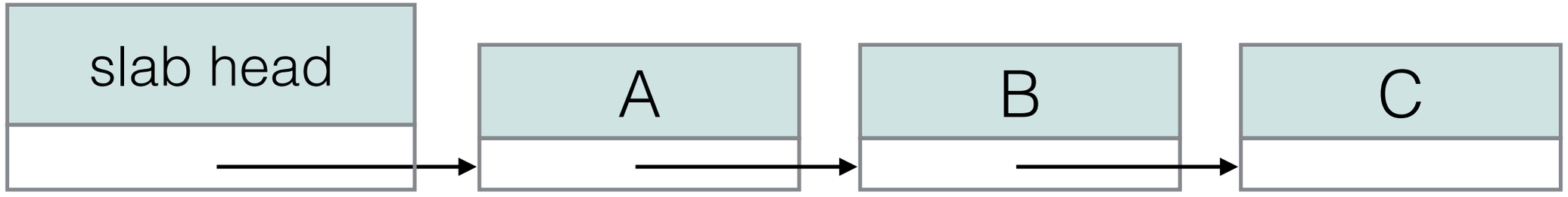
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



```
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;
}
```

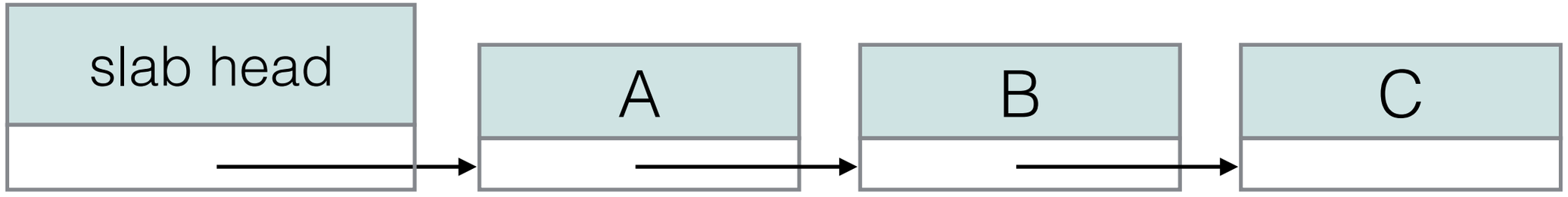


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



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

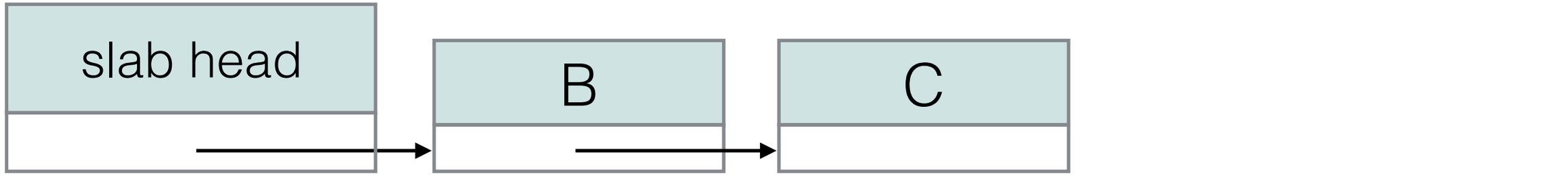
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



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

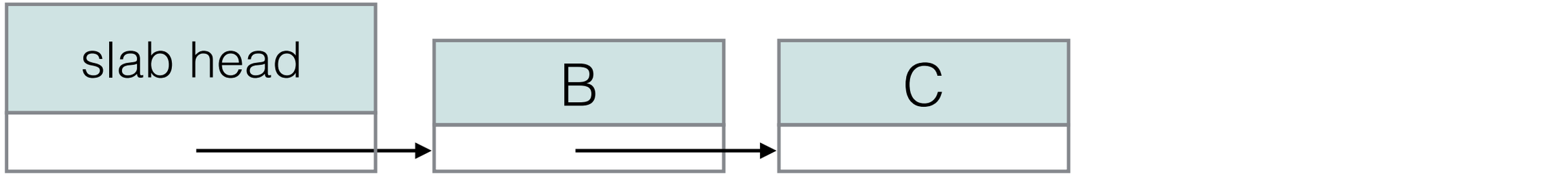
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

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

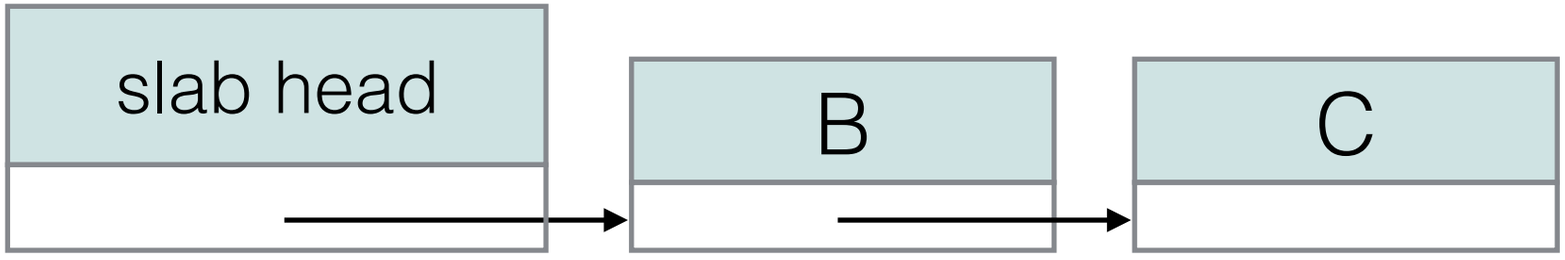


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

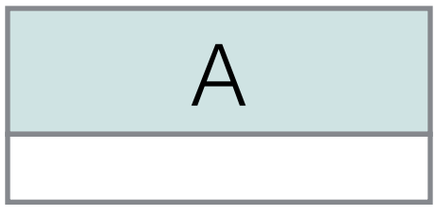
```
some_object = allocate(&shared_slab);
```



```
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;
}
```

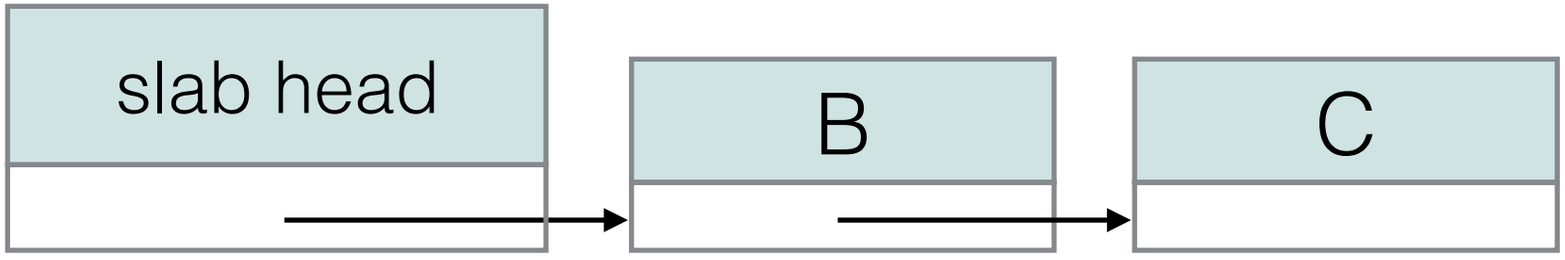


```
some_object = allocate(&shared_slab);
```

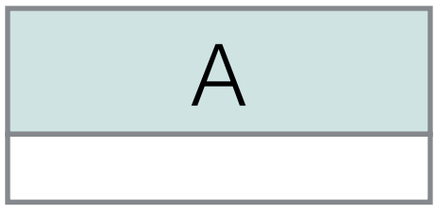


```
another_obj = allocate(&shared_slab);
```

```
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;
}
```

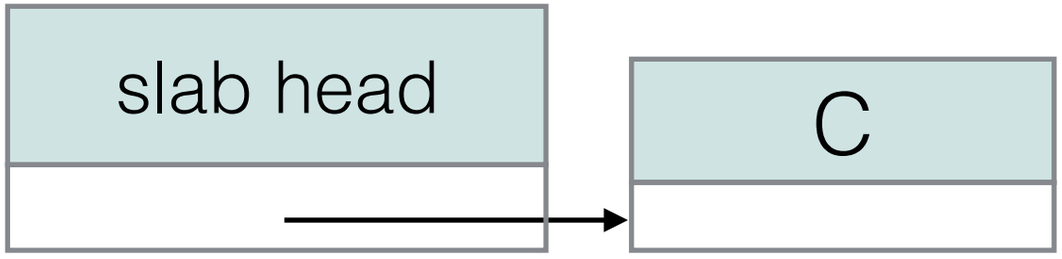


```
some_object = allocate(&shared_slab);
```

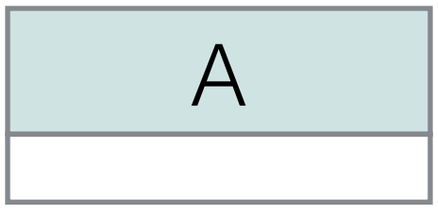


```
another_obj = allocate(&shared_slab);
```

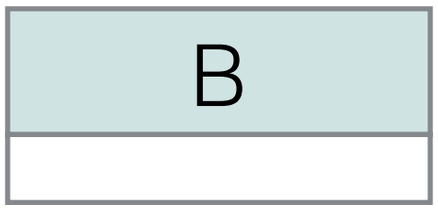
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



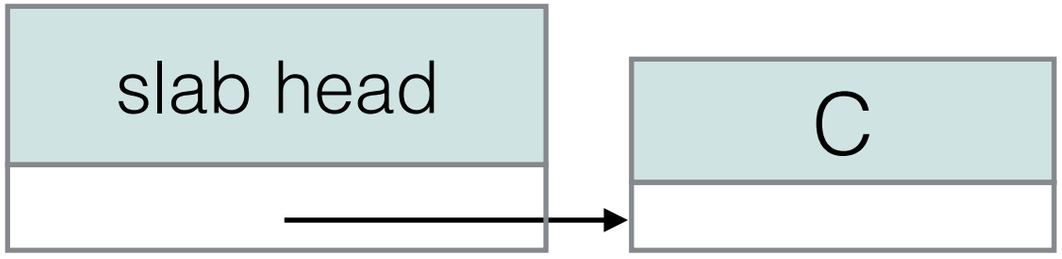
```
some_object = allocate(&shared_slab);
```



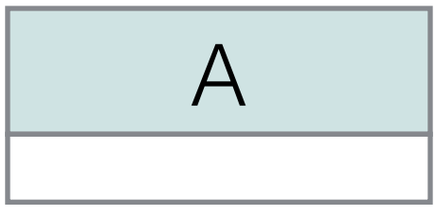
```
another_obj = allocate(&shared_slab);
```



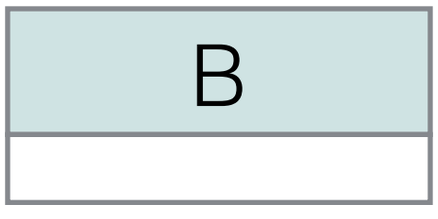
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



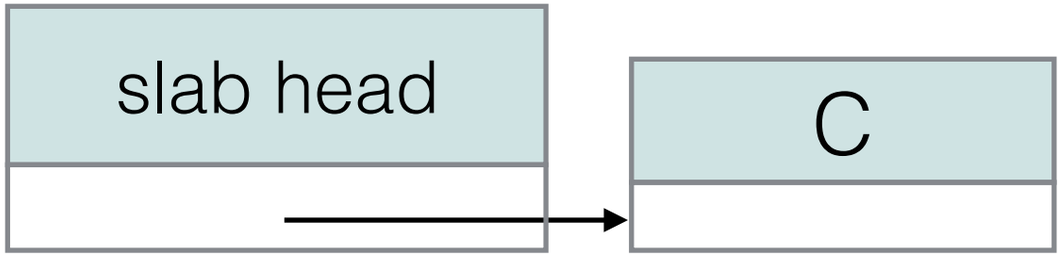
```
some_object = allocate(&shared_slab);
```



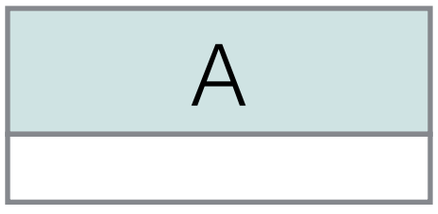
```
another_obj = allocate(&shared_slab);
```



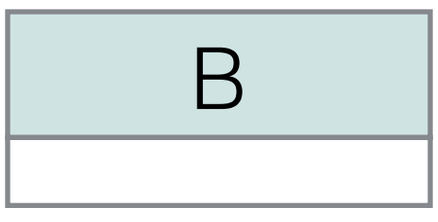
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



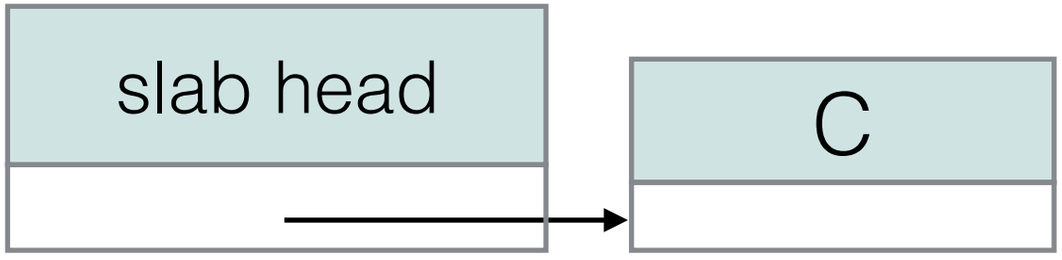
```
free(&shared_slab, some_object);
```



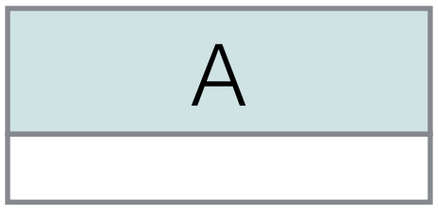
```
another_obj = allocate(&shared_slab);
```



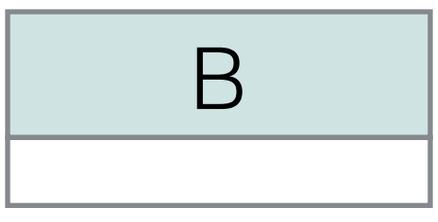
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



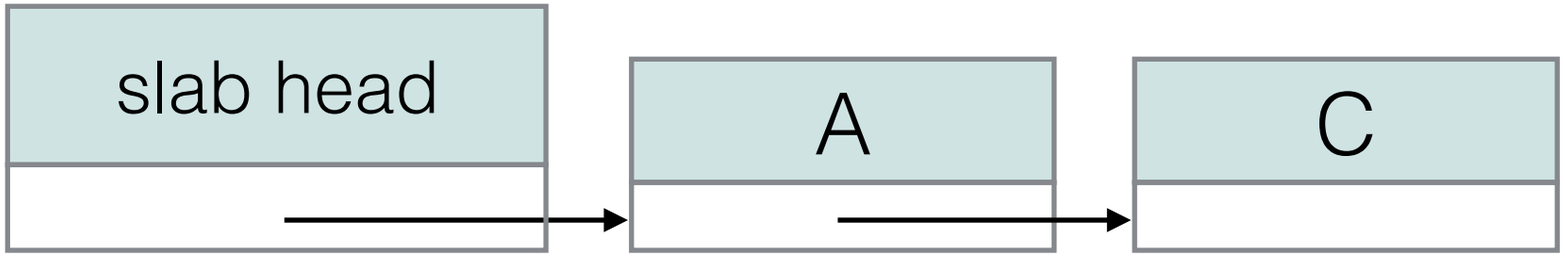
```
free(&shared_slab, some_object);
```



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

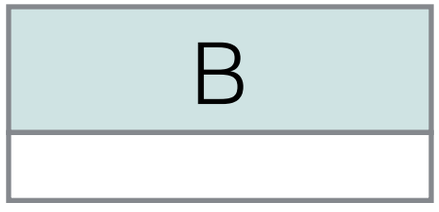


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

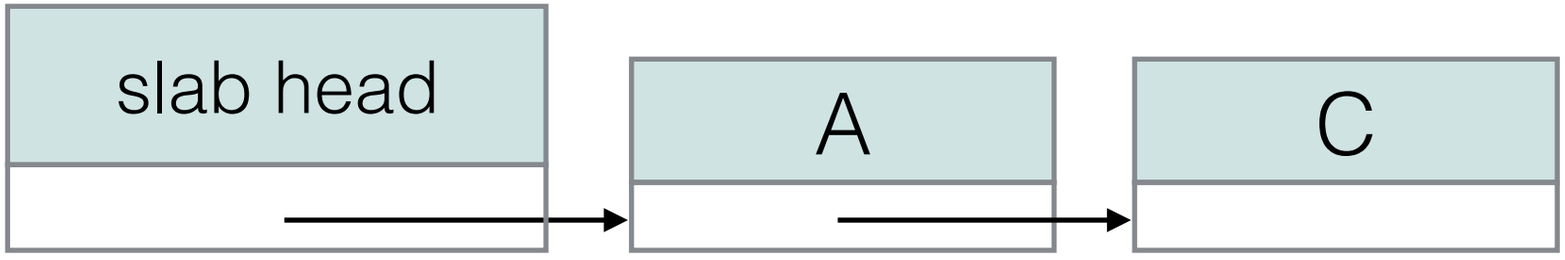


```
free(&shared_slab, some_object);
```

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

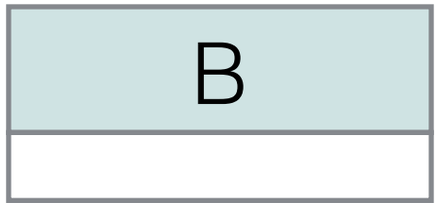


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

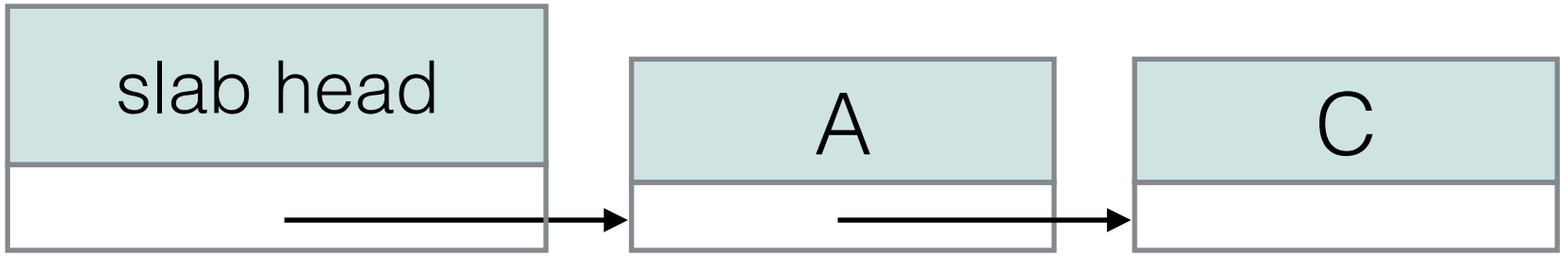


```
free(&shared_slab, some_object);
```

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

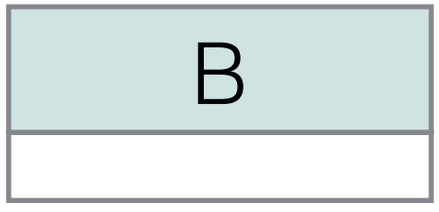


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

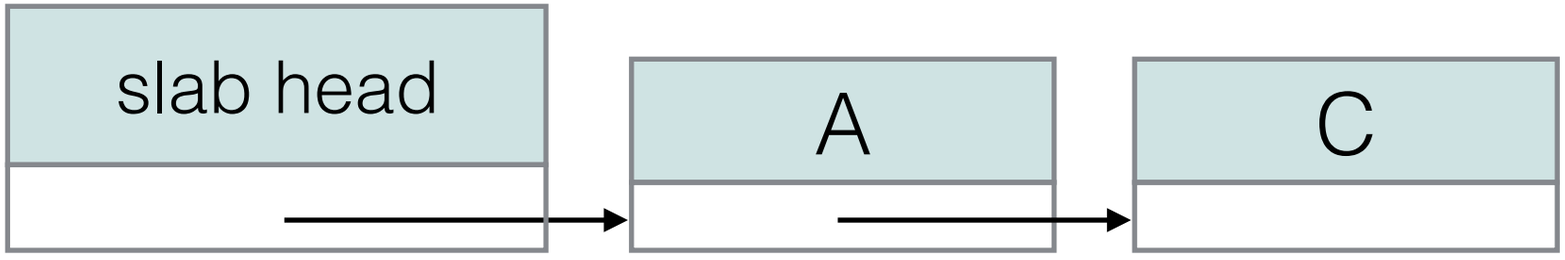


```
free(&shared_slab, some_object);
```

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

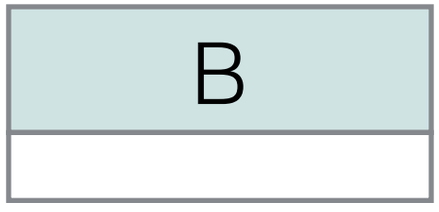


```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```

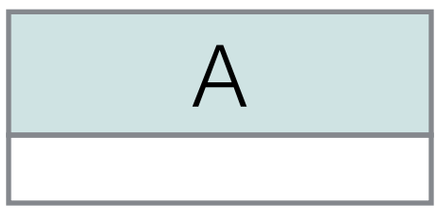


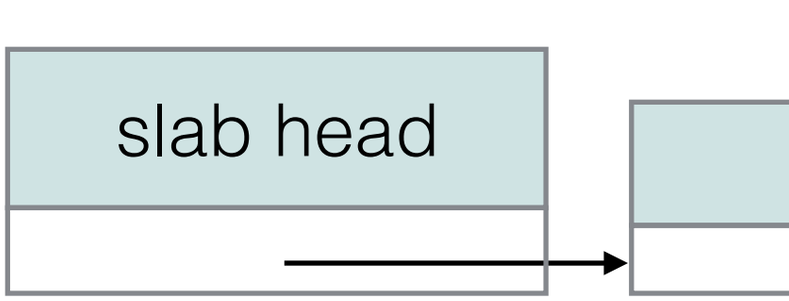
```
free(&shared_slab, some_object);
```

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



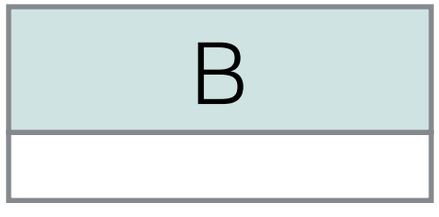
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



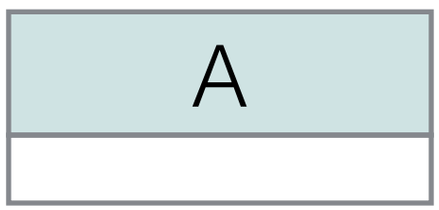


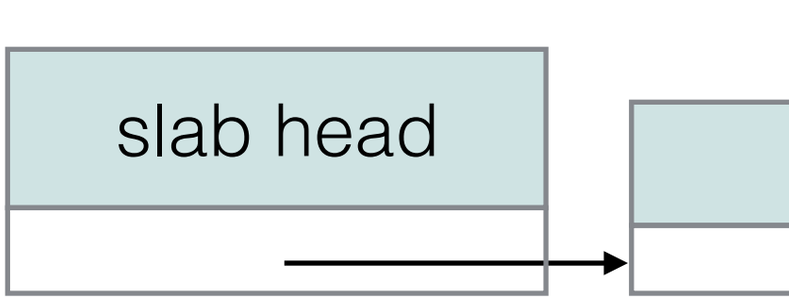
```
free(&shared_slab, some_object);
```

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



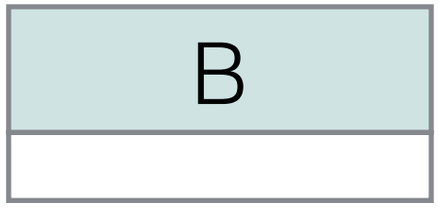
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```



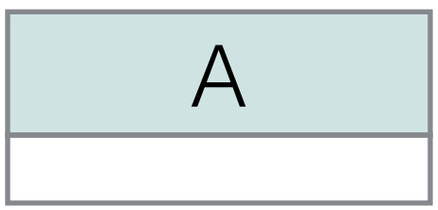


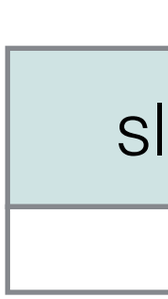
```
free(&shared_slab, some_object);
```

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



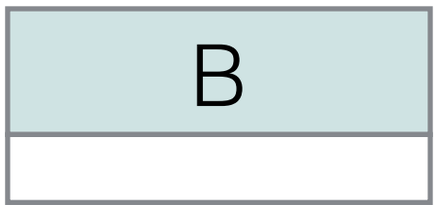
```
obj *allocate(slab *s) {
  obj *a, *b;
  do {
    a = s->head;
    if (a == NULL) return NULL;
    b = a->next;
  } while (!cas(a, b, &s->head));
  return a;
}
```





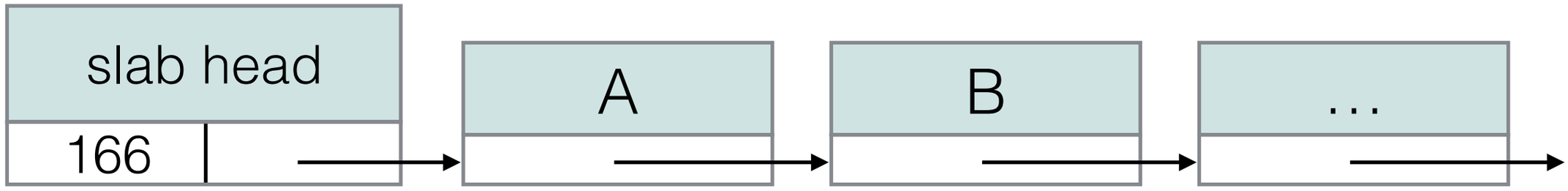
```
free(&shared_slab, some_object);
```

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

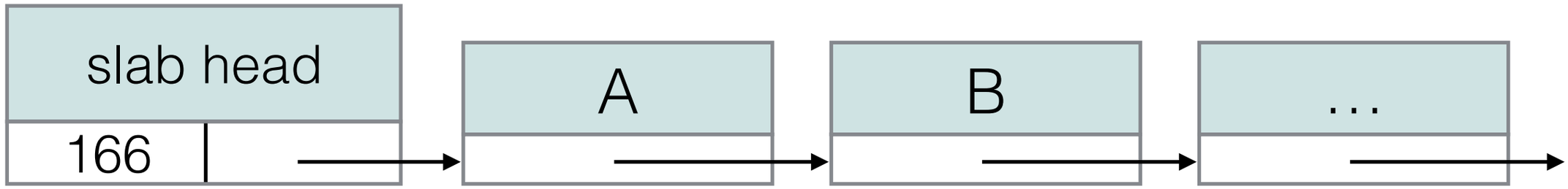


The ABA Problem

"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




```
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;
}
```

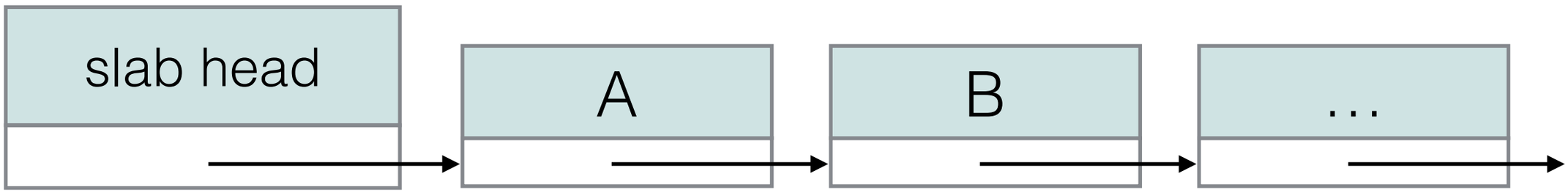



```
obj *allocate(slab *s) {
  slab orig, update;
  do {
    orig.gen = s.gen;
    orig.head = s.head;
    if (!orig.head) return NULL;
    update.gen = orig.gen + 1;
    update.head = orig.head->next;
  } while (!dcas(&orig, &update, s));
  return orig.head;
}
```

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

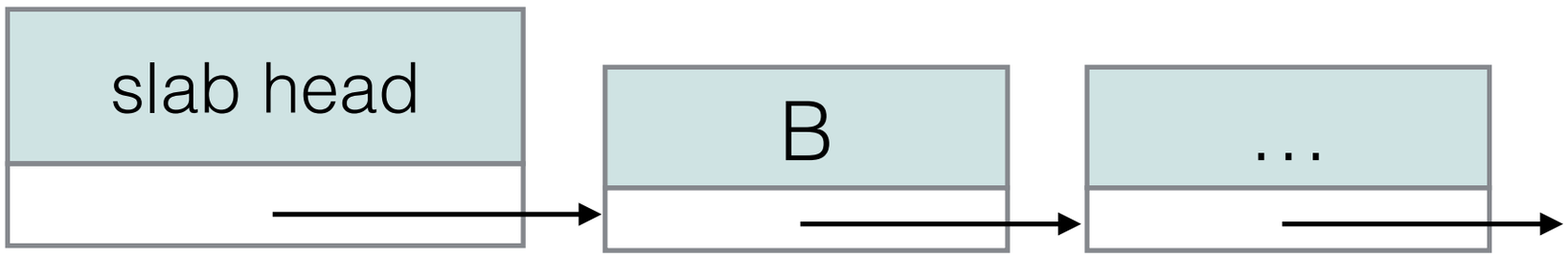
```
obj *allocate(slab *s) {
  lock(&allocator_lock);
  obj *a = s->head;
  if (a == NULL) return NULL;
  s->head = a->next;
  unlock(&allocator_lock);
  return a;
}
```

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



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

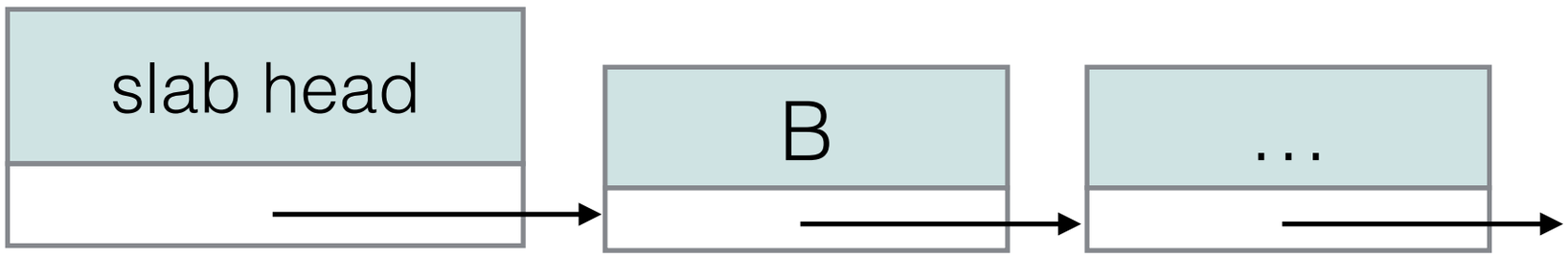
```
obj *o = allocate(&shared_slab);
```



A = allocate(&shared_slab);

A = allocate(&shared_slab);

Memory barriers



A = allocate(&shared_slab);

A = allocate(&shared_slab);

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

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

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

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

```
;;;; 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
```

```
;;;; 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
```

```
;;;; 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
```

Table C.5: Summary of Memory Ordering

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

Table C.5: Summary of Memory Ordering

	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	Y
IA64	Y	Y	Y	Y	Y	Y	Y	Y
(PA-RISC)	Y	Y	Y	Y	Y	Y	Y	Y
PA-RISC CPUs				Y				
POWER™	Y	Y	Y	Y	Y	Y	Y	Y
(SPARC RMO)	Y	Y	Y	Y	Y	Y	Y	Y
(SPARC PSO)			Y	Y	Y	Y	Y	Y
SPARC TSO				Y	Y	Y	Y	Y
x86				Y	Y	Y	Y	Y
(x86 OOStore)	Y	Y	Y	Y	Y	Y	Y	Y
zSeries®				Y	Y	Y	Y	Y

	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

	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

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.

```
;;;; 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
```

```
;;;; 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?
```

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.

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

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

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

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

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

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

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

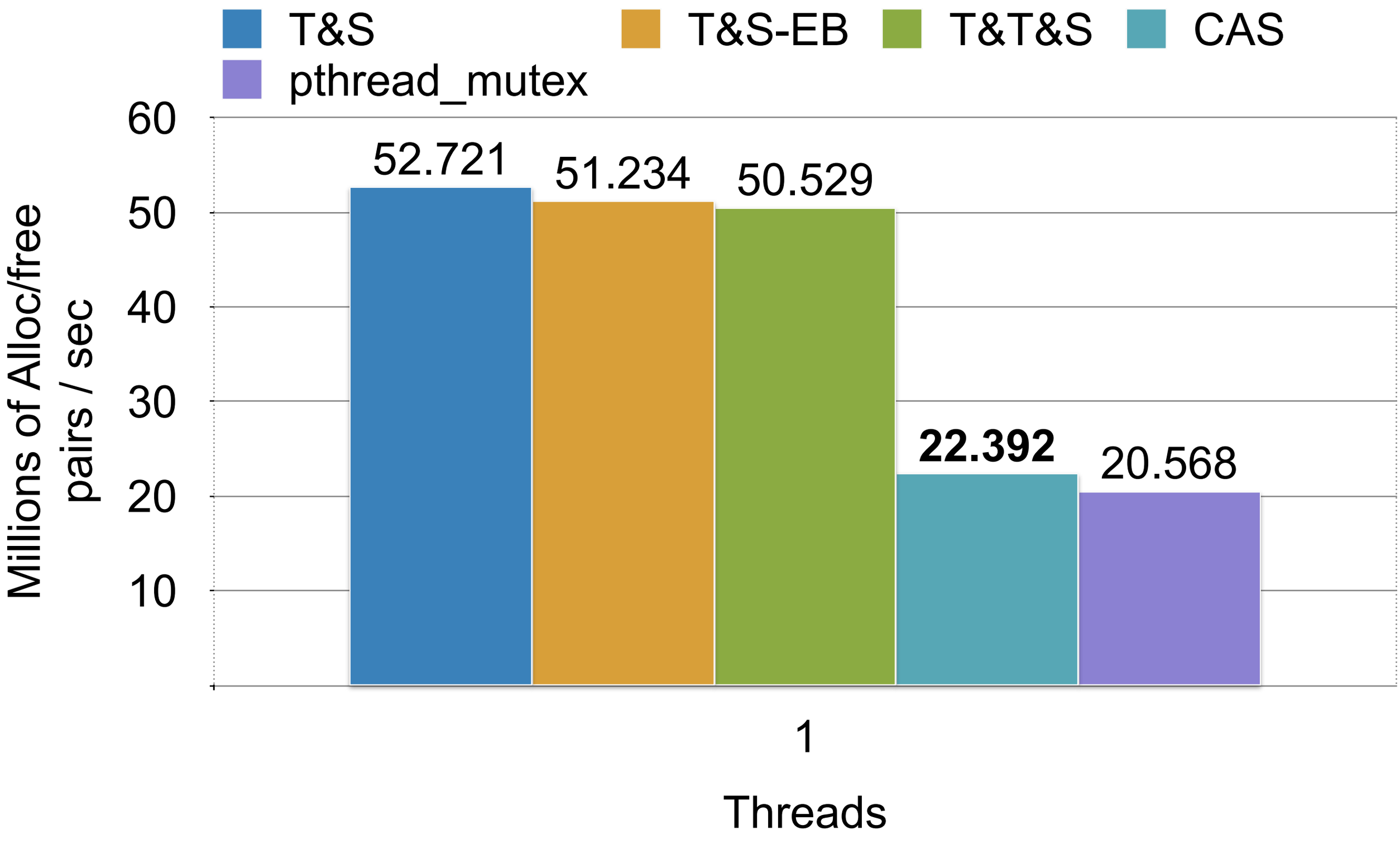
```
DMB  ; Data Memory Barrier
```

```
;;;; 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
```

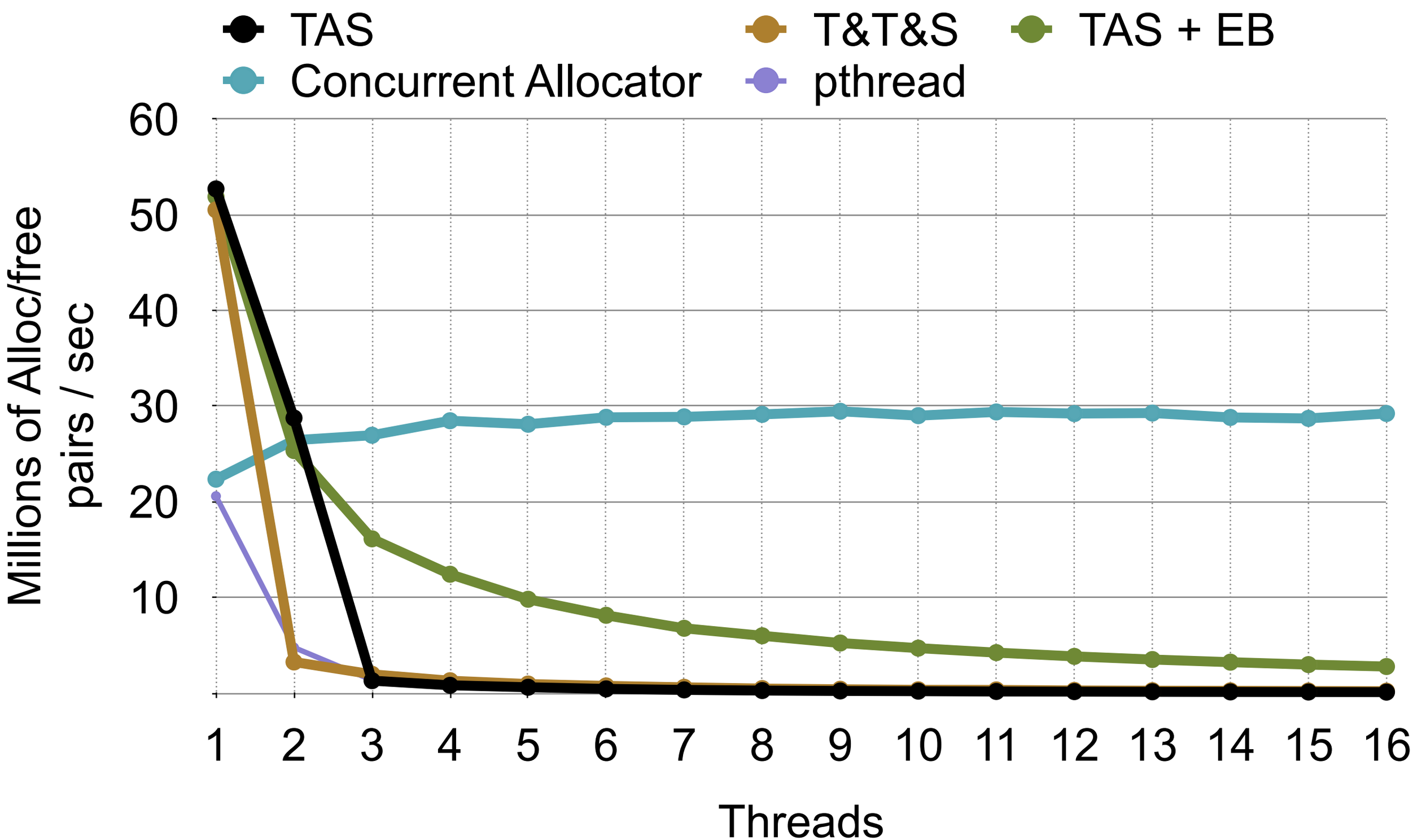
Allocator performance

```
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
```

Allocator Throughput

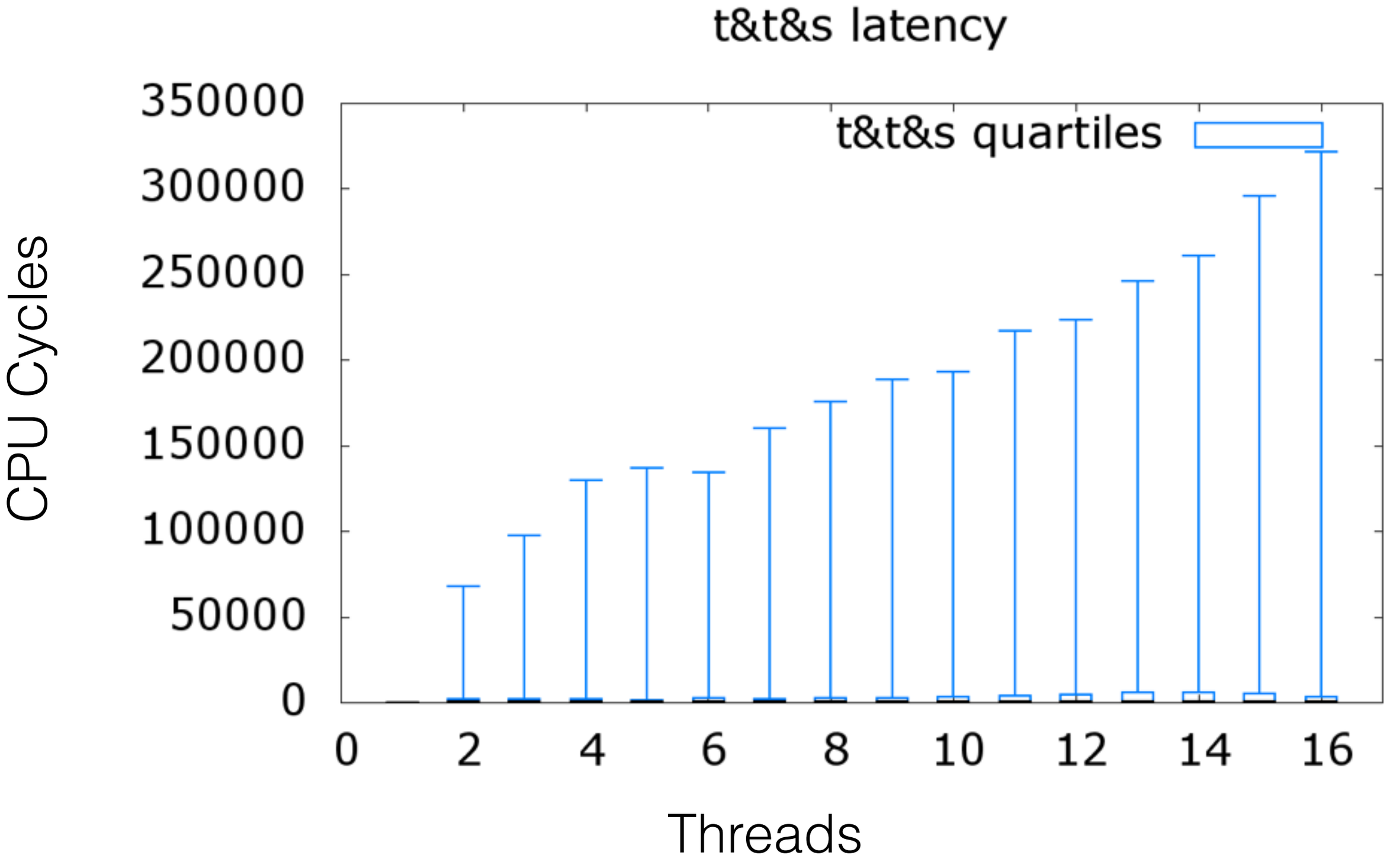


Allocator Throughput

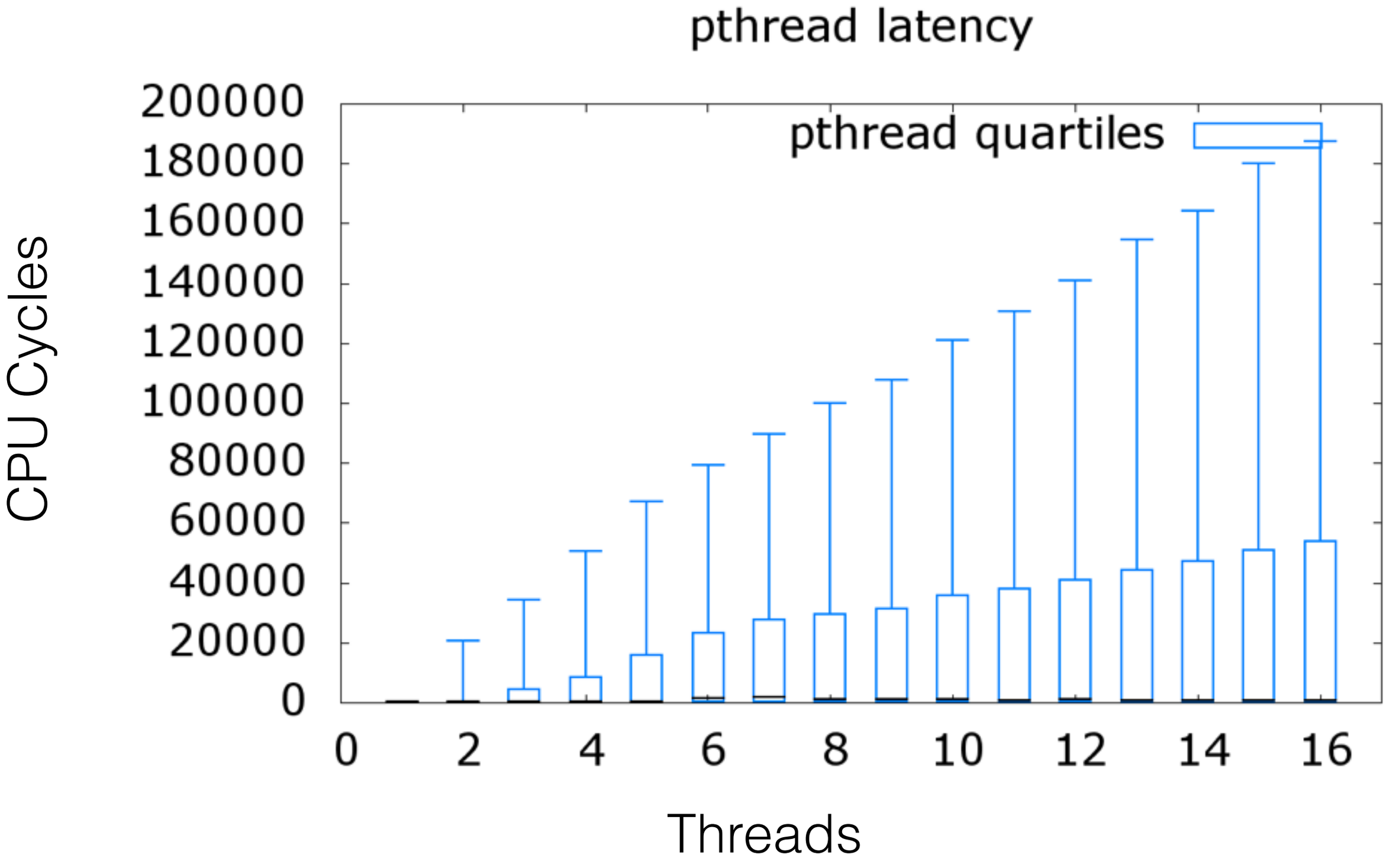


Allocator latency

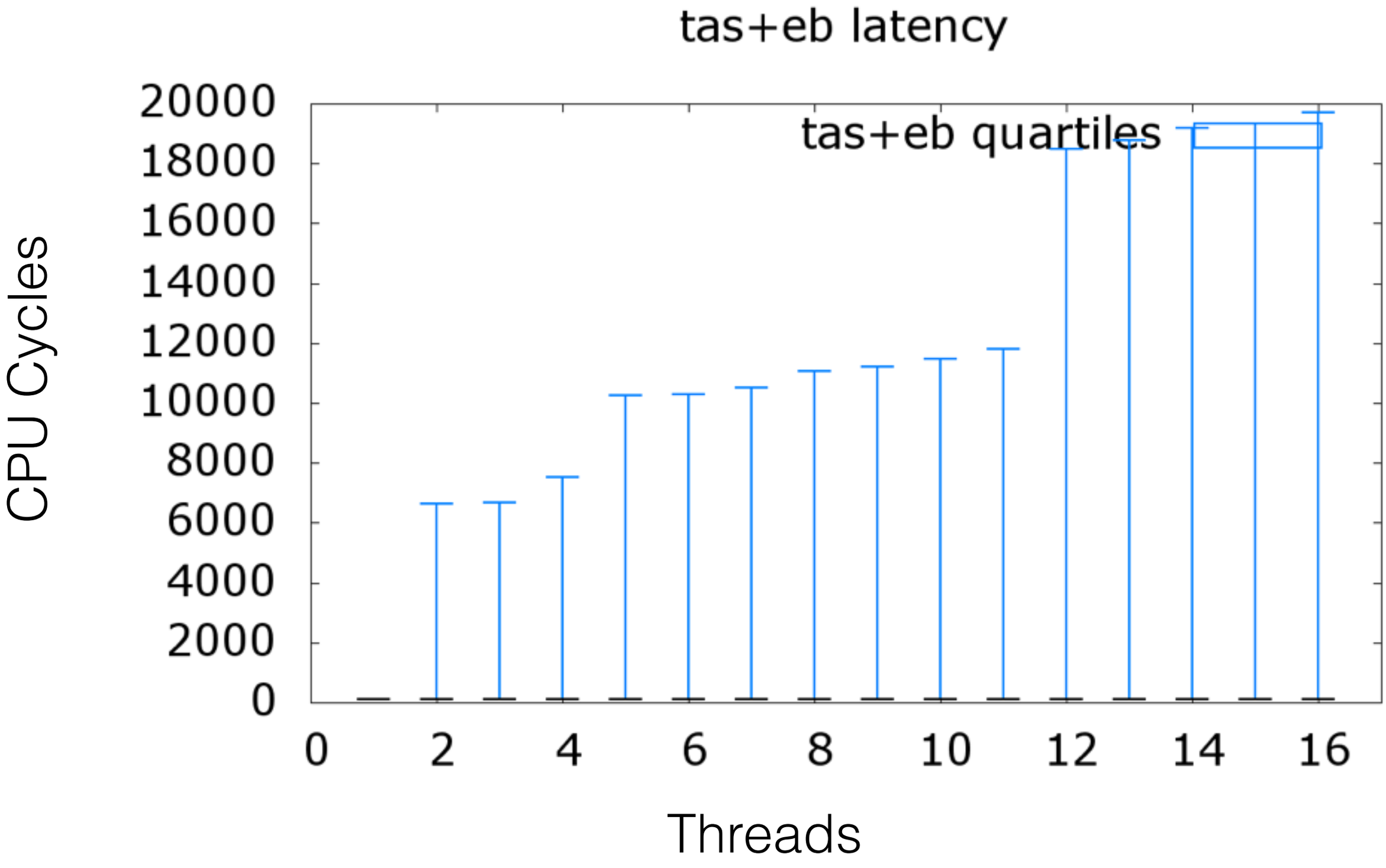
Allocator latency



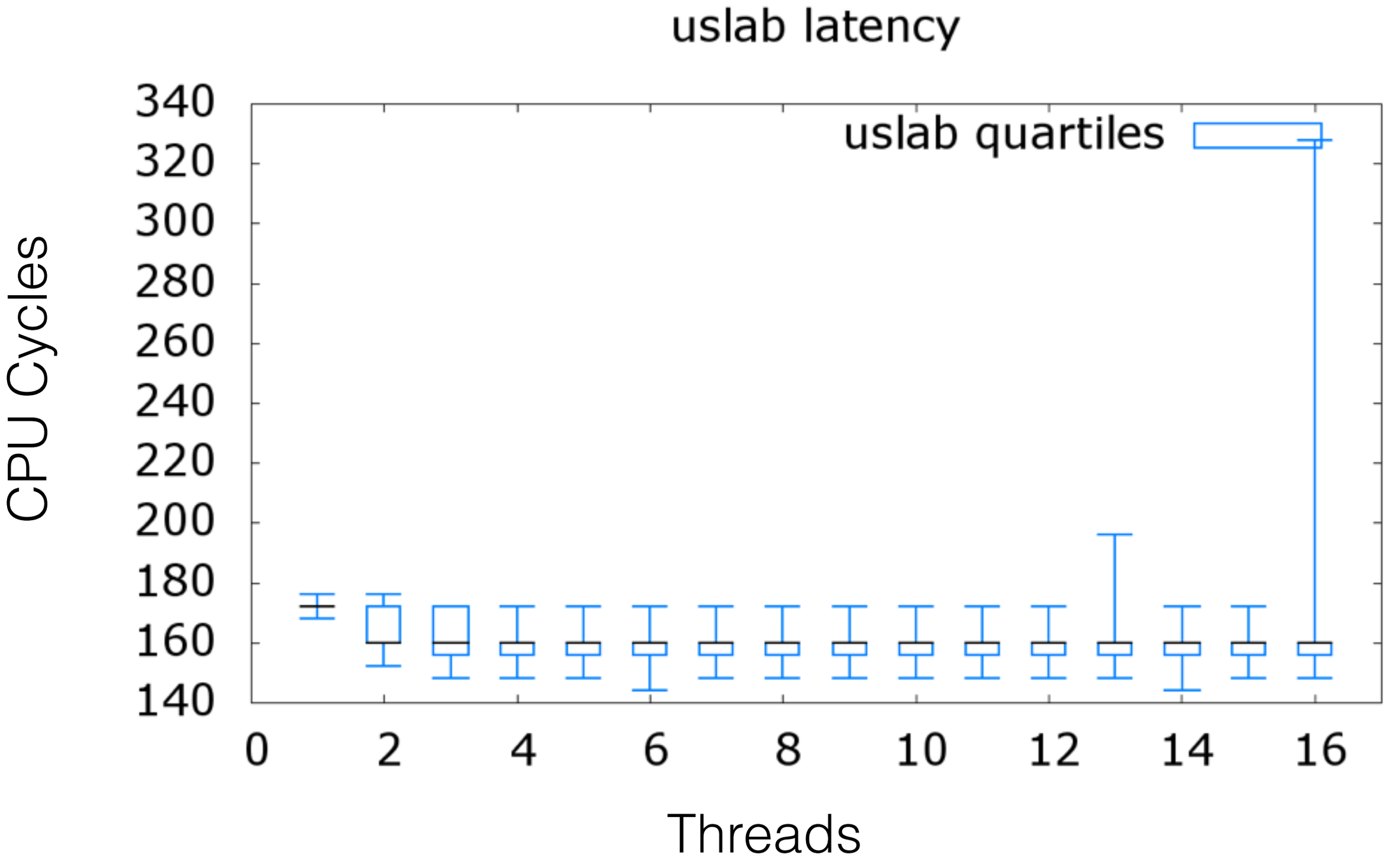
Allocator latency

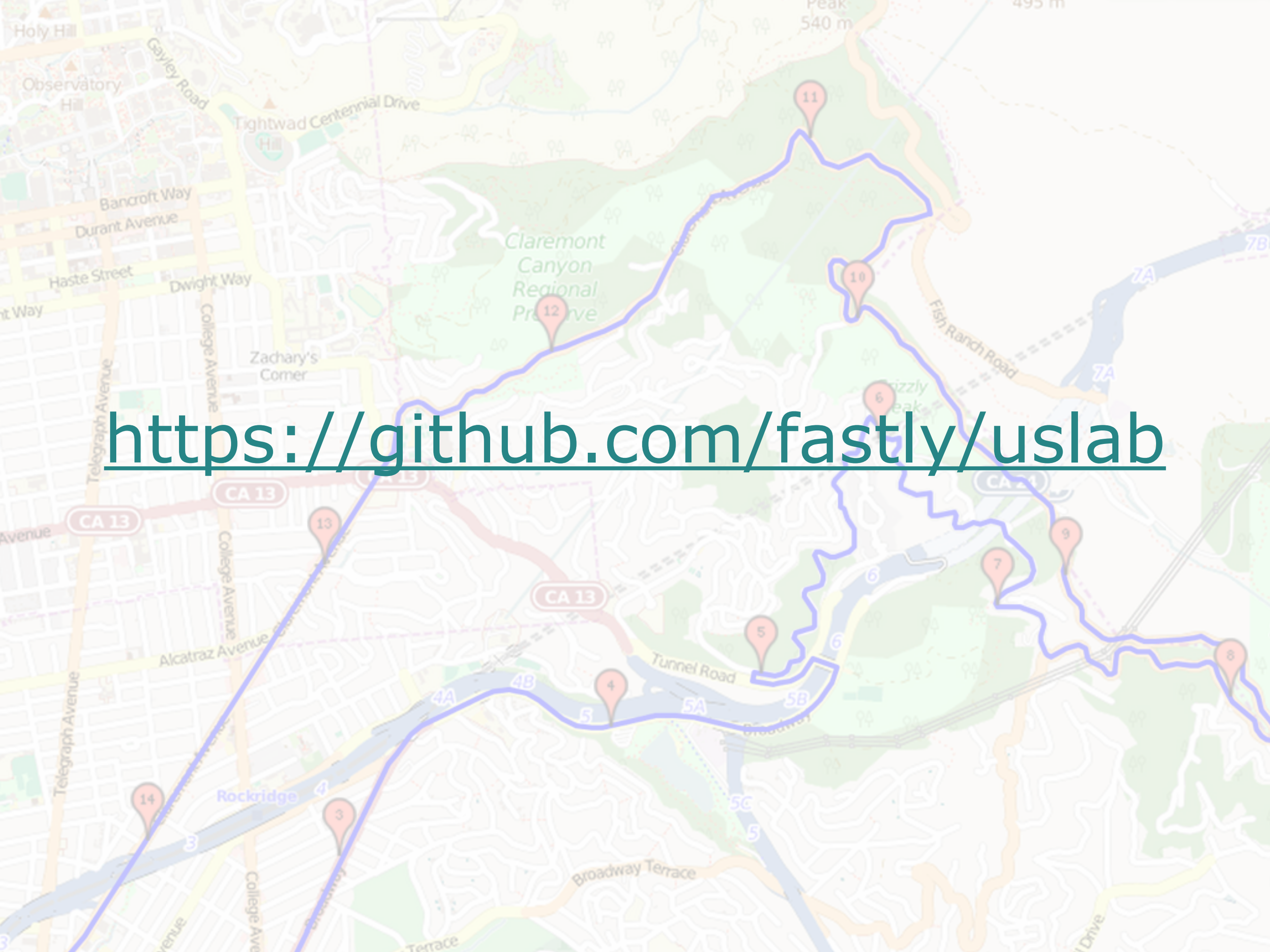


Allocator latency



Allocator latency




https://github.com/fastly/uslab

The lyf so short, the CAS so longe to lerne

- Cache coherency and NUMA architecture
- Transactional memory


#thoughtleadership
Holy Hill
Observatory Hill
Gayley Road
Tightwad Centennial Drive
Hill
Bancroft Way
Durant Avenue
Haste Street
Dwight Way
College Avenue
Zachary's Corner
Claremont Canyon Regional Preserve
Fish Ranch Road
Grizzly Peak
Alcatraz Avenue
Telegraph Avenue
Tunnel Road
Broadway
Broadway Terrace
Rockridge
CA 13
11
10
12
6
13
5
9
7
8
4
14
3
4A
4B
5
5A
5B
5C
7A
7B

When is a race

a safe race?

Holy Hill
Observatory Hill
Gayley Road
Tightwad Centennial Drive
Bancroft Way
Durant Avenue
Haste Street
Dwight Way
College Avenue
Zachary's Corner
Claremont Canyon Regional Preserve
Fish Ranch Road
Grizzly Peak
CA 13
Alcatraz Avenue
Telegraph Avenue
Tunnel Road
Broadway Terrace
Rockridge
Peak 540 m
495 m
7A
7B
11
10
12
6
13
5
9
7
8
4
14
3
4A
4B
5A
5B
5C

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

"lock-free programming is hard; let's go ride bikes"?

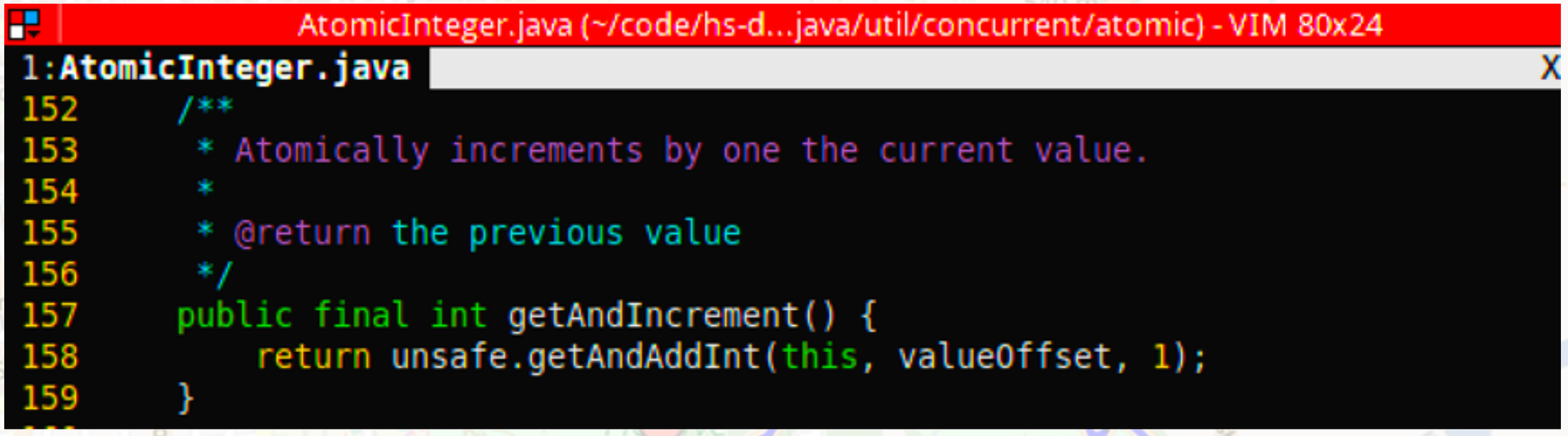
- high-level performance necessitates an understanding of low level performance

"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

"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)


AtomicInteger.java (~/code/hs-d...java/util/concurrent/atomic) - VIM 80x24
1:AtomicInteger.java
X
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 }

AtomicInteger.java (~/code/hs-d...java/util/concurrent/atomic) - VIM 80x24

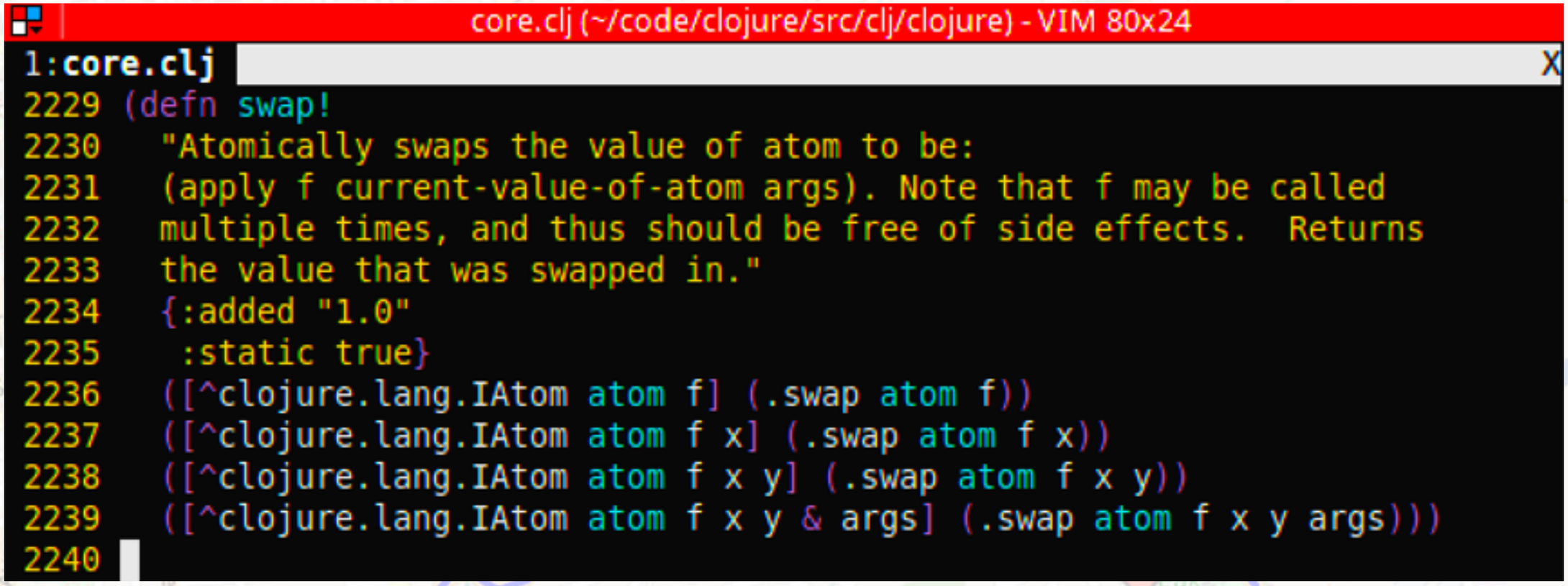
1:AtomicInteger.java

```
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     }
```


core.clj (~/code/clojure/src/clj/clojure) - VIM 80x24
1:core.clj
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))
2237   ([^clojure.lang.IAtom atom f x] (.swap atom f x))
2238   ([^clojure.lang.IAtom atom f x y] (.swap atom f x y))
2239   ([^clojure.lang.IAtom atom f x y & args] (.swap atom f x y args)))
2240

core.clj (~/code/clojure/src/clj/clojure) - VIM 80x24

1:core.clj

```
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

1:Atom.java

```
 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 }
```

core.clj (~/code/clojure/src/clj/clojure) - VIM 80x24

1:core.clj

```
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

1:Atom.java

```
 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

1:AtomicReference.java

```
115     public final boolean compareAndSet(V expect, V update) {
116         return unsafe.compareAndSwapObject(this, valueOffset, expect,
    update);
117     }
```

www.thestrangeloop.com/2015/...

USING RACE CONDITIONS IN CORRECT CONCURRENT SOFTWARE

Devon O'Dell

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