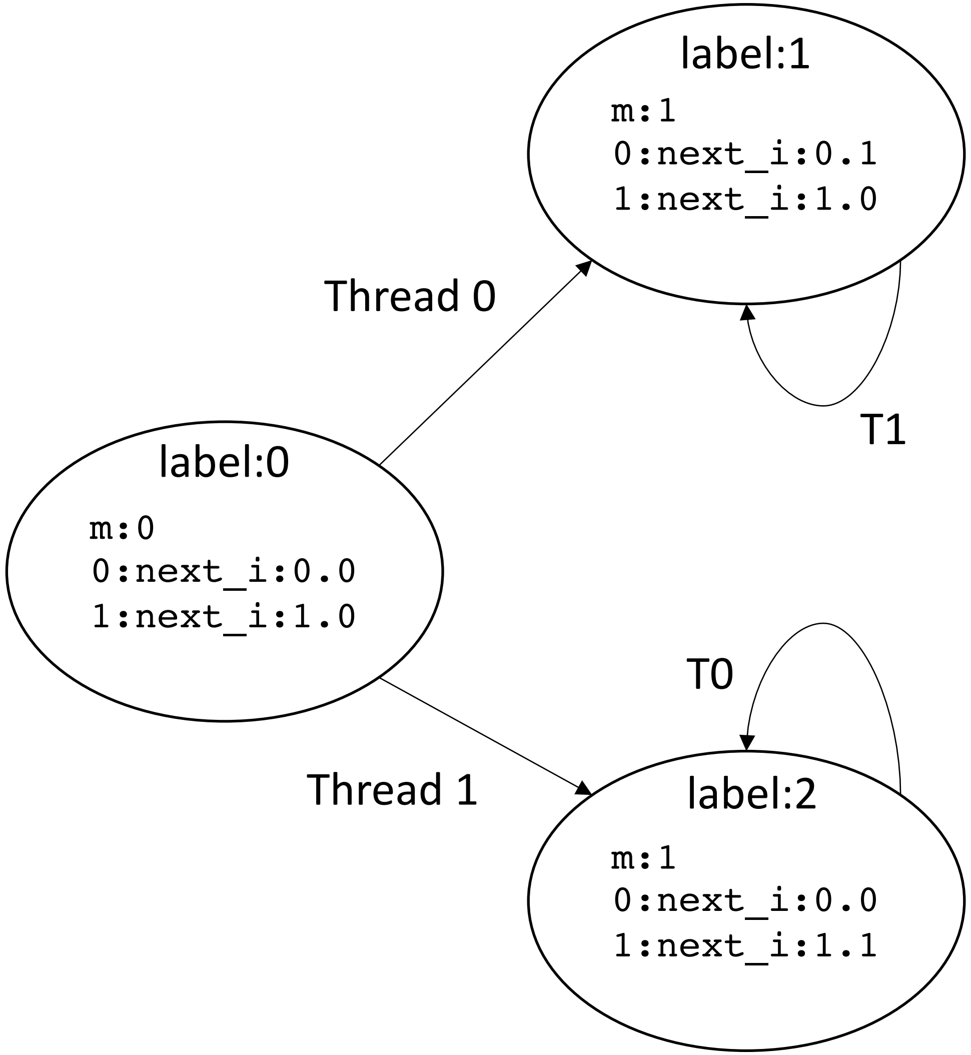
# CSE113: Parallel Programming

Feb. 25, 2022

- **Topics**:
  - Finishing up memory models
    - An cautionary tale
    - portability
  - Reasoning about schedulers
    - Labeled transition systems



# Announcements

- HW 4 is out
  - Due in 1 week
  - Please don’t share timing results until next Monday
  - We will discuss a few issues today

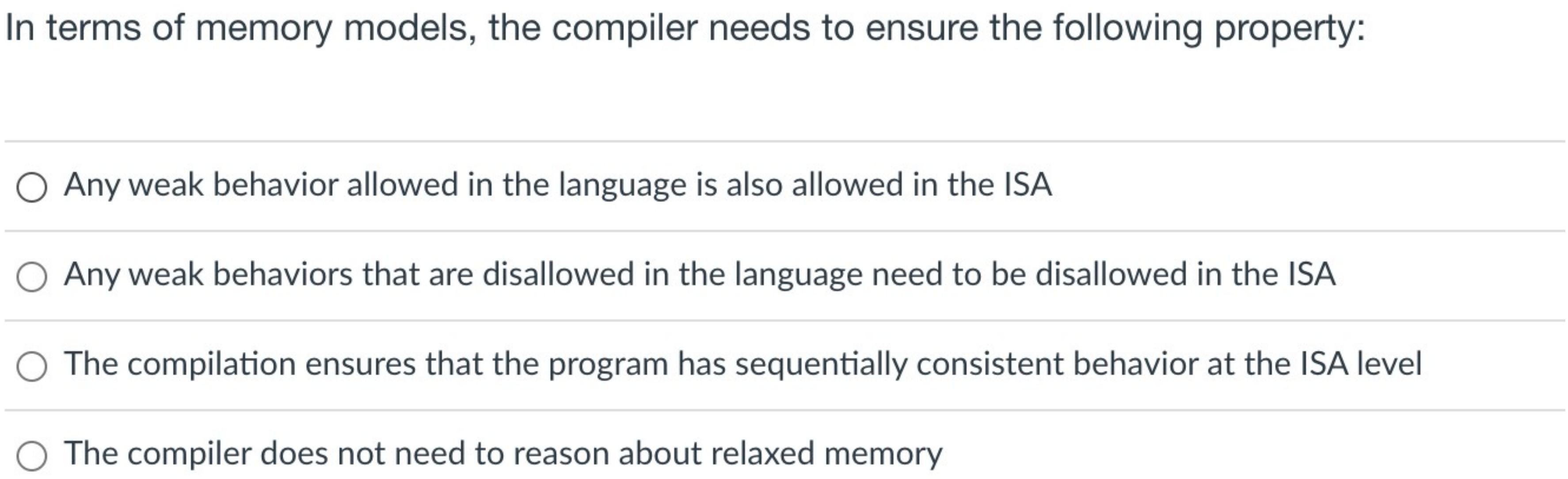
- Grades for HW 2 are out
  - Let us know by next Monday if you have any issues

- Grades for Midterm are on their way
  - Expect them by Monday

# Today’s Quiz

- Due Monday by class; please do it!

# Previous quiz



In terms of memory models, the compiler needs to ensure the following property:

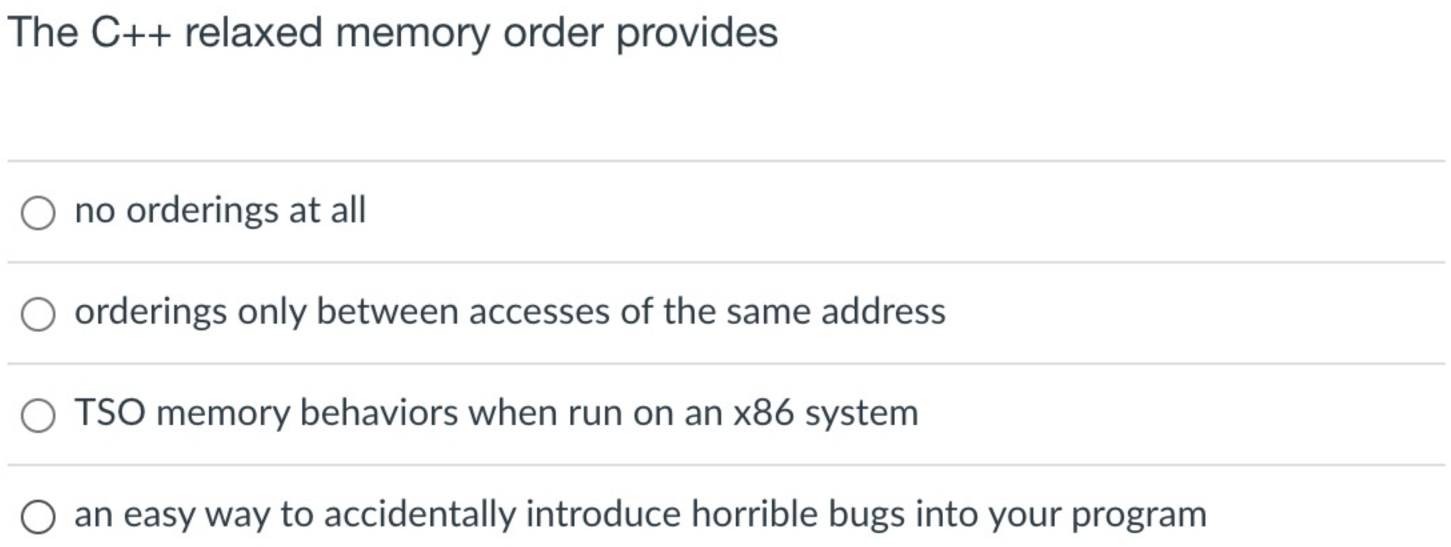
○ Any weak behavior allowed in the language is also allowed in the ISA

○ Any weak behaviors that are disallowed in the language need to be disallowed in the ISA

○ The compilation ensures that the program has sequentially consistent behavior at the ISA level

○ The compiler does not need to reason about relaxed memory

# Previous quiz



The C++ relaxed memory order provides

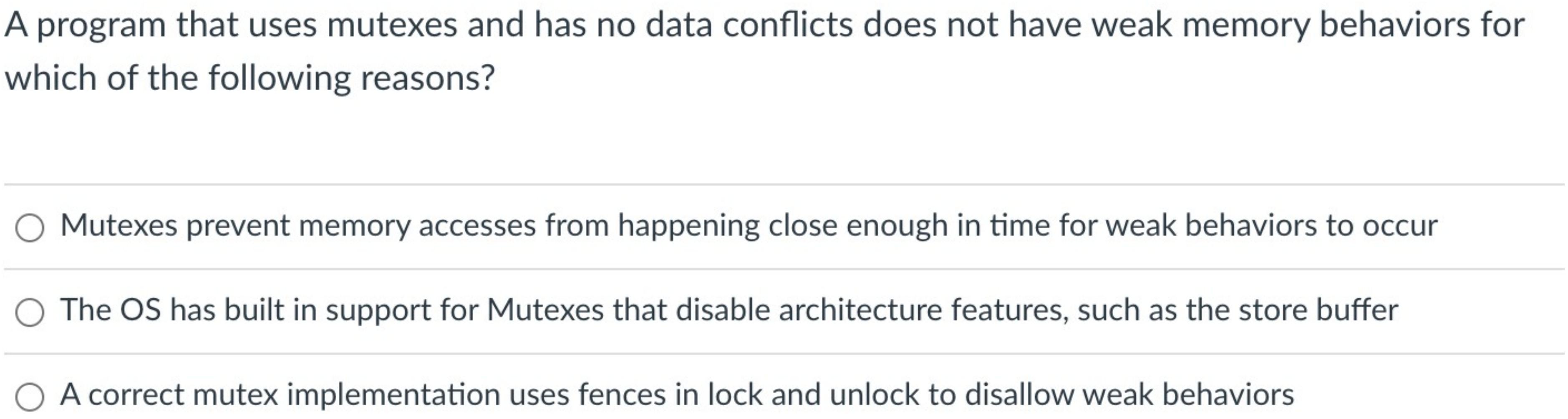
○ no orderings at all

○ orderings only between accesses of the same address

○ TSO memory behaviors when run on an x86 system

○ an easy way to accidentally introduce horrible bugs into your program

# Previous quiz



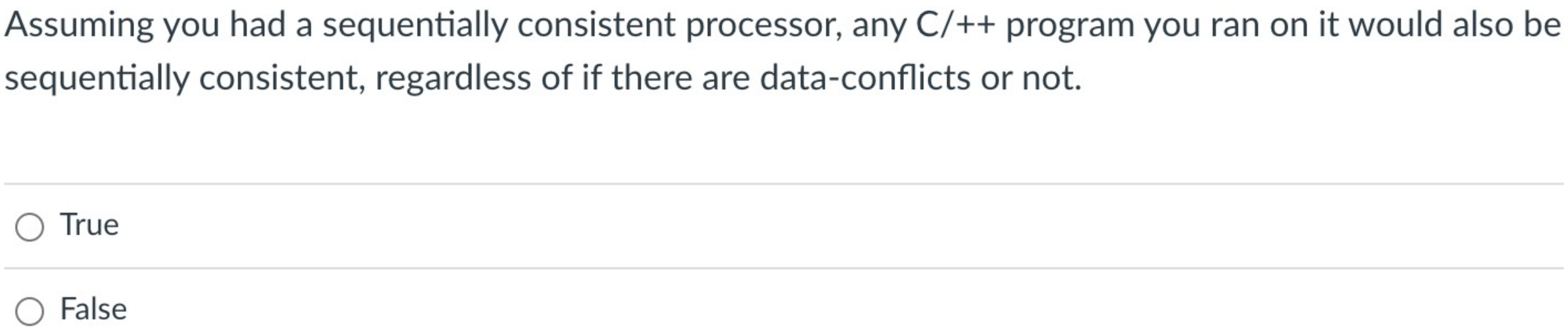
A program that uses mutexes and has no data conflicts does not have weak memory behaviors for which of the following reasons?

○ Mutexes prevent memory accesses from happening close enough in time for weak behaviors to occur

○ The OS has built in support for Mutexes that disable architecture features, such as the store buffer

○ A correct mutex implementation uses fences in lock and unlock to disallow weak behaviors

# Previous quiz

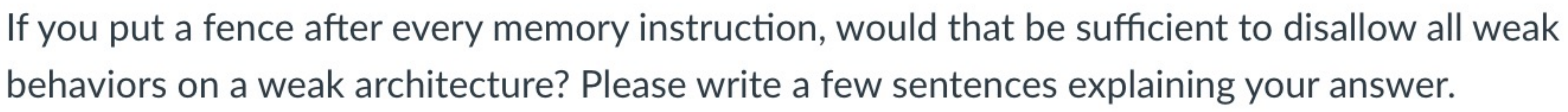


Assuming you had a sequentially consistent processor, any C/++ program you ran on it would also be sequentially consistent, regardless of if there are data-conflicts or not.

○ True

○ False

# Previous quiz



If you put a fence after every memory instruction, would that be sufficient to disallow all weak behaviors on a weak architecture? Please write a few sentences explaining your answer.

# Review

# Relaxed memory models

## C++ language

default (sequential consistency)

| | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

relaxed memory order

| | L | S |
|---|---|---|
| L | different address | different address |
| S | different address | different address |

*ISA memory models*

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

TSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |

PSO

| | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

RMO

# Weak memory examples

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

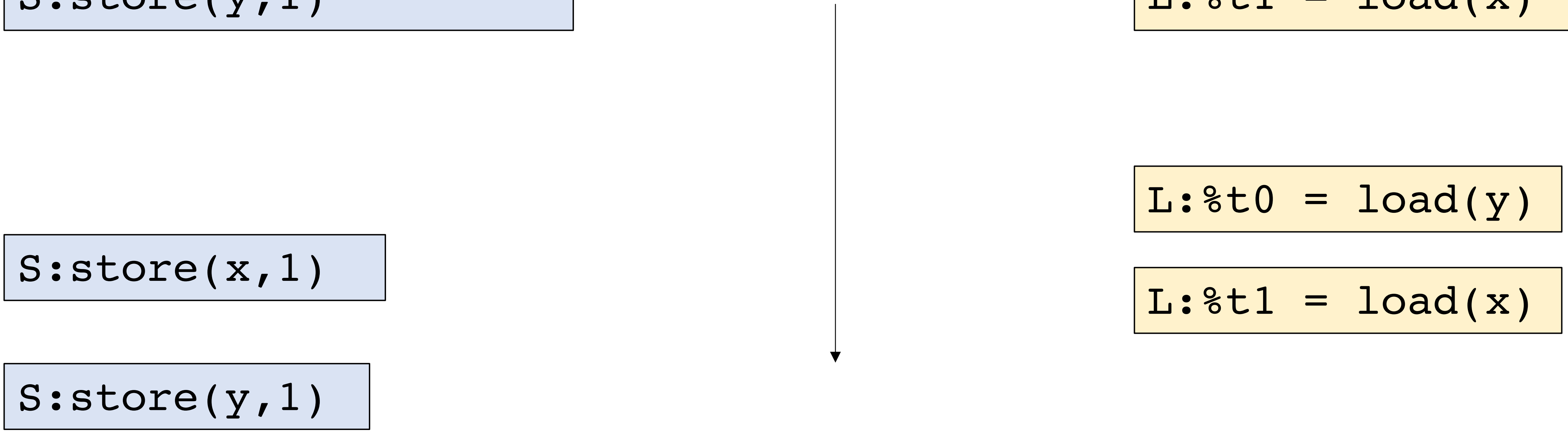
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



```
L:%t0 = load(y)
L:%t1 = load(x)
S:store(x,1)
S:store(y,1)
```

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

start off thinking about sequential consistency

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```

```
L:%t0 = load(y)
```

```
L:%t1 = load(x)
```

```
S:store(x,1)
```

```
S:store(y,1)
```

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

Question: can `t0 == 1 and t1 == 0`?

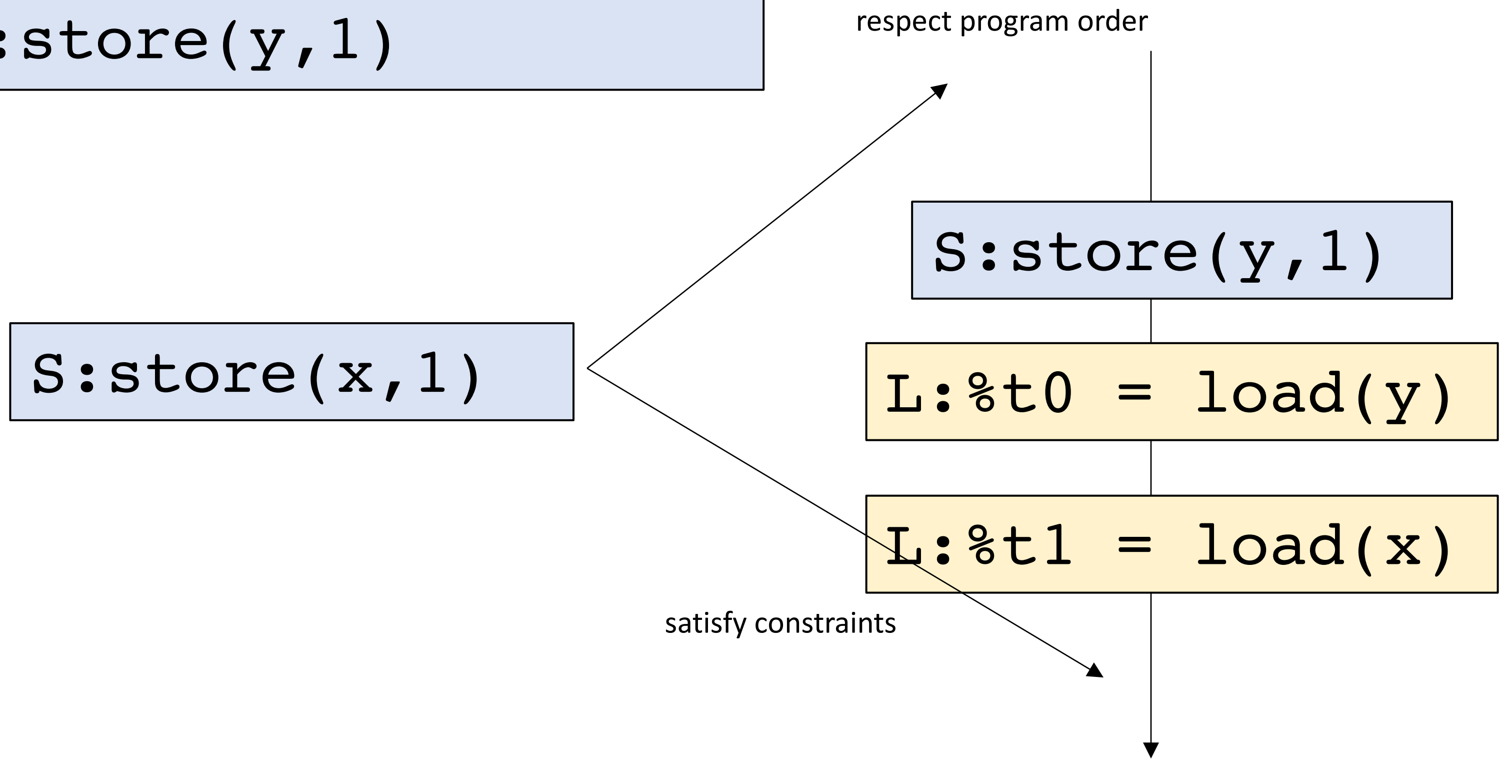
*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

start off thinking about sequential consistency

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

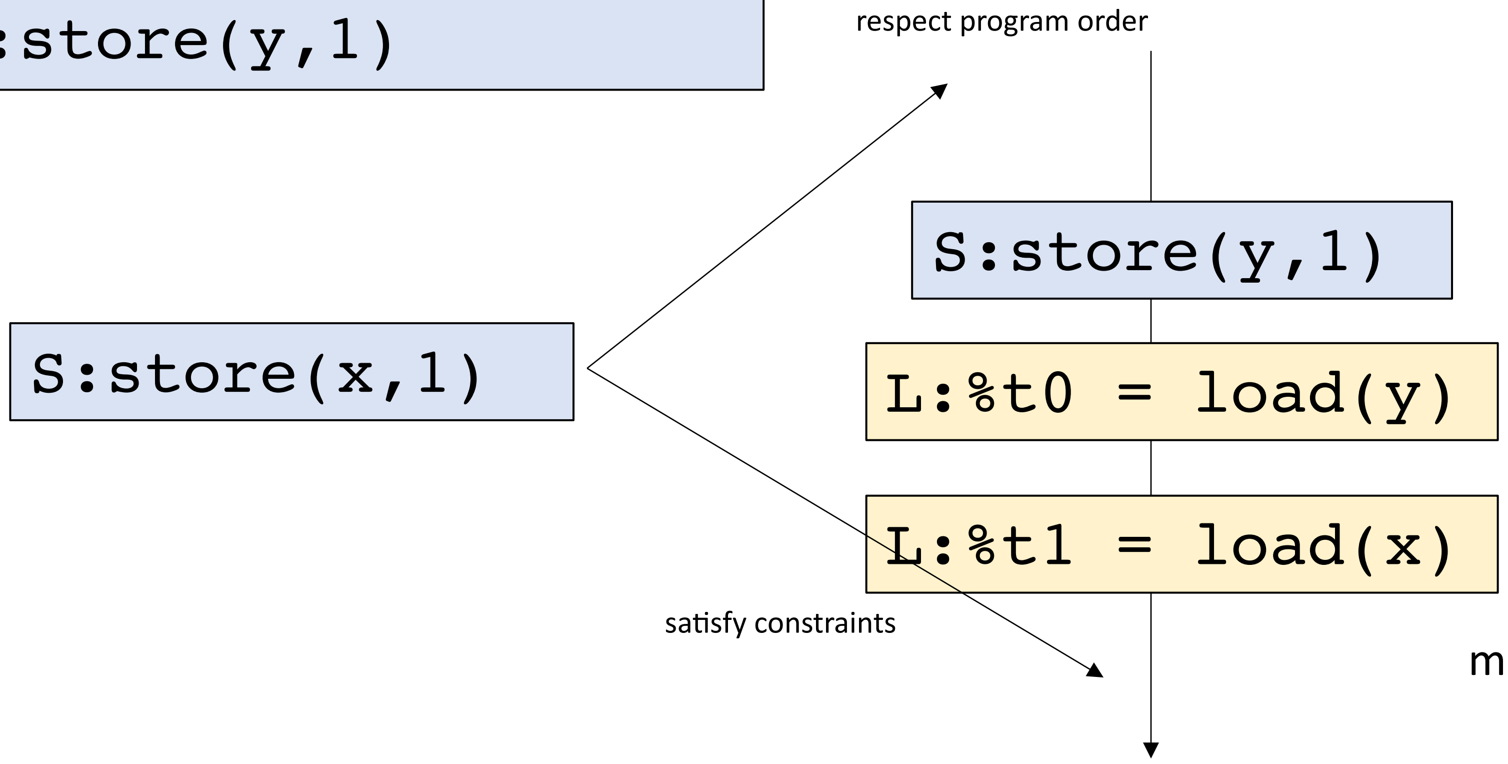
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

```
S:store(x,1)
```

satisfy constraints

| | | memory access 0 | |
|---|---|---|---|
| | | L | S |
| memory access 1 | L | NO | Different address |
| | S | NO | NO |

What about TSO?

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

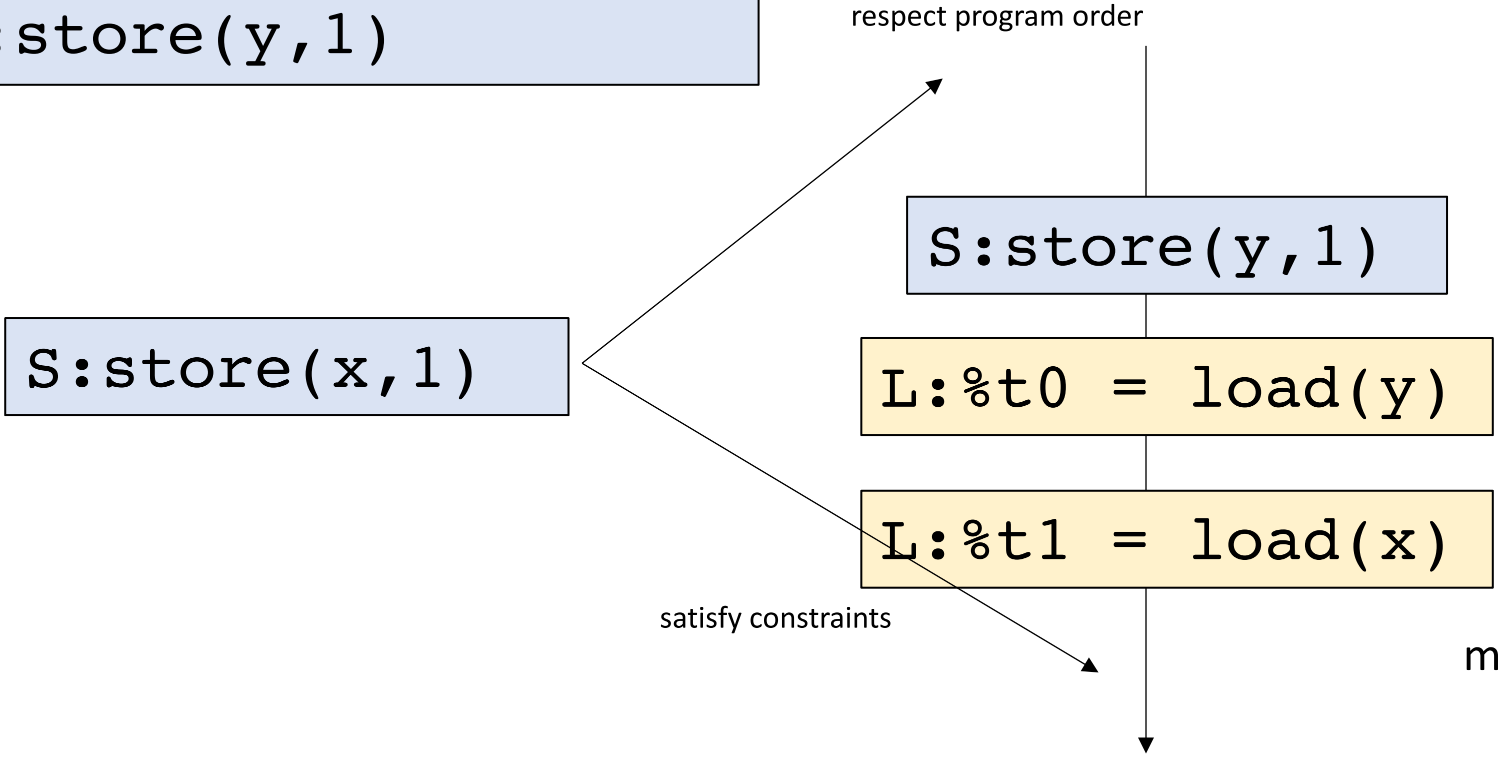
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

```
S:store(x,1)
```

satisfy constraints

| | | memory access 0 | |
|---|---|---|---|
| | | L | S |
| memory access 1 | L | NO | Different address |
| | S | NO | NO |

What about TSO? NO

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

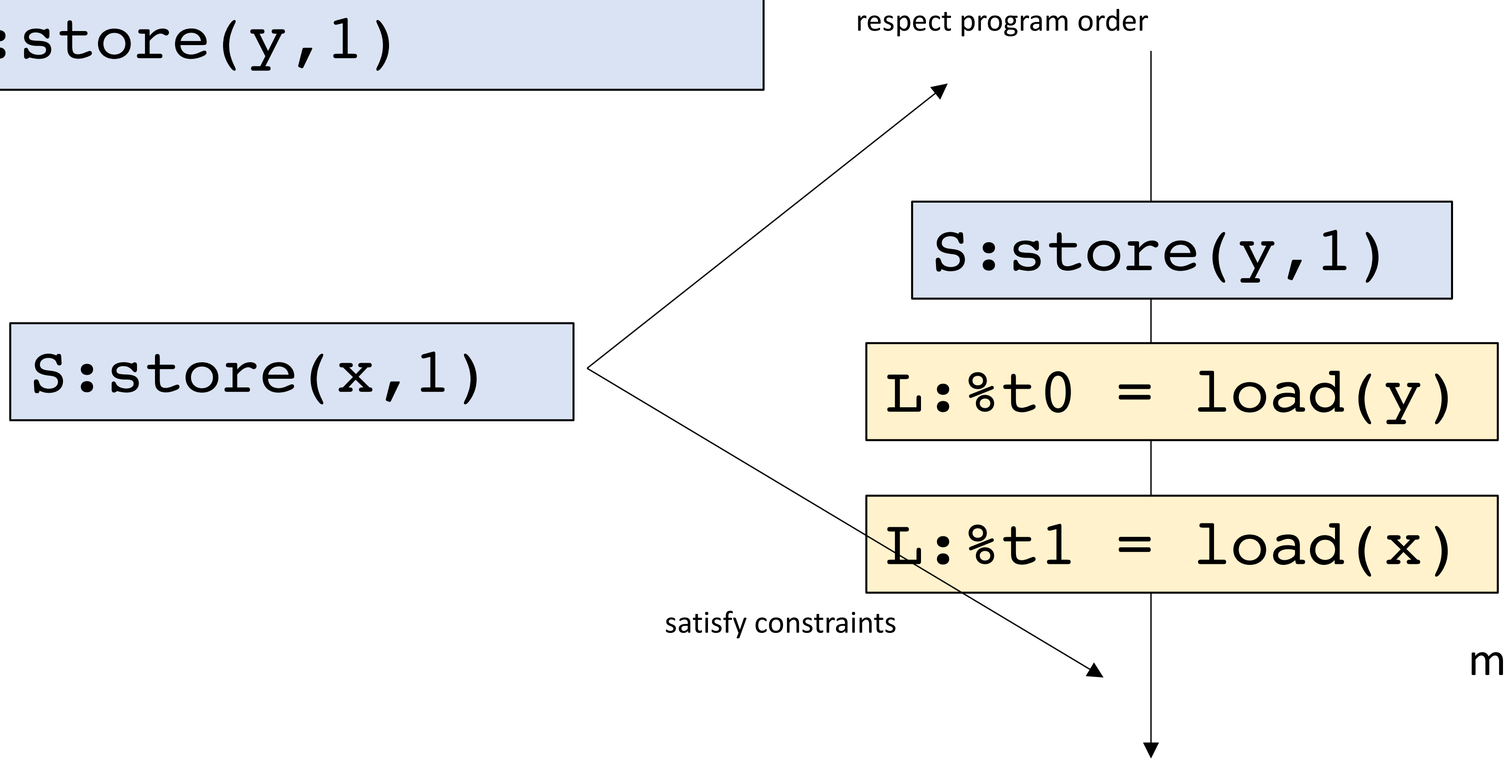
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



|  |  | memory access 0 |  |
|---|---|---|---|
|  |  | L | S |
| memory access 1 | L | NO | Different address |
|  | S | NO | Different address |

What about PSO?

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

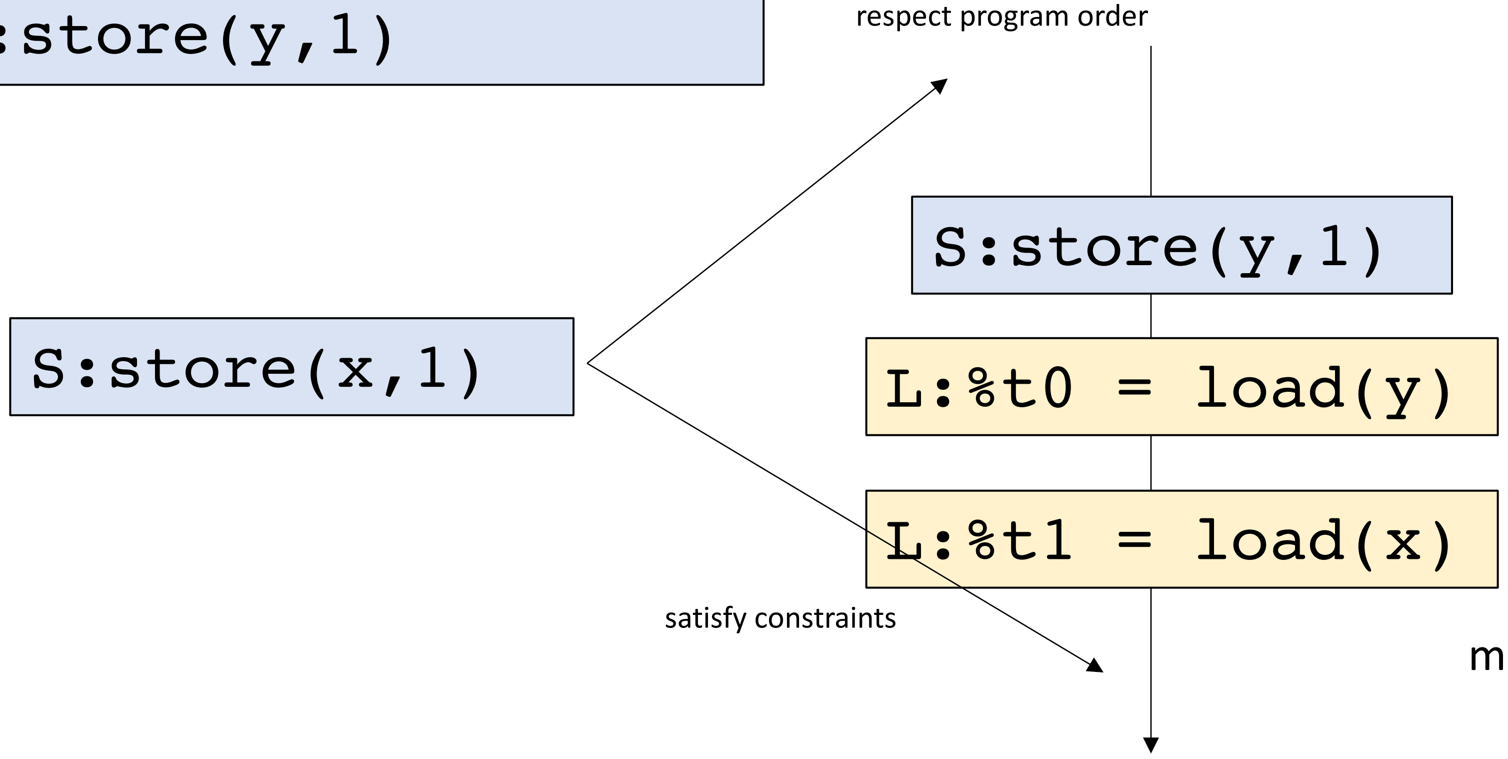
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



respect program order

```
S:store(x,1)
```

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

satisfy constraints

memory access 0 / memory access 1

| memory access 1 \ memory access 0 | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | 🟩 Different address |

What about PSO?

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

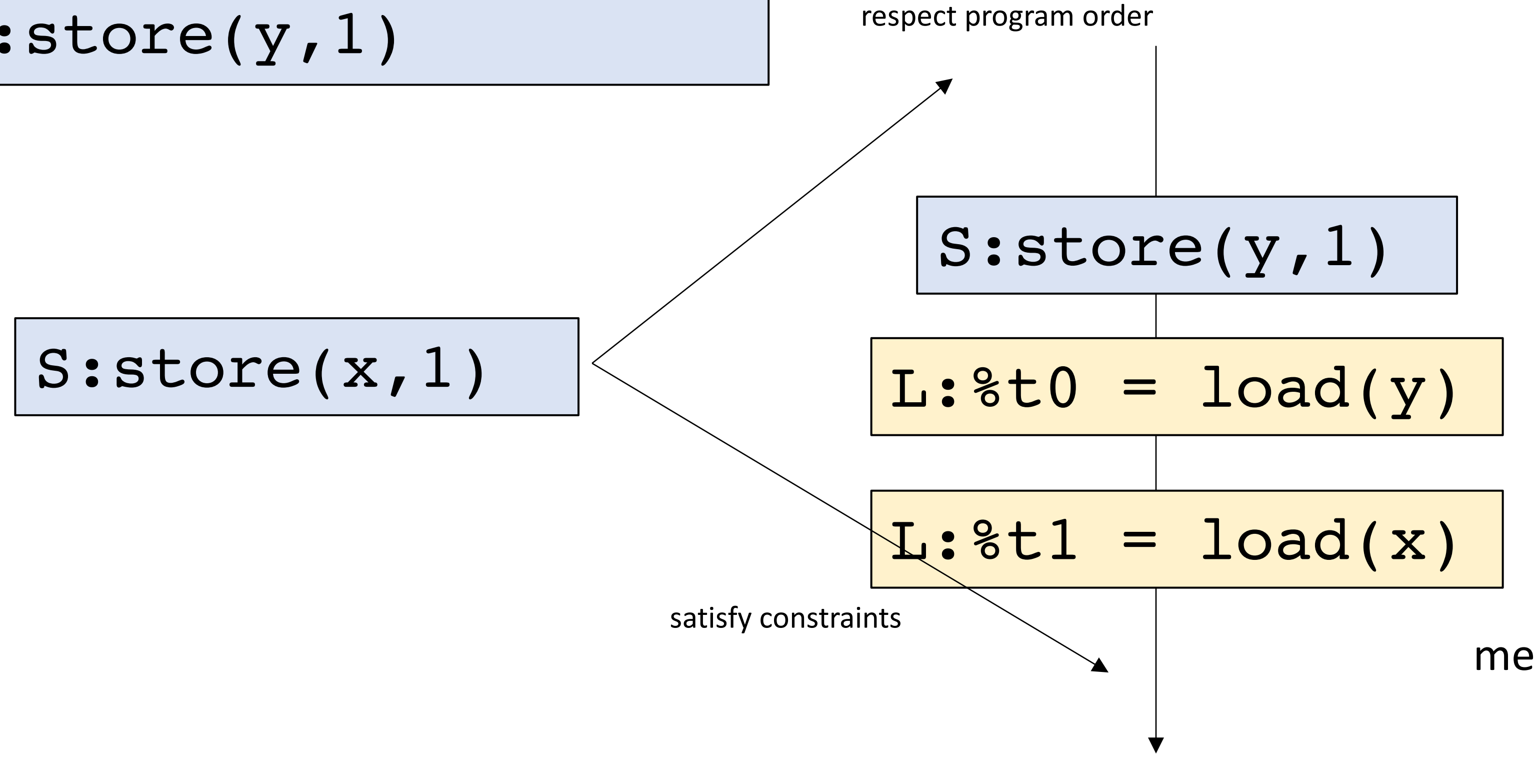
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

```
S:store(x,1)
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



respect program order

```
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

```
S:store(x,1)
```

satisfy constraints

| | | memory access 0 | |
|---|---|---|---|
| | | L | S |
| memory access 1 | L | NO | Different address |
| | S | NO | Different address |

What about PSO? YES

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

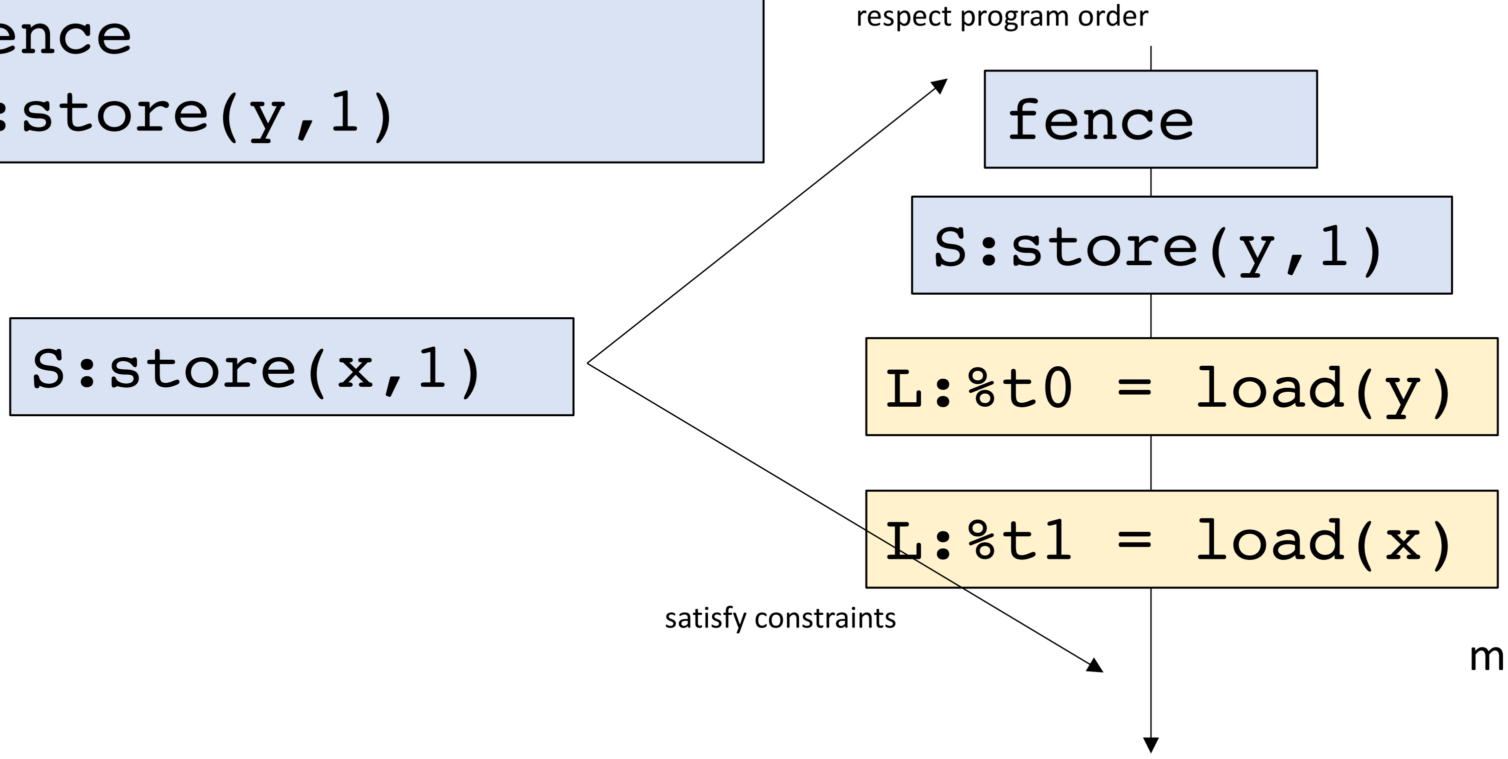
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



S:store(x,1)

respect program order

fence

S:store(y,1)

L:%t0 = load(y)

L:%t1 = load(x)

satisfy constraints

|  |  | memory access 0 |  |
|---|---|---|---|
|  |  | L | S |
| memory access 1 | L | NO | Different address |
|  | S | NO | Different address |

Now it is disallowed in PSO

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

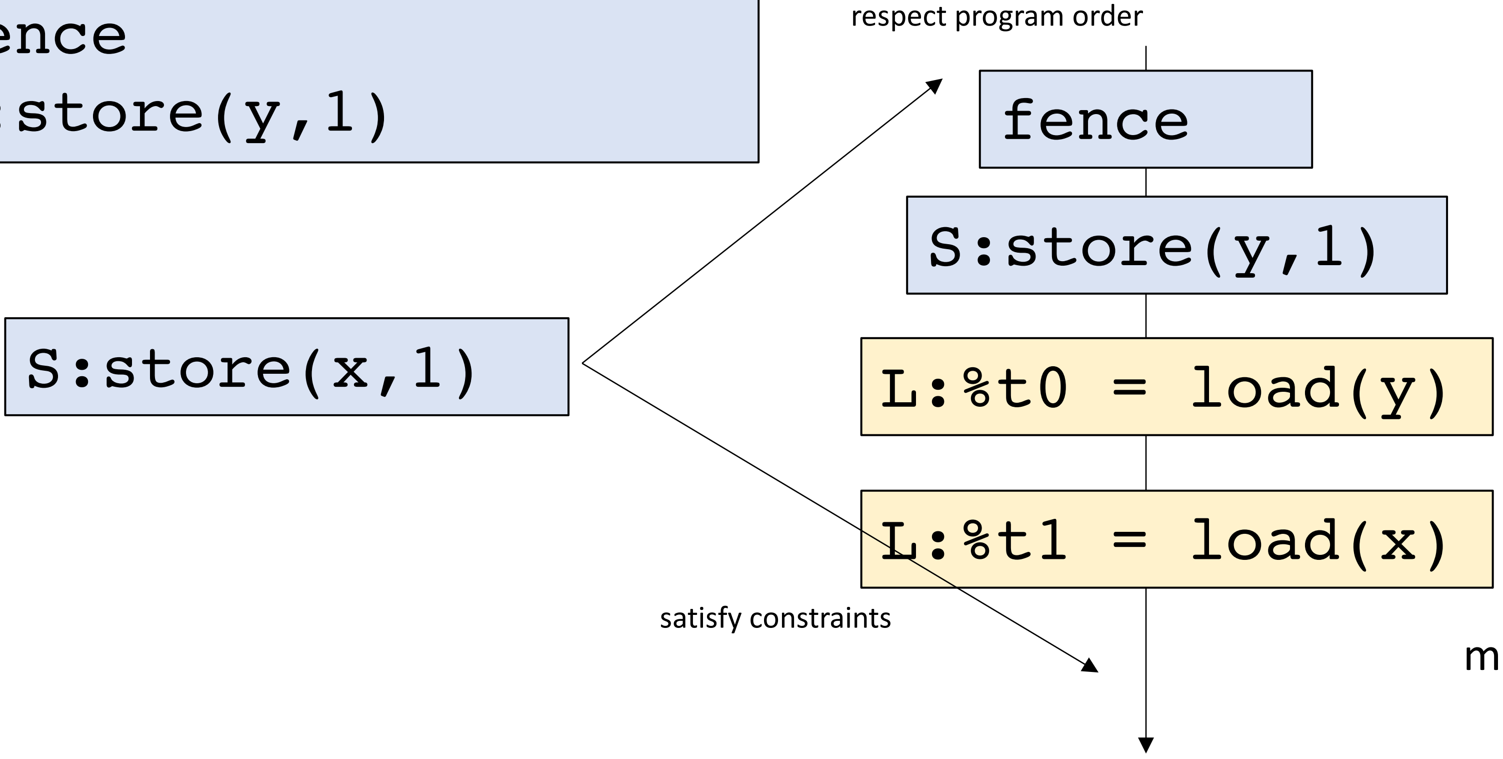
Question: can `t0 == 1 and t1 == 0`?

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



respect program order

```
fence
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

```
S:store(x,1)
```

satisfy constraints

| | | memory access 0 | |
|---|---|---|---|
| | | L | S |
| memory access 1 | L | YES | Different address |
| | S | Different address | Different address |

What about RMO?

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

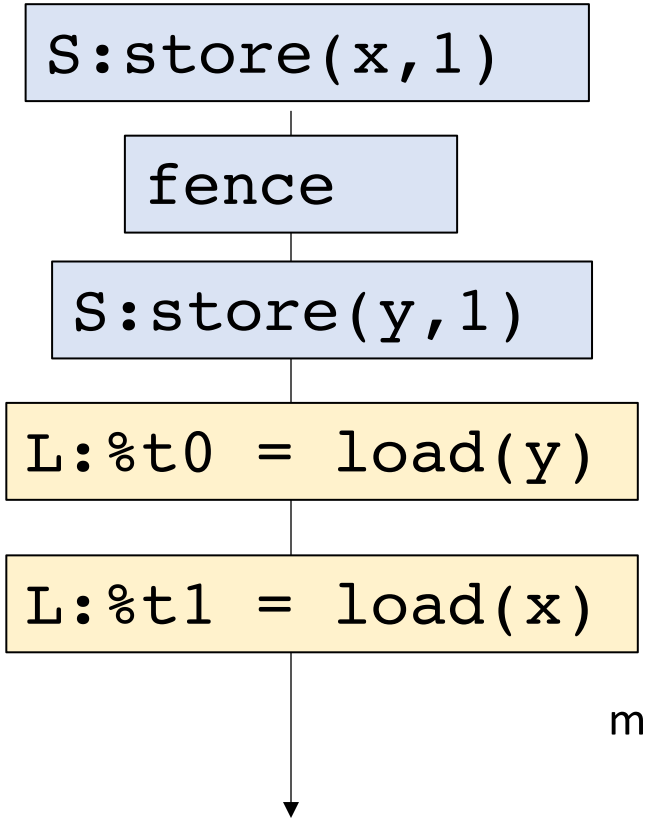
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```



```
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
L:%t1 = load(x)
```

|  |  | memory access 0 |  |
|---|---|---|---|
|  |  | L | S |
| memory access 1 | L | YES | Different address |
|  | S | Different address | Different address |

What about RMO?

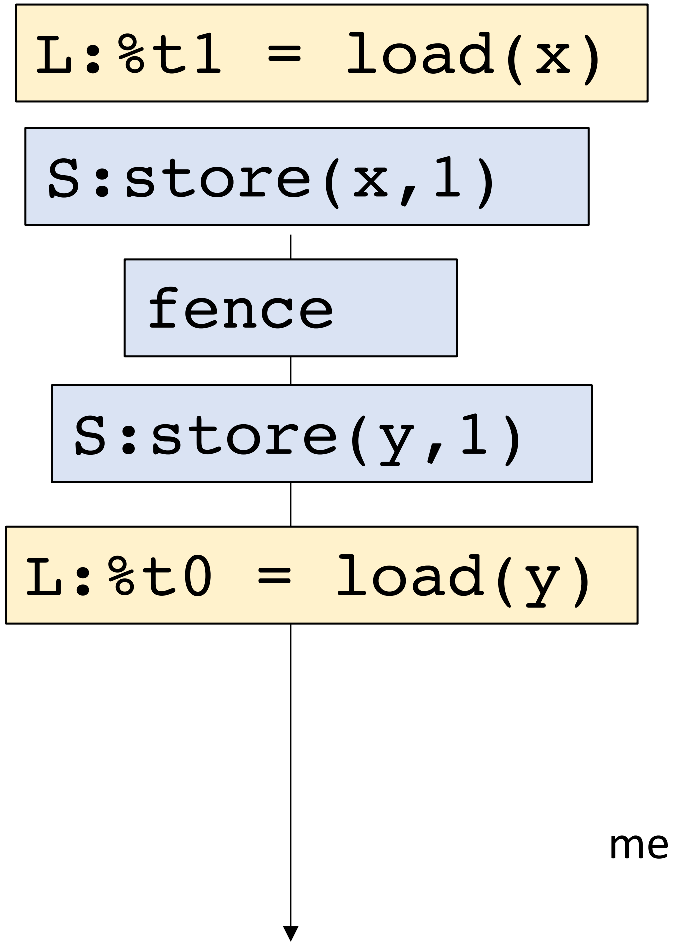
*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```



```
L:%t1 = load(x)
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
```

*Thread 1:*

```
L:%t0 = load(y)
L:%t1 = load(x)
```

memory access 0

| | | L | S |
|---|---|---|---|
| memory access 1 | L | YES | Different address |
| | S | Different address | Different address |

What about RMO? The loads can be reordered also!

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

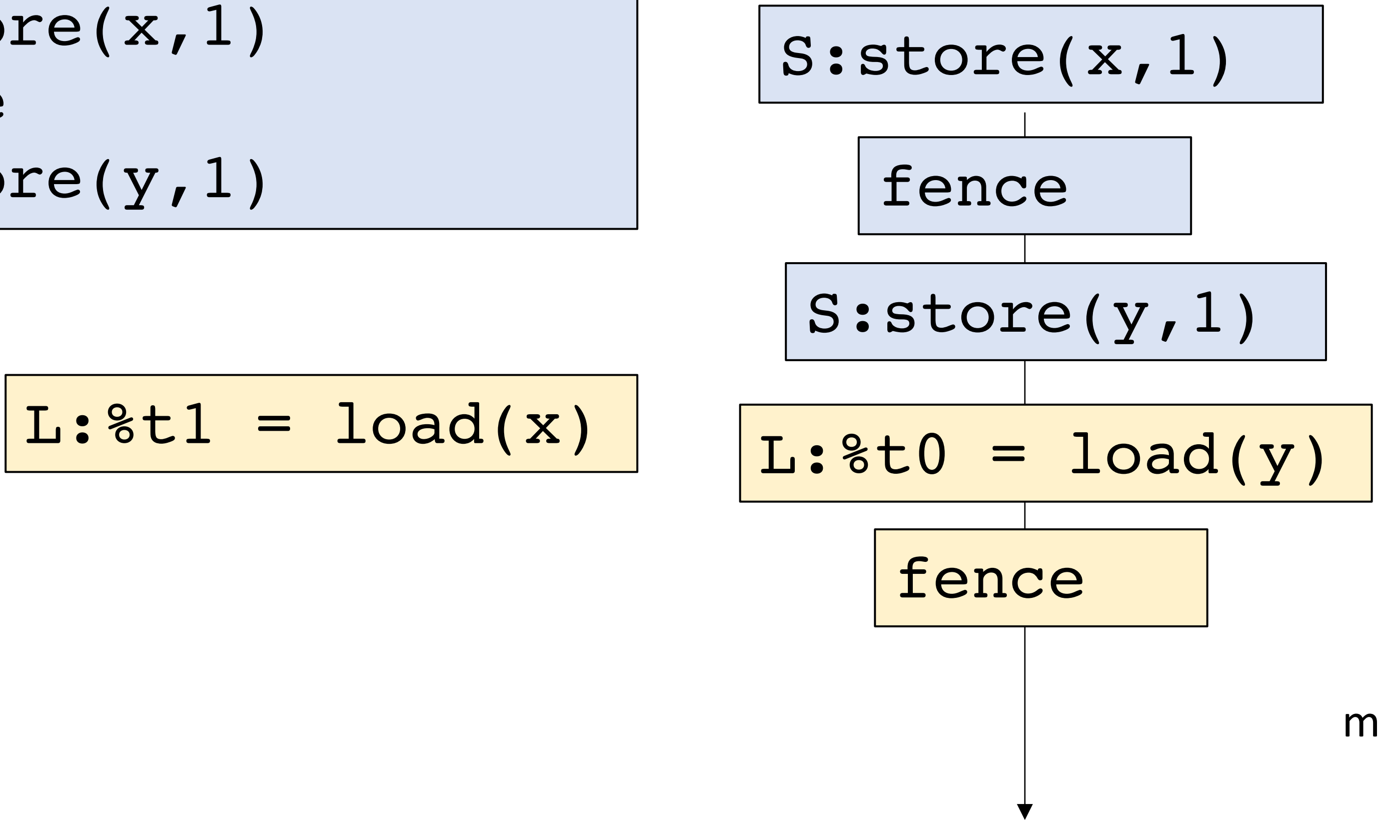
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
fence
L:%t1 = load(x)
```



```
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
fence
```

```
L:%t1 = load(x)
```

memory access 0

| | | L | S |
|---|---|---|---|
| memory access 1 | L | YES | Different address |
| | S | Different address | Different address |

What about RMO? add a fence

*Global variable:*

```
int x[1] = {0};
int y[1] = {0};
```

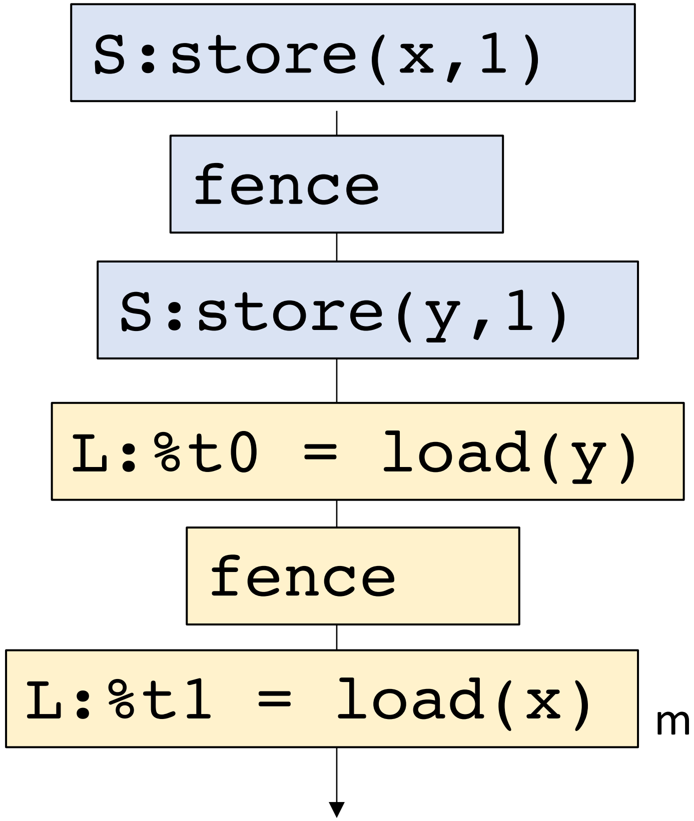
Question: can `t0 == 1 and t1 == 0?`

*Thread 0:*

```
S:store(x,1)
fence
S:store(y,1)
```

*Thread 1:*

```
L:%t0 = load(y)
fence
L:%t1 = load(x)
```



```
S:store(x,1)
fence
S:store(y,1)
L:%t0 = load(y)
fence
L:%t1 = load(x)
```

|  |  | memory access 0 |  |
|---|---|---|---|
|  |  | L | S |
| memory access 1 | L | YES | Different address |
|  | S | Different address | Different address |

Now the relaxed behavior is disallowed

# Compiling weak memory models

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

|   | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

target machine
TSO (x86)

|   | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

|   | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

find mismatch

target machine
TSO (x86)

|   | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

find mismatch

Two options:

make sure stores
are not reordered
with later loads

make sure loads
are not reordered
with earlier stores

target machine
TSO (x86)

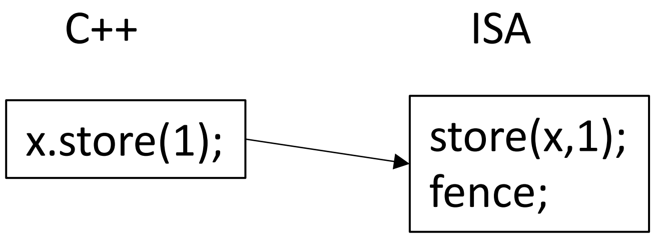
| | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

|   | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |


C++

ISA

```
x.store(1);
```

```
store(x,1);
fence;
```

or

target machine
TSO (x86)

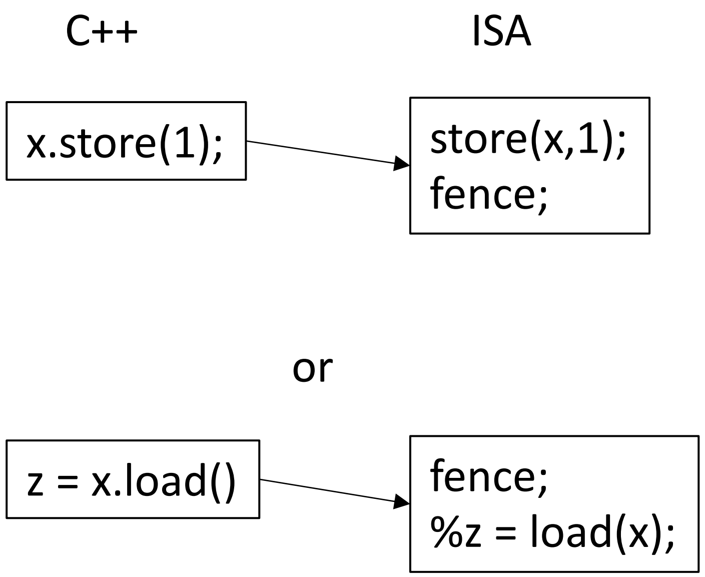
|   | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |



C++

ISA

```
x.store(1);
```

```
store(x,1);
fence;
```

or

```
z = x.load()
```

```
fence;
%z = load(x);
```

target machine
TSO (x86)

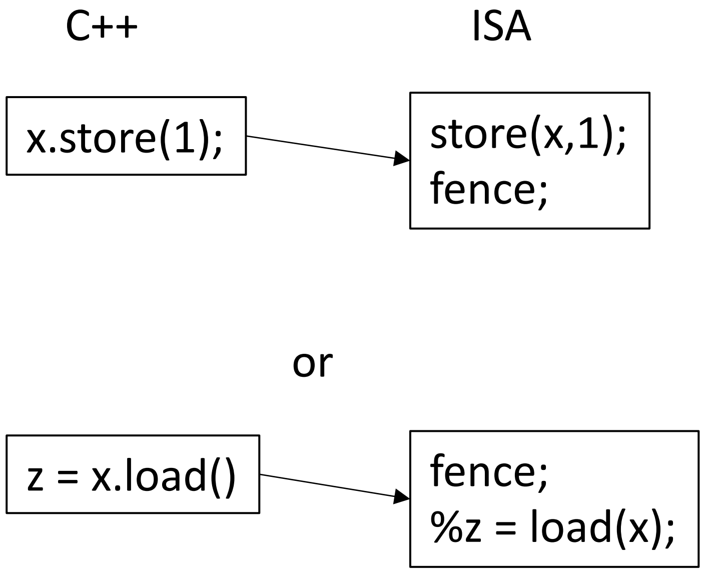
| | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

|   | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |



C++

ISA

```
x.store(1);
```

```
store(x,1);
fence;
```

or

```
z = x.load()
```

```
fence;
%z = load(x);
```

target machine
TSO (x86)

|   | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | No |

*This should help you see why you want to reduce the number of atomic load/stores in your program*

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

*How about this one?*

target machine
PSO

| | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | different address |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

| | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |

target machine
PSO

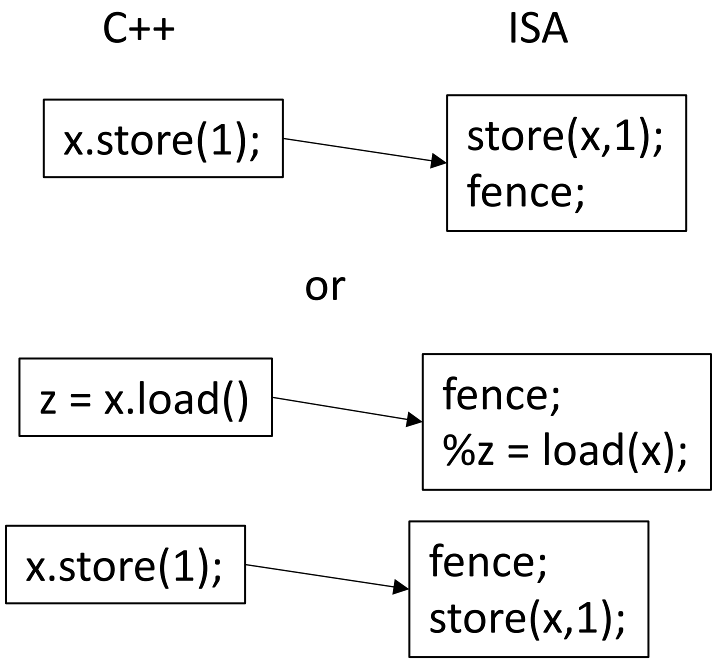
| | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | different address |

# C++11 atomic operation compilation

start with both both of the grids for the two different memory models

language
C++11 (sequential consistency)

|   | L | S |
|---|---|---|
| L | NO | NO |
| S | NO | NO |



C++ → ISA

```
x.store(1);   →   store(x,1);
                  fence;
```

or

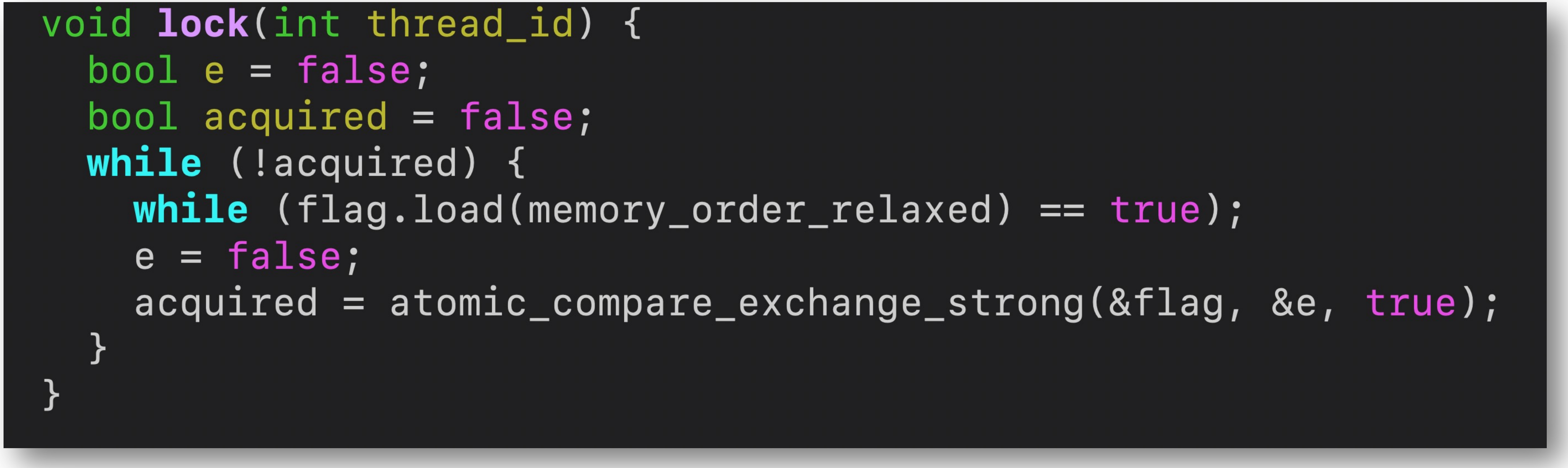
```
z = x.load()  →   fence;
                  %z = load(x);

x.store(1);   →   fence;
                  store(x,1);
```

target machine
PSO

|   | L | S |
|---|---|---|
| L | NO | different address |
| S | NO | different address |

# A few comments on homework

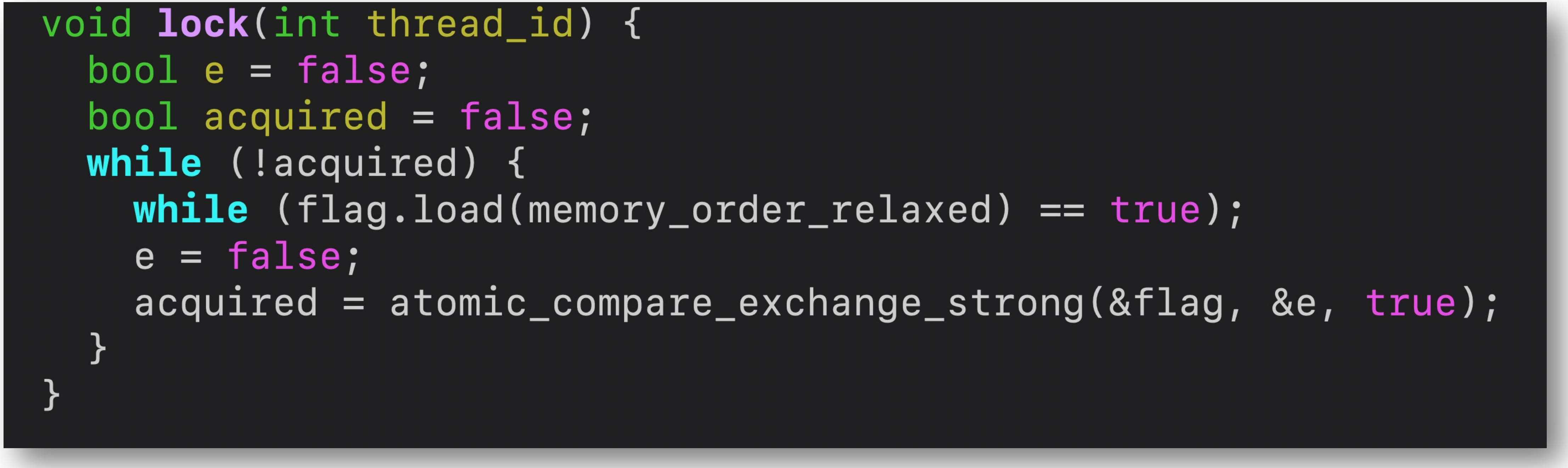
```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense.store(thread_sense[tid]);
        }
        else {
          while (sense.load() != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```



```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense.store(thread_sense[tid]);
        }
        else {
          while (sense.load(memory_order_relaxed) != thread_sense[tid]);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

*what else*
*is needed?*



```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Given a global barrier B
and a global memory location x where
initially *x = 0;

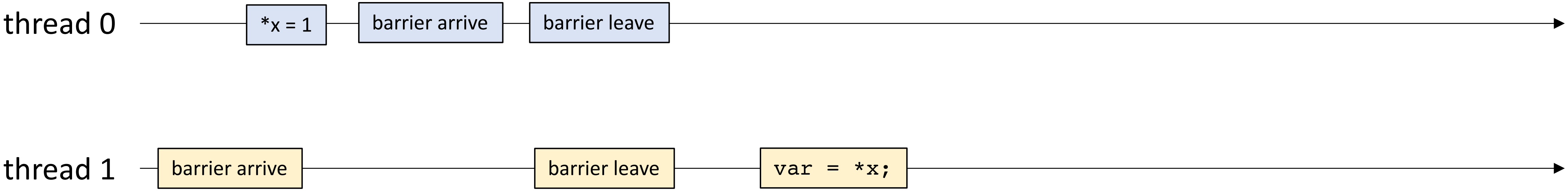
*Thread 0:*

```
*x = 1;
B.barrier();
```

*Thread 1:*

```
B.barrier();
var = *x;
```

*What happens if the store buffer isn't flushed when the barrier leaves?*



thread 0

*x = 1

barrier arrive

barrier leave

thread 1

barrier arrive

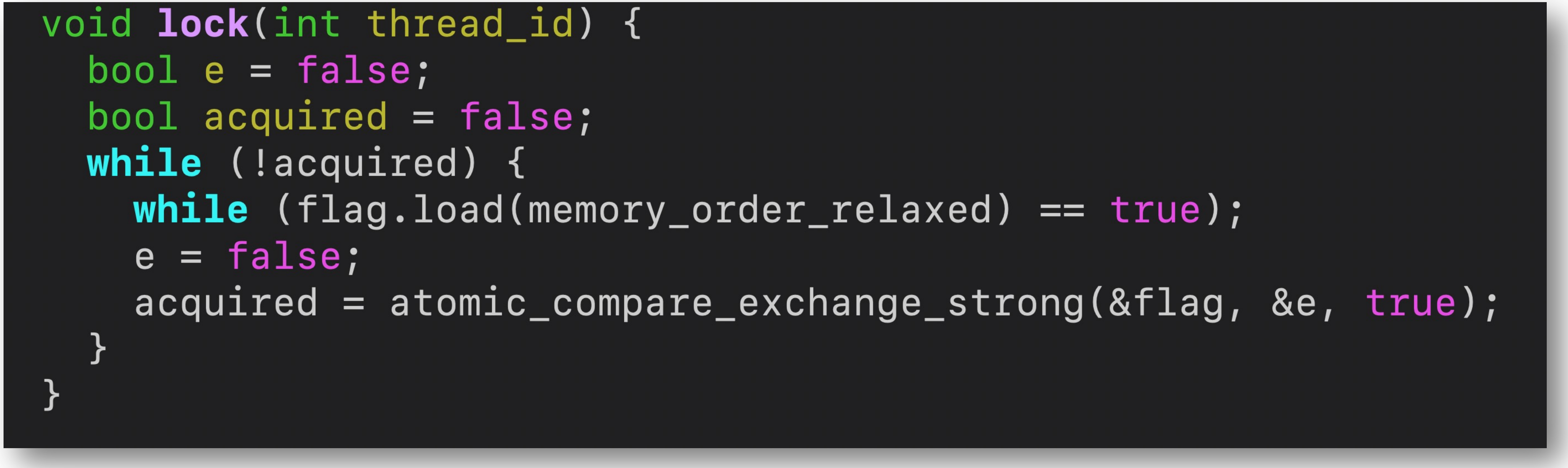
barrier leave

`var = *x;`

what is this allowed to return?

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense.store(thread_sense[tid]);
        }
        else {
          while (sense.load(memory_order_relaxed) != thread_sense[tid]);
          sense.load();
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

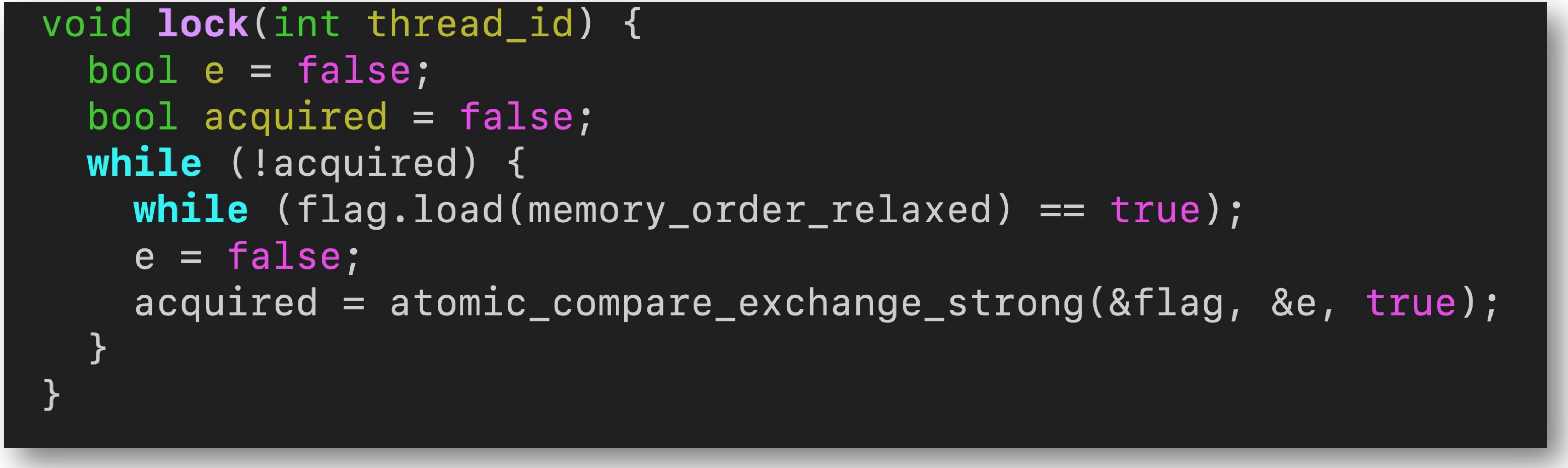
*what else is needed?*



```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

```
void barrier(int tid) {
        int arrival_num = atomic_fetch_add(&counter, 1);
        if (arrival_num == num_threads-1) {
           counter.store(0);
           sense.store(thread_sense[tid]);
        }
        else {
          while (sense.load(memory_order_relaxed) != thread_sense[tid]);
          atomic_thread_fence(memory_order_seq_cst);
        }
        thread_sense[tid] = !thread_sense[tid];
     }
```

*what else is needed?*



```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true);
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

# On to new material

# Schedule

- Finishing up relaxed memory models:
  - A cautionary tale
  - porting between memory models

- Reasoning about schedulers
  - labeled transition systems

# Schedule

- Finishing up relaxed memory models:
  - **A cautionary tale**
  - porting between memory models

- Reasoning about schedulers
  - labeled transition systems

# A cautionary tale

*Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)*

*Thread 0:*

```
m.lock();
display.enq(triangle0);
m.unlock();
```

*Thread 1:*

```
m.lock();
display.enq(triangle1);
m.unlock();
```

*Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)*

*Thread 0:*

```
m.lock();
display.enq(triangle0);
m.unlock();
```

*Thread 1:*

```
m.lock();
display.enq(triangle1);
m.unlock();
```

We know how lock and unlock are implemented

*Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)*

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
display.enq(triangle0);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
display.enq(triangle1);
store(mutex,0);
```

We know how lock and unlock are implemented
We also know how a queue is implemented

*Consider the following example: a graphics program where each thread wants to display a triangle; the display is a queue (not thread safe)*

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

We know how lock and unlock are implemented
We also know how a queue is implemented

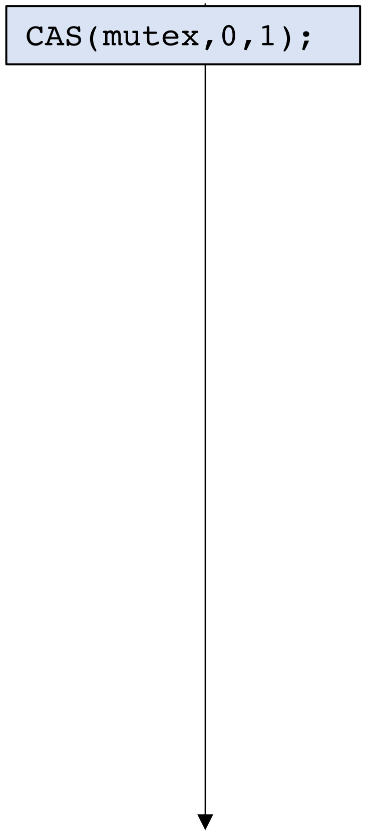
What is an execution?

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```



```
CAS(mutex,0,1);
```

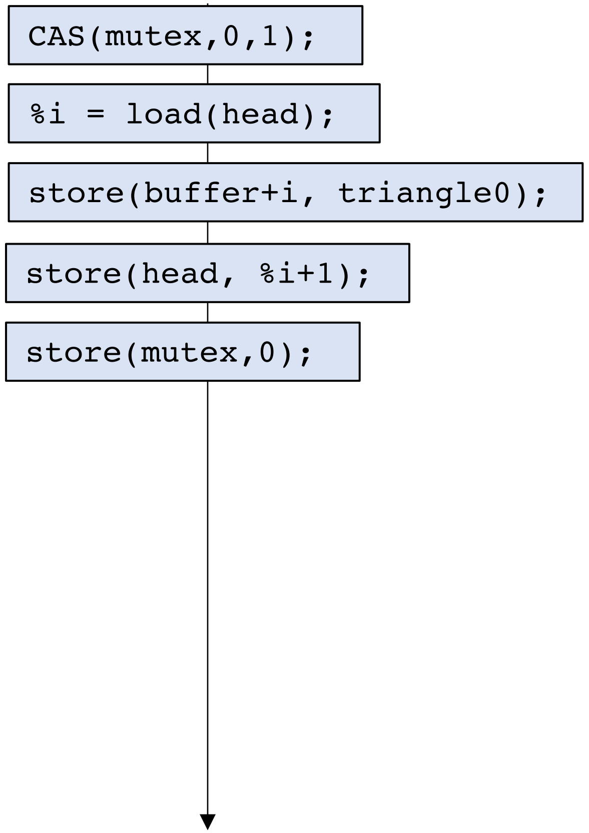
*if blue goes first*
*it gets to complete*
*its critical section*
*while thread 1 is spinning*

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```



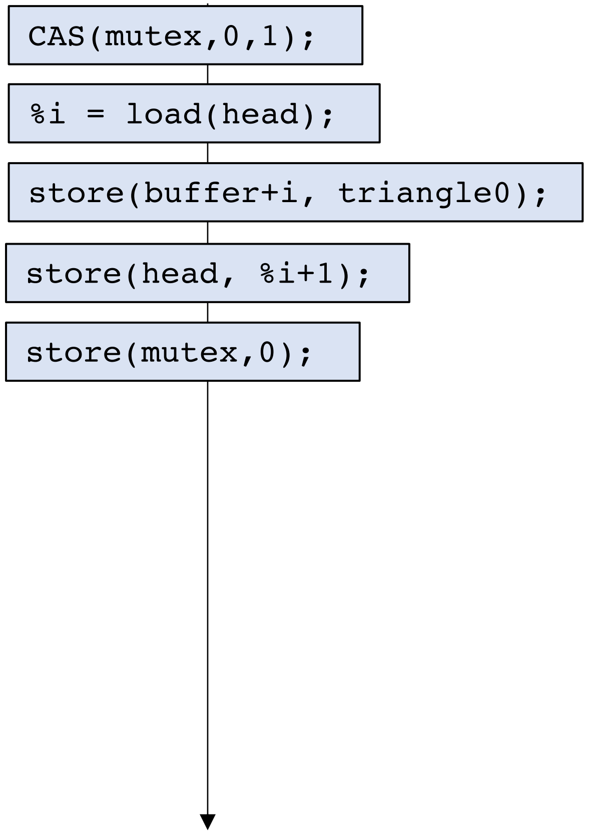
```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

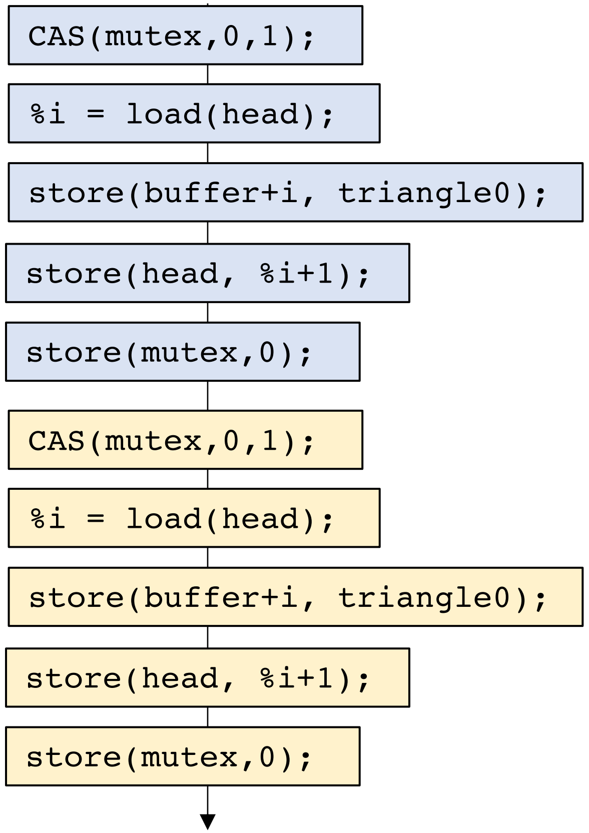
*now yellow gets a change to go*

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*now yellow gets a change to go*

*Thread 0:*

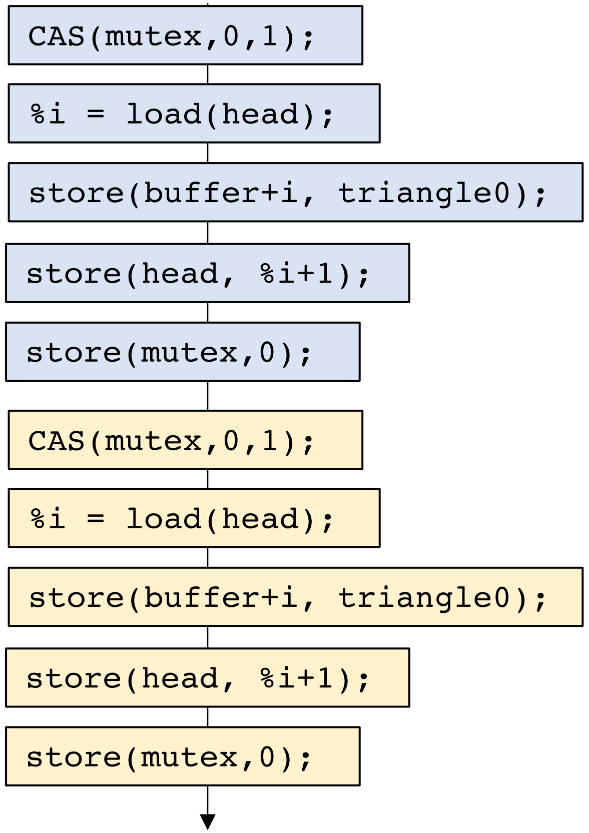
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

*what can happen in a PSO memory model?*

|   | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 0:*

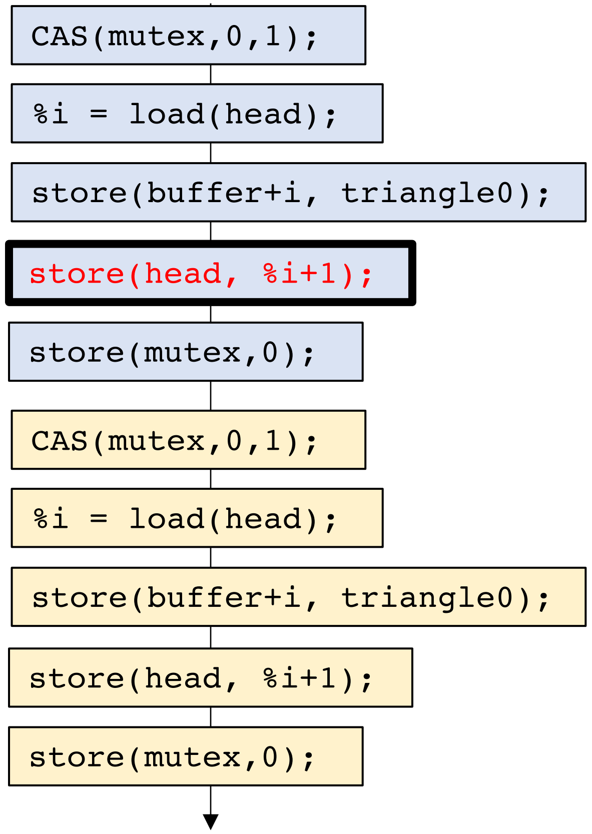
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

*what can happen in a PSO memory model?*

|   | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 0:*

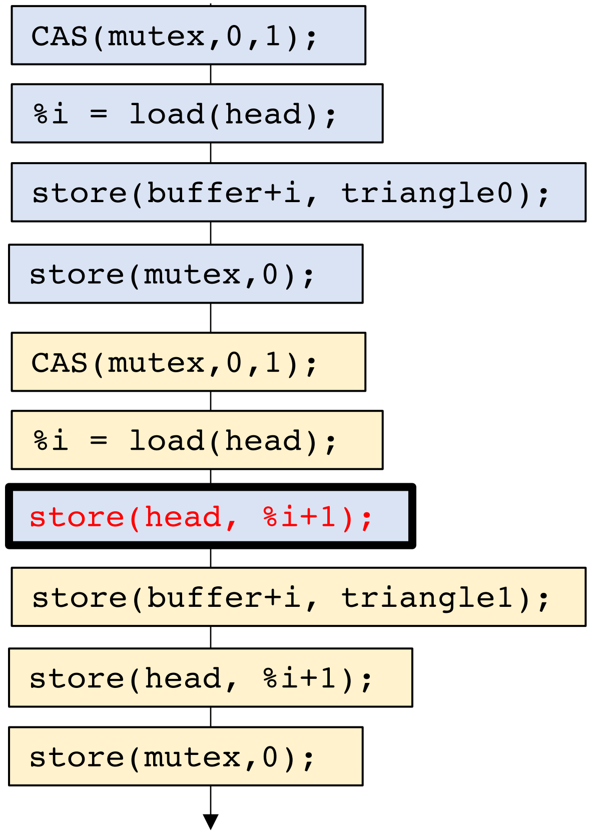
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

*what can happen in a PSO memory model?*

|   | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



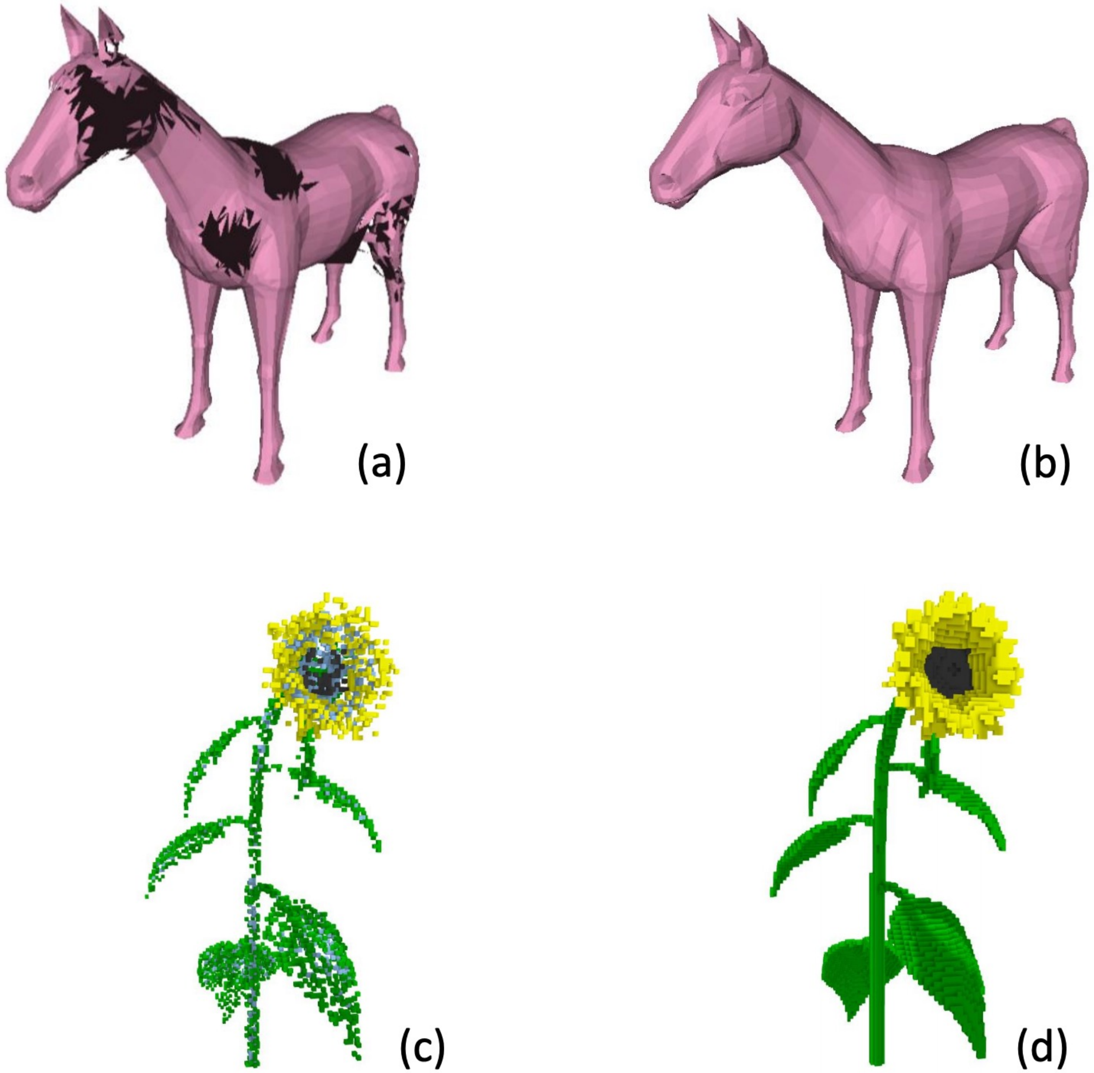
```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(head, %i+1);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

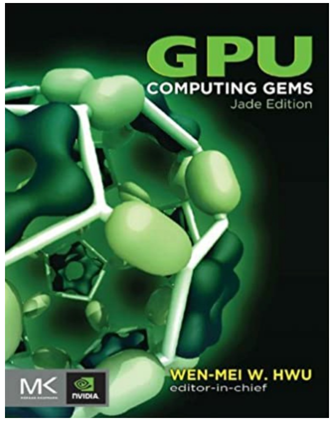
What just happened if this store moves?

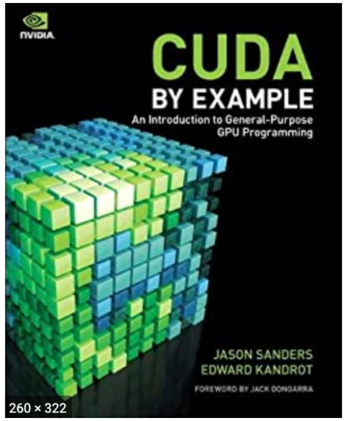
# Nvidia in 2015

- Nvidia architects implemented a weak memory model
- Nvidia programmers expected a strong memory model
- Mutexes implemented without fences!

# Nvidia in 2015







bug found in two
Nvidia textbooks

We implemented
a side-channel attack
that made the bugs
appear more frequently

These days Nvidia has
a very well-specified
memory model!

*Thread 0:*

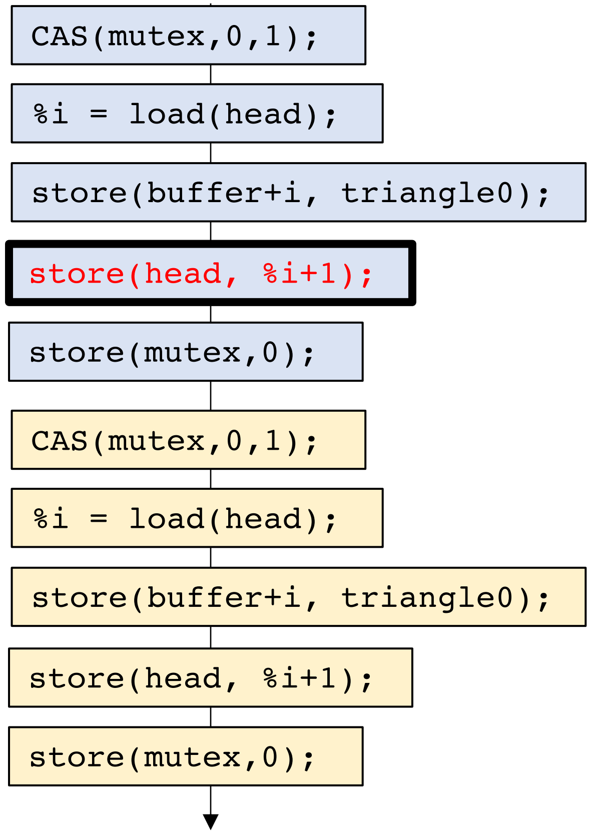
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
store(mutex,0);
```

*what can happen in a PSO memory model?*

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

How to fix the issue?

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
fence;
store(mutex,0);
```

*unlock contains fence before store!*

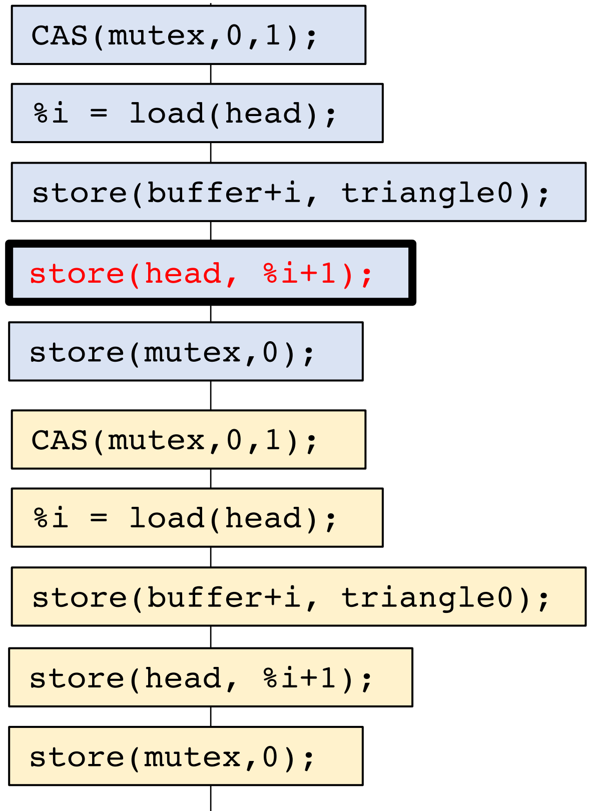
*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
fence;
store(mutex,0);
```

*unlock contains fence before store!*

*what can happen in a PSO memory model?*

|   | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

How to fix the issue?

your unlock function should contain a fence!

*Thread 0:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
fence;
store(mutex,0);
```

*unlock contains fence before store!*

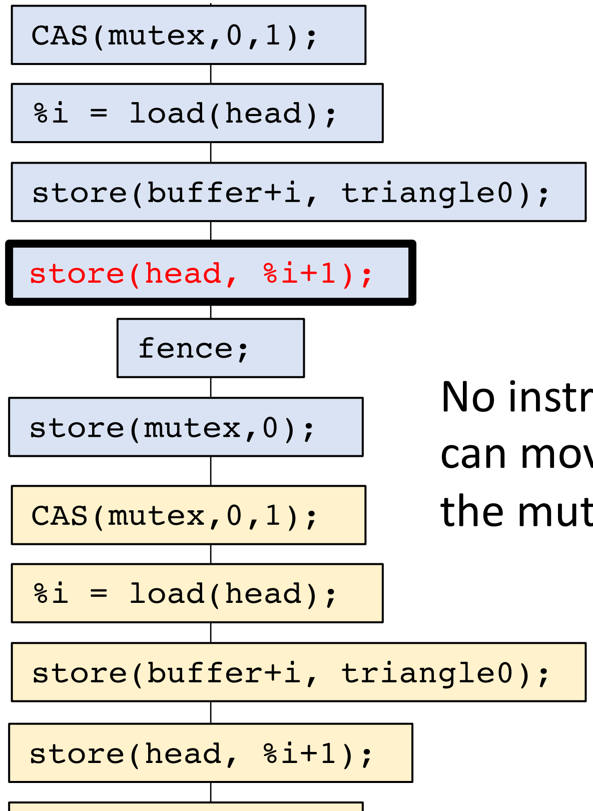
*Thread 1:*

```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle1);
store(head, %i+1);
fence;
store(mutex,0);
```

*unlock contains fence before store!*

*what can happen in a PSO memory model?*

|   | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |



```
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
fence;
store(mutex,0);
CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
```

No instructions can move after the mutex store!

How to fix the issue?

your unlock function should contain a fence!

# Schedule

- Finishing up relaxed memory models:
  - A cautionary tale
  - **porting between memory models**

- Reasoning about schedulers
  - labeled transition systems

# Memory Model Strength

- If one memory model M0 allows more relaxed behaviors than another memory model M1, then M0 is more *relaxed* (or *weaker*) than M1.

- It is safe to run a program written for M0 on M1. But not vice versa

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | NO |

TSO

| | L | S |
|---|---|---|
| L | NO | Different address |
| S | NO | Different address |

PSO

| | L | S |
|---|---|---|
| L | YES | Different address |
| S | Different address | Different address |

RMO

# Schedule

- Finishing up relaxed memory models:
    - A cautionary tale
    - porting between memory models

- **Reasoning about schedulers**
    - **labeled transition systems**

# How do we think about concurrency?

- When you write a concurrent program, how do you think about what can happen?

# How do we think about concurrency?

- When you write a concurrent program, how do you think about what can happen?

- Interleavings?

# How do we think about concurrency?

- When you write a concurrent program, how do you think about what can happen?

- Interleavings?

- RMWs?

# How do we think about concurrency?

- When you write a concurrent program, how do you think about what can happen?

- Interleavings?

- RMWs?

- Thread Sanitizer?

# How do we think about concurrency?

- When you write a concurrent program, how do you think about what can happen?
- Interleavings?
- RMWs?
- Thread Sanitizer?
- Run the program and pray to the gods of concurrency?

## Think about two threads accessing the bank account

getting paid

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

buying coffee

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```

*assuming sequential consistency*

global timeline

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```

step 1 pick a thread

*assuming sequential consistency*

global timeline

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```

step 1 pick a thread

*assuming sequential consistency*

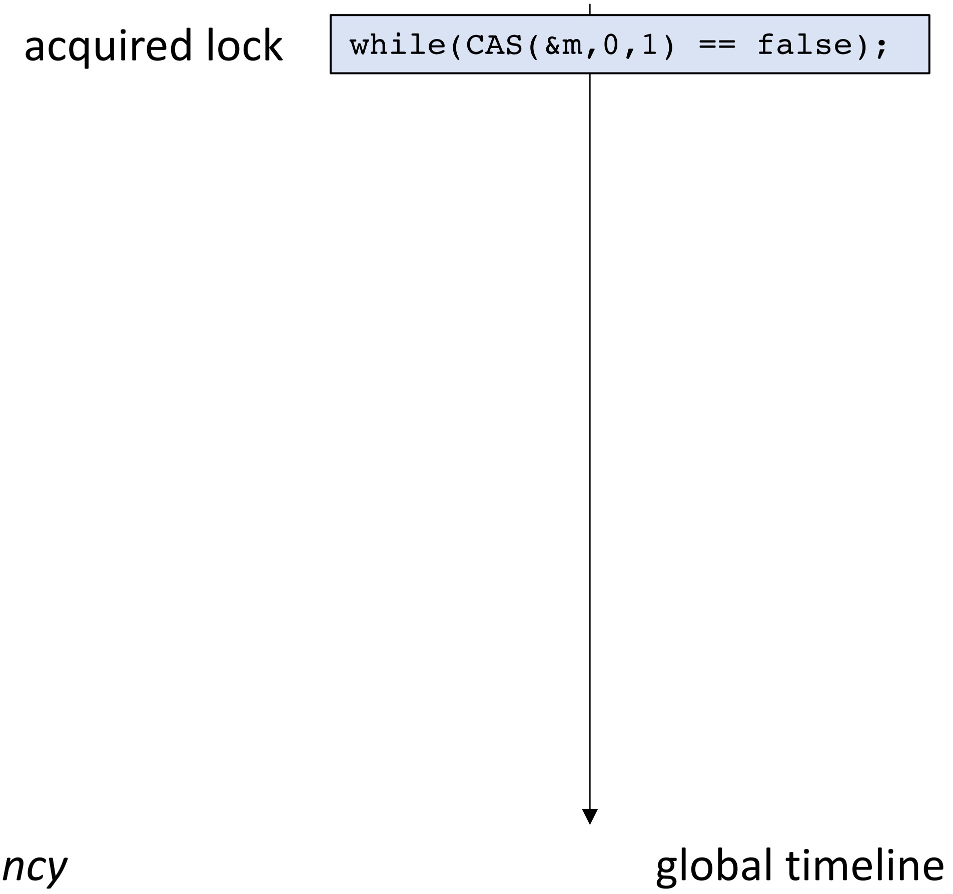
global timeline

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

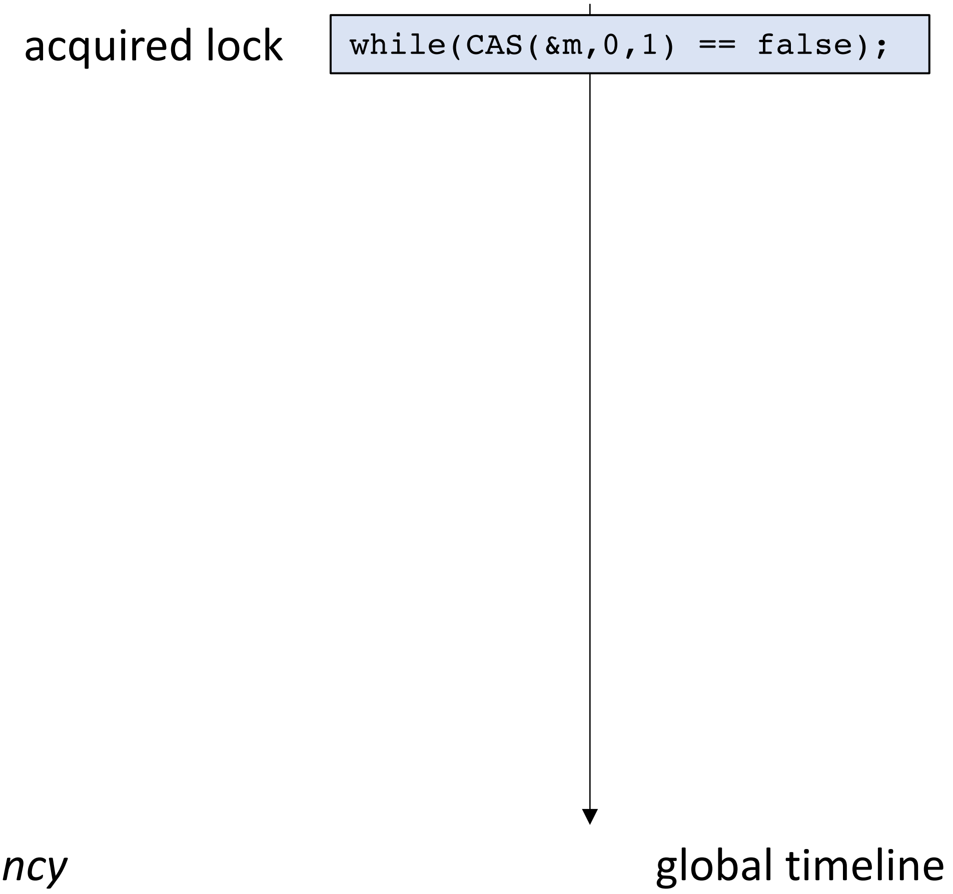
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

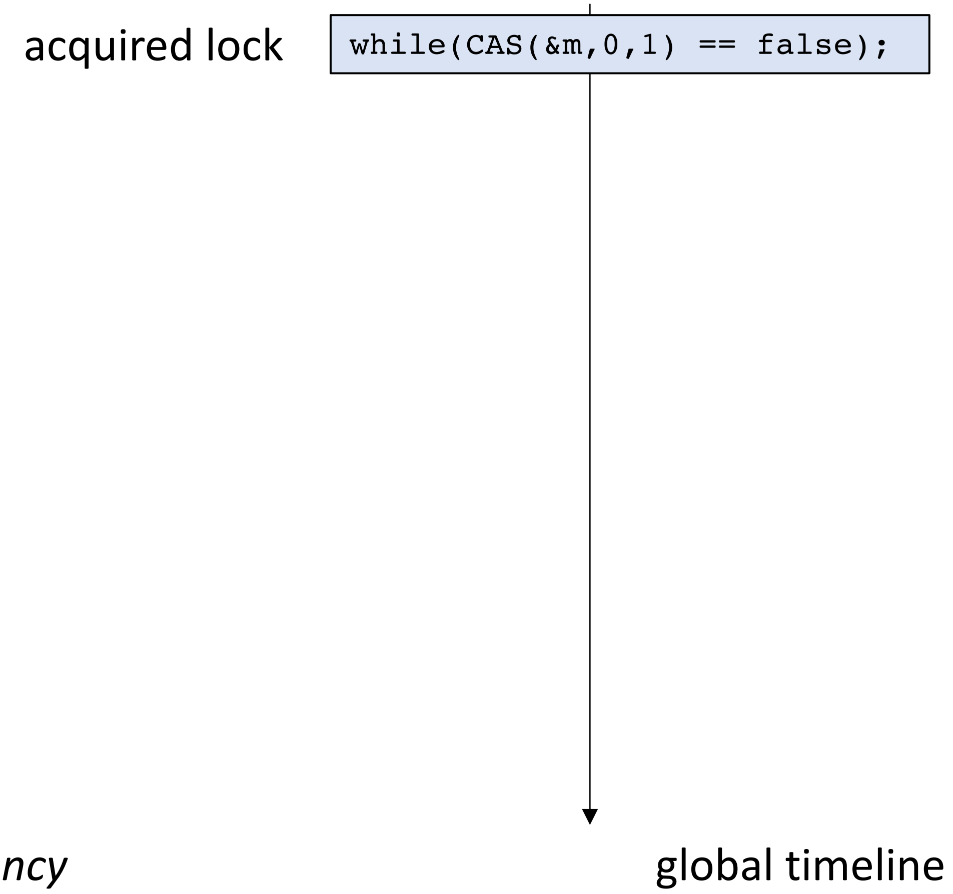
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



```
while(CAS(&m,0,1) == false);
```

step 1 pick a thread

Keep track of next instruction to execute

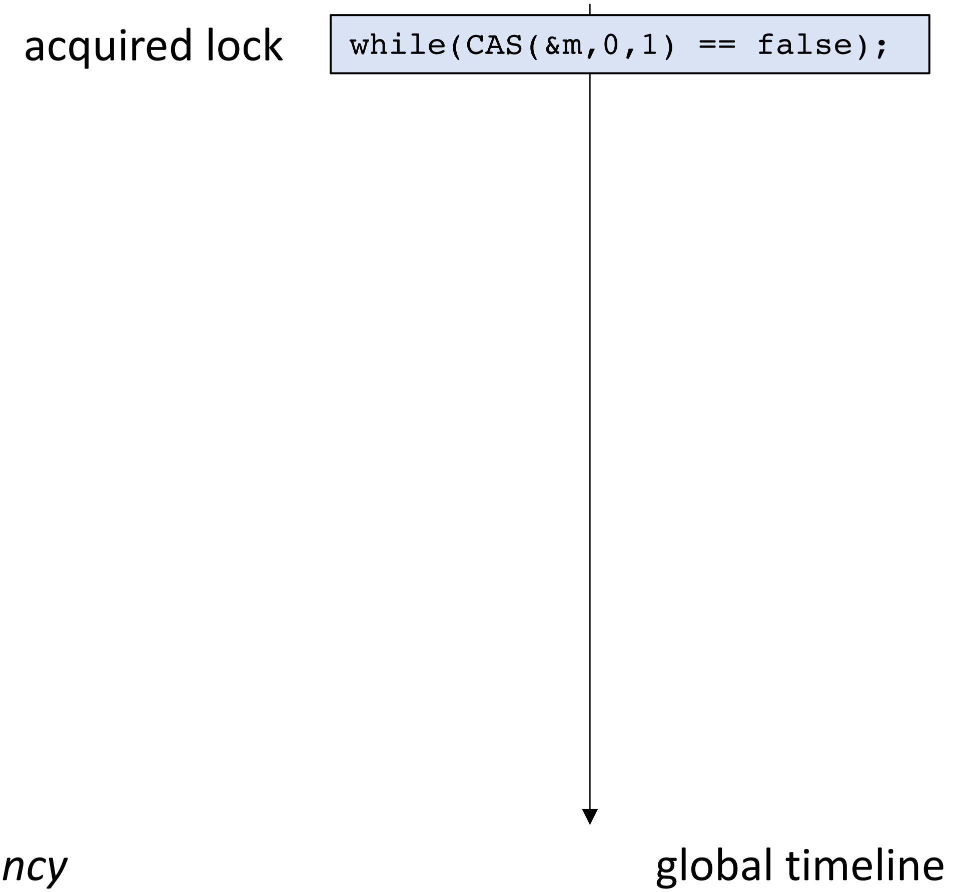
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

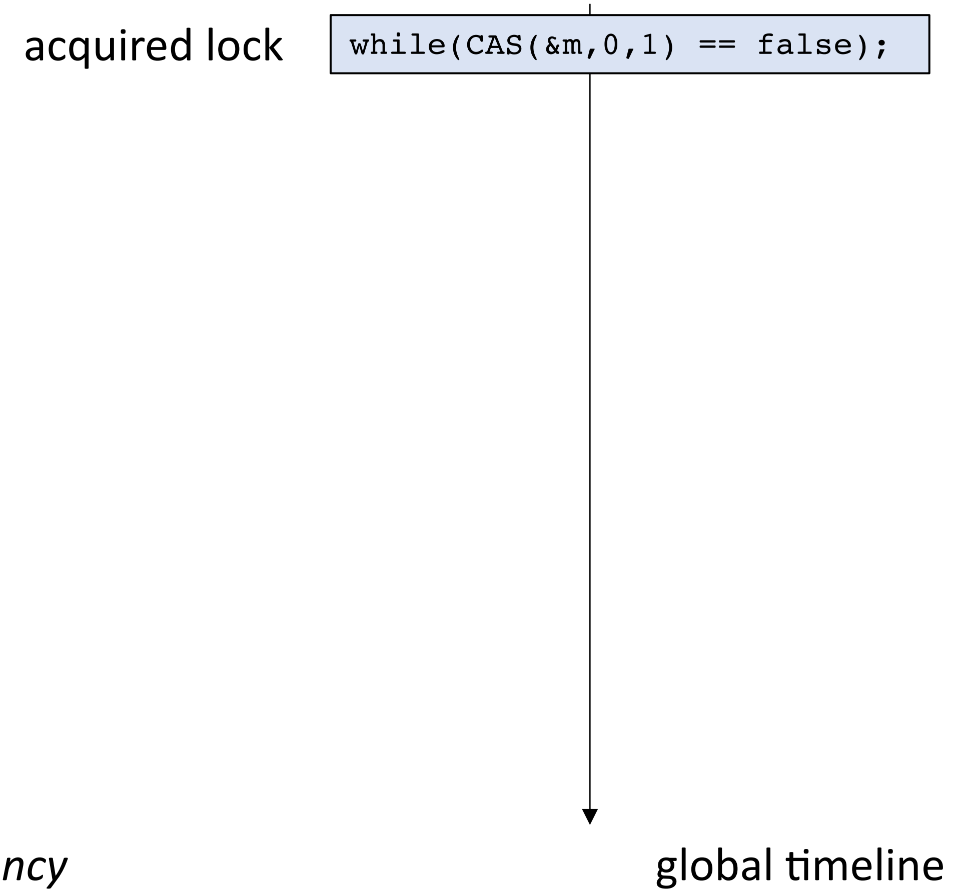
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

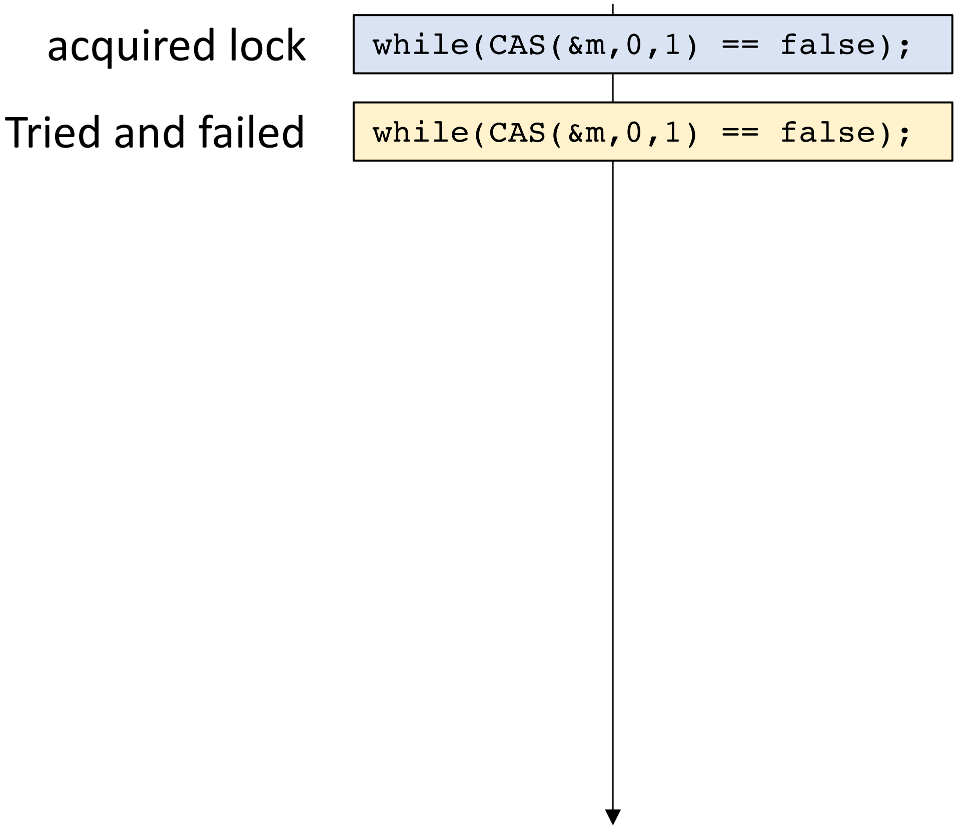
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

*assuming sequential consistency*

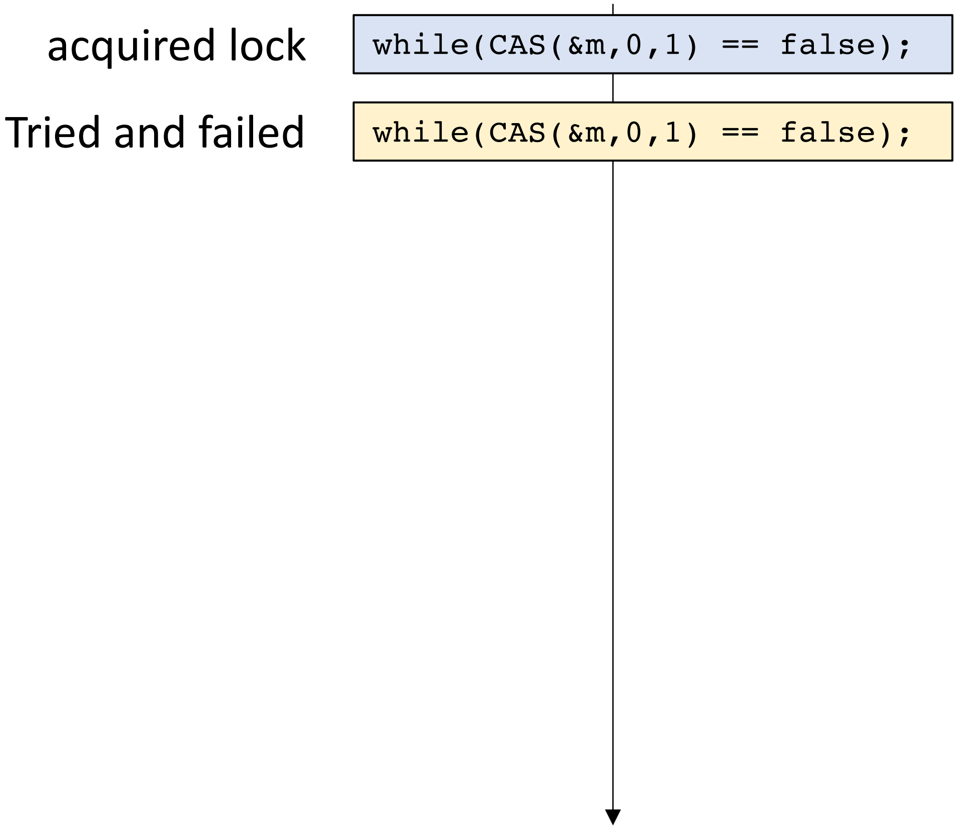
global timeline

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

Keep track of next instruction to execute

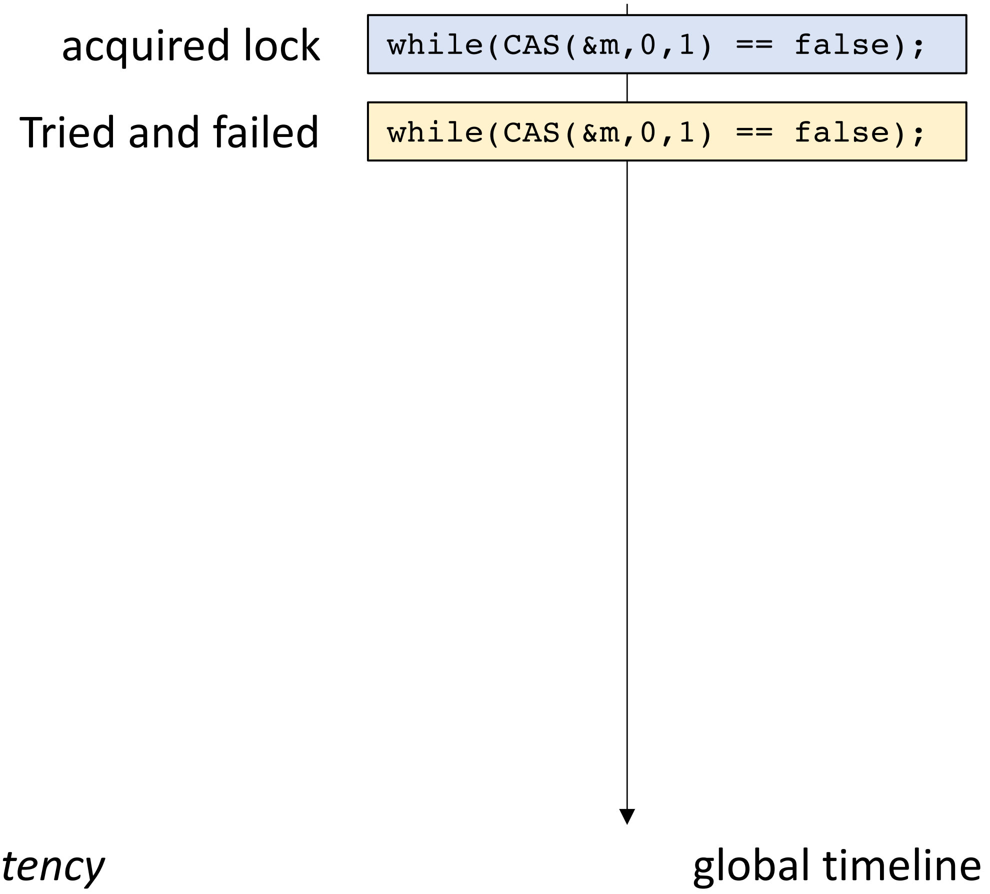
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

Keep track of next instruction to execute

which thread to pick next?

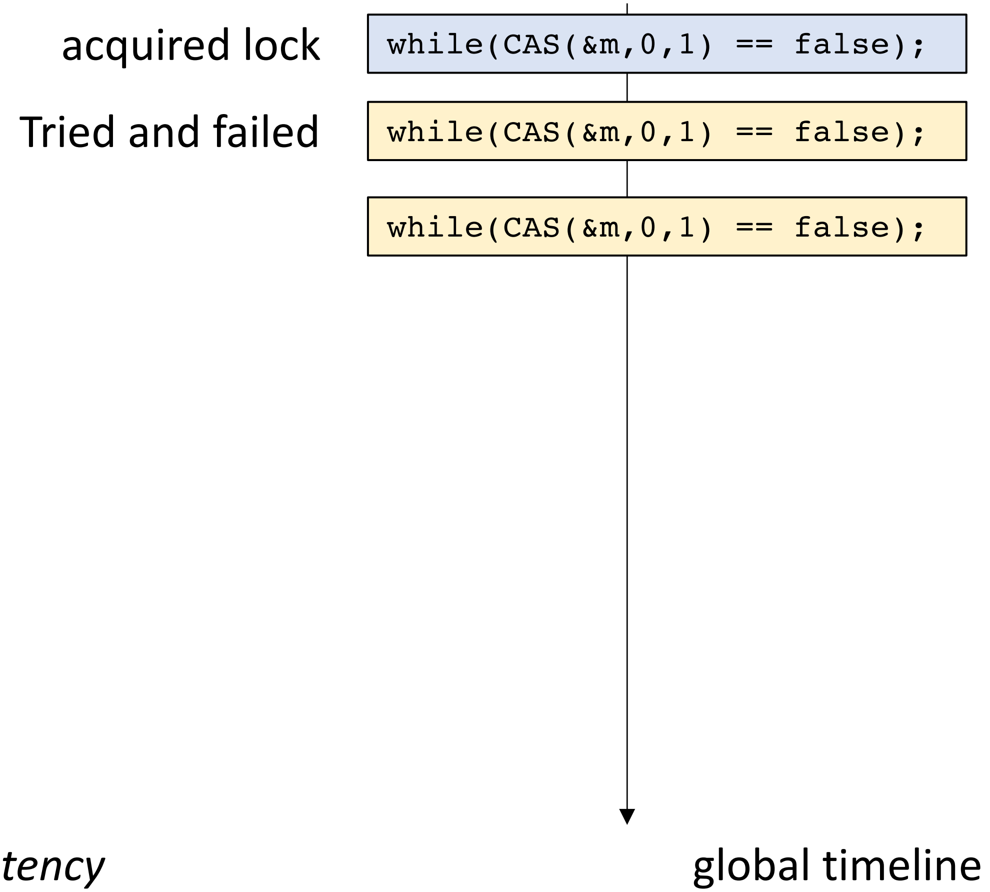
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

Keep track of next instruction to execute

which thread to pick next?

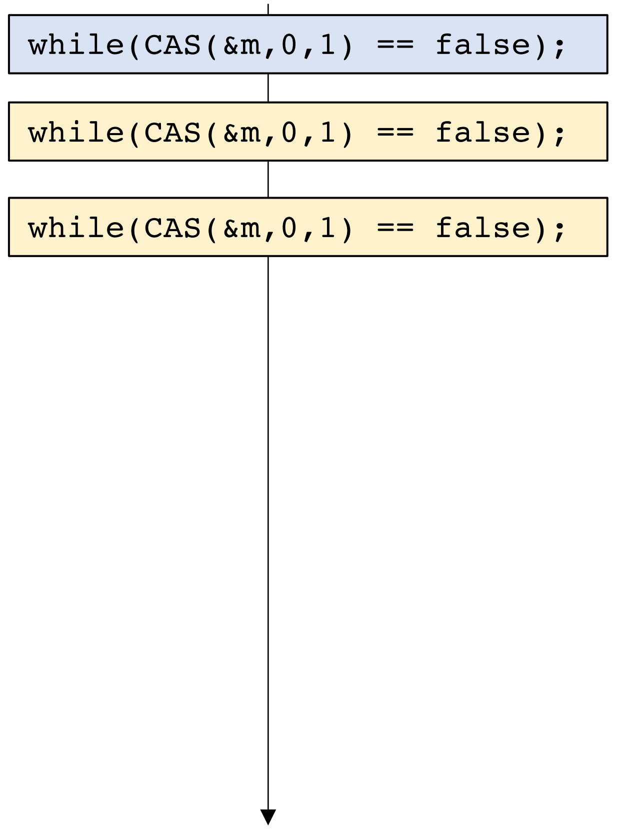
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

```
while(CAS(&m,0,1) == false);
```

global timeline

step 1 pick a thread

Keep track of next instruction to execute

pick the next thread to execute

Keep track of next instruction to execute

What happens if we keep picking thread 1?

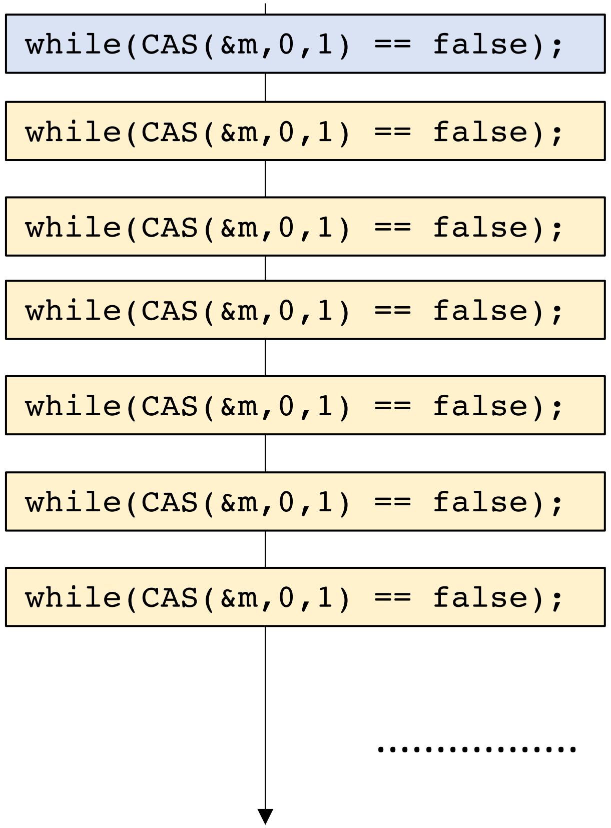
*assuming sequential consistency*

*Thread 0:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp++;
*bank_account = tmp;
m.store(0); //unlock
```

*Thread 1:*

```
//lock
while(CAS(&m,0,1) == false);
int tmp = *bank_account;
tmp--;
*bank_account = tmp;
m.store(0); //unlock
```



acquired lock

```
while(CAS(&m,0,1) == false);
```

Tried and failed

```
while(CAS(&m,0,1) == false);
```

Thread 1 waiting for Thread 0 to release the lock

```
while(CAS(&m,0,1) == false);
while(CAS(&m,0,1) == false);
while(CAS(&m,0,1) == false);
while(CAS(&m,0,1) == false);
while(CAS(&m,0,1) == false);
```

global timeline

Can this keep going forever?

Is this program guaranteed to terminate?

Why? Why not?

*assuming sequential consistency*

# A new way to represent concurrent executions

- Global timeline fails to capture the full picture

- Introducing Labelled Transition System (LTS)
  - Concurrent execution in a graph form.

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

Lets only think about the locks and unlocks
assume any critical section

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

program location

Lets only think about the locks and unlocks
assume any critical section

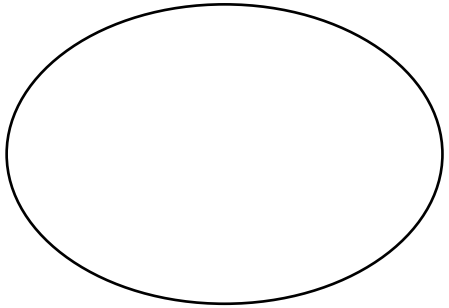
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

Start making our graph, with a starting node:



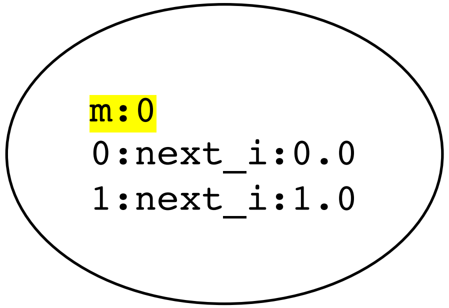
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

Start making our graph, with a starting node:



```
m:0
0:next_i:0.0
1:next_i:1.0
```

global variable values

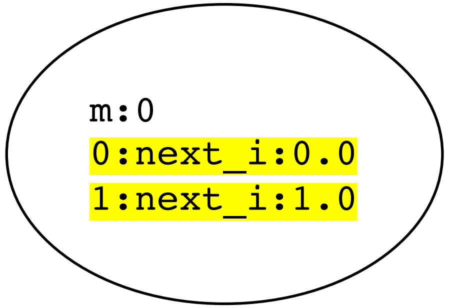
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

Start making our graph, with a starting node:



```
m:0
0:next_i:0.0
1:next_i:1.0
```

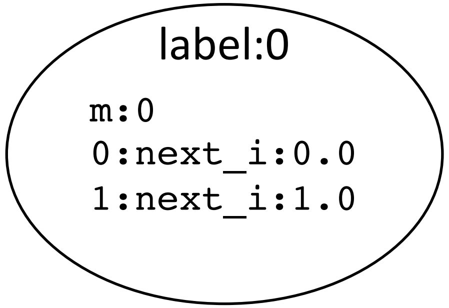
global variable values
next instructions to execute

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0

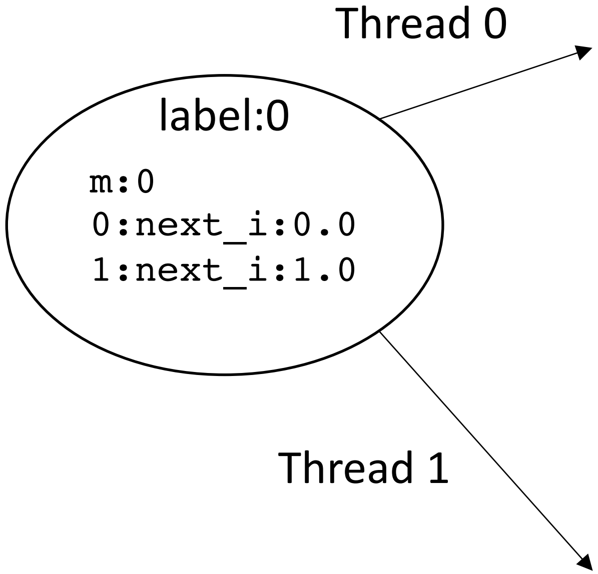
```
m:0
0:next_i:0.0
1:next_i:1.0
```

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



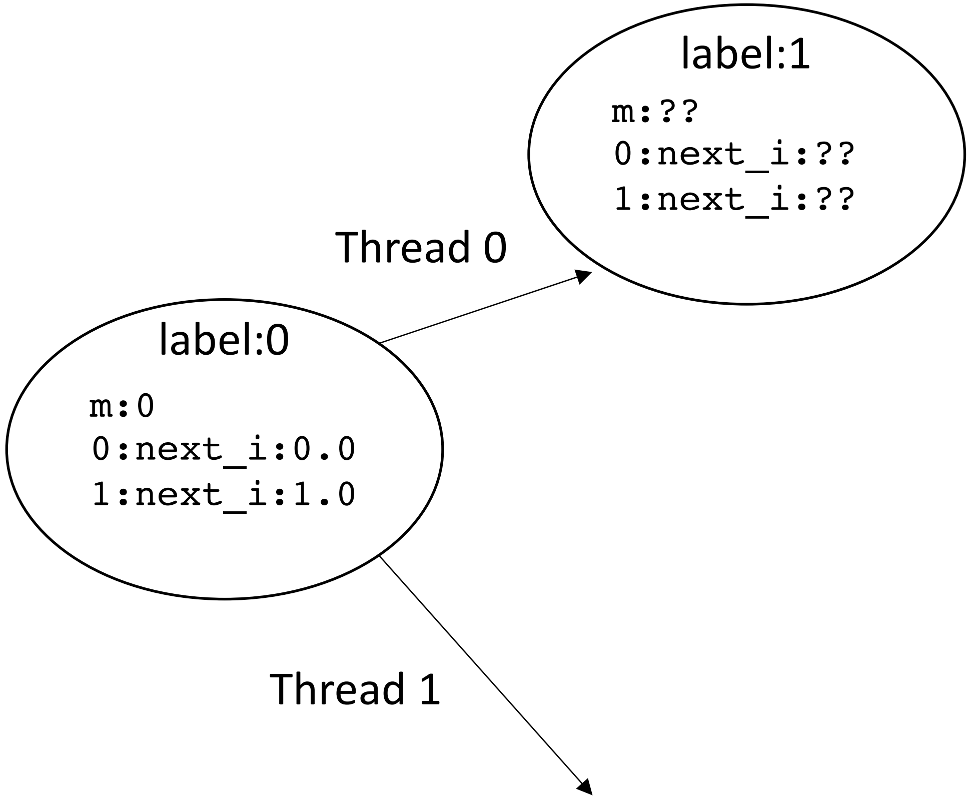
two choices:
thread 0 executes, or thread 1 executes

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

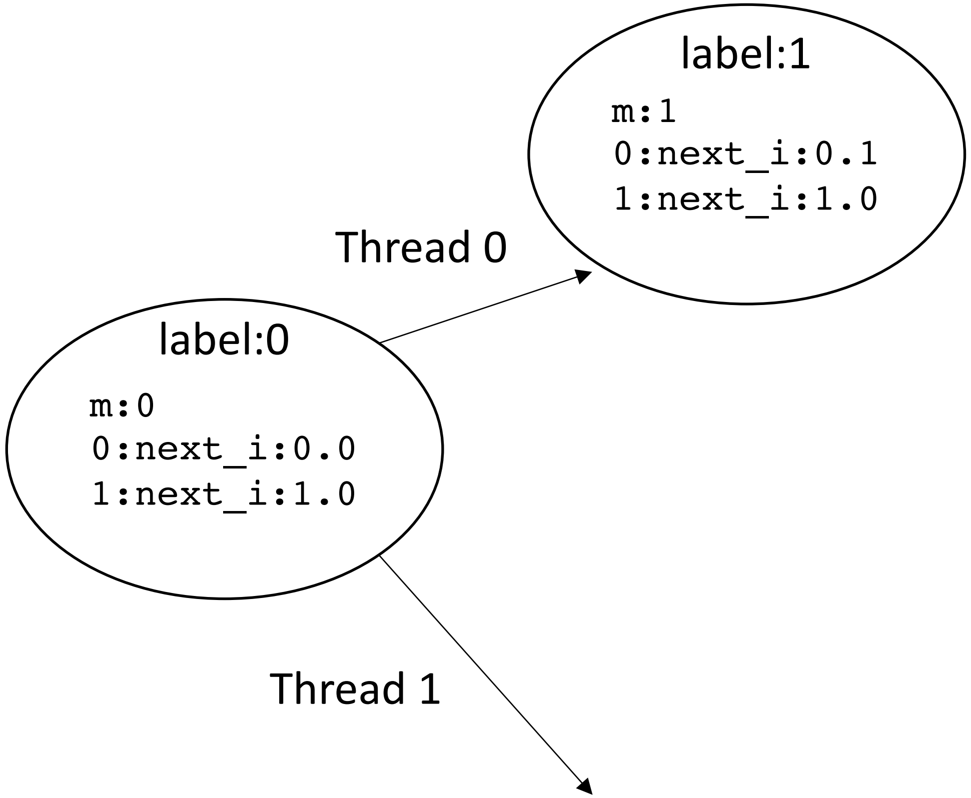


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0

```
m:0
0:next_i:0.0
1:next_i:1.0
```

Thread 0

label:1

```
m:1
0:next_i:0.1
1:next_i:1.0
```

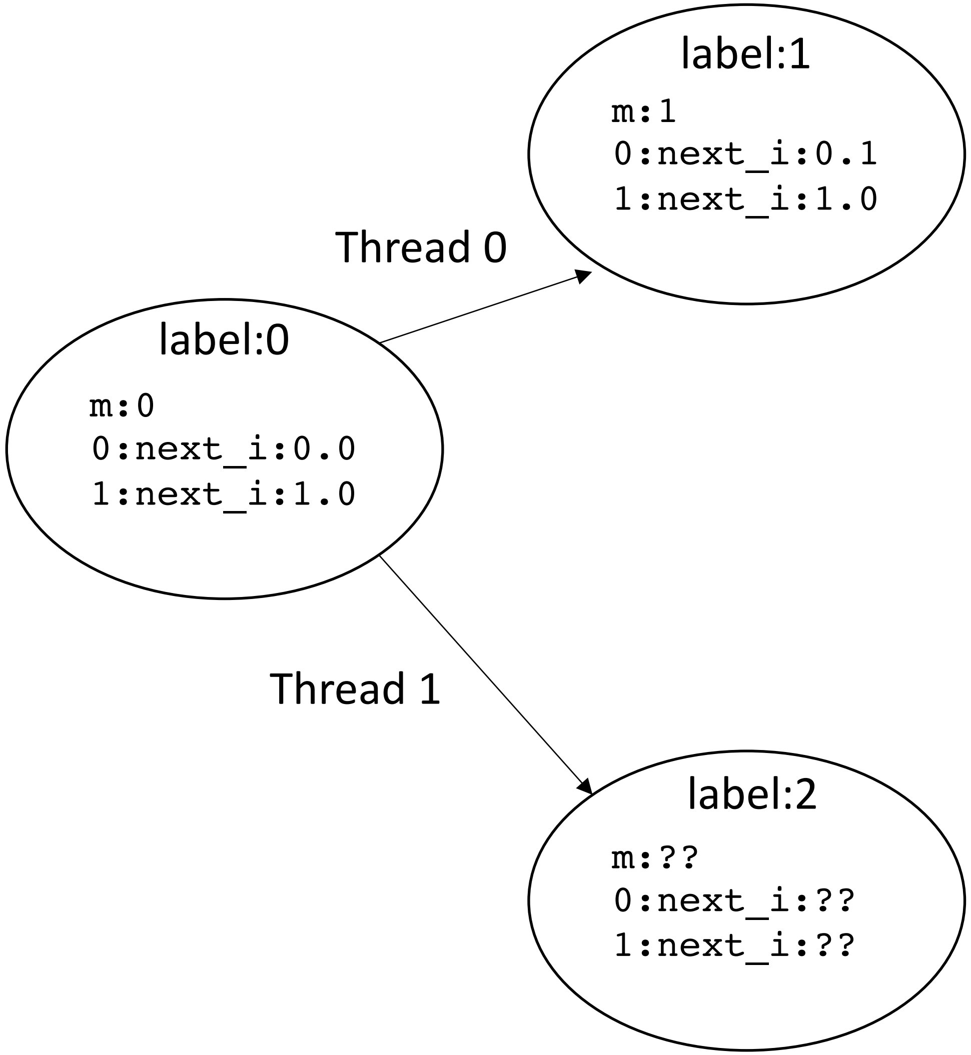
Thread 1

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

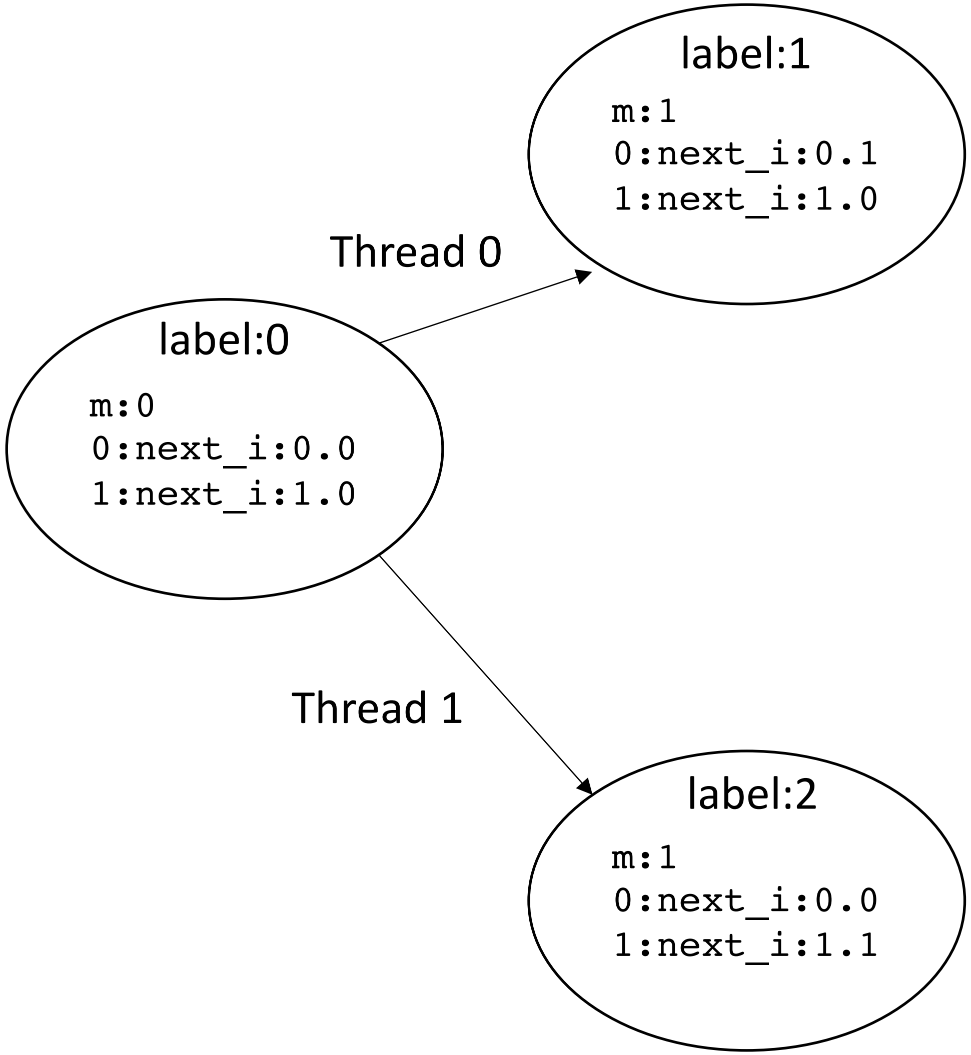


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

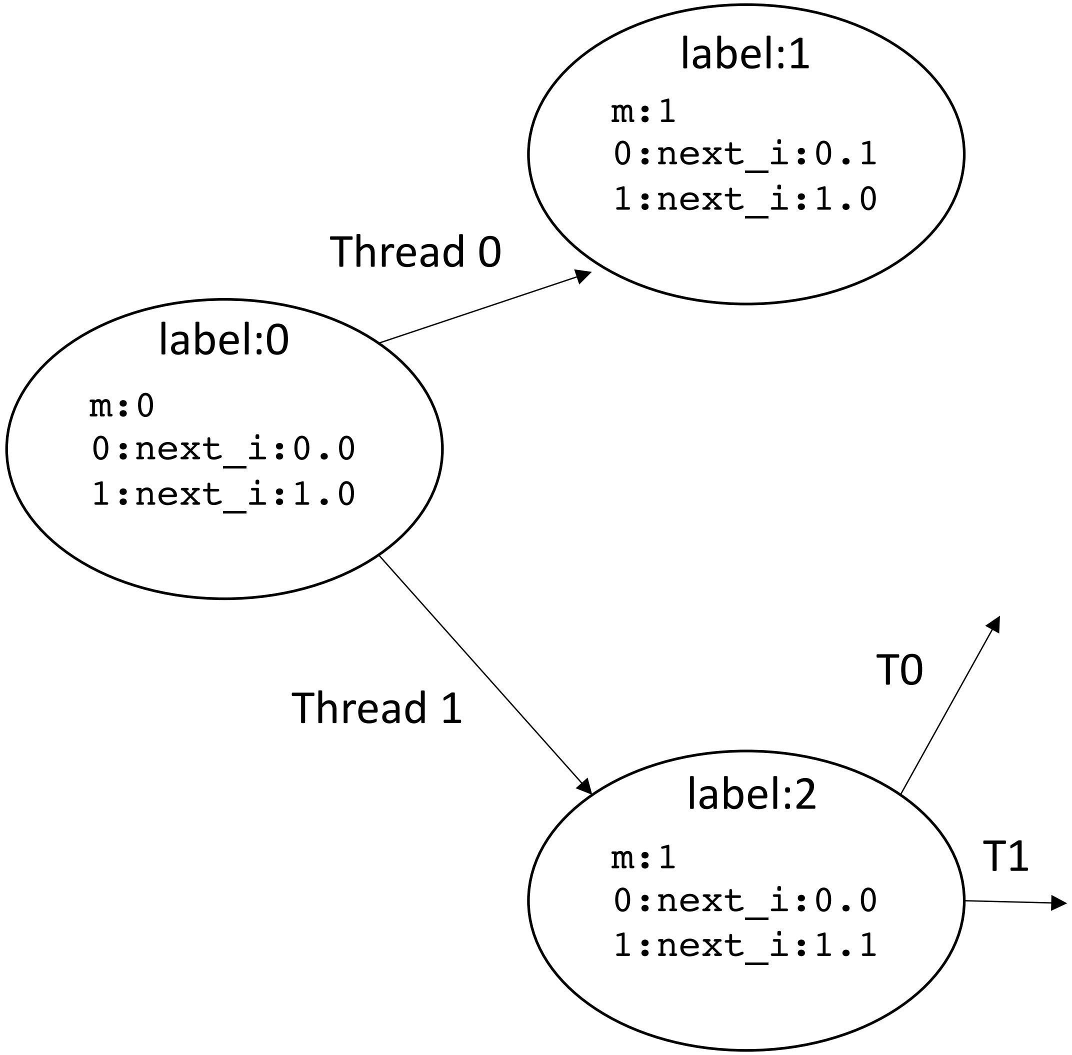


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



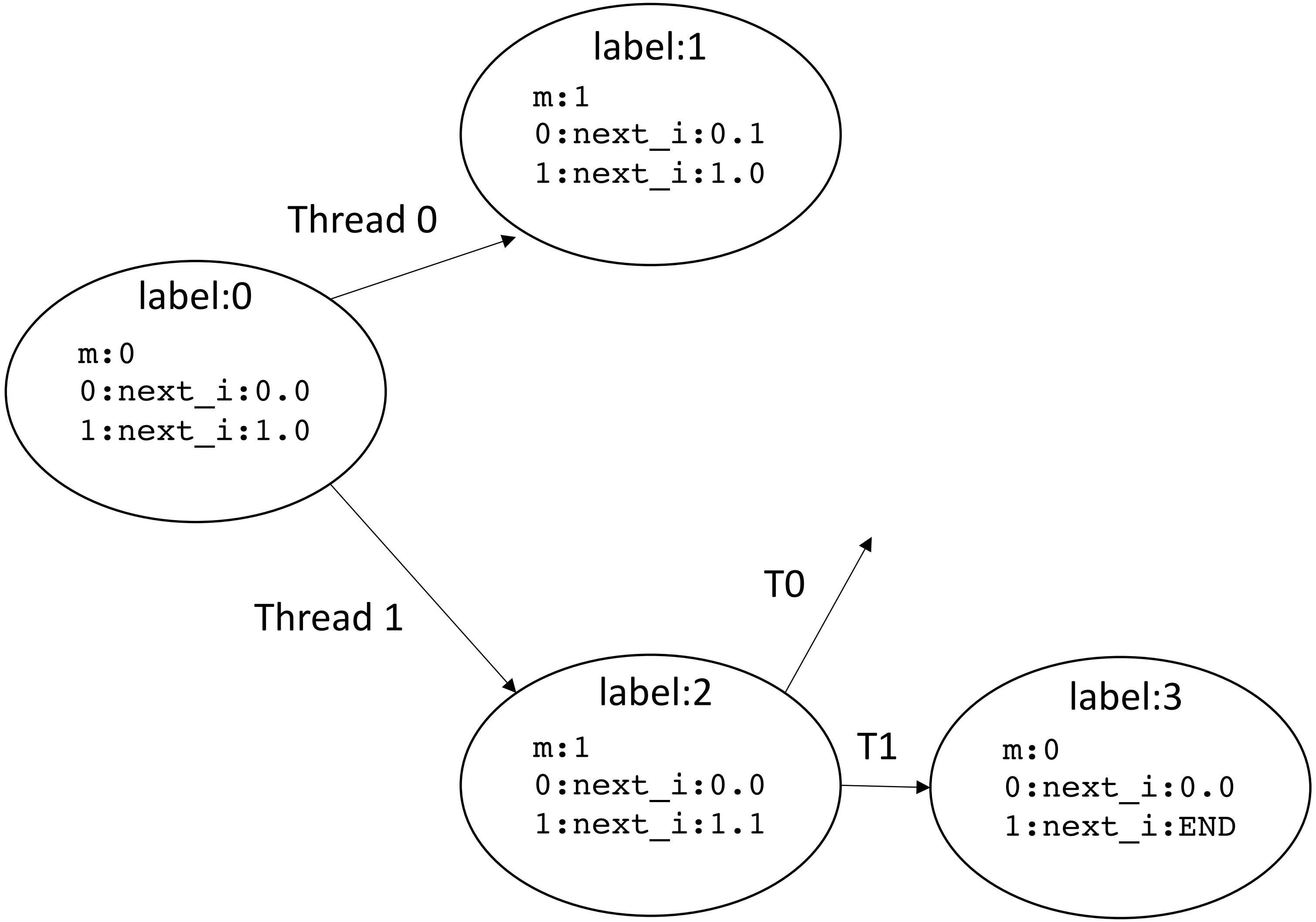
Lets do the states out of label 2

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0);  //unlock
```



label:0

m:0
0:next_i:0.0
1:next_i:1.0

Thread 0

label:1

m:1
0:next_i:0.1
1:next_i:1.0

Thread 1

label:2

m:1
0:next_i:0.0
1:next_i:1.1

T0

T1

label:3

m:0
0:next_i:0.0
1:next_i:END

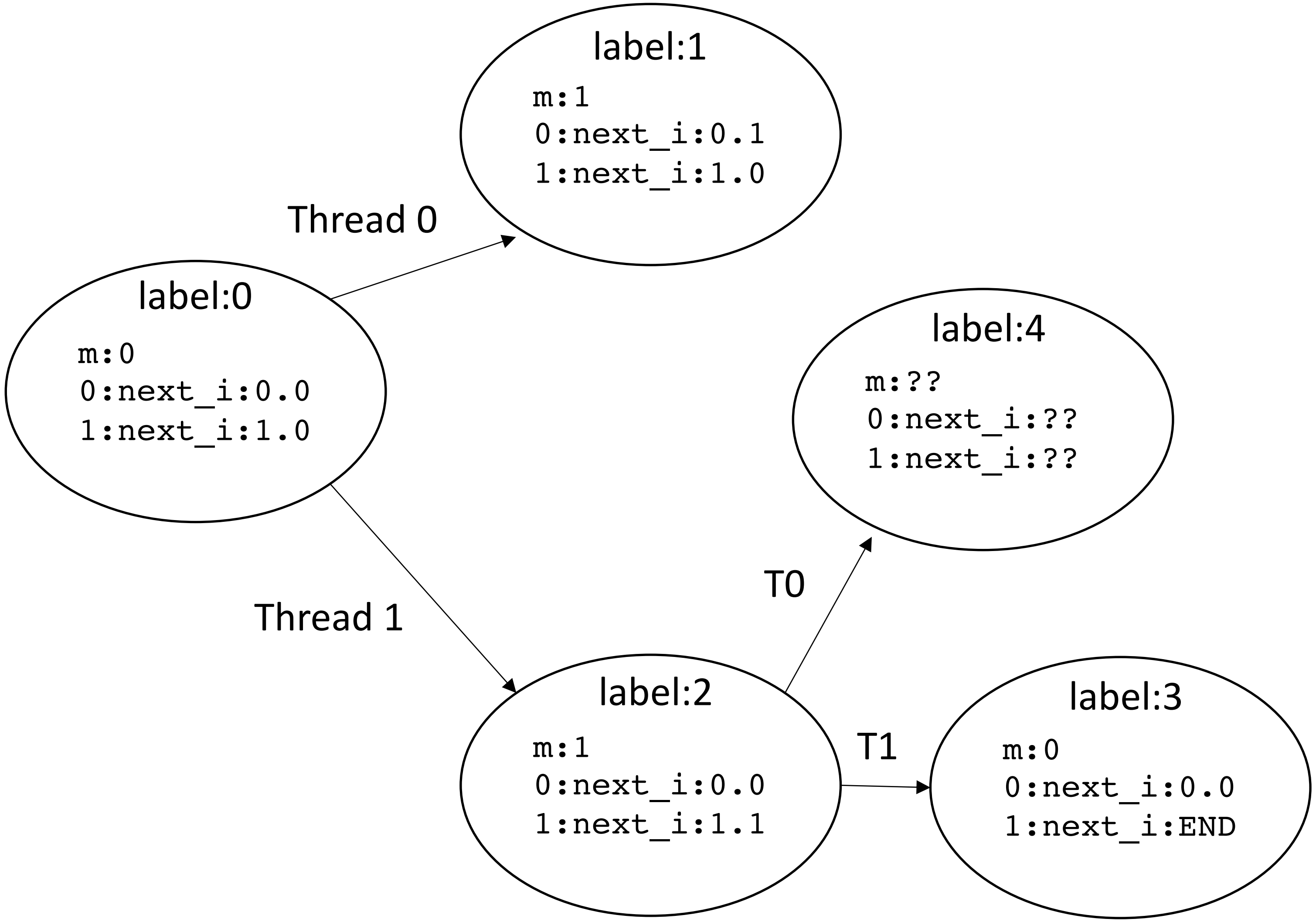
Lets do the states out of label 2

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0
m:0
0:next_i:0.0
1:next_i:1.0

Thread 0

label:1
m:1
0:next_i:0.1
1:next_i:1.0

Thread 1

label:2
m:1
0:next_i:0.0
1:next_i:1.1

T0

label:4
m:??
0:next_i:??
1:next_i:??

T1

label:3
m:0
0:next_i:0.0
1:next_i:END

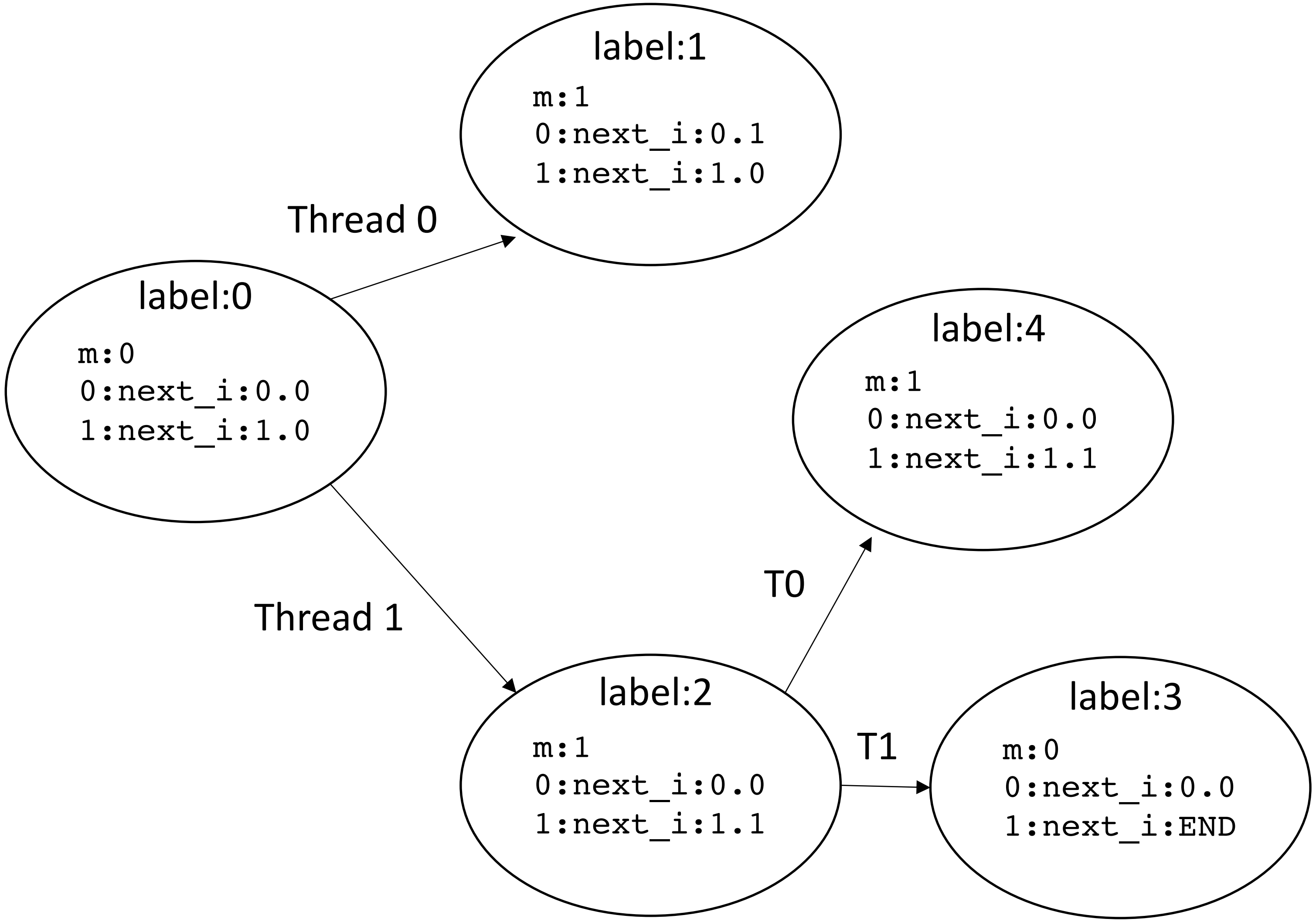
Lets do the states out of label 2

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



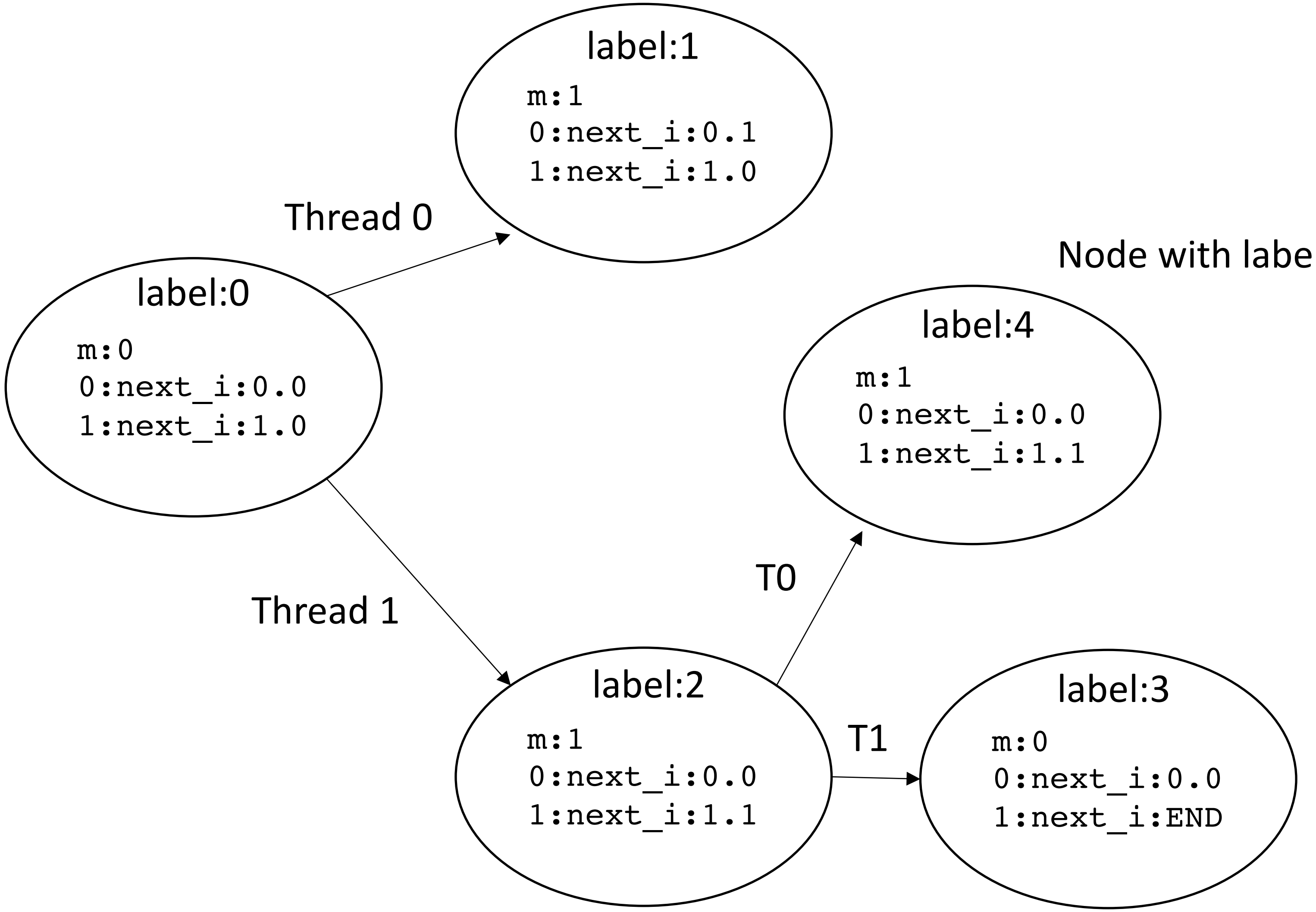
Lets do the states out of label 2

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0

```
m:0
0:next_i:0.0
1:next_i:1.0
```

Thread 0

label:1

```
m:1
0:next_i:0.1
1:next_i:1.0
```

Thread 1

label:2

```
m:1
0:next_i:0.0
1:next_i:1.1
```

T0

label:4

```
m:1
0:next_i:0.0
1:next_i:1.1
```

T1

label:3

```
m:0
0:next_i:0.0
1:next_i:END
```

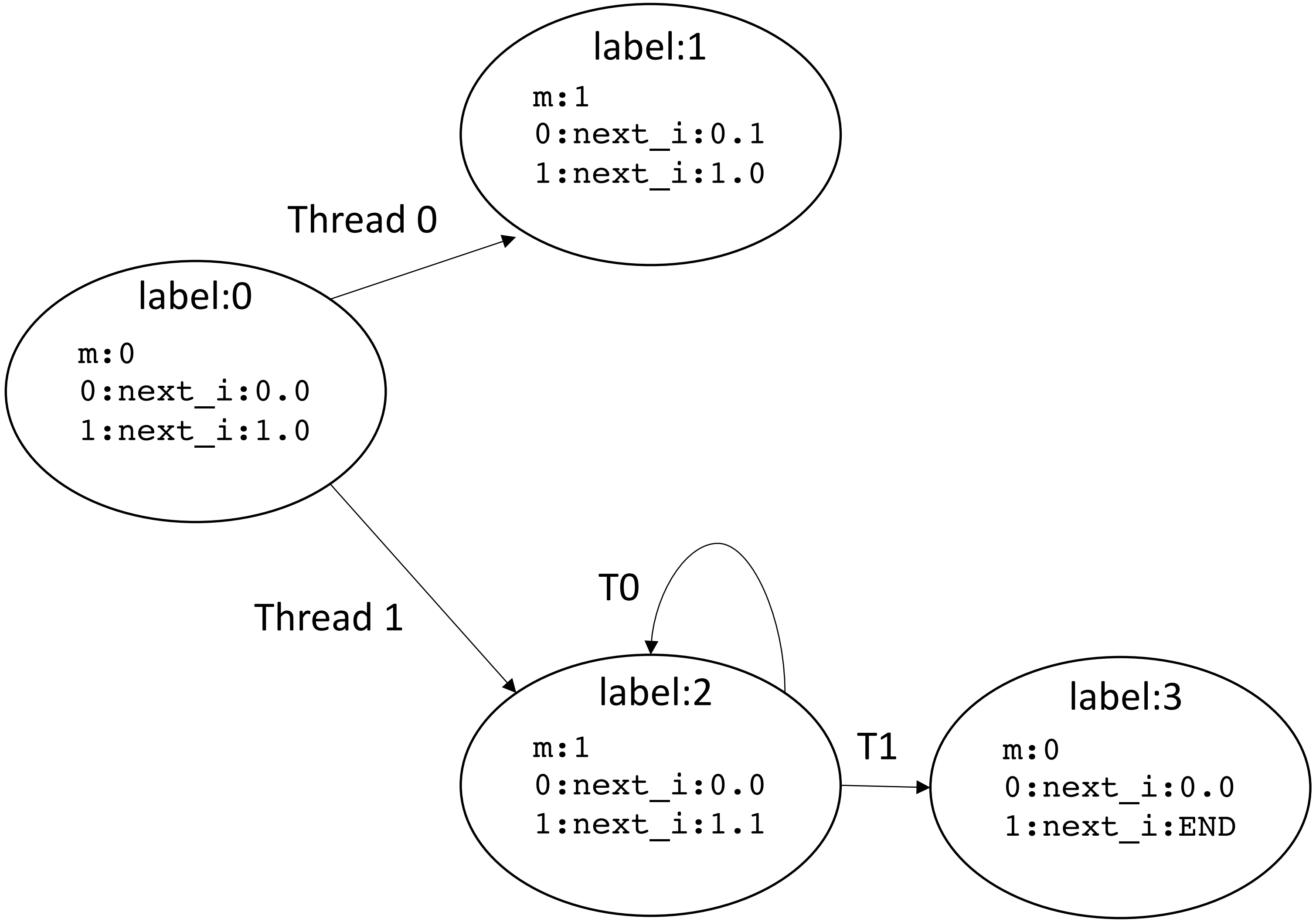
Node with label 4 looks familiar....

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

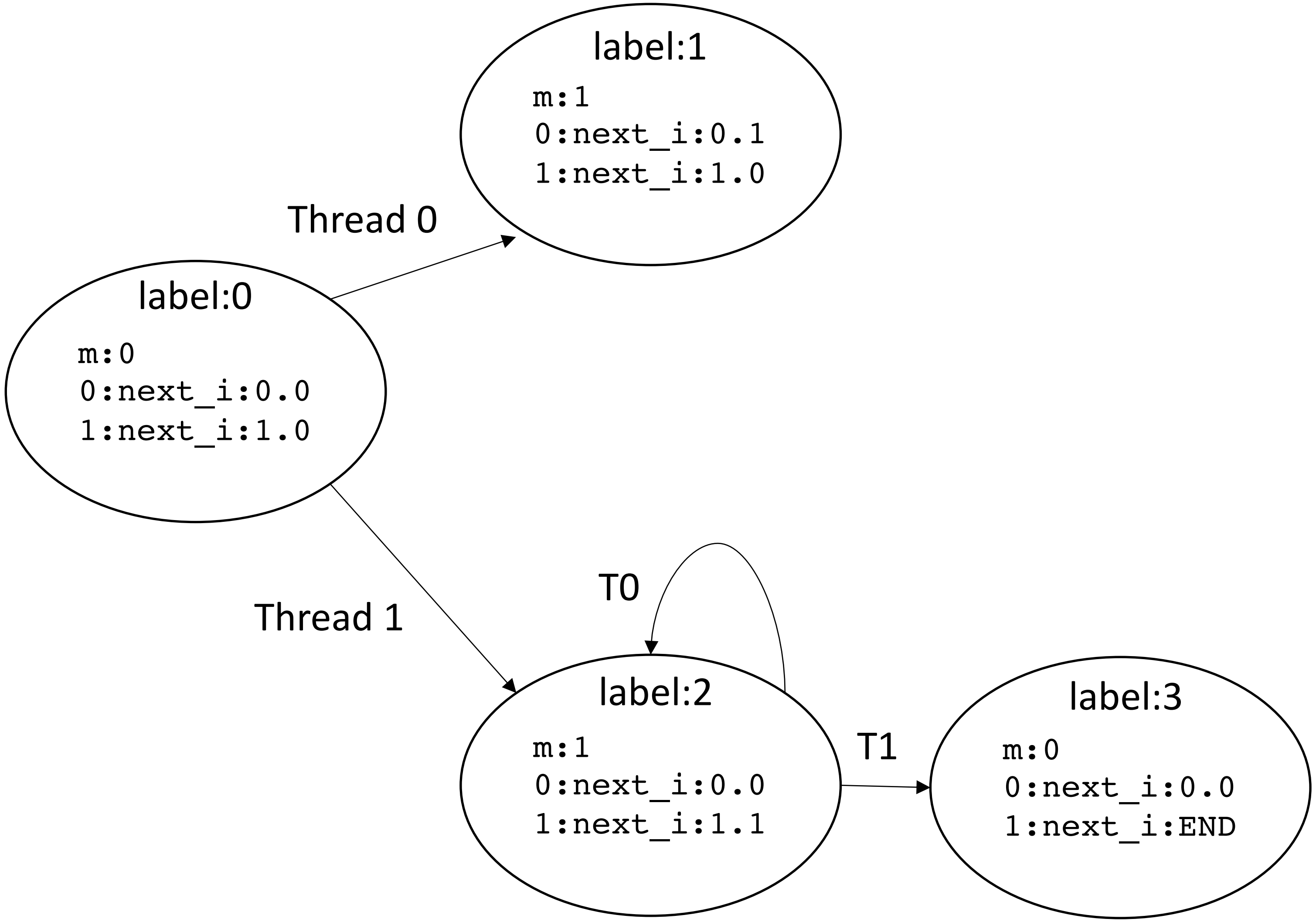


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0

```
m:0
0:next_i:0.0
1:next_i:1.0
```

Thread 0

label:1

```
m:1
0:next_i:0.1
1:next_i:1.0
```

Thread 1

label:2

```
m:1
0:next_i:0.0
1:next_i:1.1
```

T0

T1

label:3

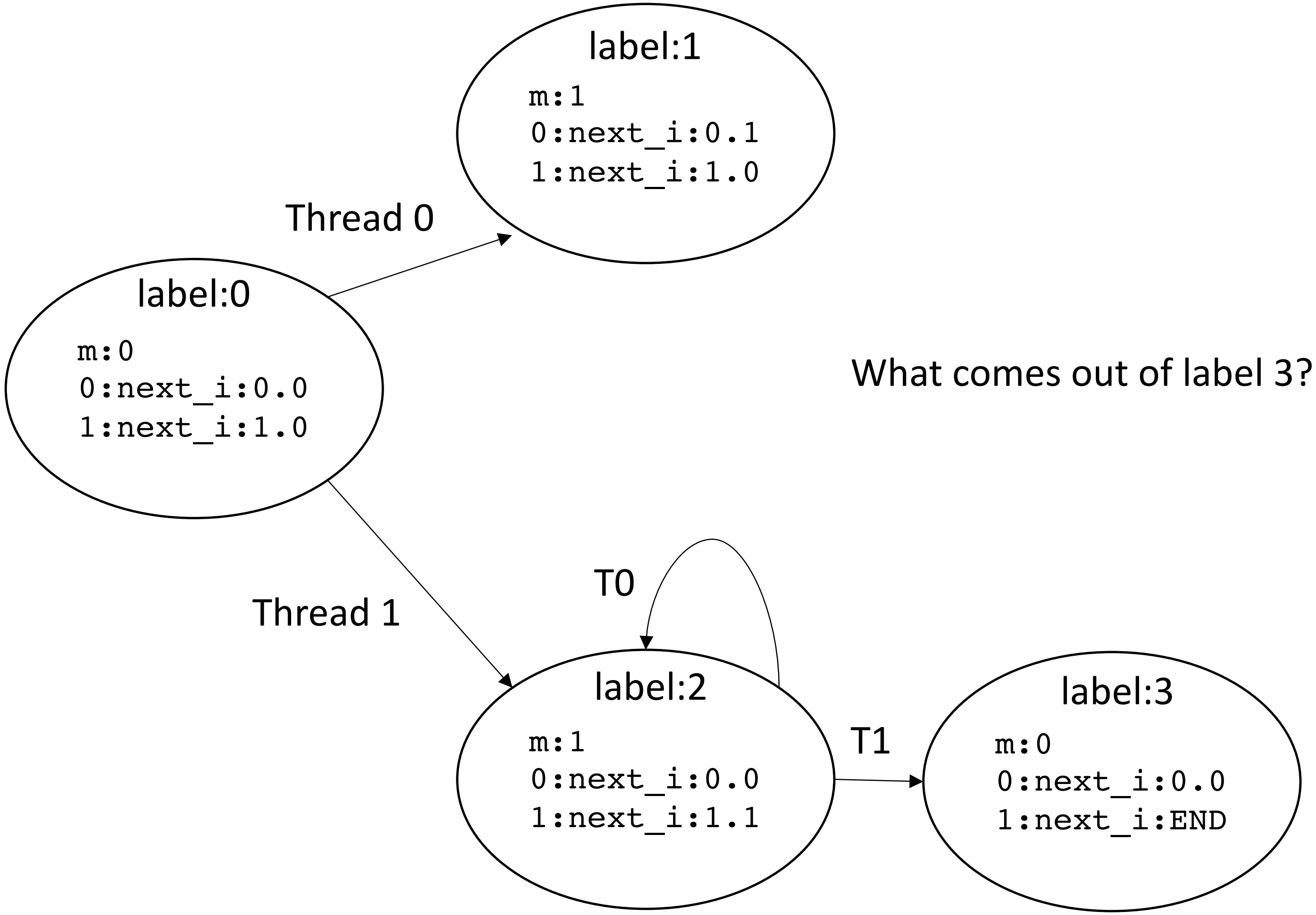
```
m:0
0:next_i:0.0
1:next_i:END
```

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



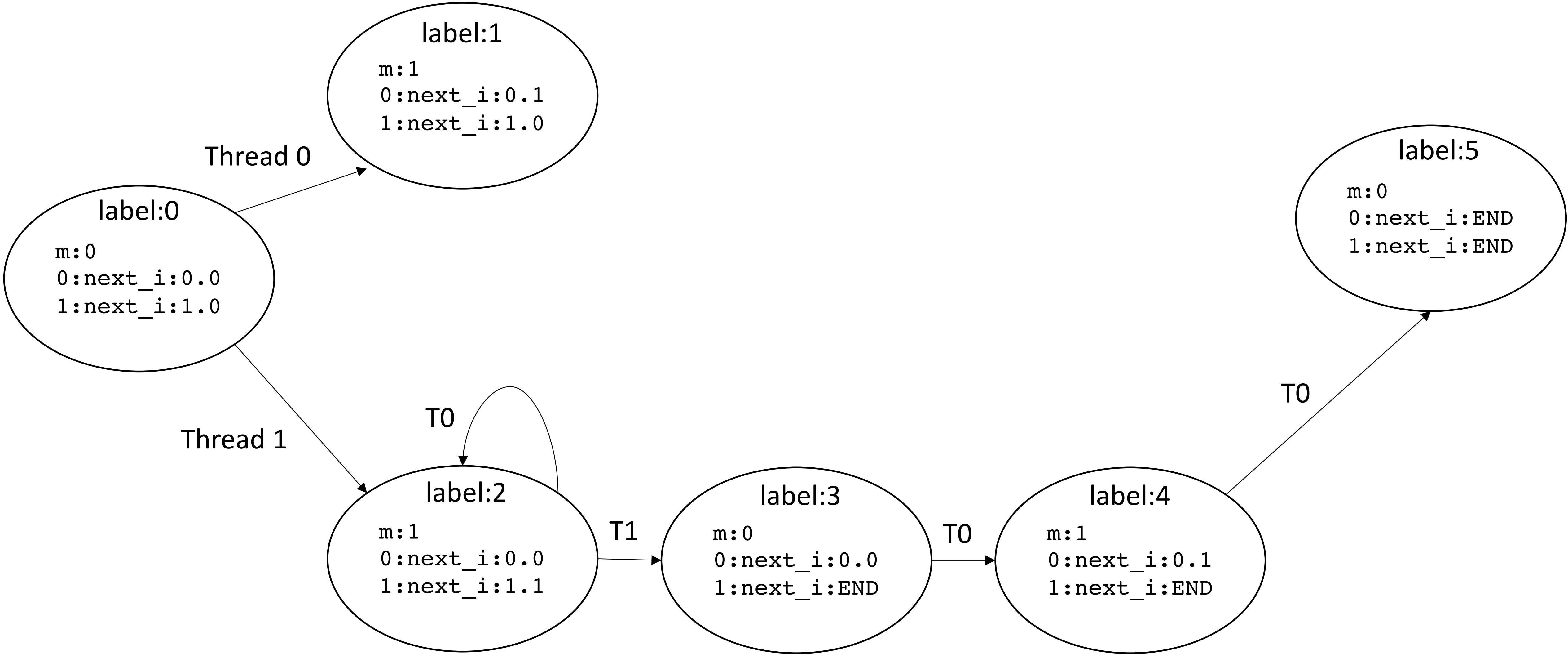
What comes out of label 3?

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

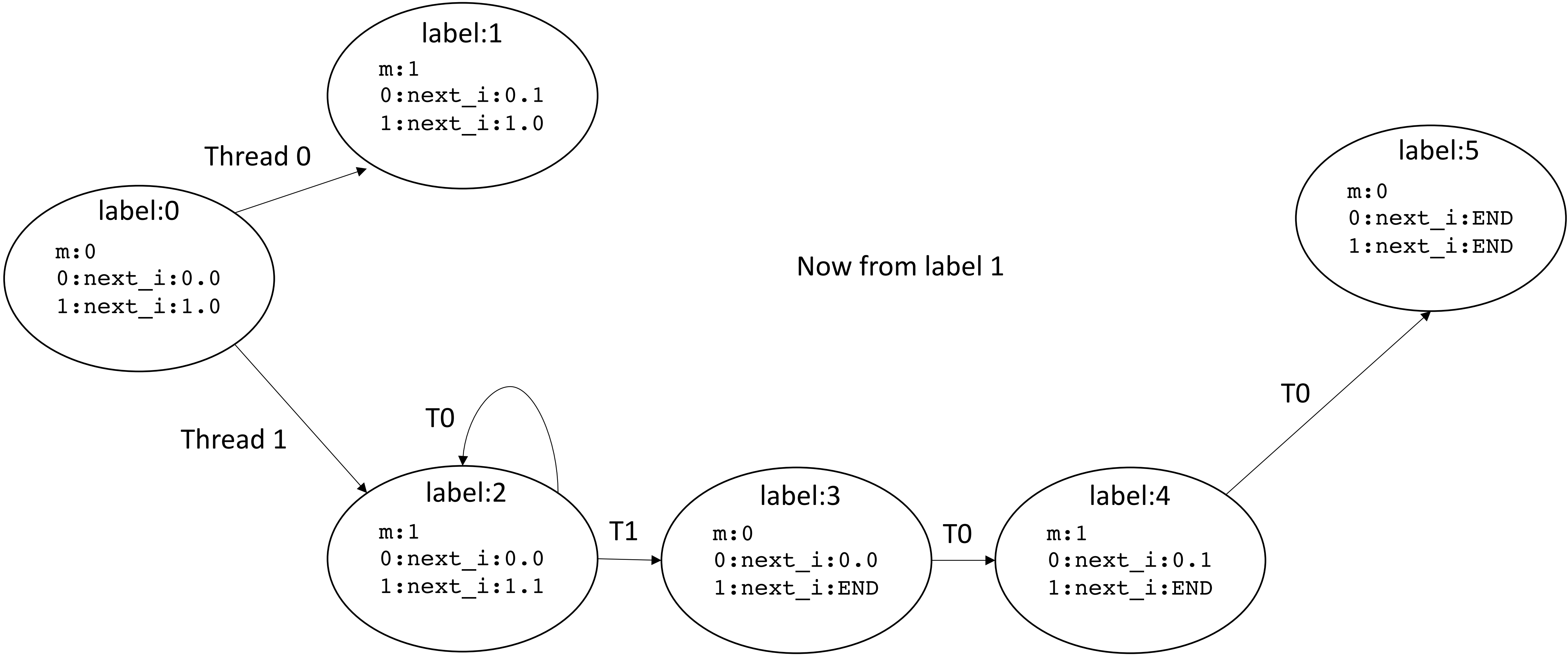


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



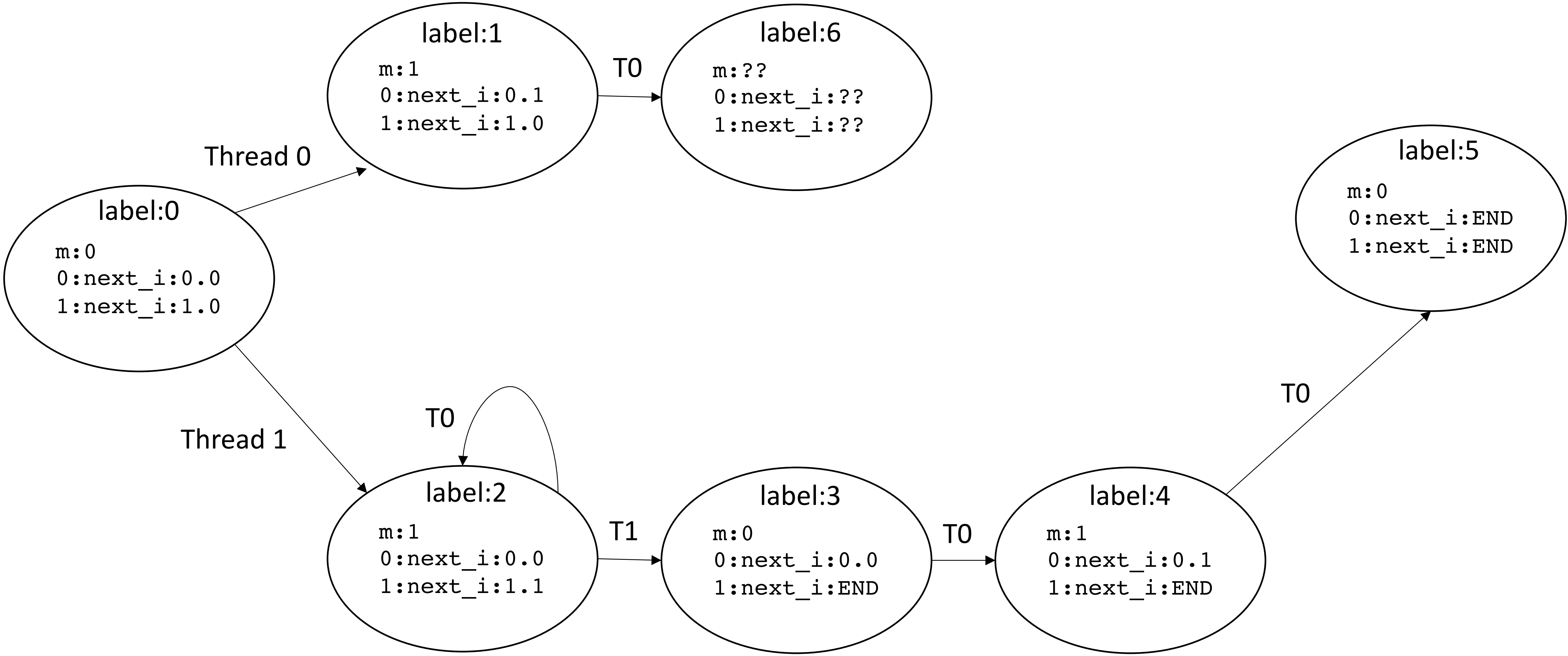
Now from label 1

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0
m:0
0:next_i:0.0
1:next_i:1.0

Thread 0

label:1
m:1
0:next_i:0.1
1:next_i:1.0

T0

label:6
m:??
0:next_i:??
1:next_i:??

Thread 1

label:2
m:1
0:next_i:0.0
1:next_i:1.1

T0

T1

label:3
m:0
0:next_i:0.0
1:next_i:END

T0

label:4
m:1
0:next_i:0.1
1:next_i:END

T0

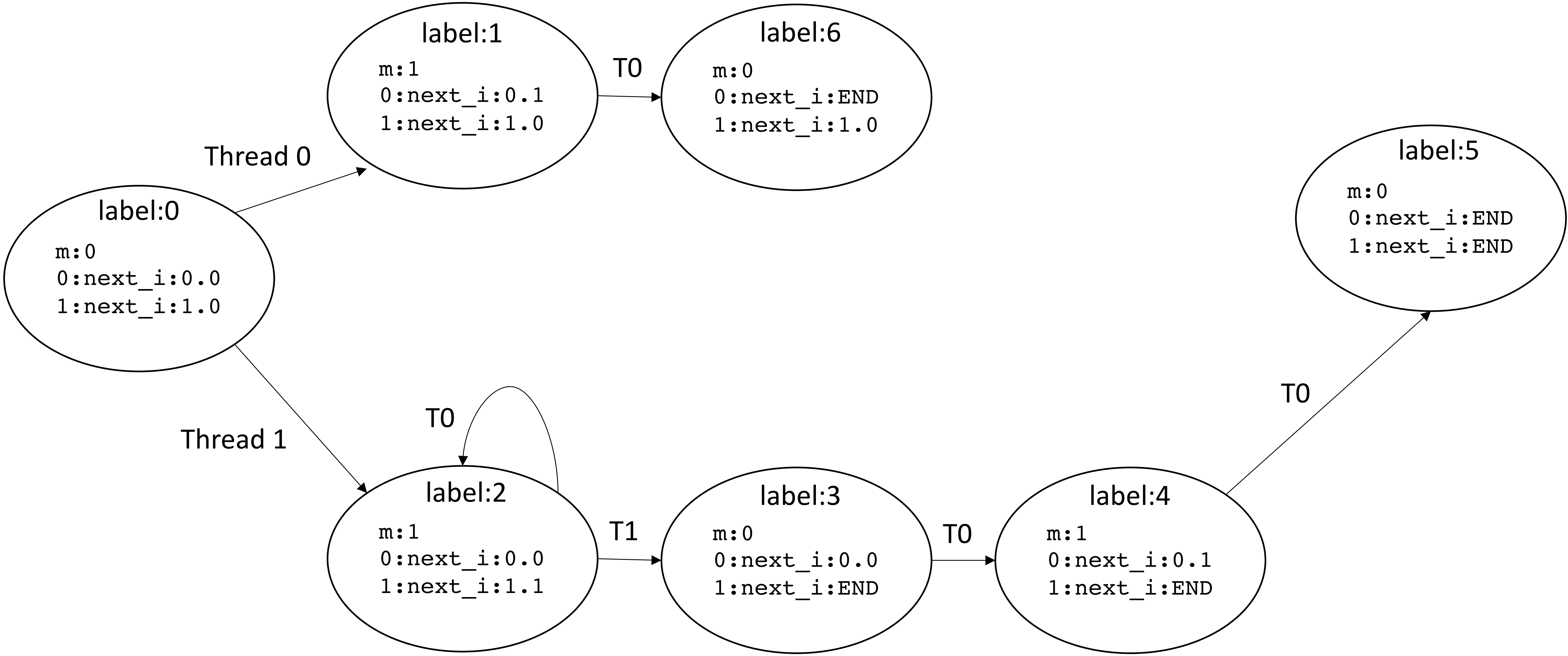
label:5
m:0
0:next_i:END
1:next_i:END

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

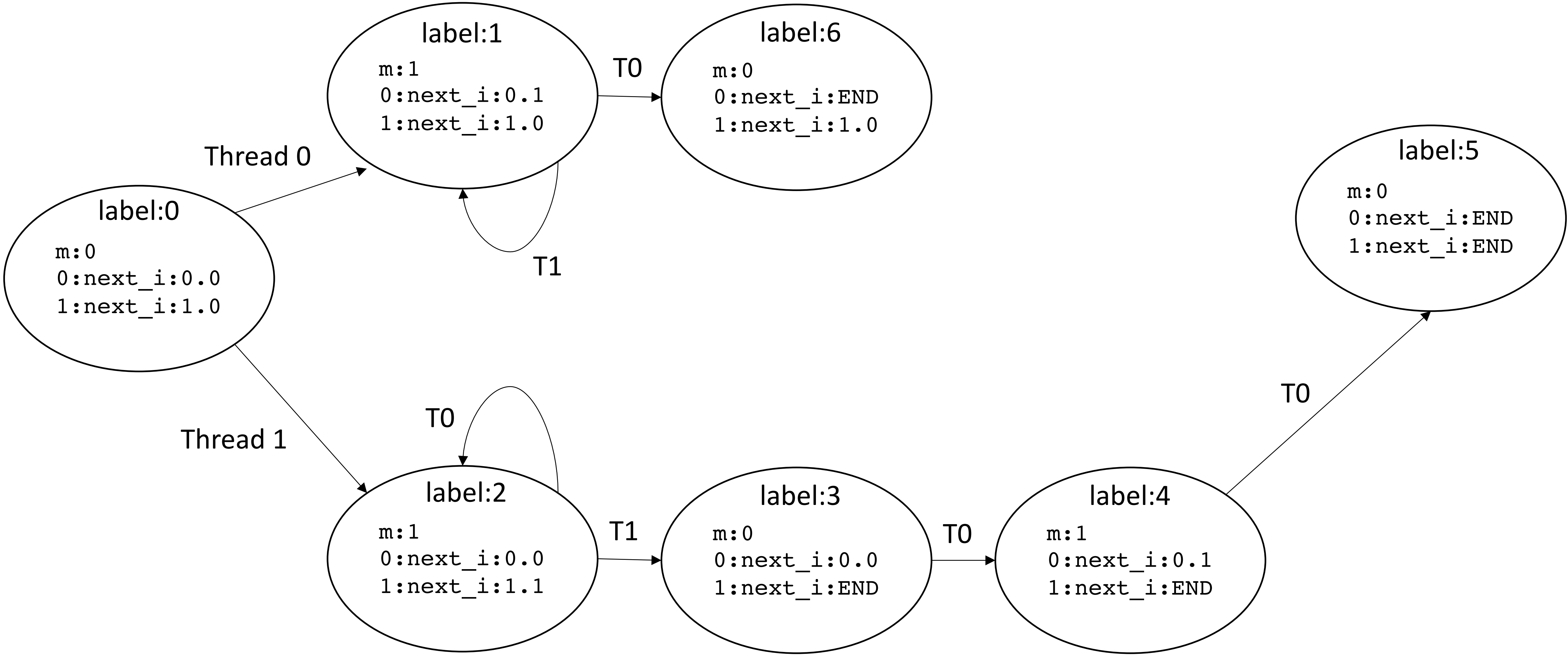


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

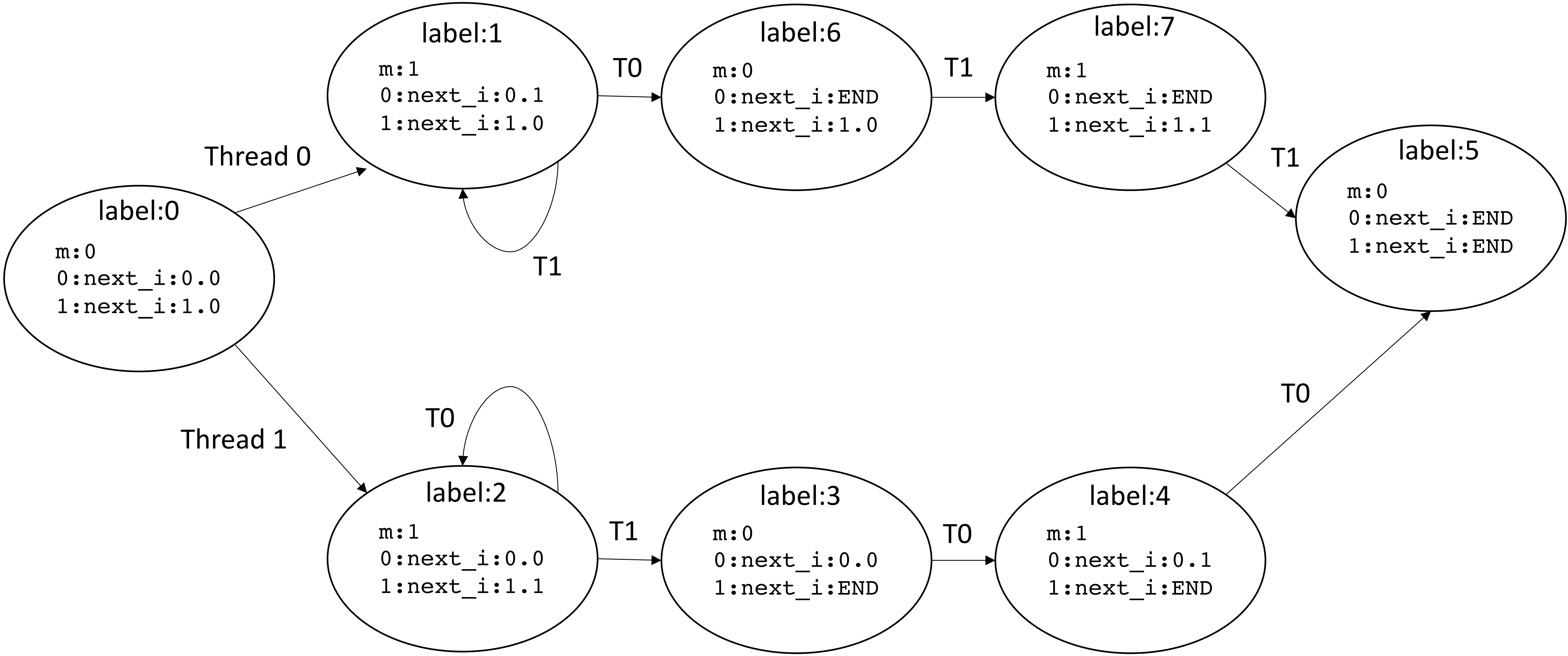


*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



# This is called an LTS

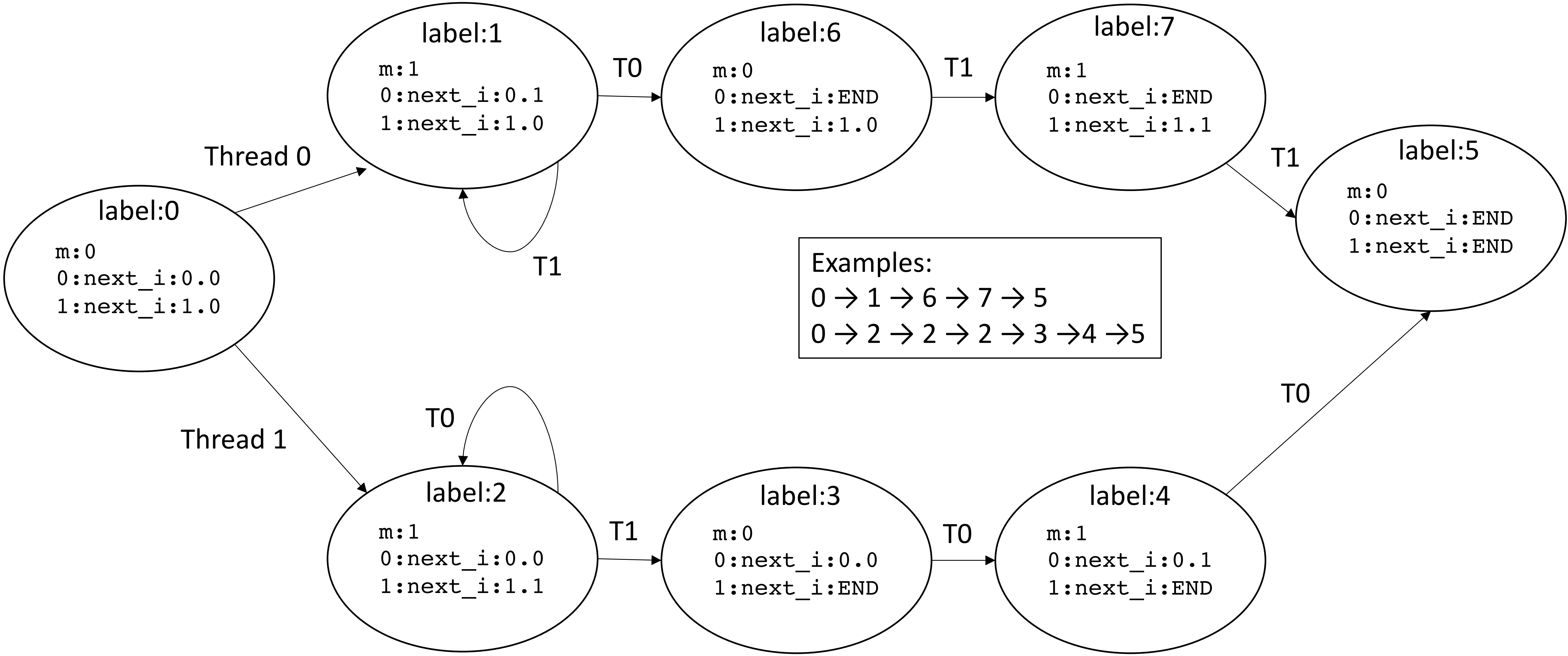
- A graph:
  - Each state encodes all variables/values and what the next instruction to execute is
  - Each edge out of a node is the different threads that can execute
  - A concurrent execution is any path through the LTS

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



Examples:

0 → 1 → 6 → 7 → 5

0 → 2 → 2 → 2 → 3 →4 →5

# What is this good for?

- Given this LTS, what kind of questions can we ask?

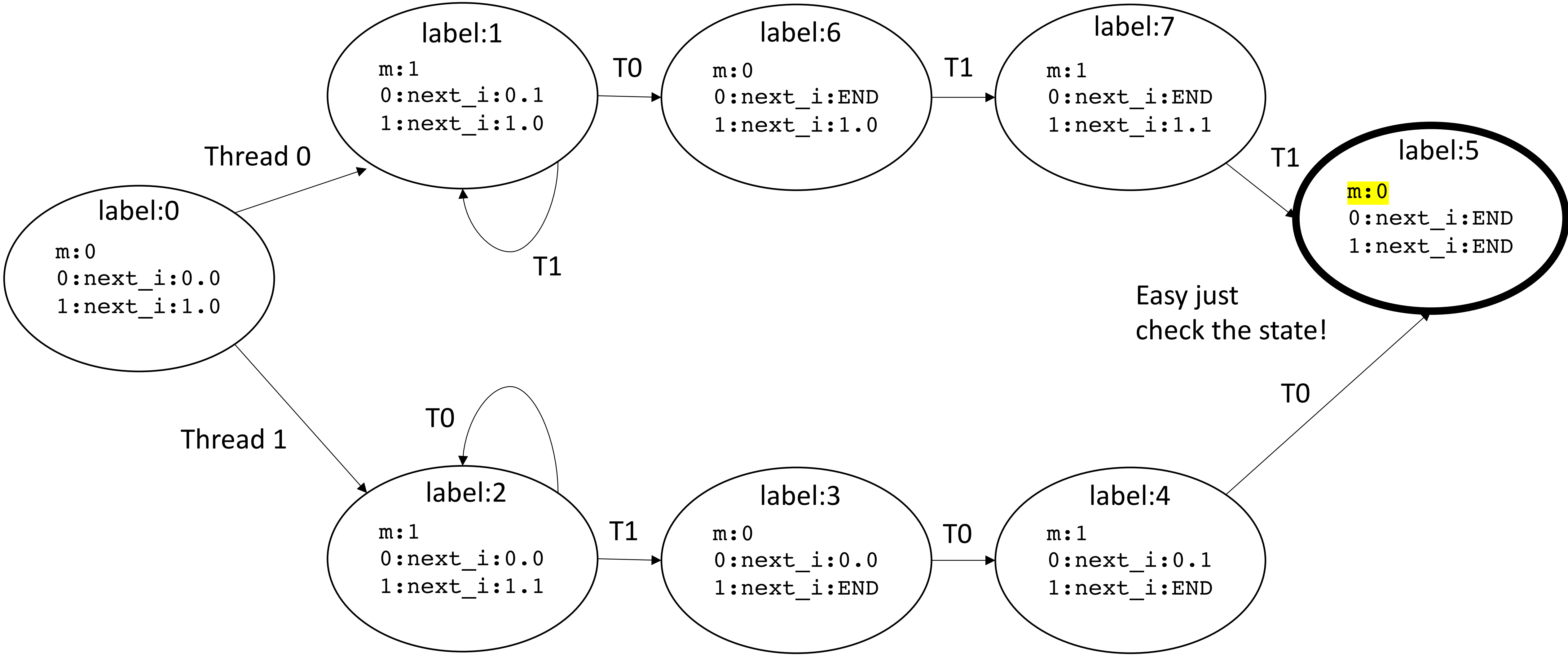
- Example:
  - At the end of the program, I want to prove that the mutex will not be taken

*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



label:0
m:0
0:next_i:0.0
1:next_i:1.0

Thread 0

label:1
m:1
0:next_i:0.1
1:next_i:1.0

T1

T0

label:6
m:0
0:next_i:END
1:next_i:1.0

T1

label:7
m:1
0:next_i:END
1:next_i:1.1

T1

label:5
m:0
0:next_i:END
1:next_i:END

Thread 1

label:2
m:1
0:next_i:0.0
1:next_i:1.1

T0

T1

label:3
m:0
0:next_i:0.0
1:next_i:END

T0

label:4
m:1
0:next_i:0.1
1:next_i:END

T0

Easy just
check the state!

# Safety property

- ***Something bad will never happen***
    - i.e. the program will not exit with the mutex taken
    - can be specified with assert statements in the program

- Easy to check in a LTS: just search the states
    - You have all the values, easy to check if something is wrong!

# However…

- *Safety is only half of the picture*

- Self driving car example:
    - Design a car that never crashes (safety property)

# However…

- *Safety is only half of the picture*

- Self driving car example:
  - Design a car that never crashes (safety property)
  - **Easy**! Just design a car that can’t move!

  - We need include something else in the specification:

# Liveness property

- Something good will eventually happen

- Examples:
  - The mutex program *will eventually terminate*
  - The self driving car *will eventually reach its destination*

- More difficult to reason about that safety properties

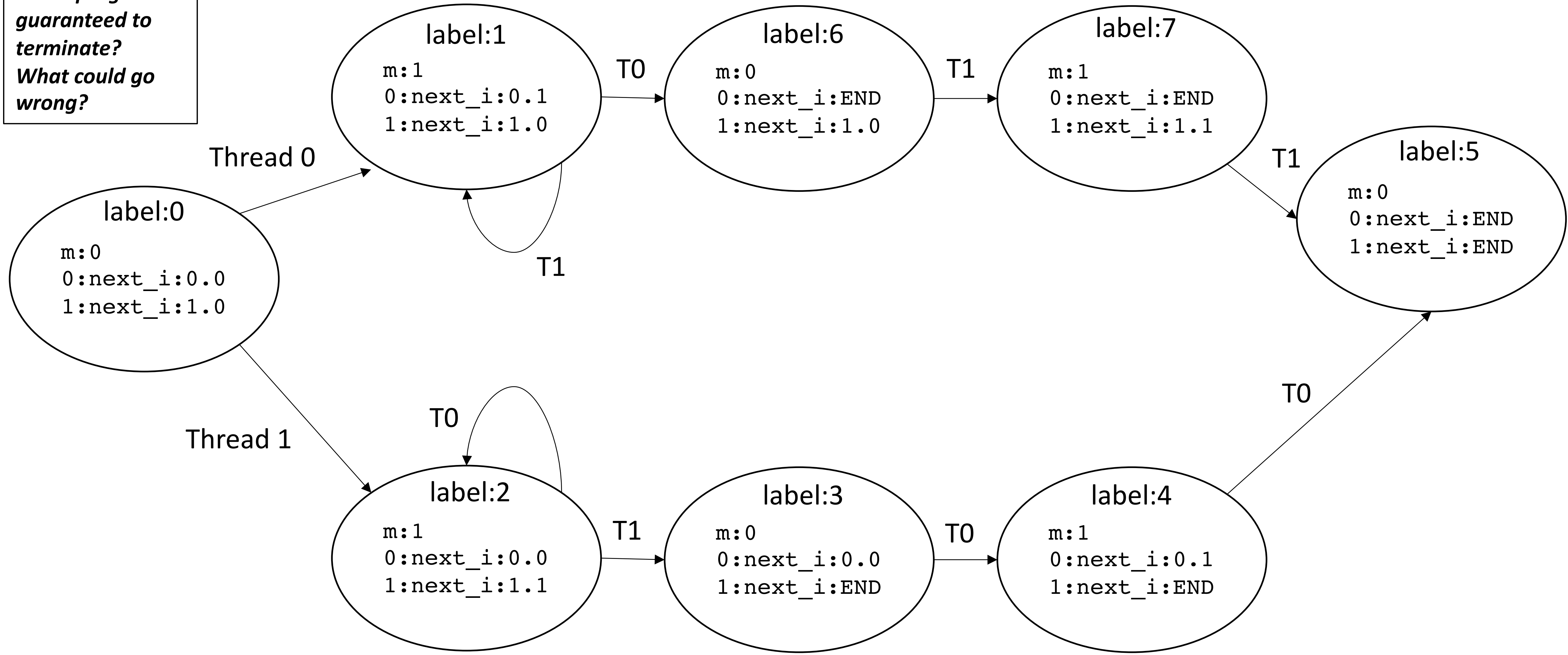
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***Is this program guaranteed to terminate? What could go wrong?***



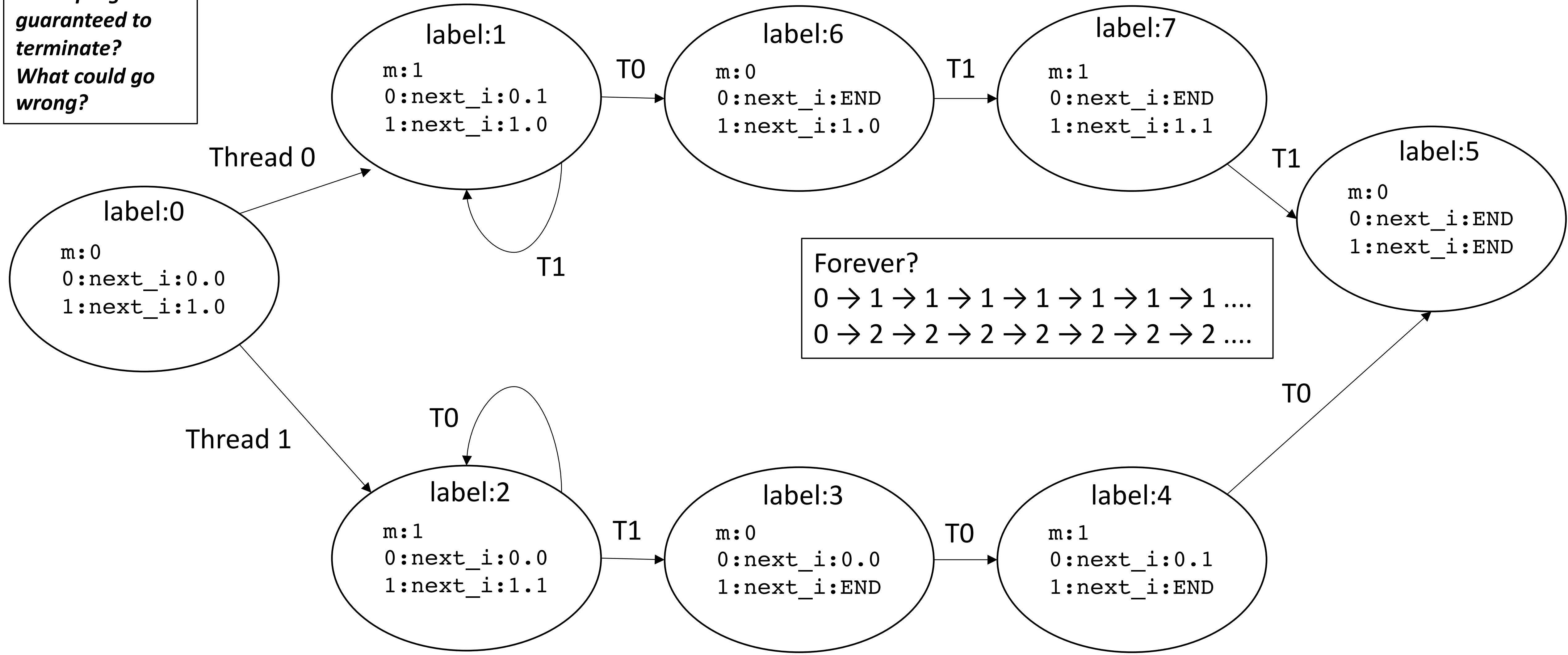
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***Is this program guaranteed to terminate? What could go wrong?***



Forever?

0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ….

0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ….

# Liveness

- Starvation cycles
  - There exists a thread that can break the system out of a cycle, but that thread never executes (i.e. it is starved).

- Can starvation cycles happen?

# Liveness

- Starvation cycles
  - There exists a thread that can break the system out of a cycle, but that thread never executes (i.e. it is starved).

- Can starvation cycles happen?
  - Depends on your scheduler!
  - With no scheduler guarantees, they cannot be ruled out!

# Liveness

- Starvation cycles
  - There exists a thread that can break the system out of a cycle, but that thread never executes (i.e. it is starved).
- Can starvation cycles happen?
  - Depends on your scheduler!
  - With no scheduler guarantees, they cannot be ruled out!
- Note that we are talking about scheduler ***specifications***, actual implementations are very complicated (take an OS class to learn more)

# The fair scheduler

- every thread that has not terminated will “eventually” get a chance to execute.
  - “concurrent forward progress”: defined by C++
    not guaranteed, but encouraged (and likely what you will observe)
  - “weakly fair scheduler”: defined by classic concurrency textbooks
- The fair scheduler disallows starvation cycles
  - waiting will always be finite (but no bounds on time)

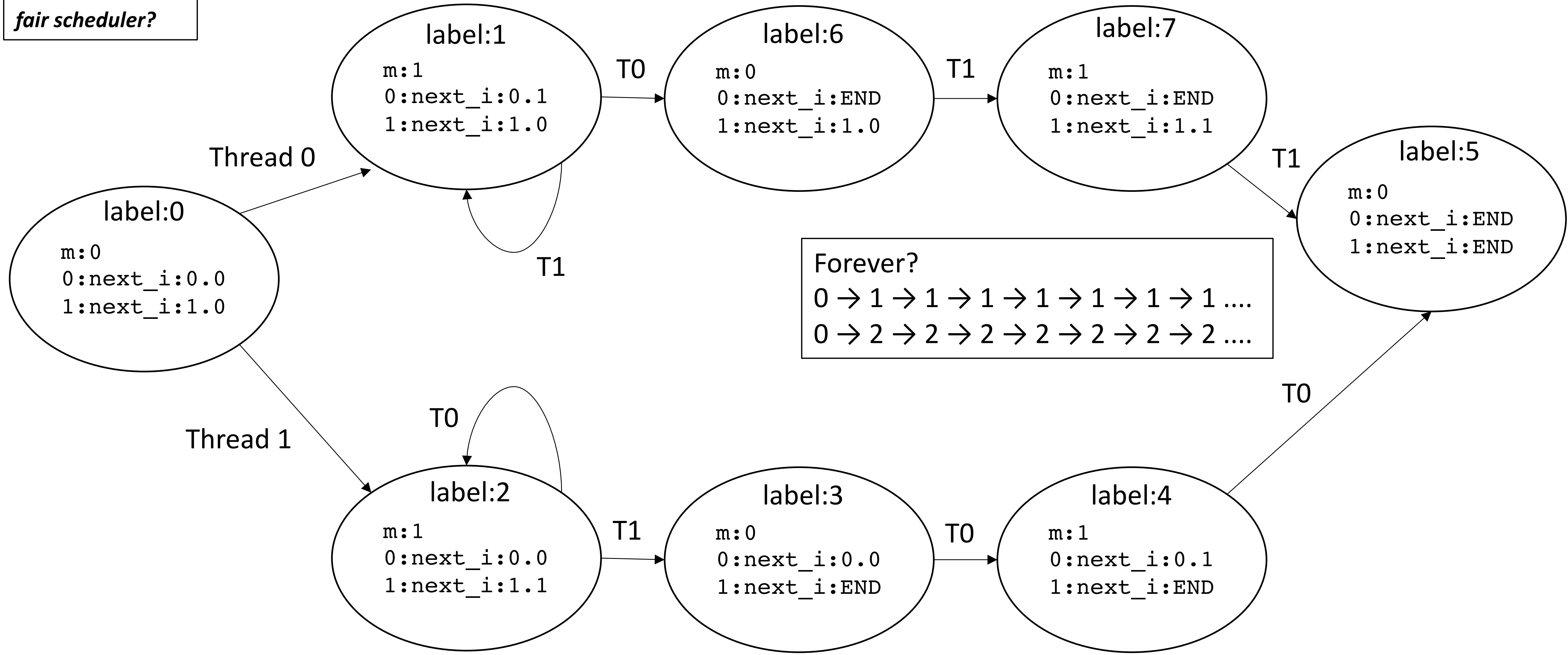
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***What about a fair scheduler?***



Forever?

0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ....

0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ....

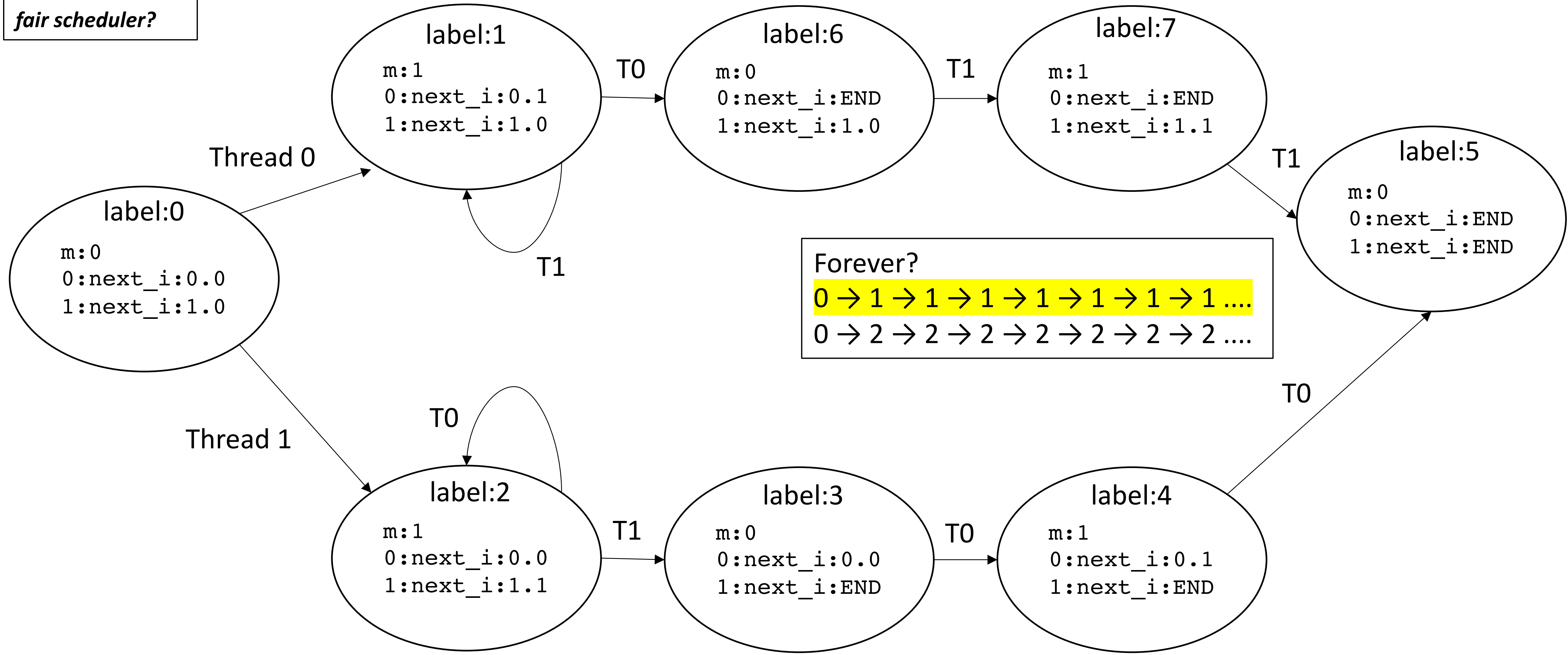
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***What about a fair scheduler?***



Forever?

0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ....

0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ....

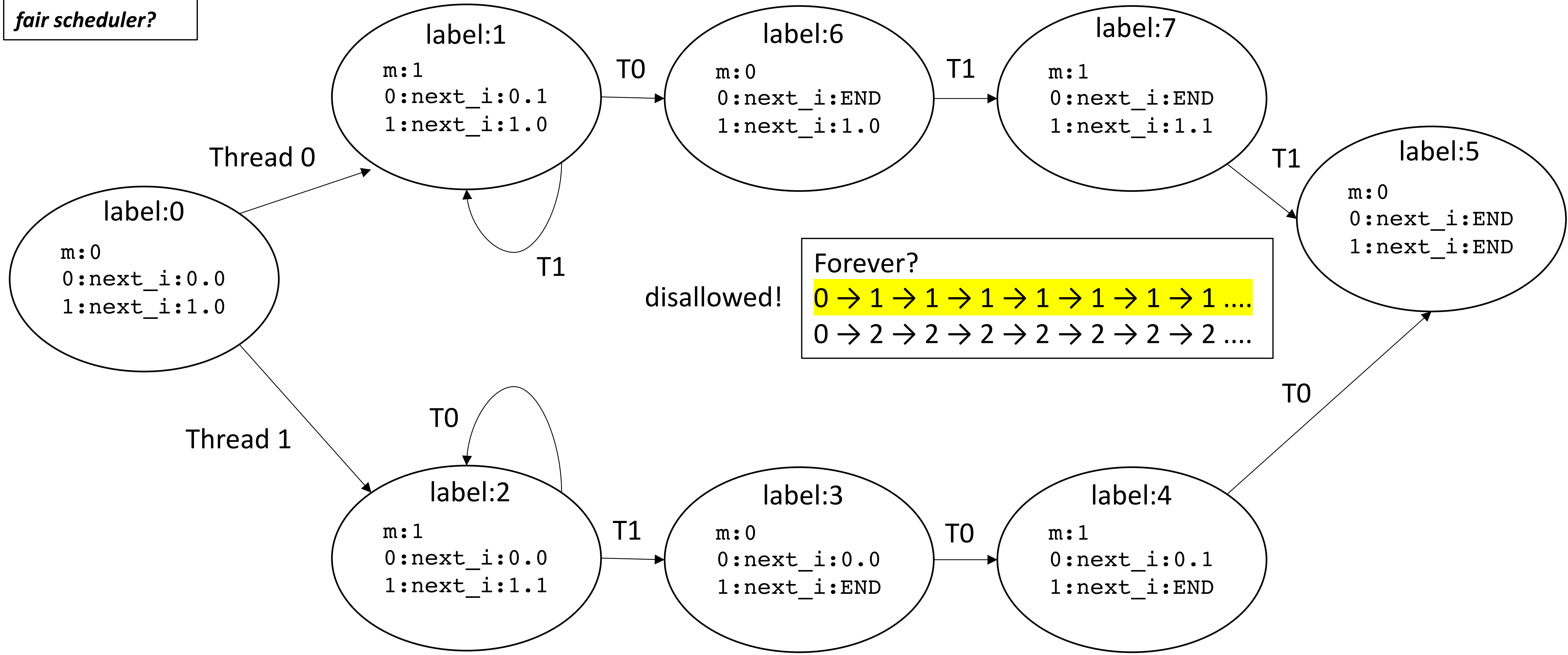
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***What about a fair scheduler?***



Forever?

0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ....

0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ....

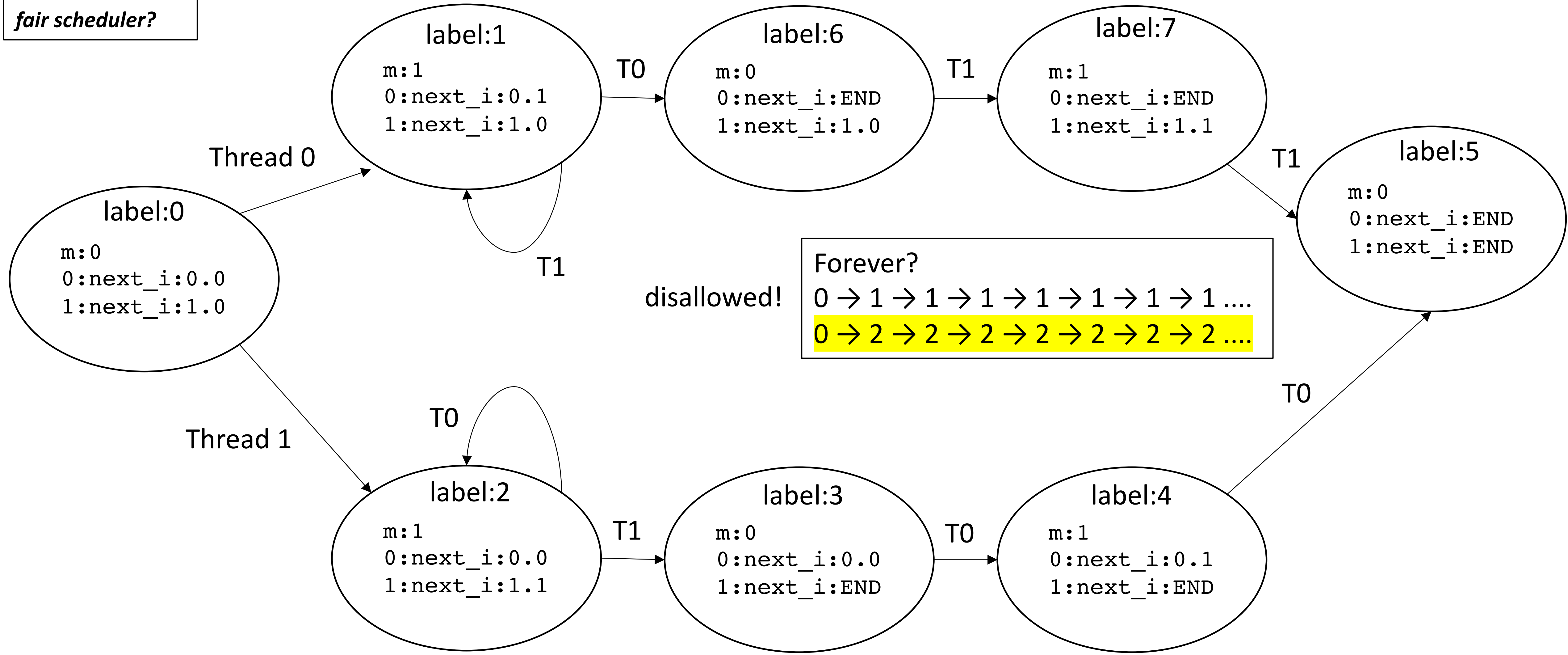
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***What about a fair scheduler?***



disallowed!

Forever?

0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ....

0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ....

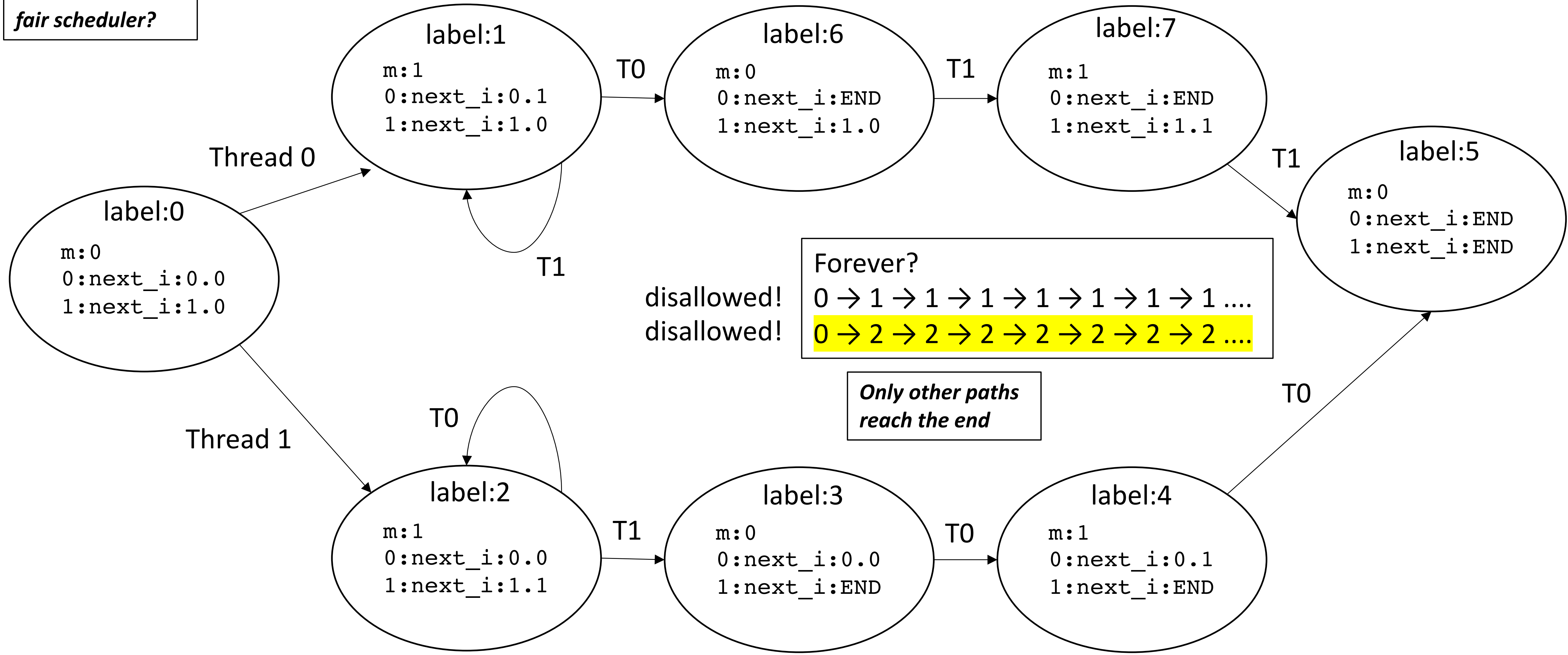
*Thread 0:*

```
0.0: while(CAS(&m,0,1) == false); //lock
     // critical section
0.1: m.store(0); //unlock
```

*Thread 1:*

```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

***What about a fair scheduler?***



Forever?

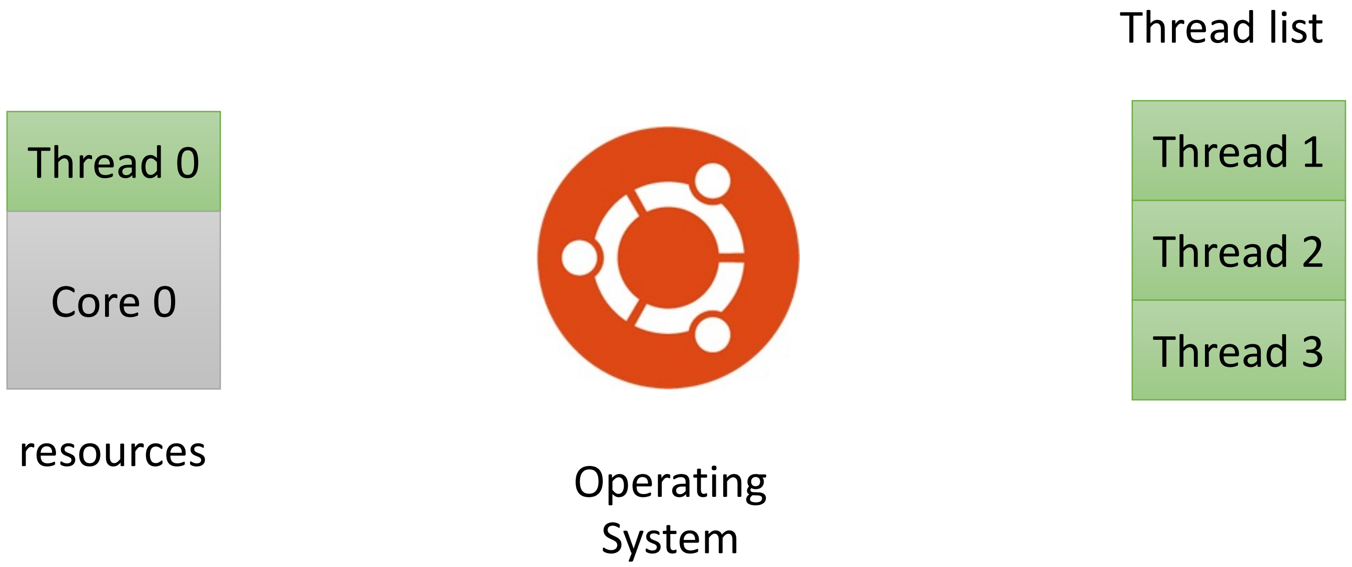
disallowed! 0 → 1 → 1 → 1 → 1 → 1 → 1 → 1 ….

disallowed! 0 → 2 → 2 → 2 → 2 → 2 → 2 → 2 ….

***Only other paths reach the end***

# Schedulers

- A fair scheduler typically requires preemption



Thread list

Thread 0

Core 0

resources

Operating System

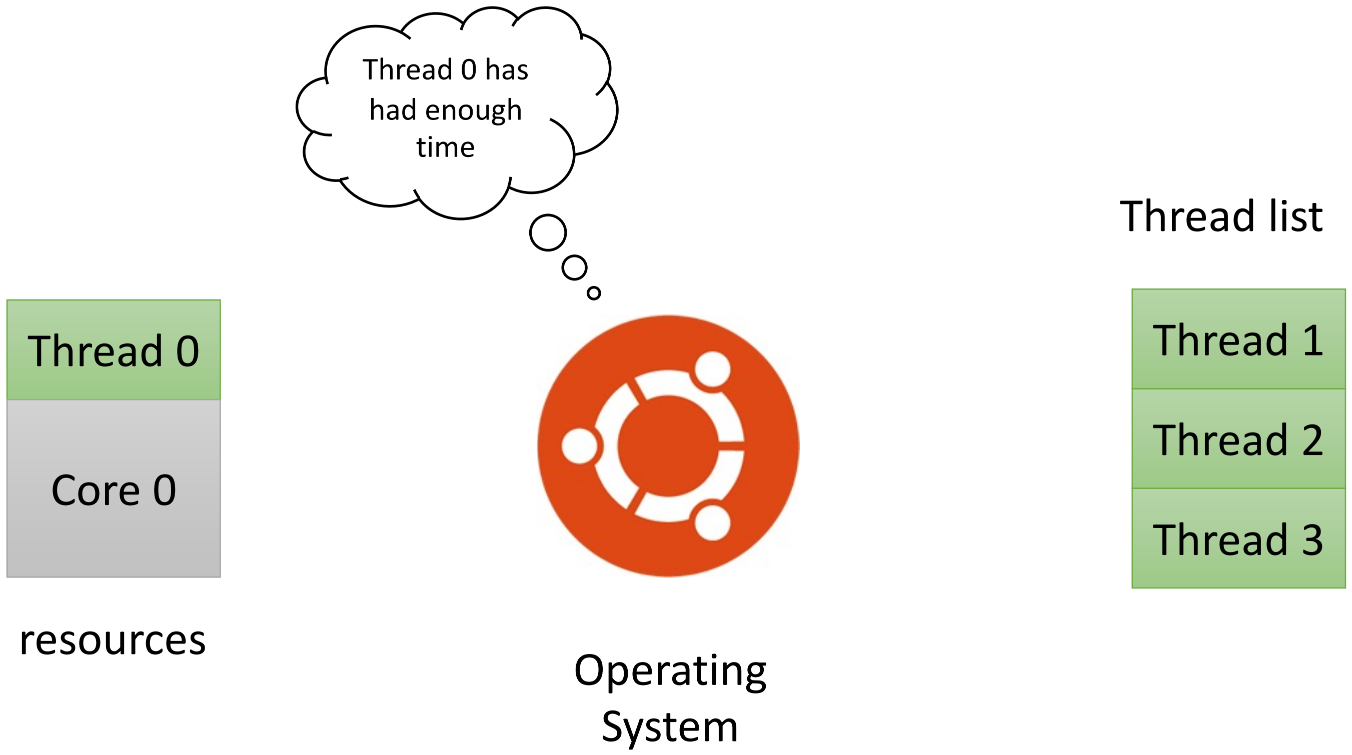
Thread 1

Thread 2

Thread 3

# Schedulers

- A fair scheduler typically requires preemption



Thread 0 has had enough time

Thread 0

Core 0

resources

Operating System

Thread list

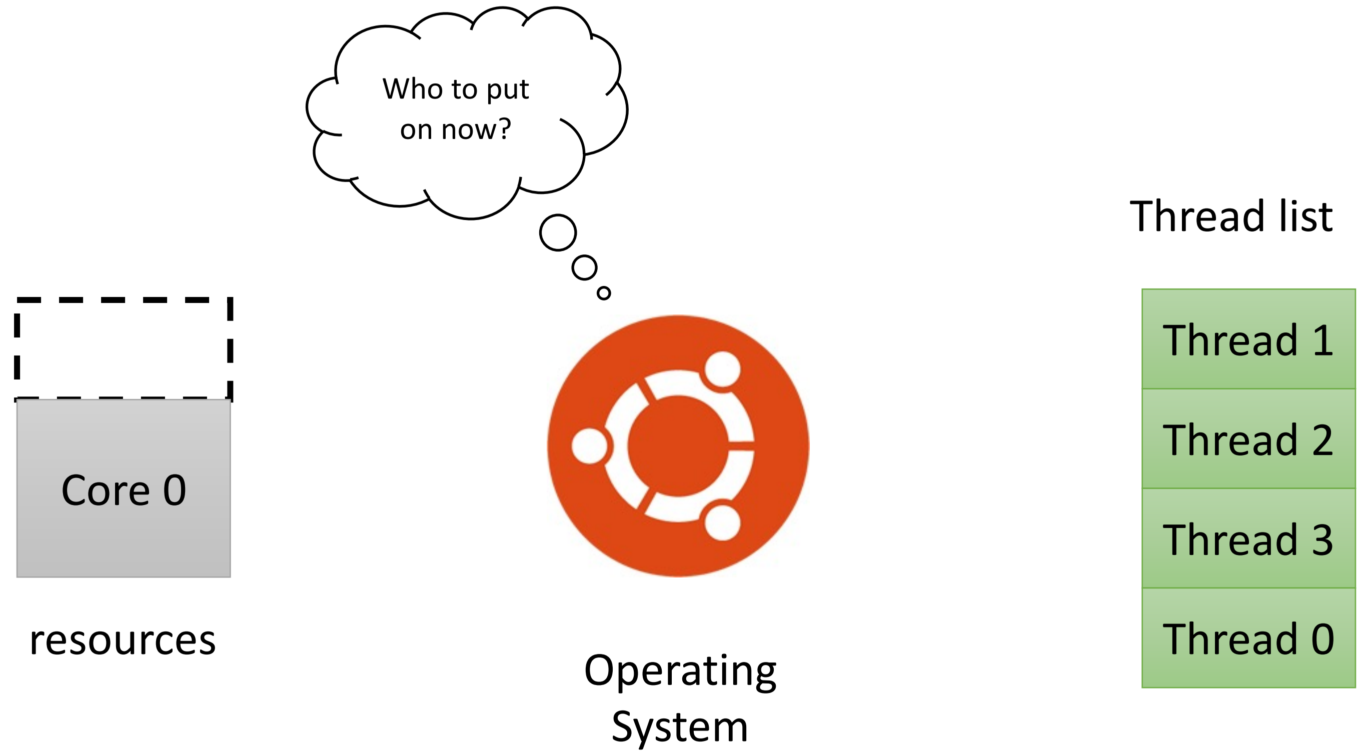
Thread 1

Thread 2

Thread 3

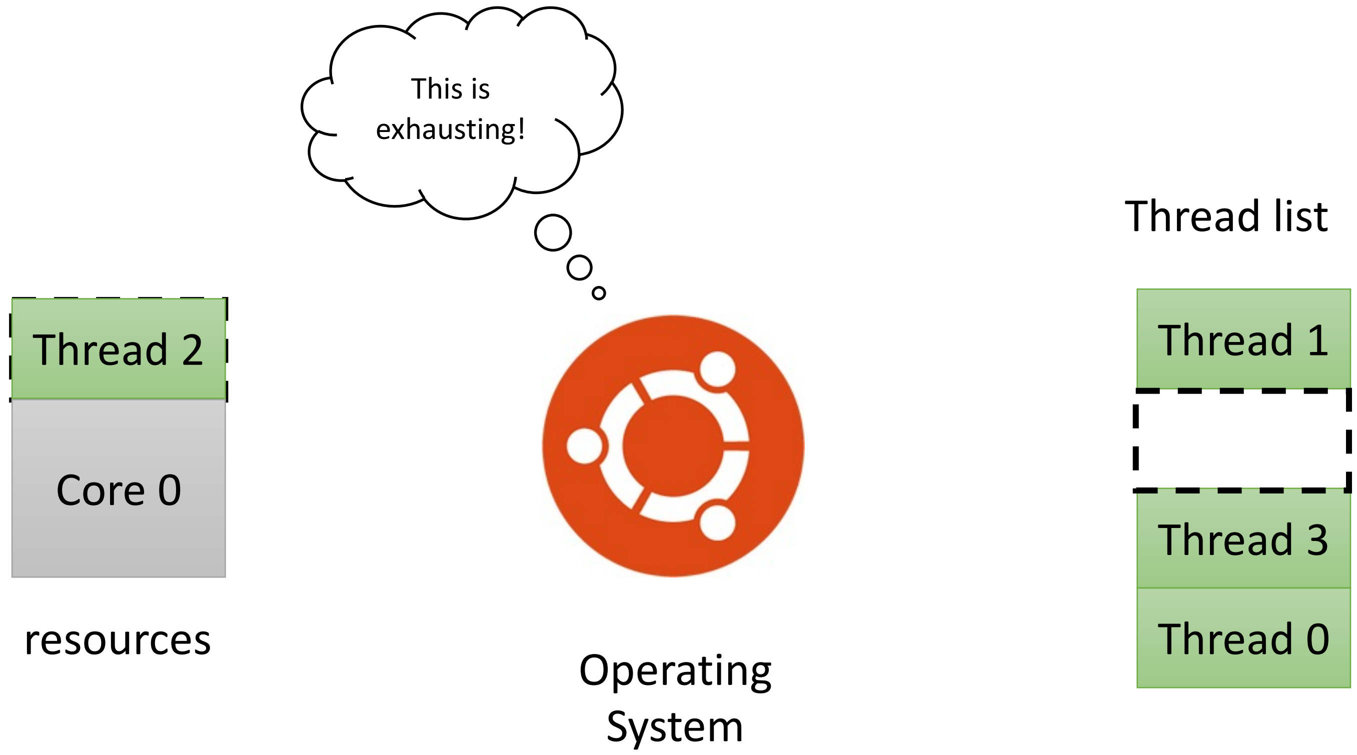
# Schedulers

- A fair scheduler typically requires preemption



# Schedulers

- A fair scheduler typically requires preemption



This is exhausting!

Thread 2

Core 0

resources

Operating System

Thread list

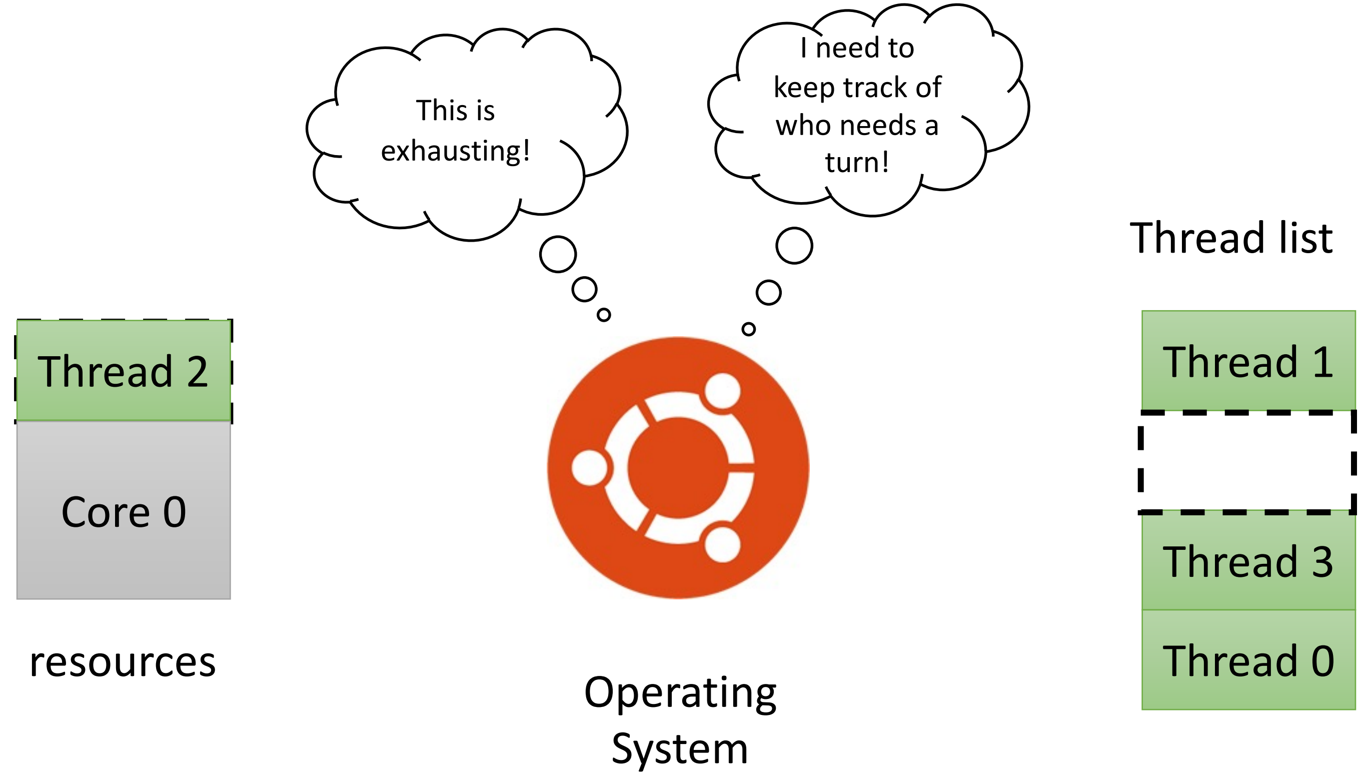
Thread 1

Thread 3

Thread 0

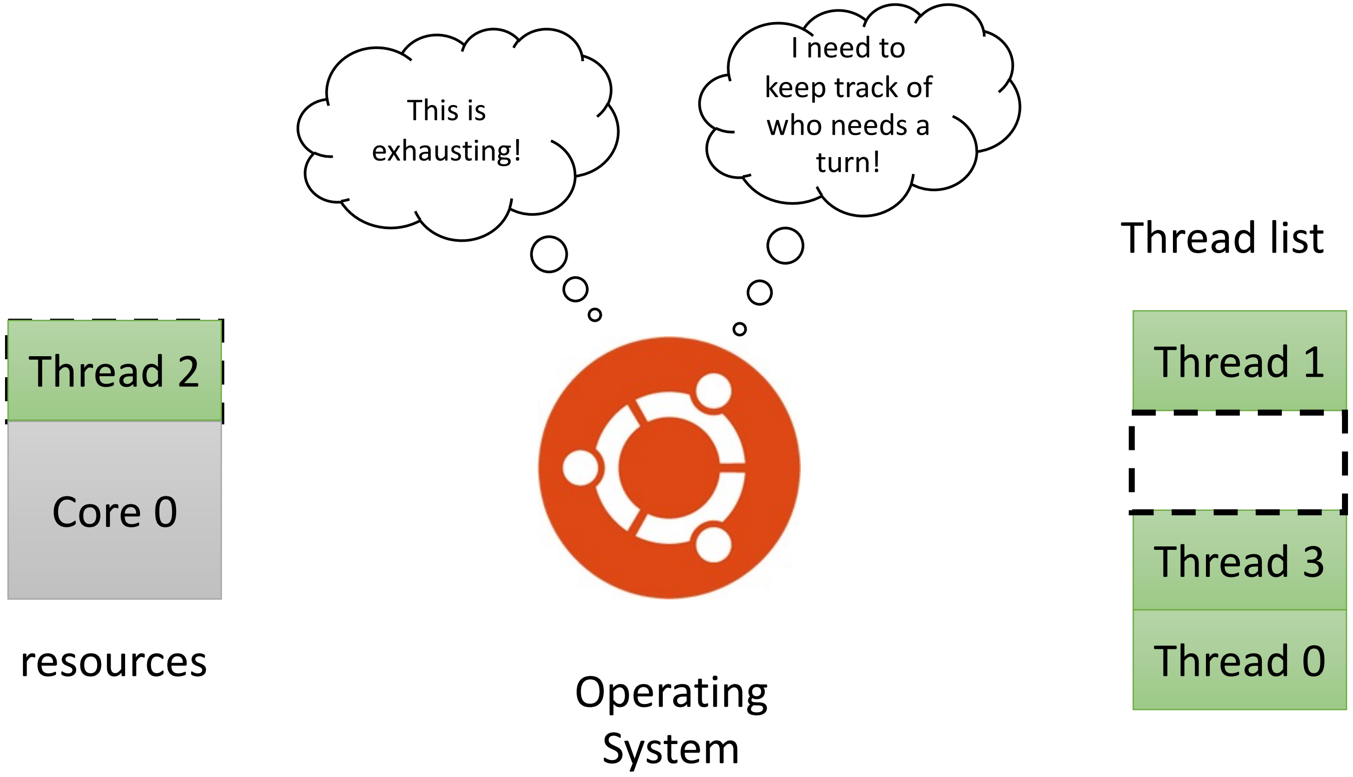
# Schedulers

- A fair scheduler typically requires preemption



# Schedulers

- A fair scheduler typically requires preemption



This is exhausting!

I need to keep track of who needs a turn!

Thread 2

Core 0

resources

Operating System

Thread list

Thread 1

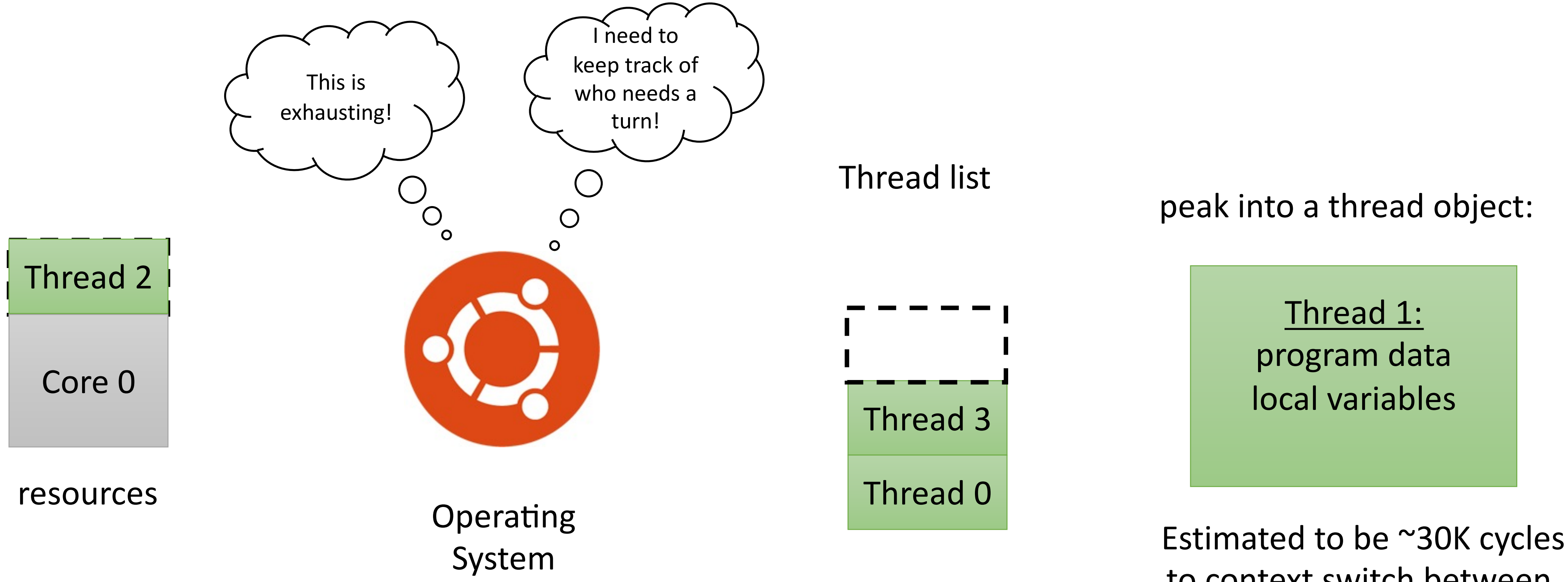
Thread 3

Thread 0

peak into a thread object:

# Schedulers

- A fair scheduler typically requires preemption



Estimated to be ~30K cycles to context switch between threads

# Schedulers

- Systems might not support preemption: e.g. GPUs
- We will study some weaker schedulers on Monday

# See you on Monday!

- Get started on HW 4

- Let us know if there are any issues with HW 2 grades

- Finishing up module 4 on Monday: weaker schedules
    - Then on to GPUs!