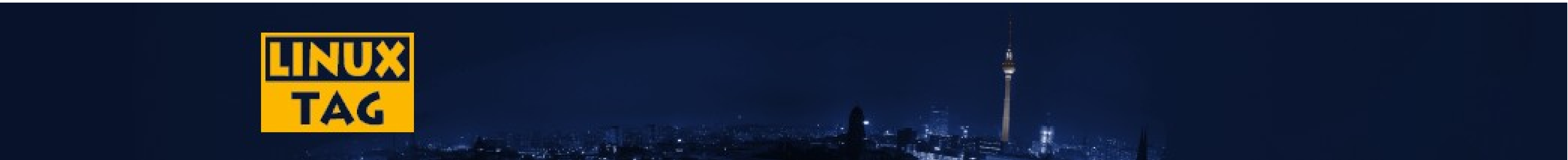
Fast, less-complicated, lock-free Data Structures

Ulrich Drepper
ulrich.drepper@gs.com



Accelerate Code

- Not (much) through new hardware
- Split into independent pieces
  - Splitting comes at a cost
  - Marshaling between stages
  - Increased latency for pipeline
- Realistically:

Parallelization needed!

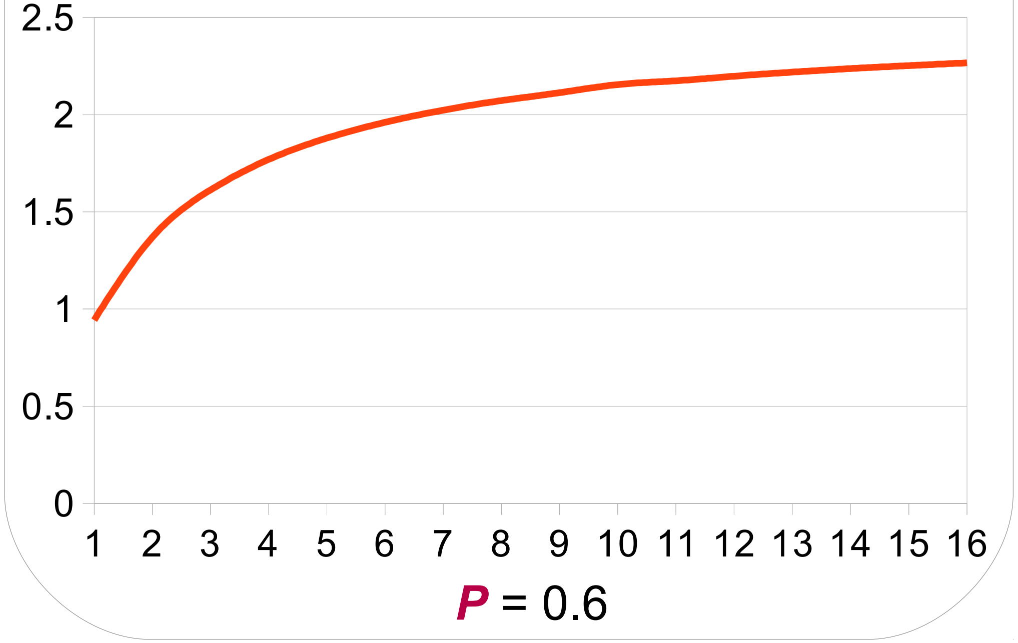
Parallelization

- Alternatives
  - Multi-process

    or
  - Multi-thread
- Error prone
- High level of parallelization needed
- Keep cost of parallelization (O_p) low

Extended "Amdahl's Law"

$$S = \frac{1}{(1-P) + \frac{P}{N}(1+O_P)}$$



Parallelization

- Collaboration through shared memory
- Synchronized access
  - Synchronized access to data structures
  - Atomic data structures

    (mostly based on Compare-And-Swap)

```
bool __sync_bool_compare_and_swap(TYPE *ptr, TYPE oldval, TYPE newval) {
   if (*ptr != oldval) return false;
   *ptr = newval;
   return true;
}
```

Lock-Free Data Structures

		LIFO	FIFO	Hash	Single Linked	Double Linked
No Priority	1:1	CAS	CAS			
	1:N	CAS				
	N:1	CAS	CAS			
	M:N	CAS				
Priority	1:1	CAS	CAS			
	1:N					
	N:1	CAS	CAS			
	M:N					

x86 Special

		LIFO	FIFO	Hash	Single Linked	Double Linked
No Priority	1:1	CAS	CAS			
	1:N	CAS	DWCAS			
	N:1	CAS	CAS			
	M:N	CAS	DWCAS			
Priority	1:1	CAS	CAS			
	1:N					
	N:1	CAS	CAS			
	M:N					

Double-wide CAS

Extended CAS

- Wider, more complicated CAS not the answer

DCAS is not a Silver Bullet for Nonblocking Algorithm Design

Doherty, Detlefs, Groves, Flood, Luchangco, Martin, Moir, Shavit, Steele, SPAA '04, 2004

Locking

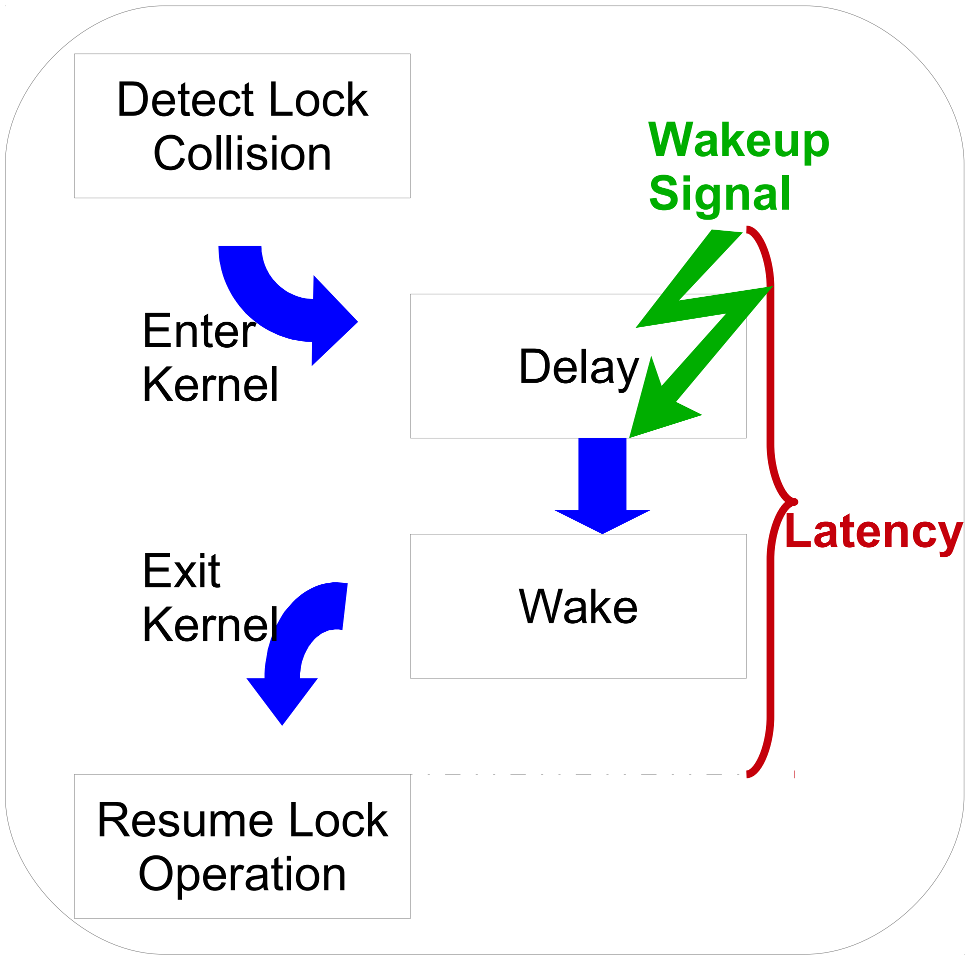
- Bane of Programming
- Interface design: explicit or implicit locking?
  - Often unnecessary overhead
- Composability problem
  - AB-BA locking problem

```
void move(dbllist<T> &target, dbllist<T>::it &prev,
          dbllist<T> &source, dbllist<T>::it &elem);
```

How to implement internal locking?

Locking and Latency

- Yes, there are spinlocks
- Fairer/more power efficient locking requires sleep
- Sleep requires wakeup



Way Forward

Two complementary approaches

- Improve implementation of locking to
  - Reduce contention
  - Reduce cost of the operation

- Replace concept of locking

Way Forward

Two complementary approaches

- Improve implementation of locking to
  - Reduce contention
  - Reduce cost of the operation

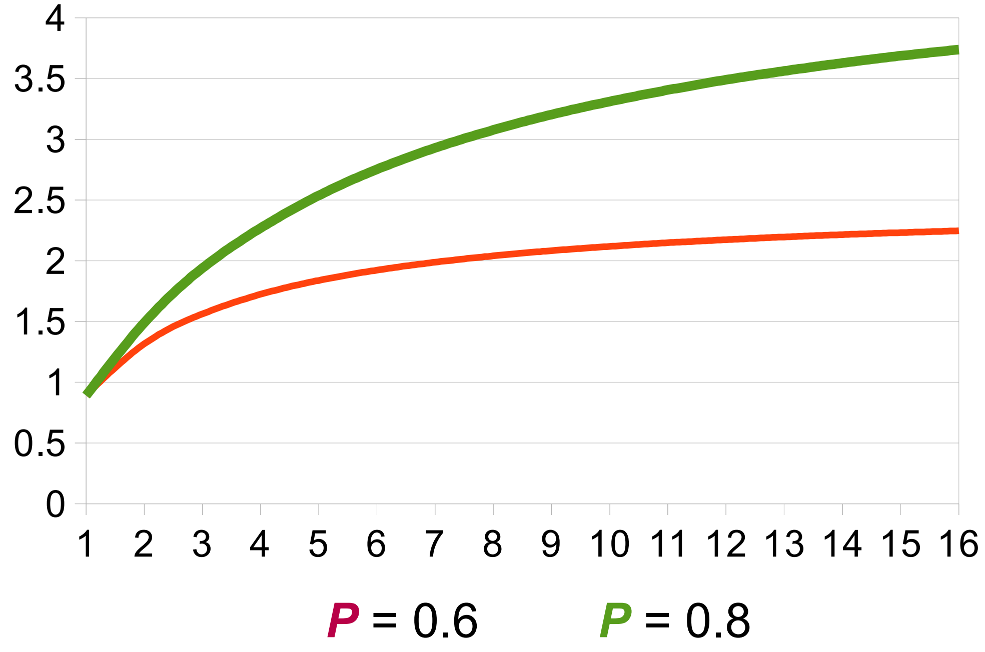
Hardware Lock Elision (HLE)

- Replace concept of locking

Transactional Memory (TM)

Increase Parallelism

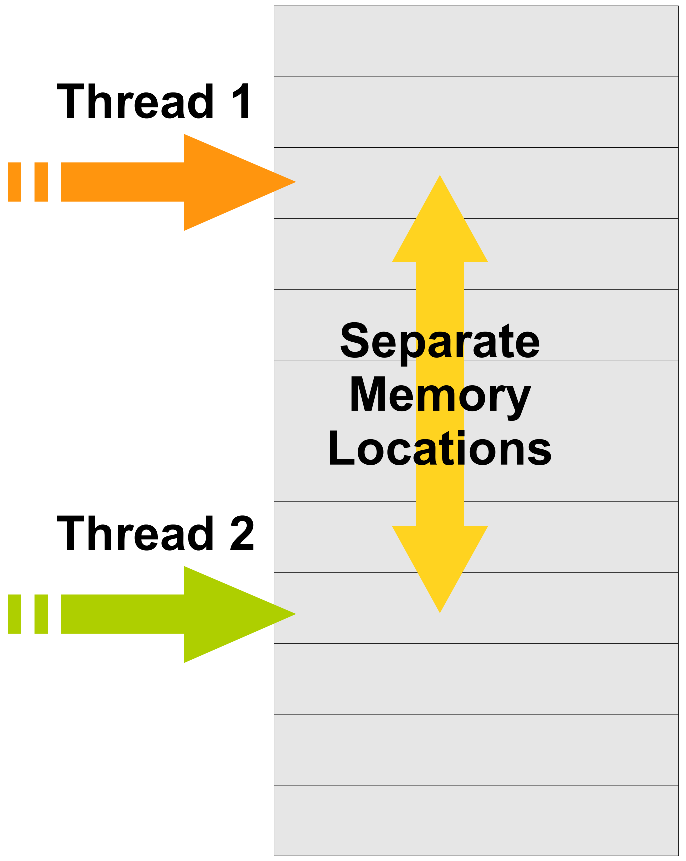
- Reduce lock contention
- Avoid “optimizations” like reader-writer locks
- Enable more code to be parallelized



Running Example

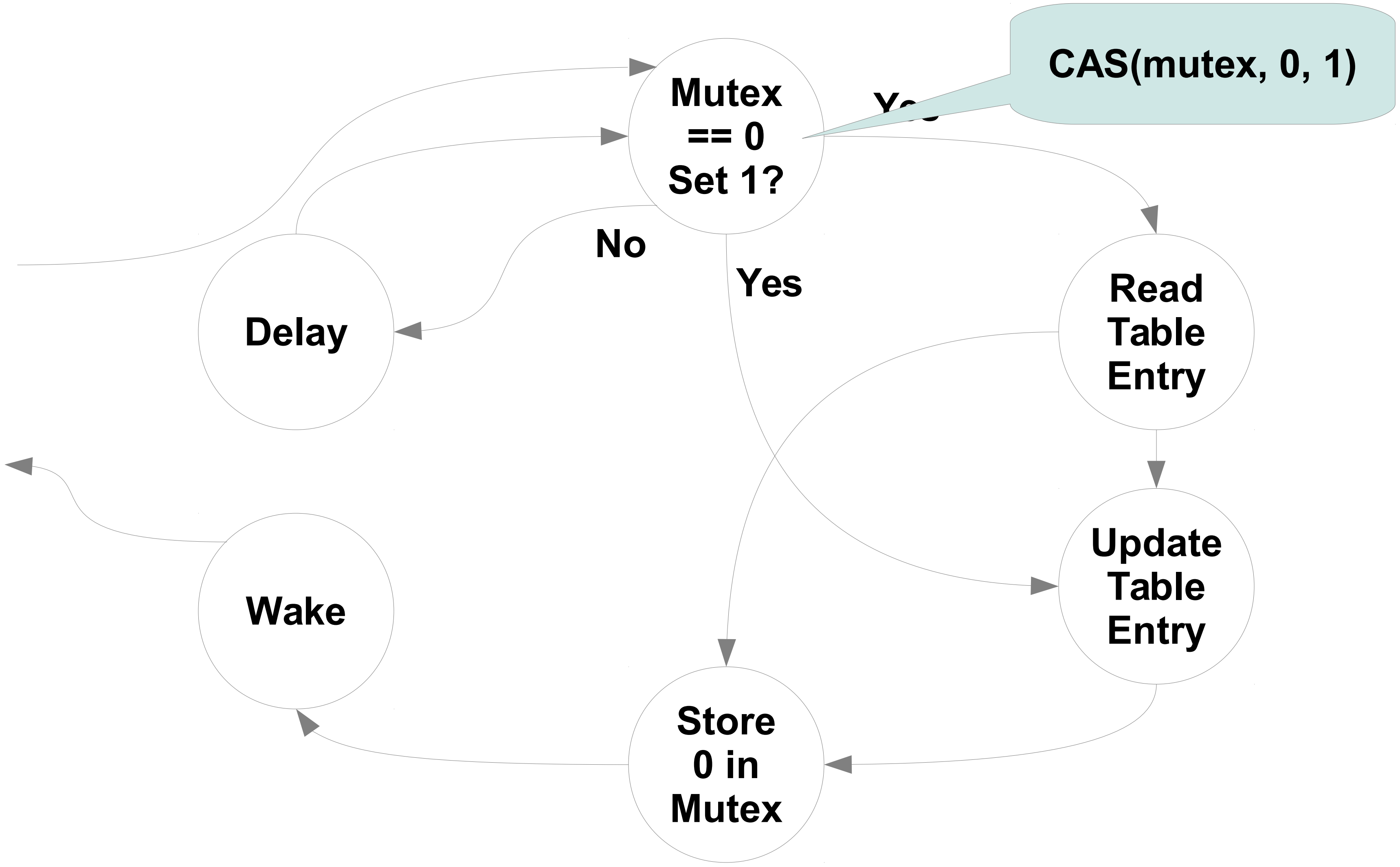
Locking Hash Tables

- Designed for concurrent accesses
- In practice mostly read accesses
- Even write accesses likely will not conflict
- Locking is overkill

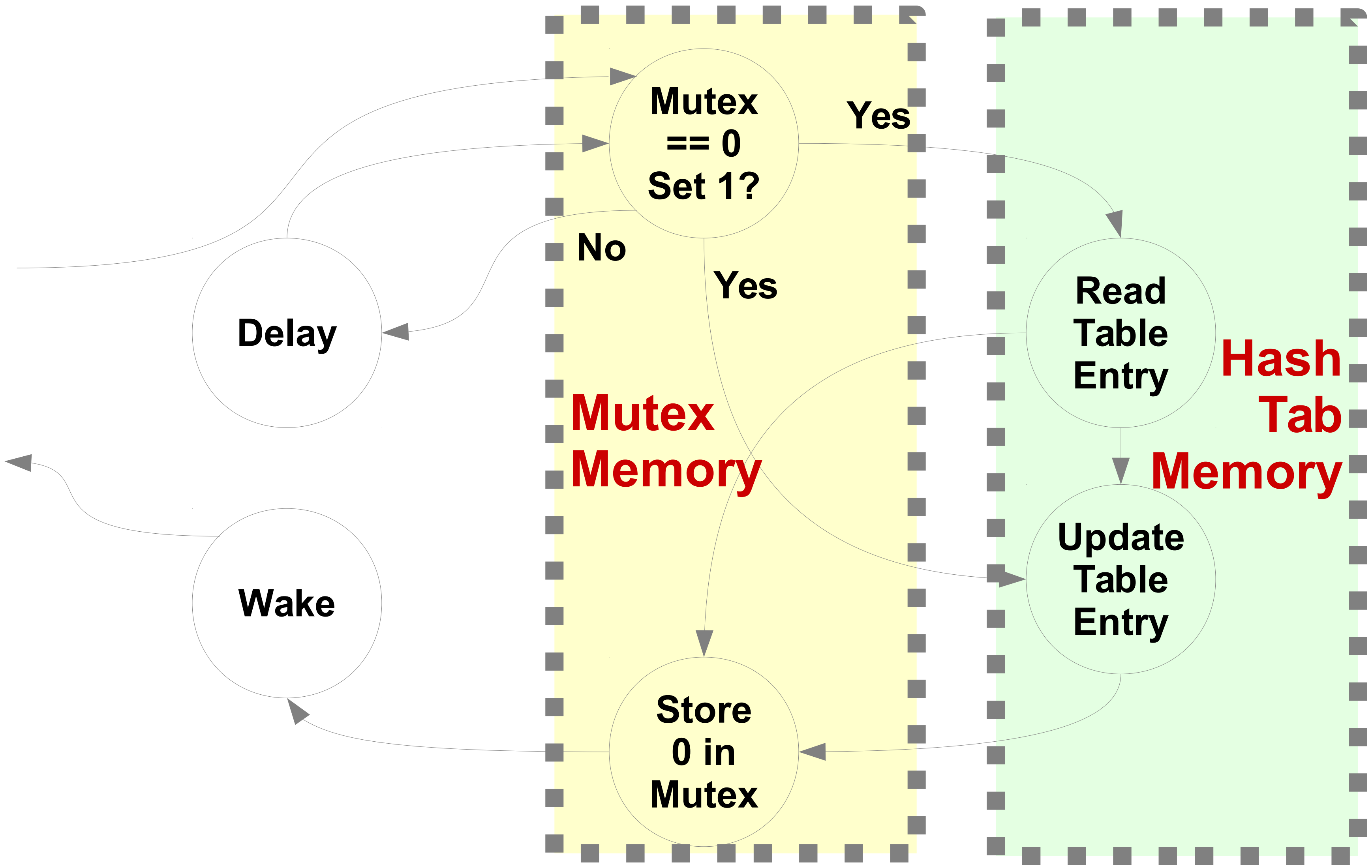


Hash Table
With locking

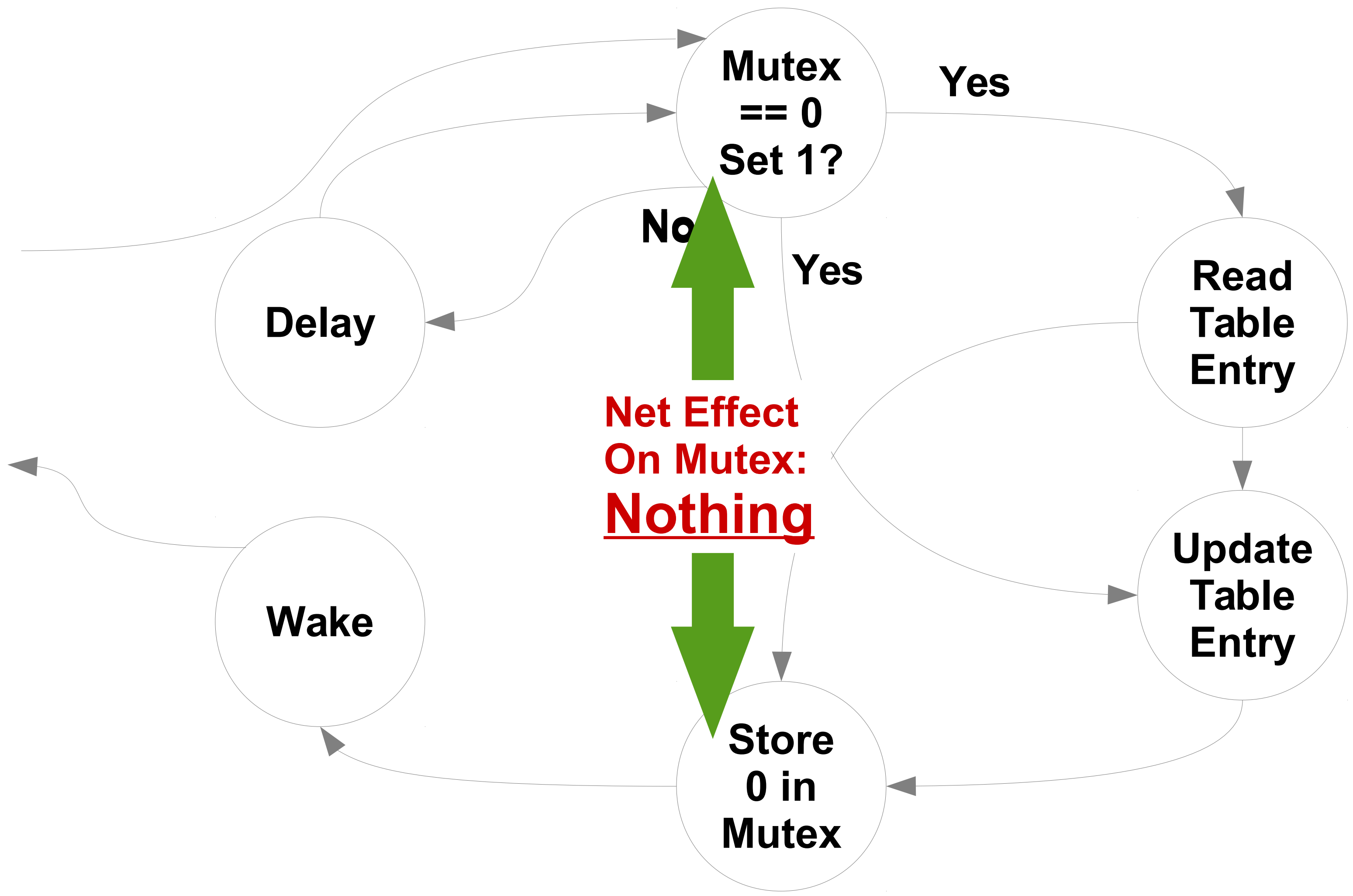
Mutually Exclusive Access



Mutually Exclusive Access

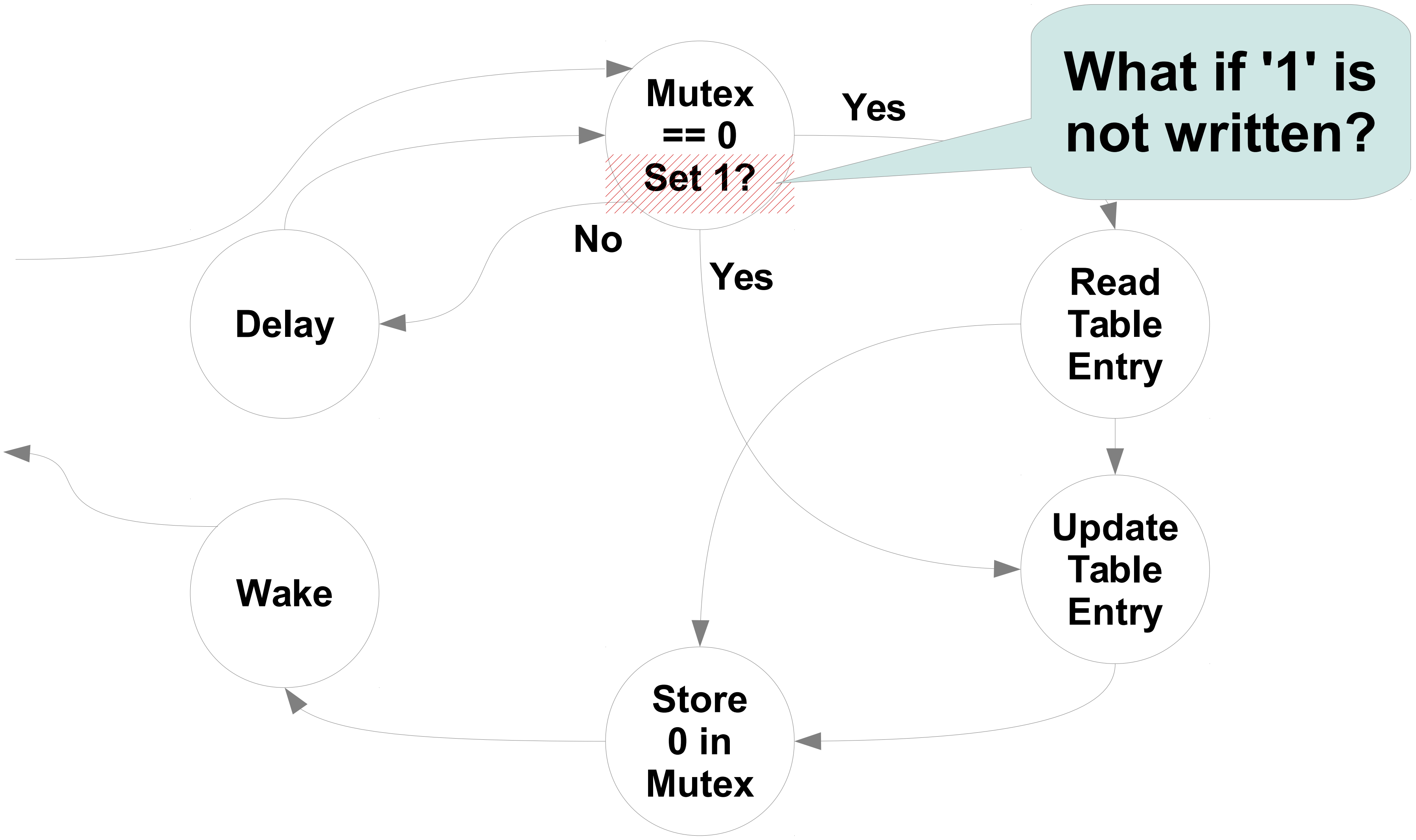


Mutually Exclusive Access

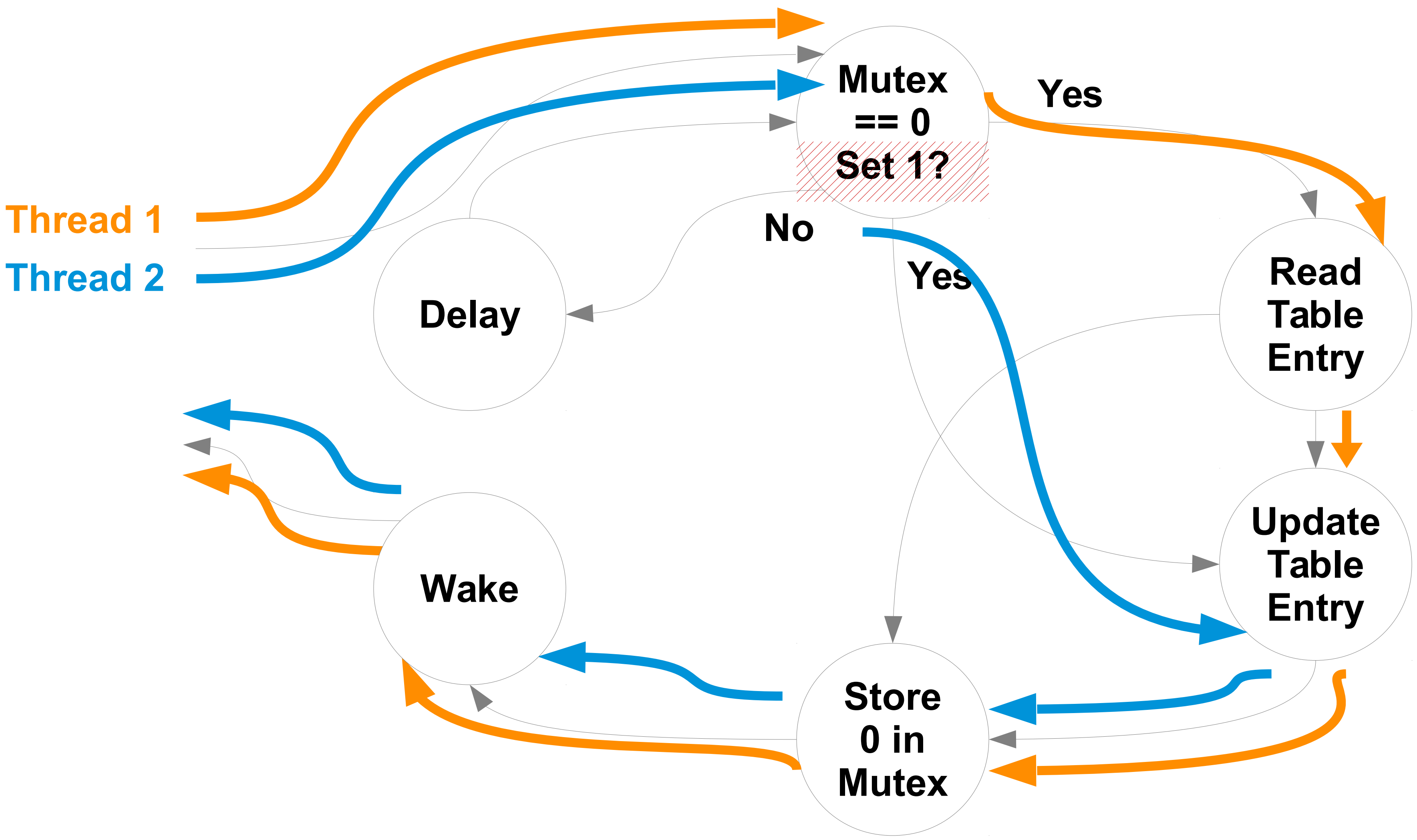


Hardware Lock Elision

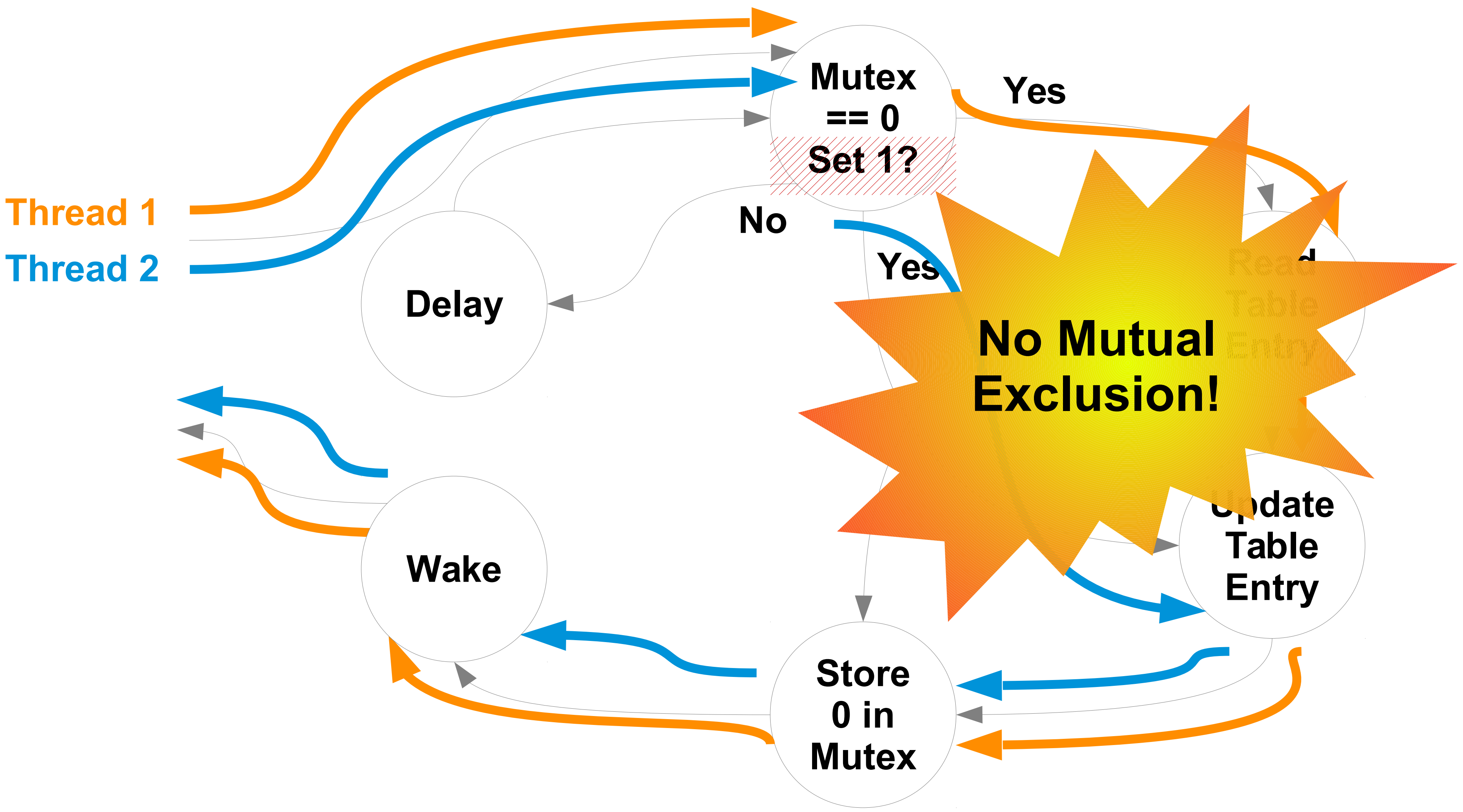
With Lock Elision



With Lock Elision

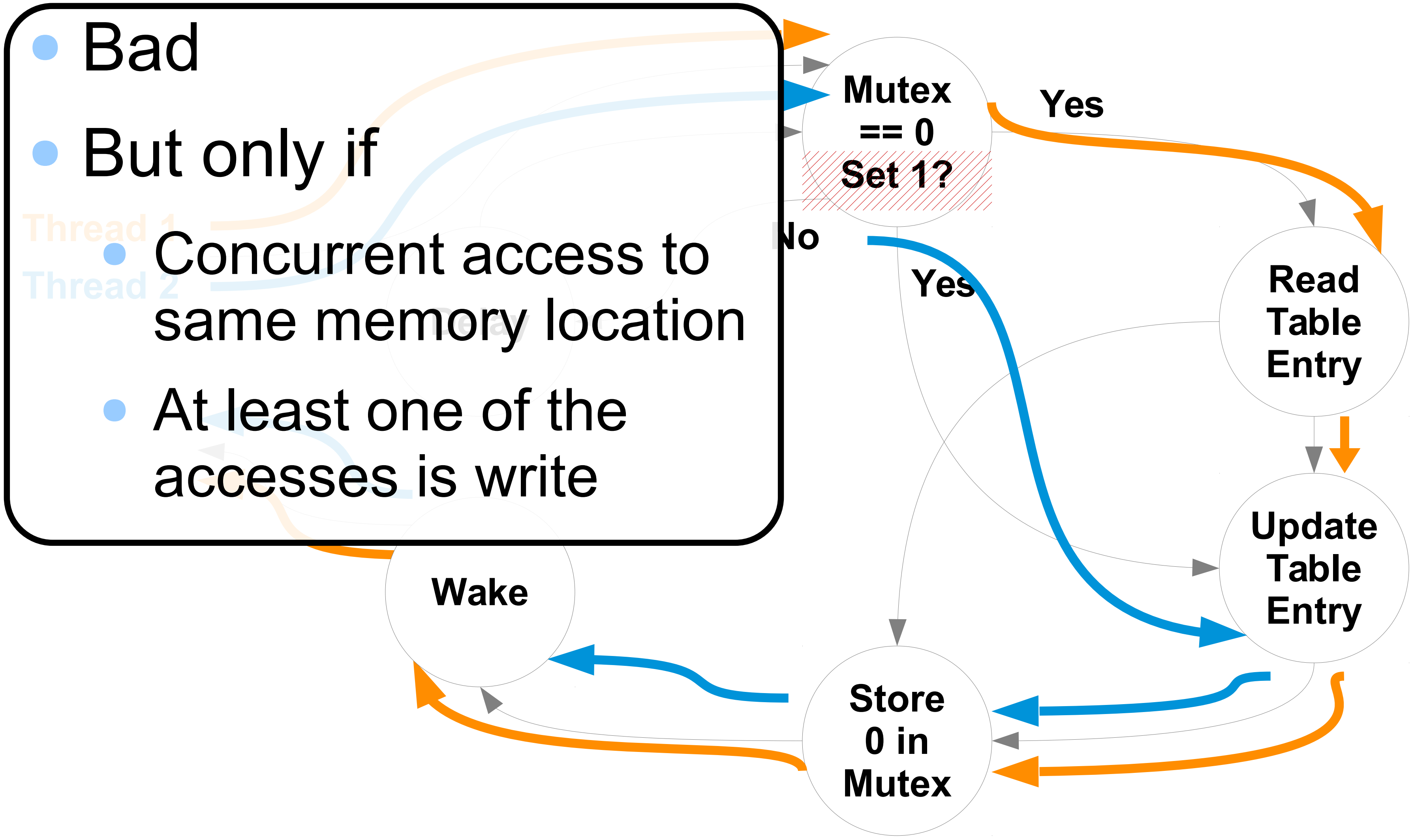


With Lock Elision

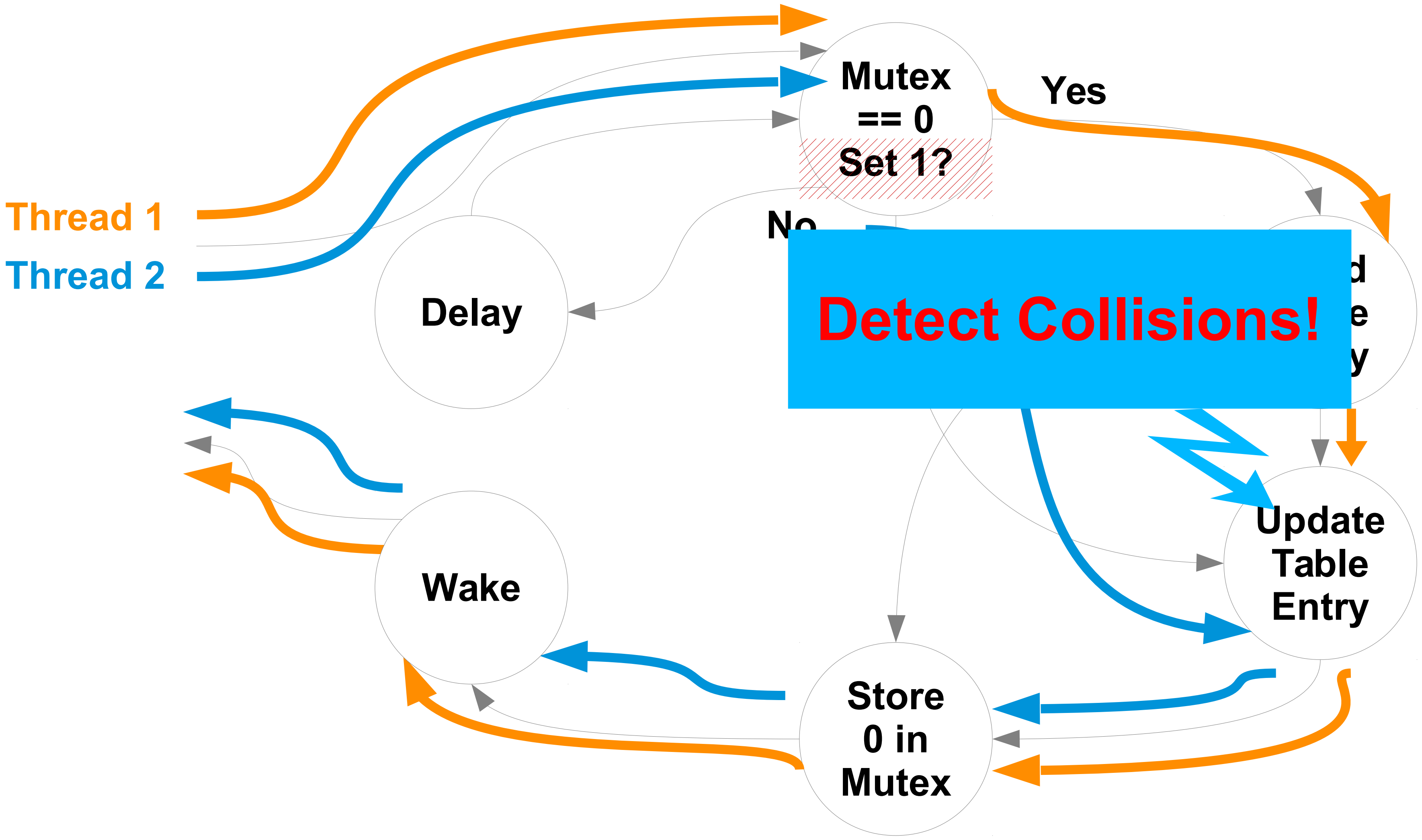


No Mutual Exclusion

- Bad
- But only if
  - Concurrent access to same memory location
  - At least one of the accesses is write



Alternative



Intel HLE

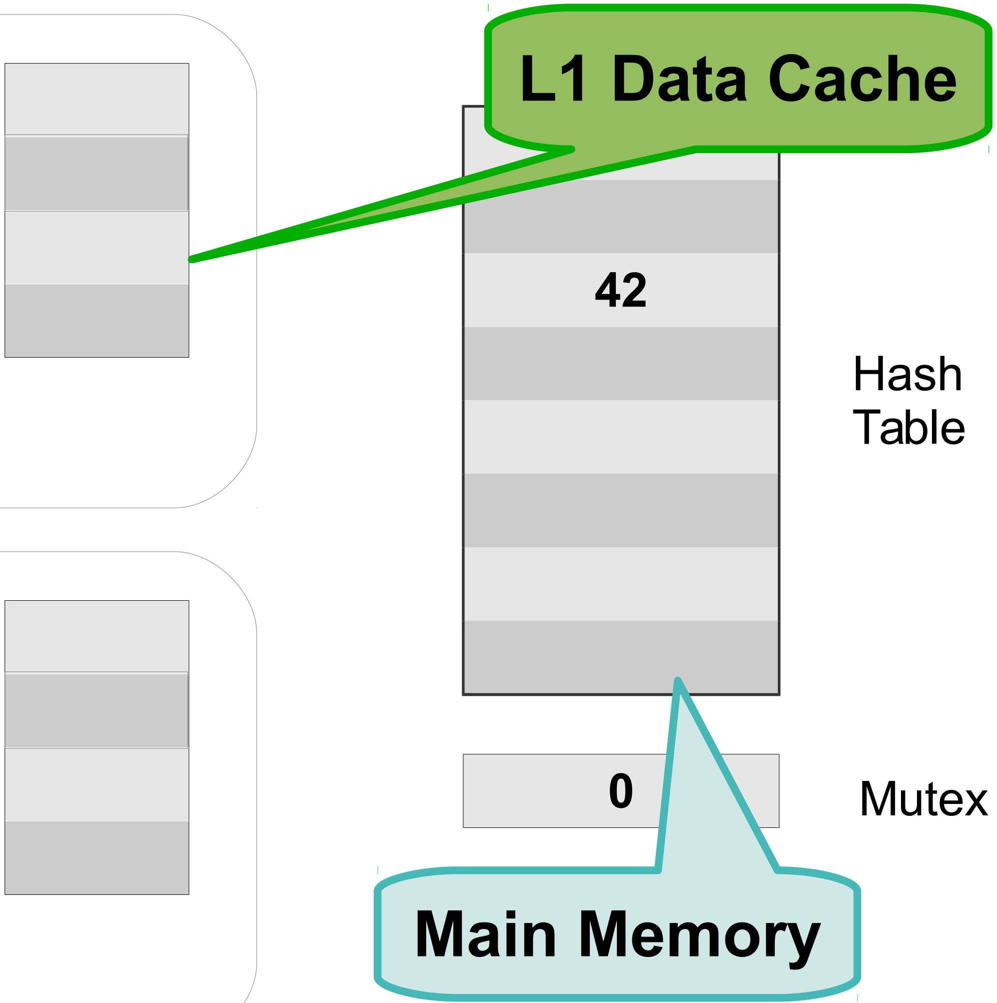
x86 code for Hash Table

Thread 1

```
lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
mov $0, mut
call wake
```

Thread 2

```
lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
mov $0, mut
call wake
```



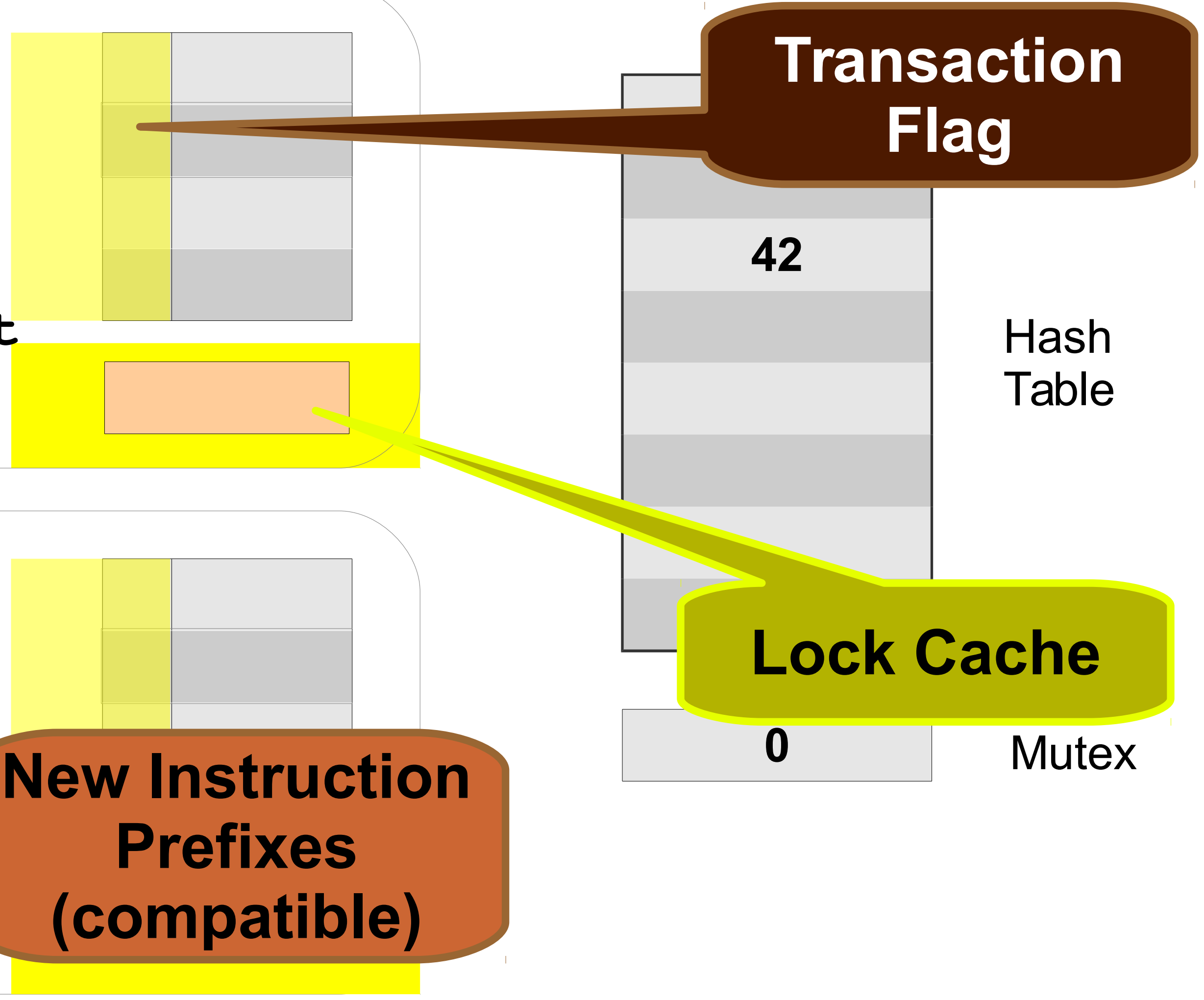
New in Intel HLE

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```



Successful Concurrent Use

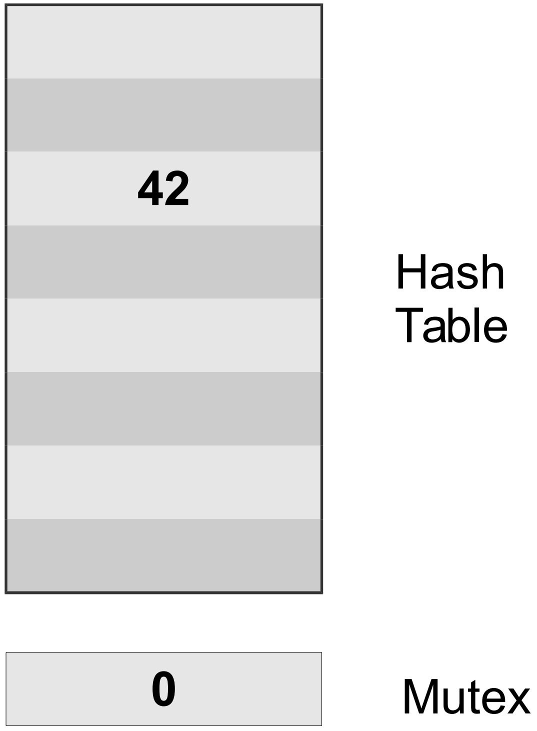
No Collision

Thread 1

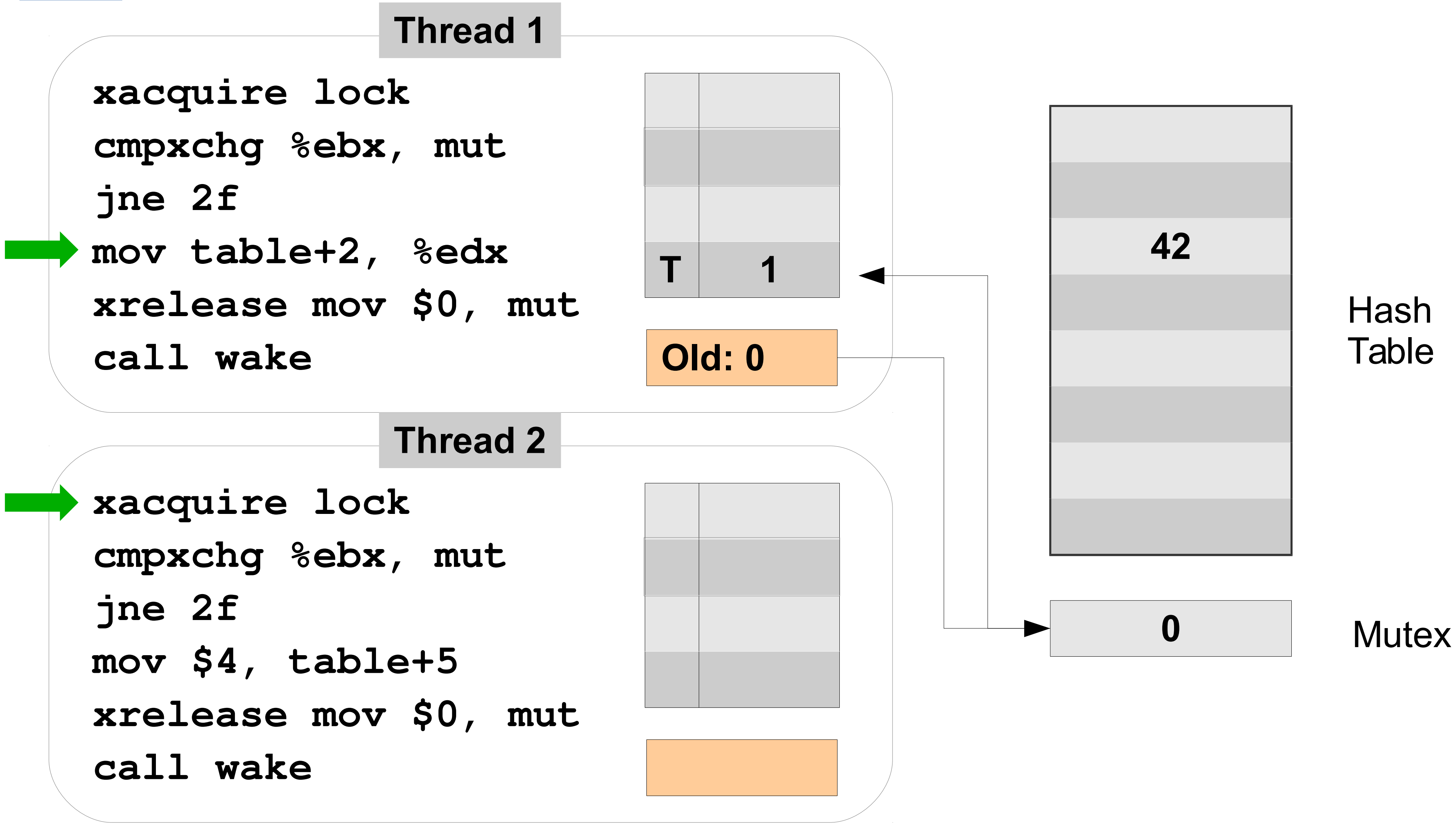
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```



No Collision



```
  xacquire lock
  cmpxchg %ebx, mut
  jne 2f
→ mov table+2, %edx
  xrelease mov $0, mut
  call wake
```

```
→ xacquire lock
  cmpxchg %ebx, mut
  jne 2f
  mov $4, table+5
  xrelease mov $0, mut
  call wake
```

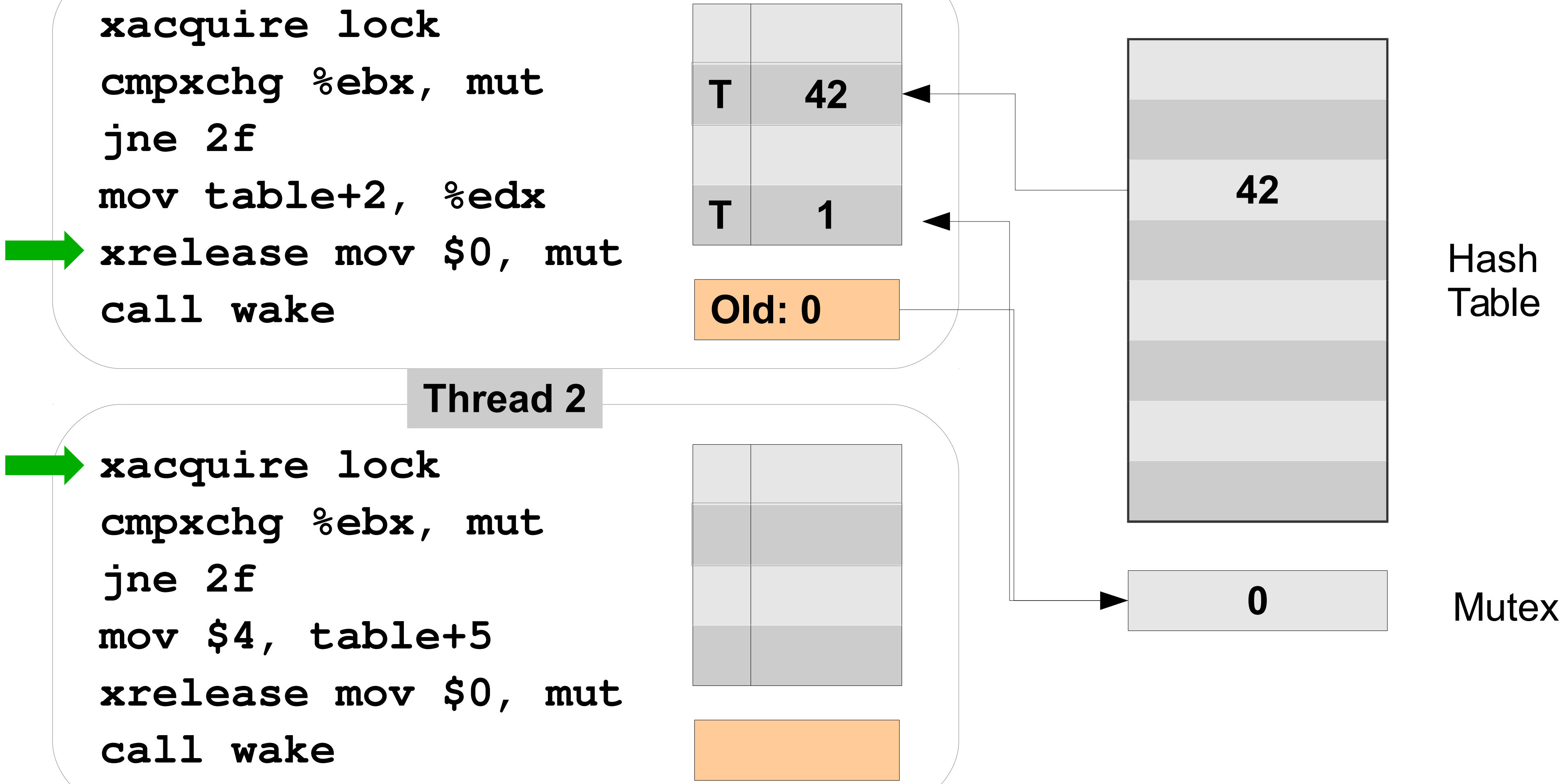
No Collision

Thread 1

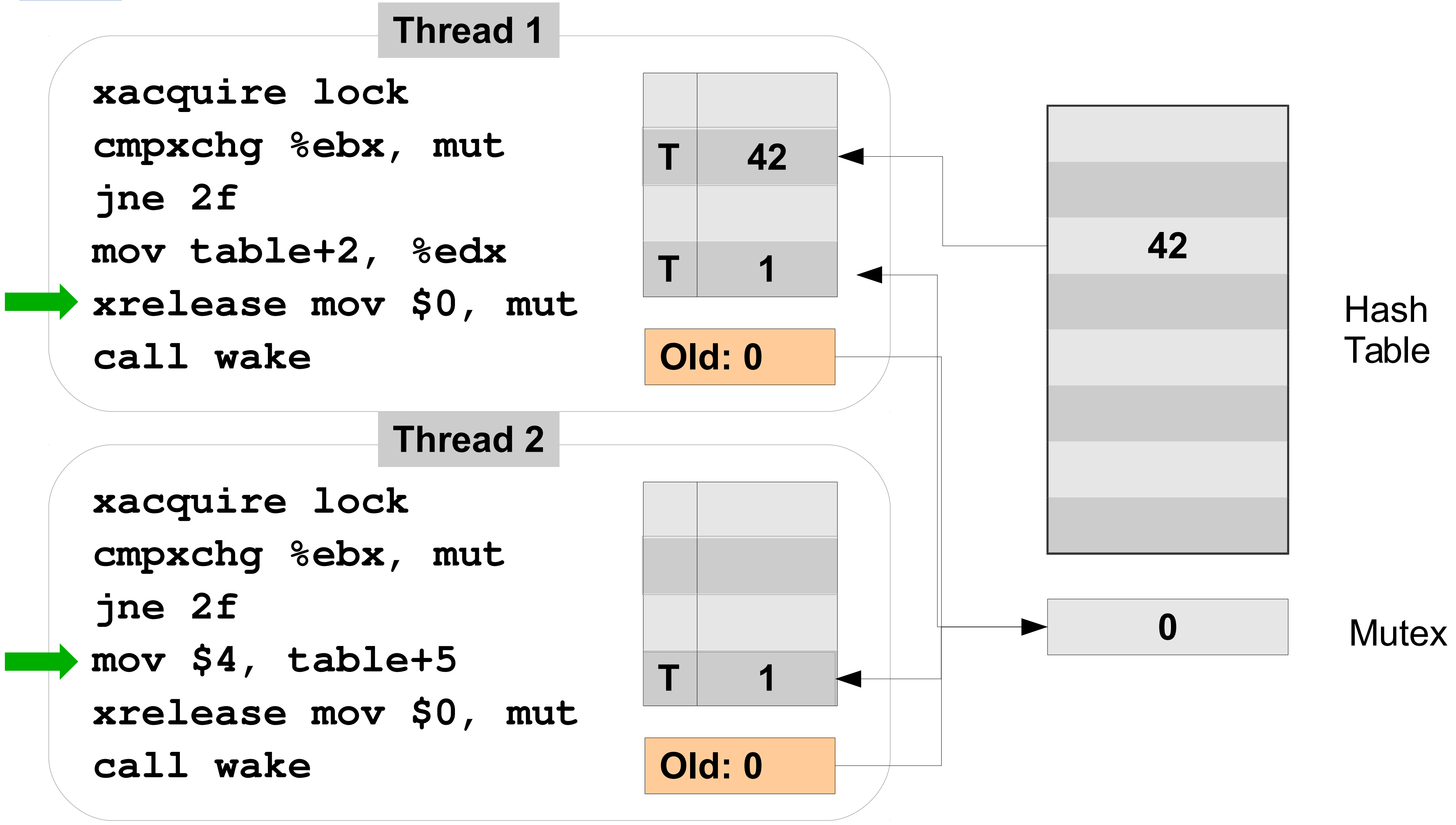
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```



No Collision



Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```

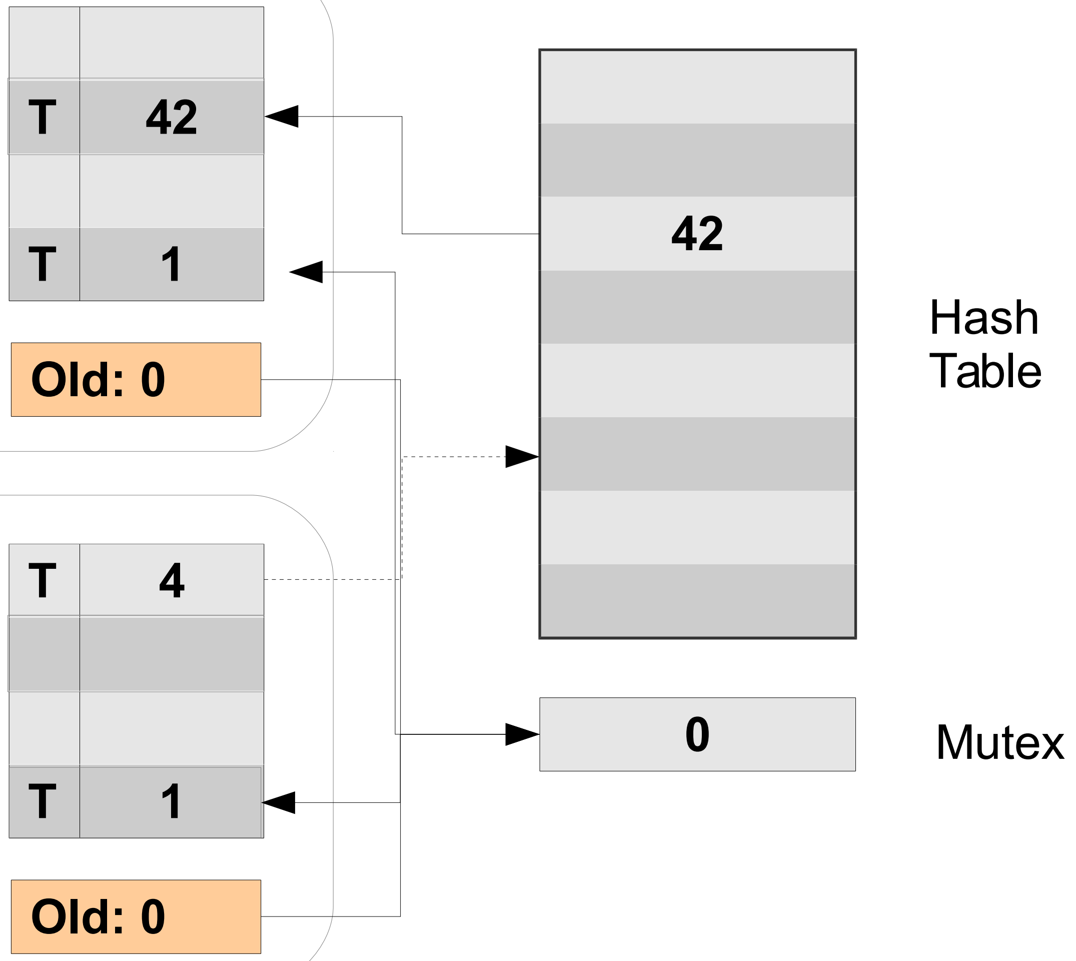
No Collision

Thread 1

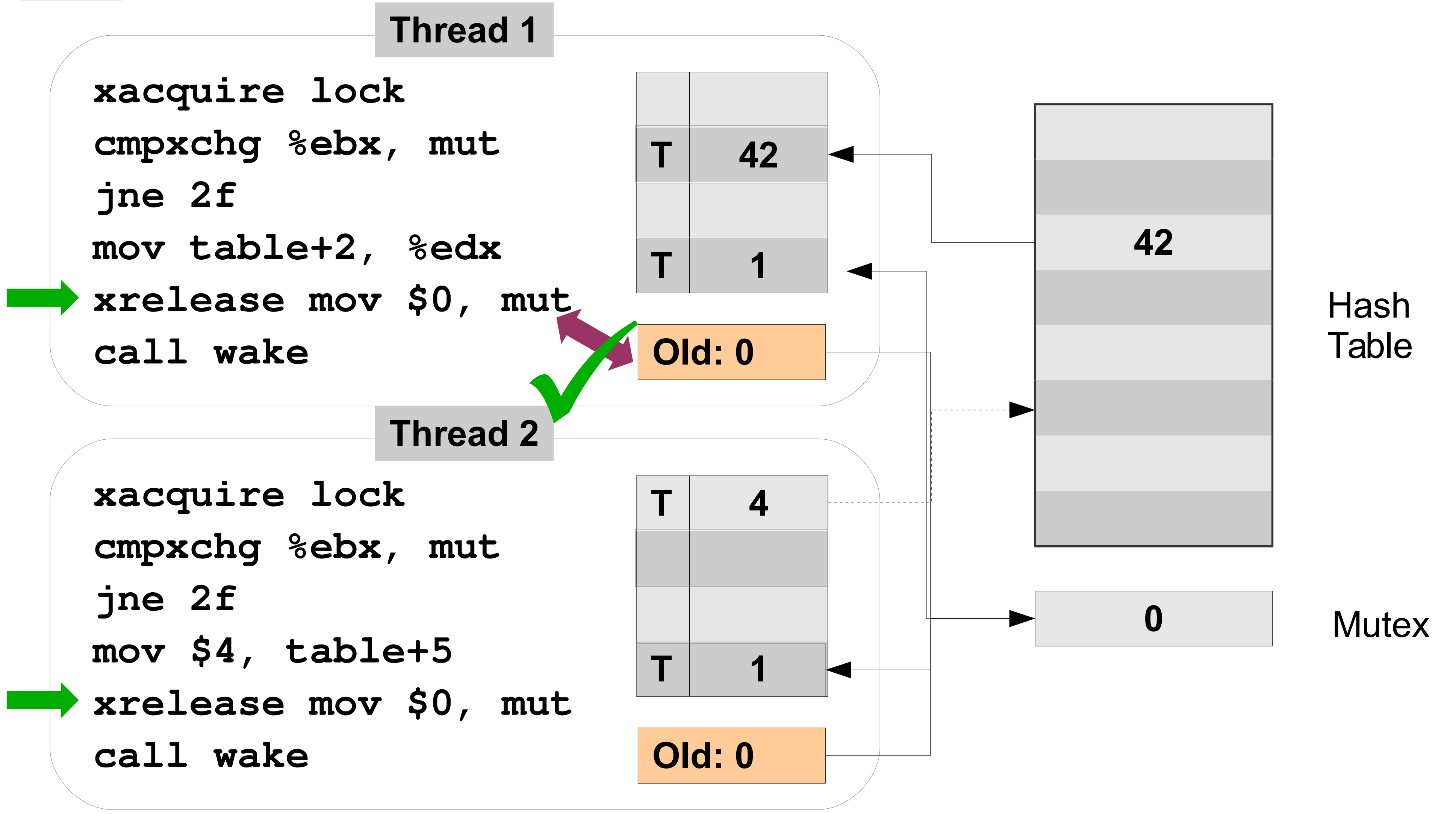
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```



No Collision



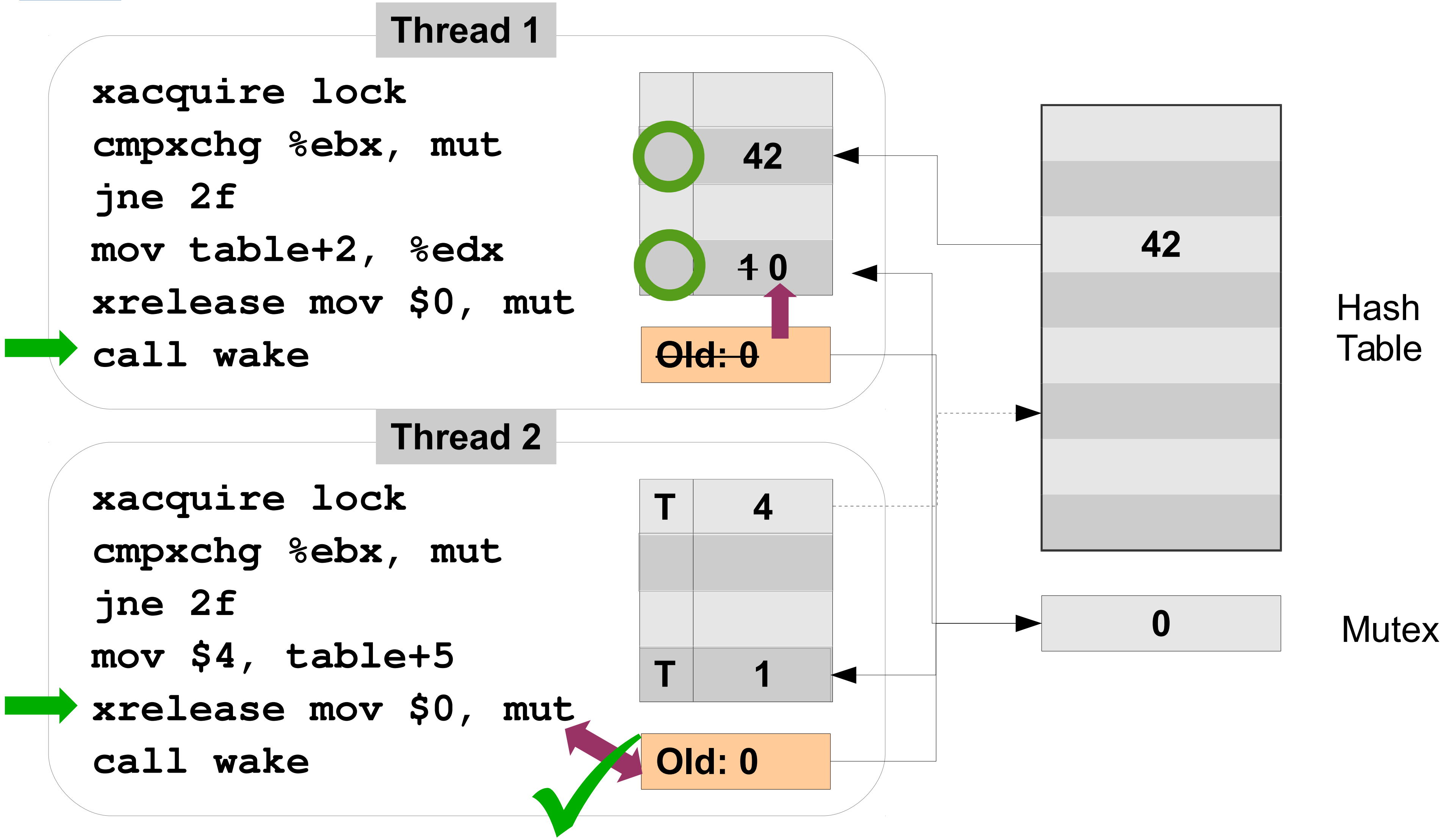
Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```

No Collision



Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```

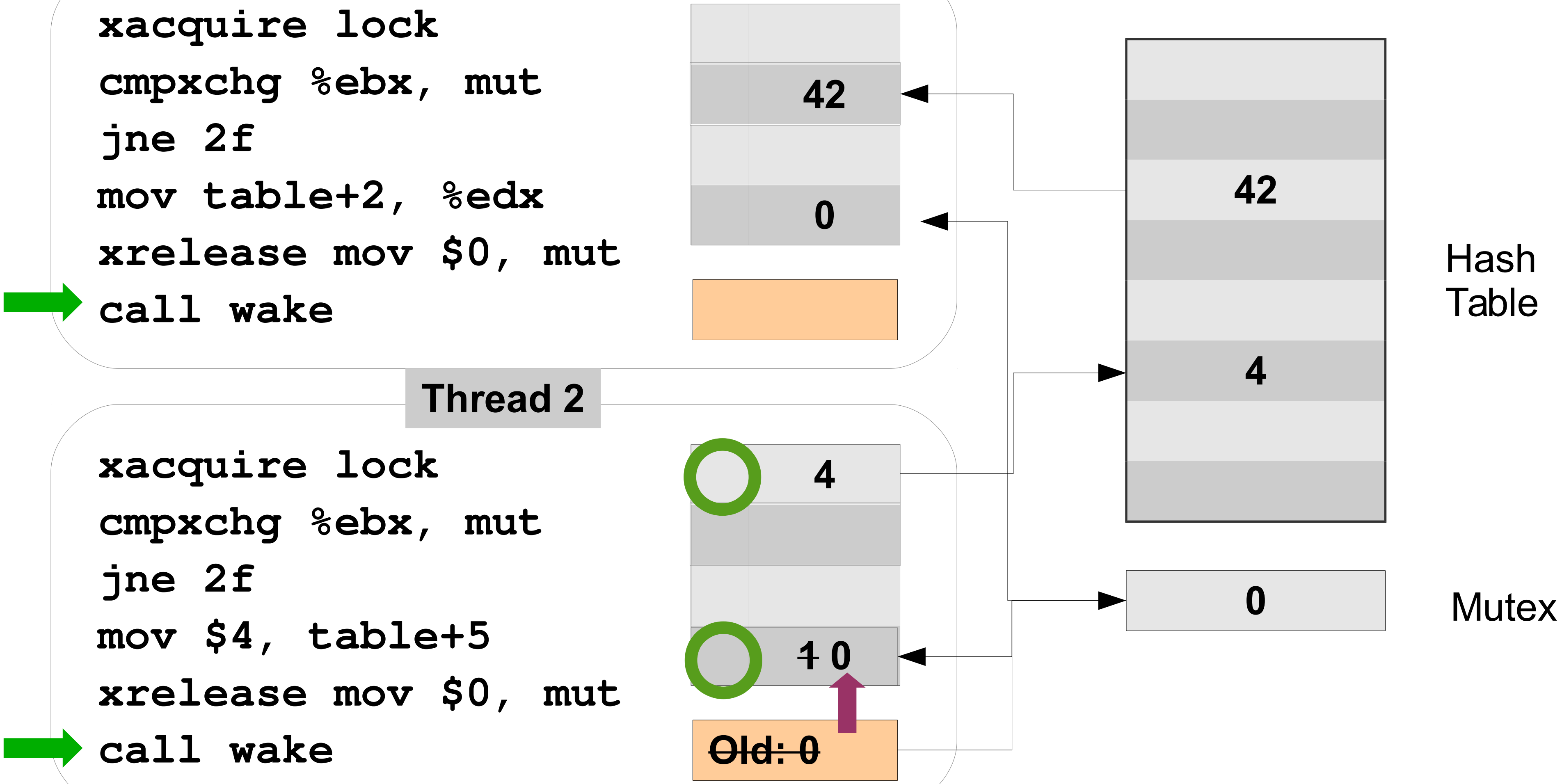
No Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+5
xrelease mov $0, mut
call wake
```



Unsuccessful Concurrent Use

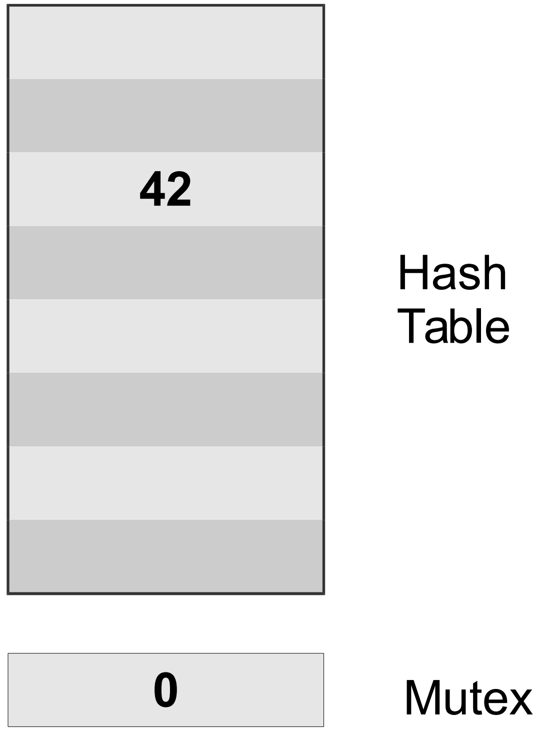
With Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



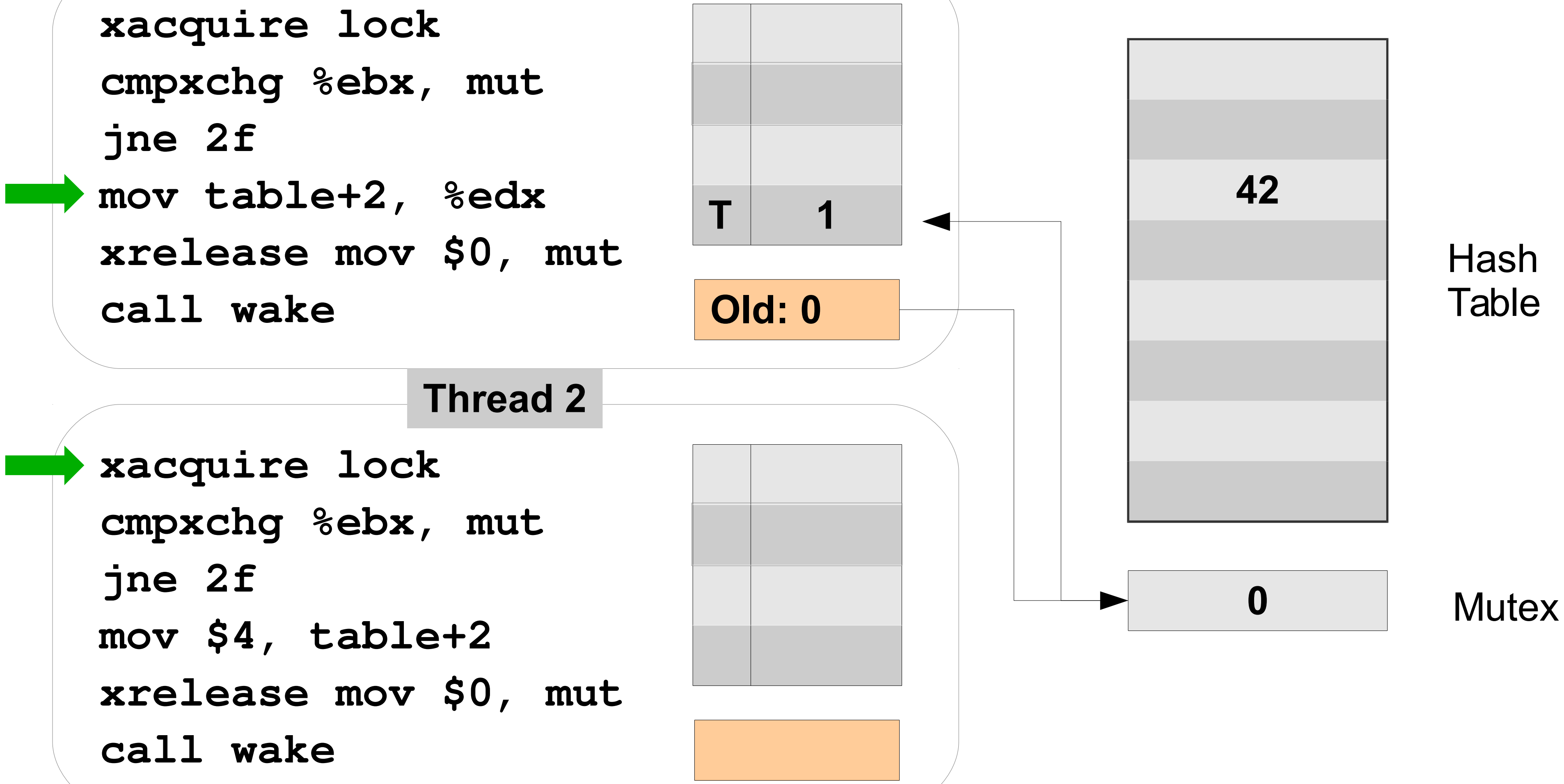
With Collision

Thread 1

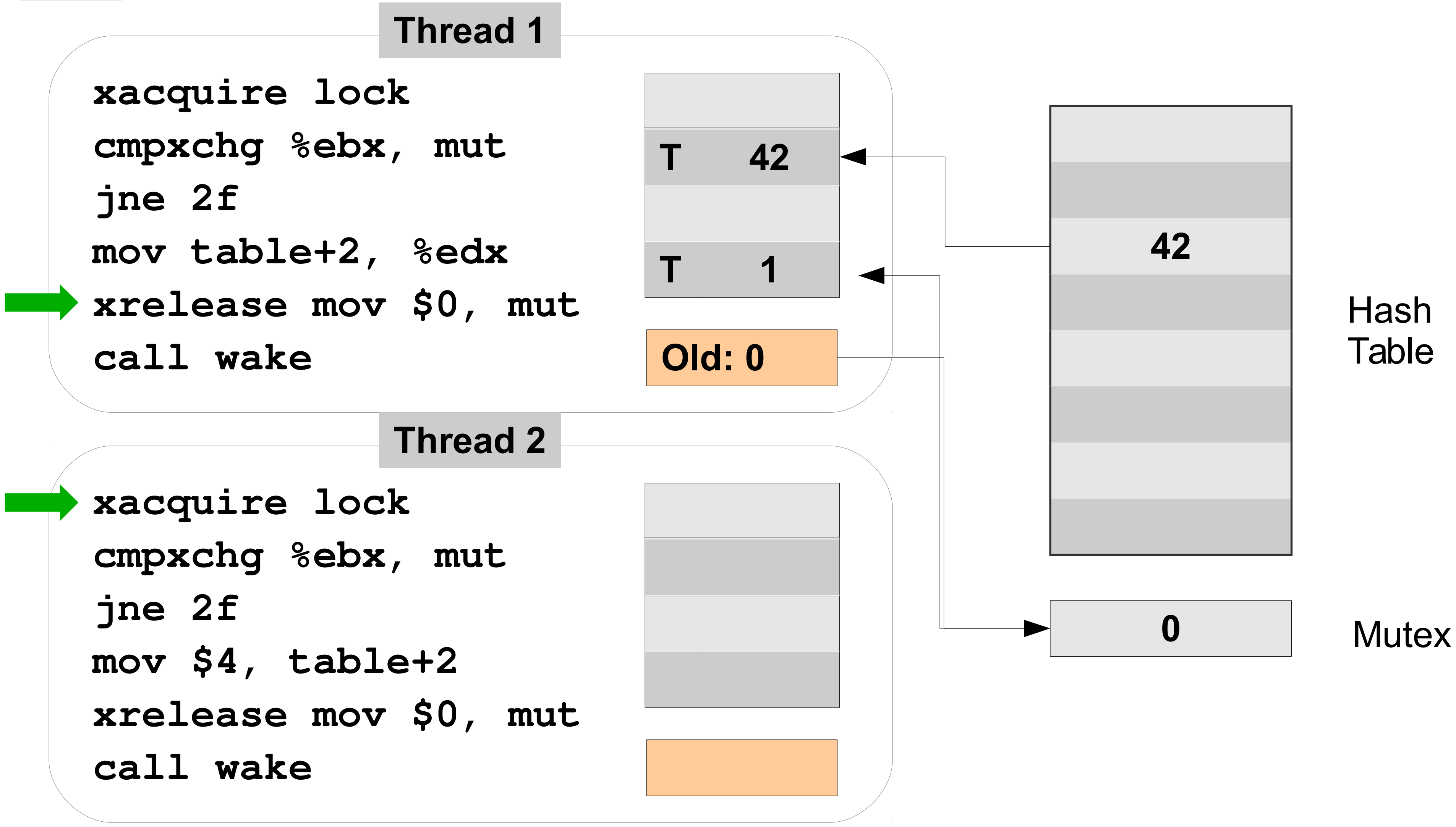
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



With Collision



Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```

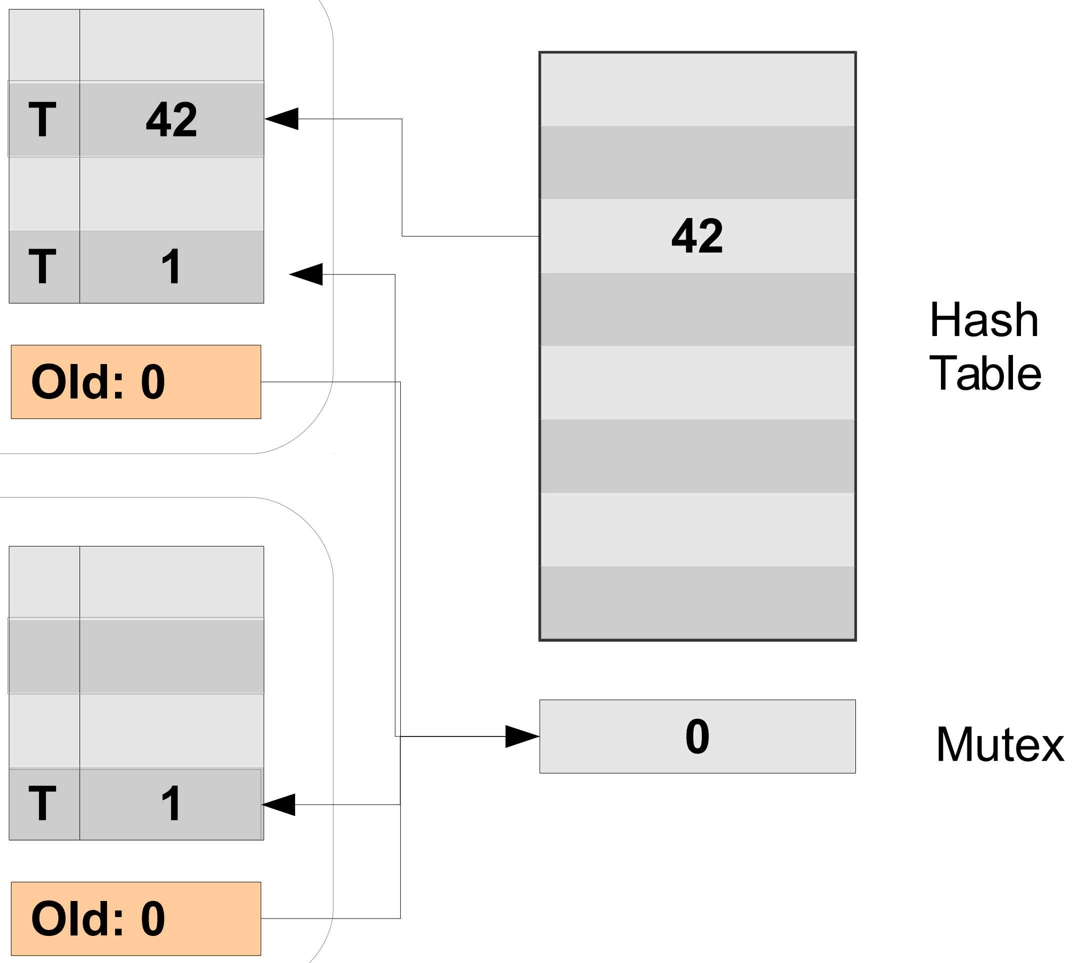
With Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



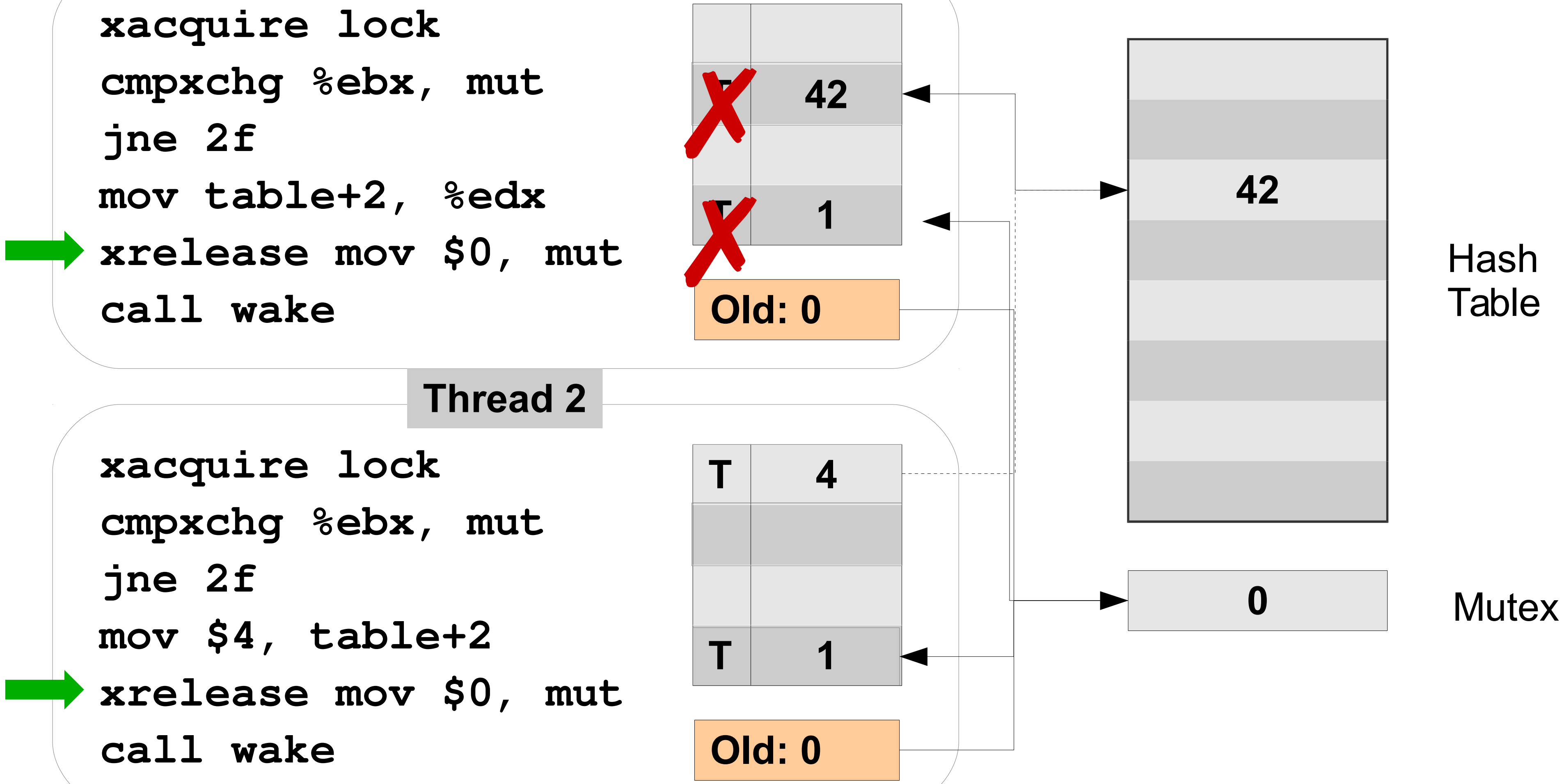
With Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



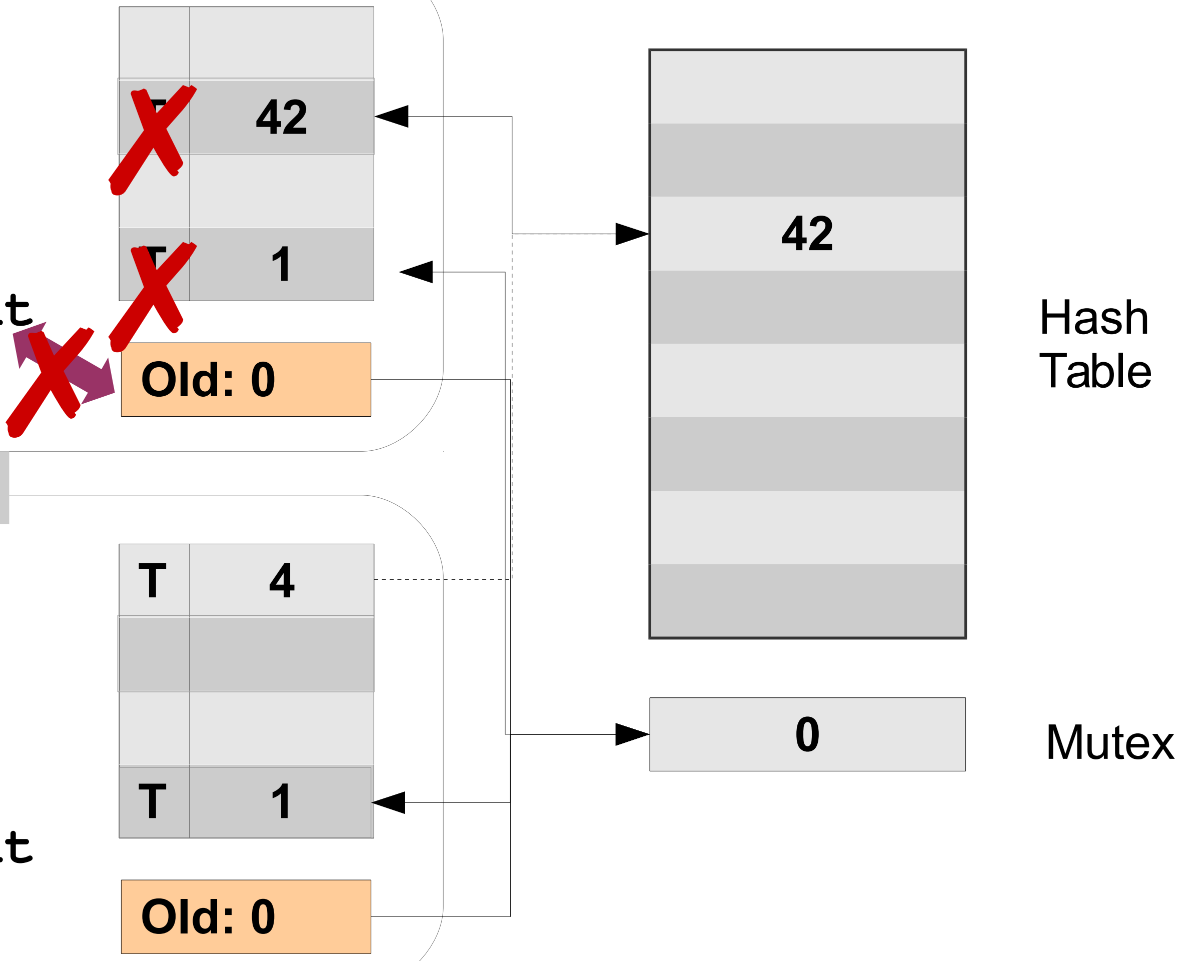
With Collision

Thread 1

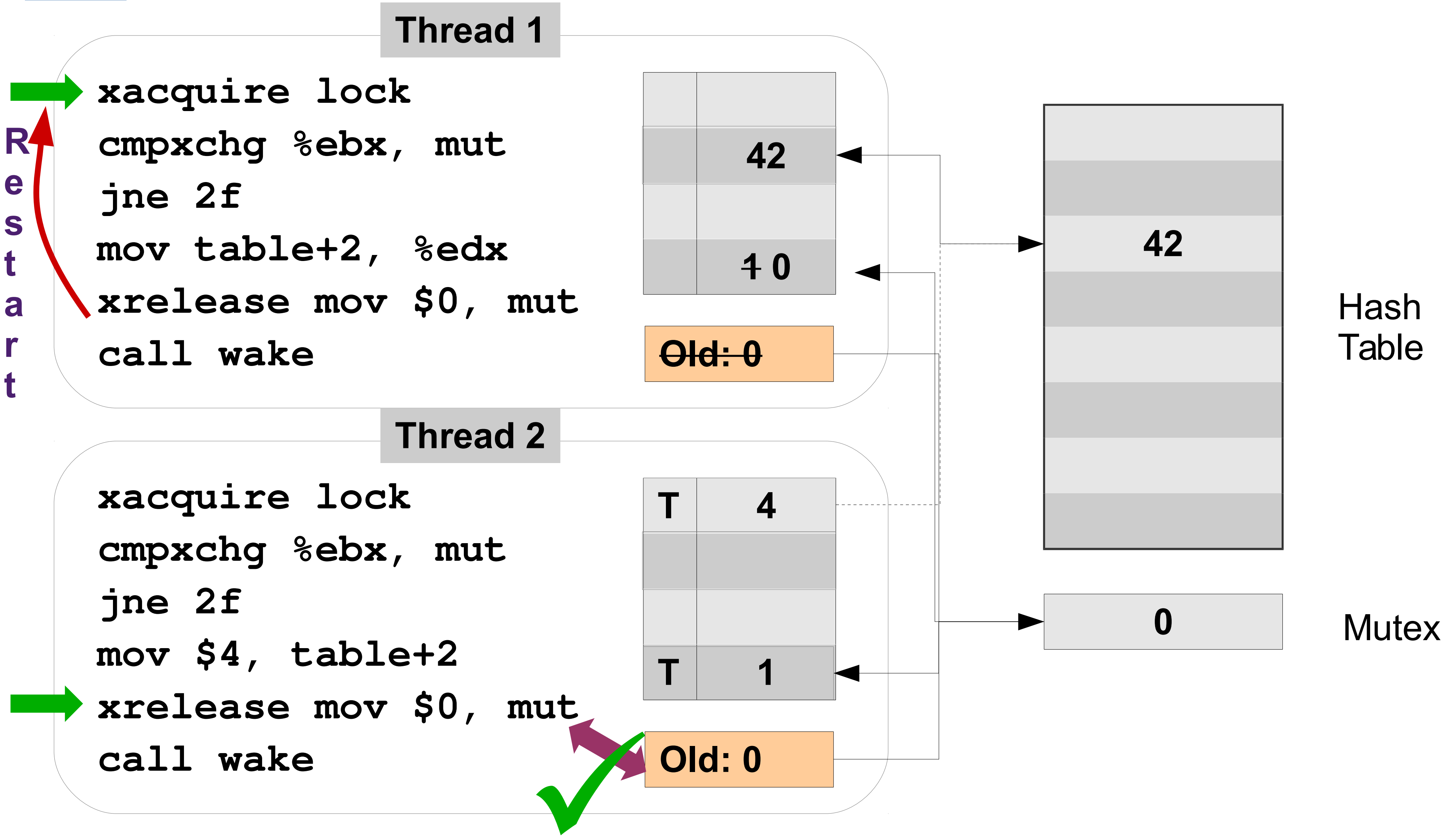
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



With Collision



Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```

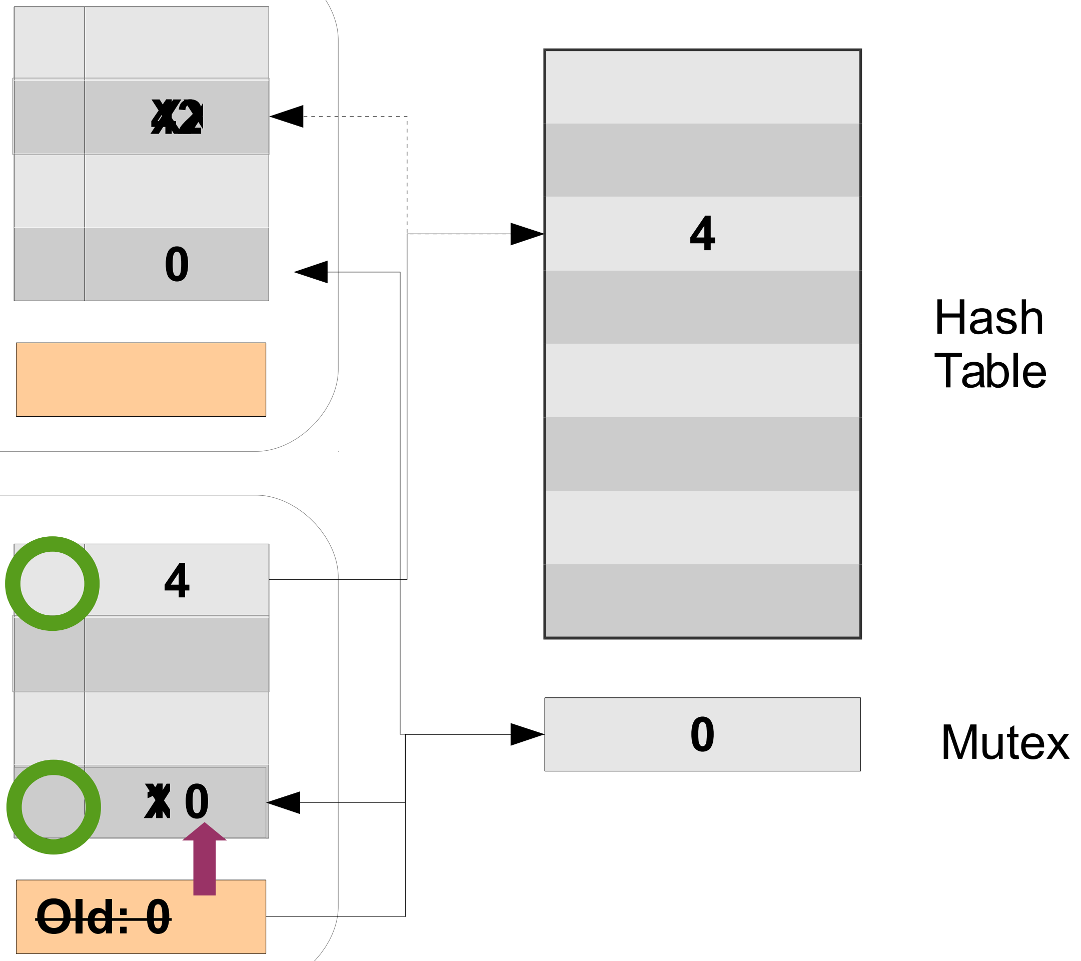
With Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



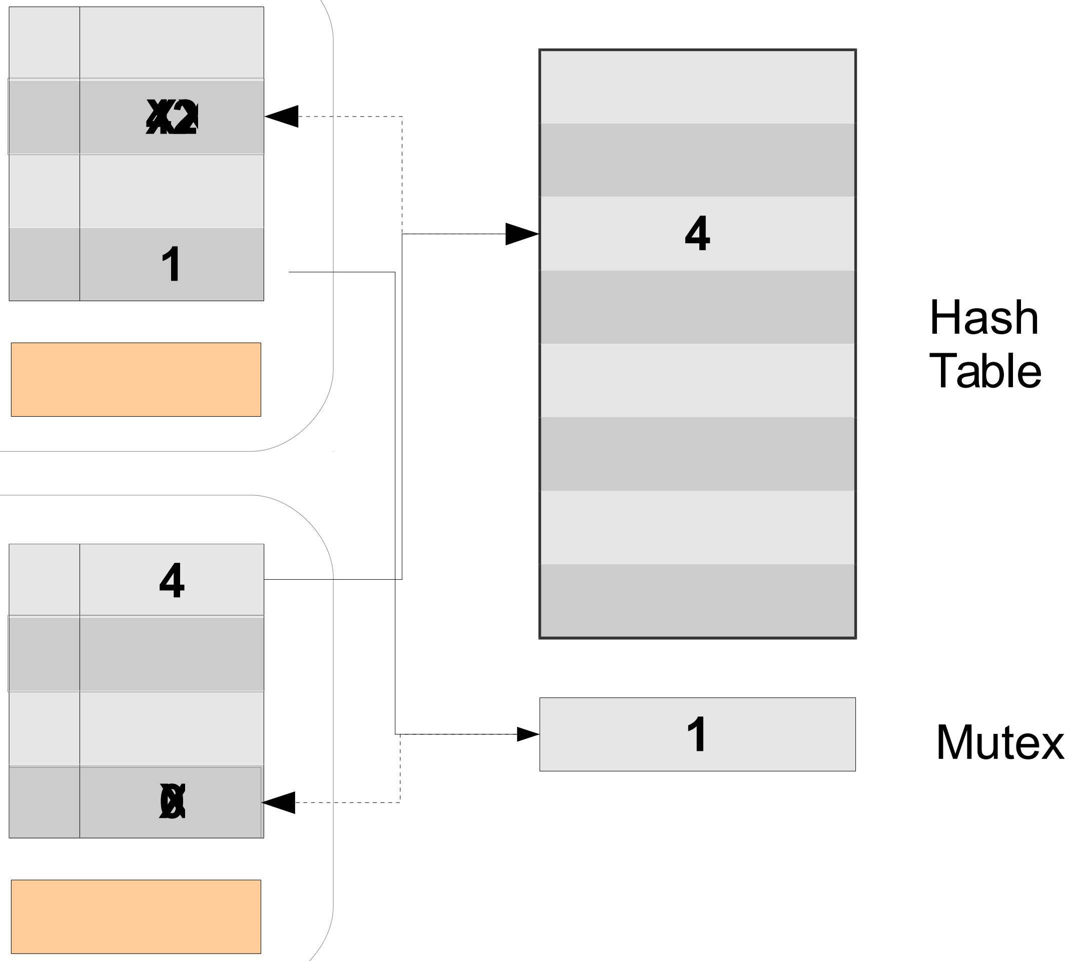
With Collision

Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



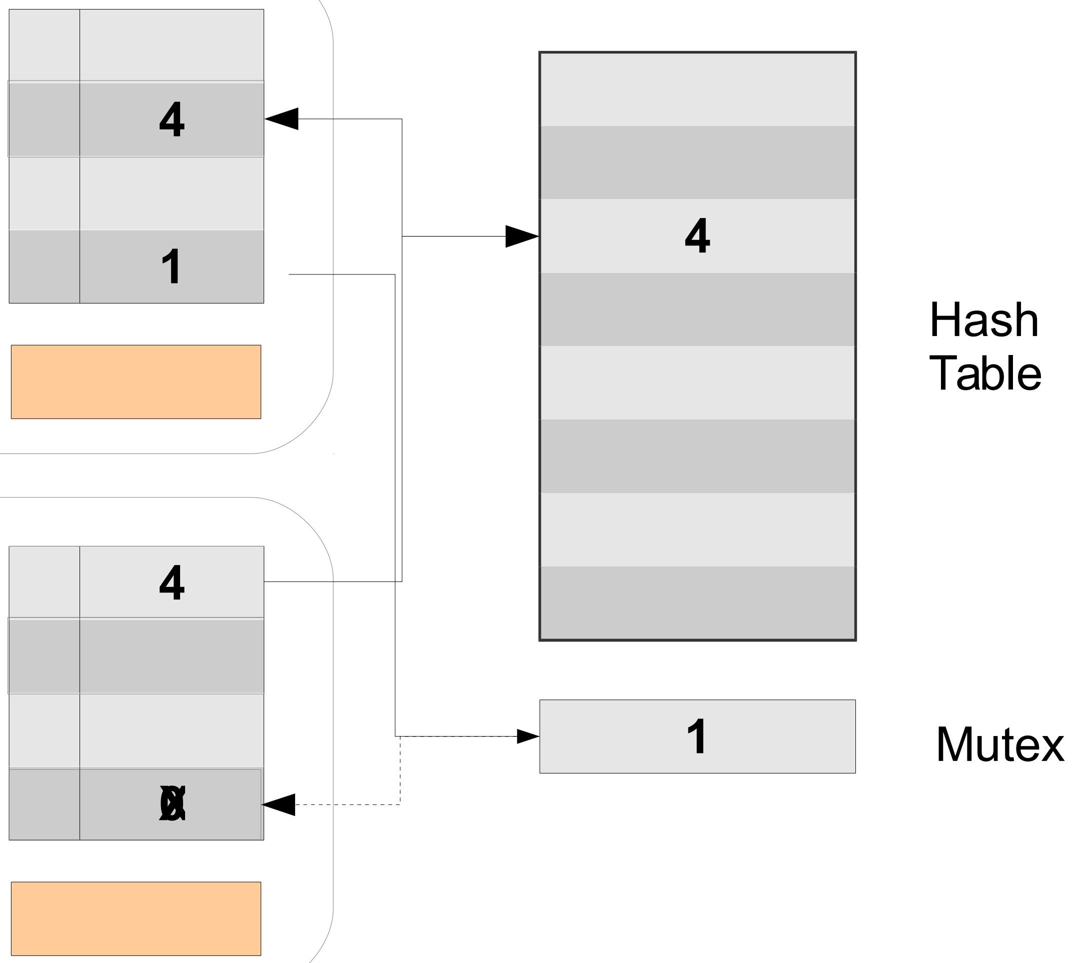
With Collision

Thread 1

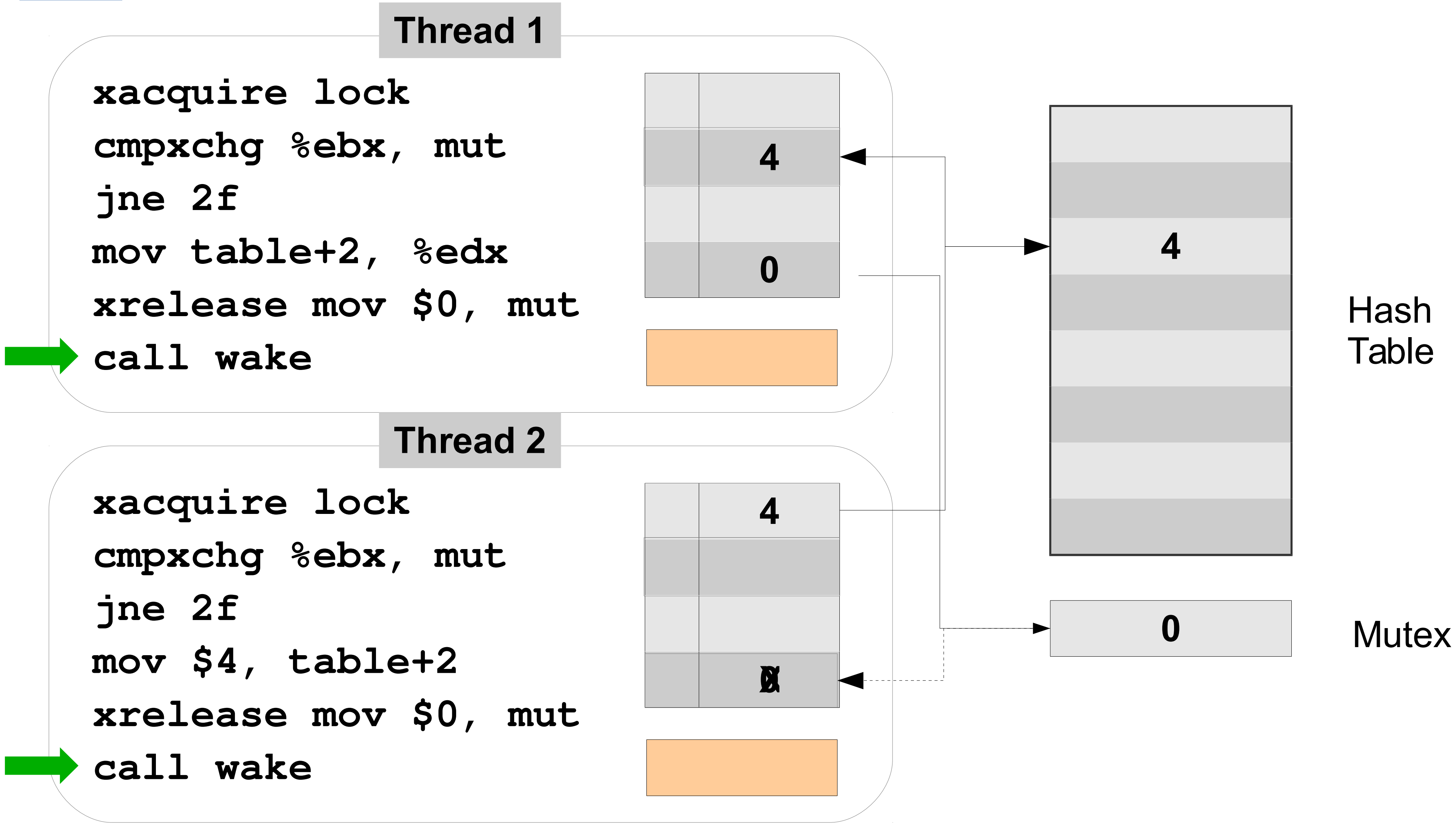
```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```



With Collision



Thread 1

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov table+2, %edx
xrelease mov $0, mut
call wake
```

Thread 2

```
xacquire lock
cmpxchg %ebx, mut
jne 2f
mov $4, table+2
xrelease mov $0, mut
call wake
```

Benefits for HLE

Lock-Free with HLE

		LIFO	FIFO	Hash	Single Linked	Double Linked
No Priority	1:1	CAS	CAS	HLE*	HLE**	HLE**
	1:N	CAS	DWCAS	HLE*	HLE**	HLE**
	N:1	CAS	CAS	HLE*	HLE**	HLE**
	M:N	CAS	DWCAS	HLE*	HLE**	HLE**
Priority	1:1	CAS	CAS	HLE*	HLE**	HLE**
	1:N	HLE	HLE	HLE*	HLE**	HLE**
	N:1	CAS	CAS	HLE*	HLE**	HLE**
	M:N	HLE	HLE	HLE*	HLE**	HLE**

Lock-Free with HLE

		LIFO	FIFO	Hash	Single Linked	Double Linked
No Priority	1:1	CAS	CAS	HLE*	HLE**	HLE**
	1:N			HLE*	HLE**	HLE**
	N:1			HLE*	HLE**	HLE**
	M:N	CAS	DWCAS	HLE*	HLE**	HLE**
Priority	1:1	CAS	CAS	HLE*	HLE**	HLE**
	1:N	HLE	HLE	HLE*	LE**	HLE**
	N:1	CAS	C		**	HLE**
	M:N	HLE	HLE	HLE	HLE**	HLE**

* = reasonable high limit for internal or external hashing

** = list length limited by cache size

Problems

Problems with HLE

- Granularity: cache lines
- Wrong conflicts through false shaing
  - Possible slowdown
- Compact data structures needed
- Abort/restart policy not architected
- Usable only for simple operations
  - No system calls, page faults, ...

Transactional Memory

History

- Available for some time

Wait-Free Synchronization, Maurice Herlihy, ACM Transactions on Programming Languages and Systems, 1991

Goldman Sachs

Software TM

Support added to C and C++

```
void insert(node *p) {
  guard g(lock);
  node **prev = &list;
  node *l = list;
  while (l &&
         l->val < p->val) {
    prev = &l->next;
    l = l->next;
  }
  p->next = l;
  *prev = p;
}
```

```
void insert(node *p) {
  tm_atomic {
    node **prev = &list;
    node *l = list;
    while (l &&
          l->val < p->val){
      prev = &l->next;
      l = l->next;
    }
    p->next = l;
    *prev = p;
  }
}
```

Software TM

- No locking needed
- Concurrency enabled
- Exception-safe
- Transparent use of hardware TM support through compiler mode

```
void insert(node *p) {
  tm_atomic {
    node **prev = &list;
    node *l = list;
    while (l &&
           l->val < p->val){
      prev = &l->next;
      l = l->next;
    }
    p->next = l;
    *prev = p;
  }
}
```

Intel RTM

Thread-Unsafe List

```
    mov    list(%rip),%rax
    mov    $list,%edx
    test   %rax,%rax
    je     1f
    mov    0x8(%rdi),%ecx
    jmp    2f
3:  mov    %rax,%rdx
    mov    (%rax),%rax
    test   %rax,%rax
    je     1f
2:  cmp    %ecx,0x8(%rax)
    jl     3b
1:  mov    %rax,(%rdi)
    mov    %rdi,(%rdx)

    ret
```

```
void insert(node *p) {
  tm_atomic {
    node **prev = &list;
    node *l = list;
    while (l &&
           l->val < p->val){
      prev = &l->next;
      l = l->next;
    }
    p->next = l;
    *prev = p;
  }
}
```

Thread-Unsafe List

```
    mov     list(%rip),%rax
    mov     $list,%edx
    test    %rax,%rax
    je      1f
    mov     0x8(%rdi),%ecx
    jmp     2f
3:  mov     %rax,%rdx
    mov     (%rax),%rax
    test    %rax,%rax
    je      1f
2:  cmp     %ecx,0x8(%rax)
    jl      3b
1:  mov     %rax,(%rdi)
    mov     %rdi,(%rdx)

    ret
```

```
void insert(node *p) {

    node **prev = &list;
    node *l = list;
    while (l &&
           l->val < p->val){
      prev = &l->next;
      l = l->next;
    }
    p->next = l;
    *prev = p;

}
```

Thread-Safe List

Restart

```
    movl    $MAX, cnt(%rsp)
0:  xbegin  .Labort
    mov     list(%rip),%rax
    mov     $list,%edx
    test    %rax,%rax
    je      1f
    mov     0x8(%rdi),%ecx
    jmp     2f
3:  mov     %rax,%rdx
    mov     (%rax),%rax
    test    %rax,%rax
    je      1f
2:  cmp     %ecx,0x8(%rax)
    jl      3b
1:  mov     %rax,(%rdi)
    mov     %rdi,(%rdx)
    xend
    ret
```

```
void insert(node *p) {
  tm_atomic {
    node **prev = &list;
    node *l = list;
    while (l &&
           l->val < p->val){
      prev = &l->next;
      l = l->next;
    }
    p->next = l;
    *prev = p;
  }
}
```

Thread-Safe List

Restart

```
    movl    $MAX, cnt(%rsp)
0:  xbegin  .Labort
    mov     list(%rip),%rax
    mov     $list,%edx
    test    %rax,%rax
    je      1f
    mov     0x8(%rdi),%ecx
    jmp     2f
3:  mov     %rax,%rdx
    mov     (%rax),%rax
    test    %rax,%rax
    je      1f
2:  cmp     %ecx,0x8(%rax)
    jl      3b
1:  mov     %rax,(%rdi)
    mov     %rdi,(%rdx)
    xend
    ret
```

```
.Labort:
    test    $2, %rax
    jz      .Ltrylocking
    decl    cnt(%rsp)
    jne     0b
.Ltrylocking:
    ...
```

Composition

```
tm_atomic {
  if ((i = find(l1.begin(), l1.end(), val)) != l1.end()) {
    l2.push_front(*i);
    l1.erase(i);
  }
}
```

```
xbegin                                reference count: 1
  find:        xbegin                                  2
               ...
               xend                                    1
  push_front:  xbegin                                  2
               ...
               xend                                    1
  erase:       xbegin                                  2
               ...
               xend                                    1
xend                                                   0   COMMIT!
```

Problems

Problems with RTM

- Cache line granularity
  - False sharing can lead to unpredictable aborts
- Composability limited
  - No system calls etc
  - Limited TM compiler knowledge so far
    - More knowledge about libraries and more function annotation needed
- No experience with abort handlers yet
- STM fallback solution <u>very</u> slow

Summary

Summary

- Increase level of parallelization through HLE
  - Opportunistic execution
  - Fully backward compatible
  - No more need for reader/writer locks
- Solve composition with RTM
  - Available through language extensions
- Problems
  - How to handle cache line-granularity?

Questions?