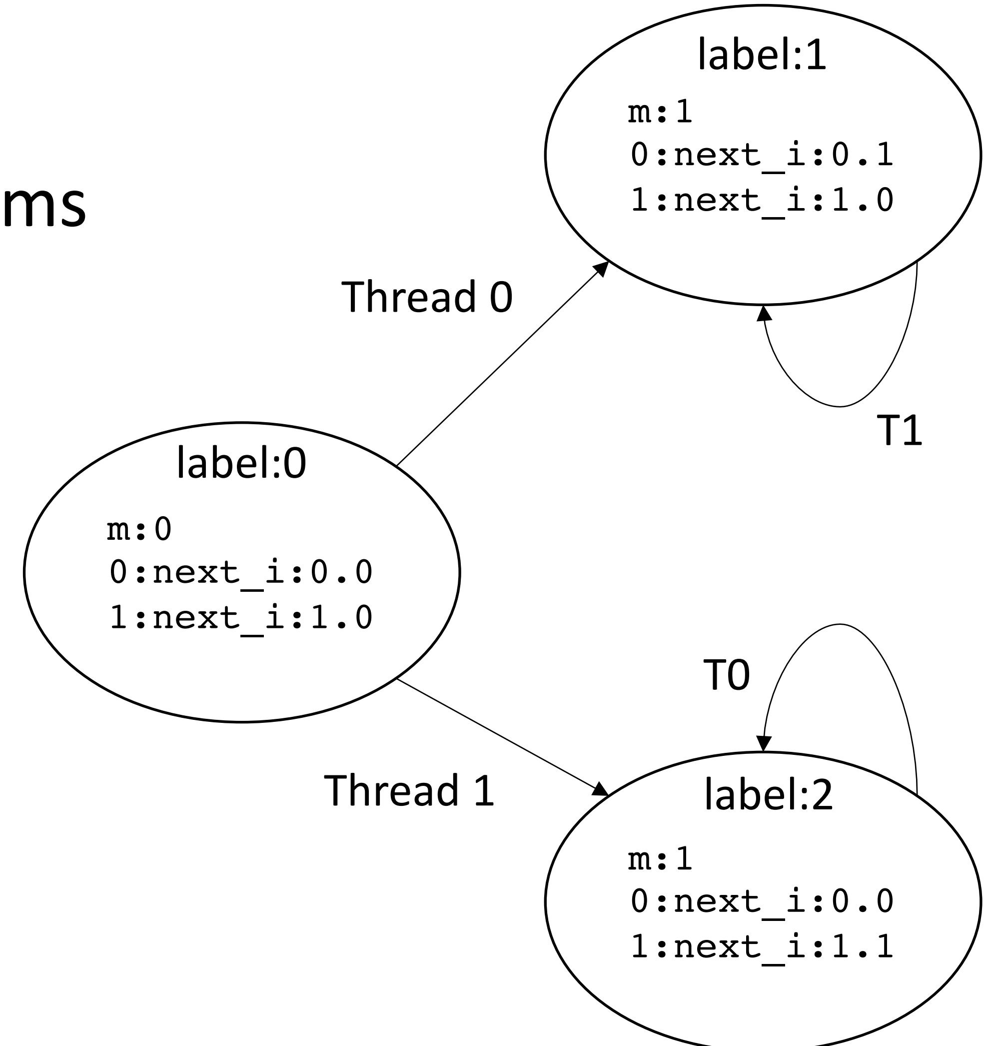
# CSE113: Parallel Programming

May 25, 2021

- **Topic**: Reasoning about concurrent programs
  - Labelled Transition Systems
  - Liveness Properties
  - Schedulers



# Announcements

- Homework 4 is posted:
  - Due June 7th
  - PLEASE do not turn this one in late!
  - Already a typo posted in piazza (will fix tonight)
- Behind schedule a bit (grading, HW assigned, etc)
  - HW 5 is canceled
  - Each HW is worth 12.5% of your final grade
  - For those interested in GPGPU programming: The book CUDA by Example is linked on the class resource page
- HW2 grades planning on being released by midnight tomorrow!

# Announcements

- SETs are out:
  - Please remember to fill them out! They are very important, especially for new classes and new faculty
  - Any other feedback is welcome! Feel free to email or discuss during office hours

# Announcements

- Today will finish up Module 4

- Last 3 lectures: heterogeneous and distributed computing:
  - 2 lectures by me about GPGPU programming
  - last lecture by Reese about distributed system programming

# Quiz

# Quiz

- Discuss answers

# Schedule

- **Labeled Transition Systems**
- Scheduler specifications

# 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);
```

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);
```

step 1 pick a thread

Keep track of next instruction 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);
```

step 1 pick a thread

Keep track of next instruction 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);
```

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);
```

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);
```

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);`

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*

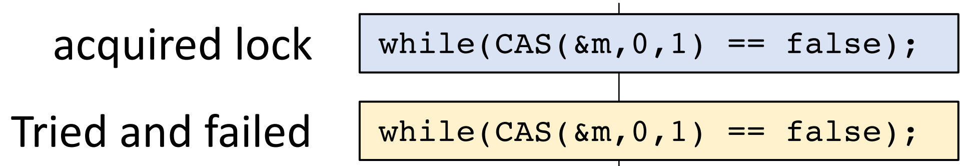
global timeline

*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

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*

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);
```

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

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*

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);
```

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

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*

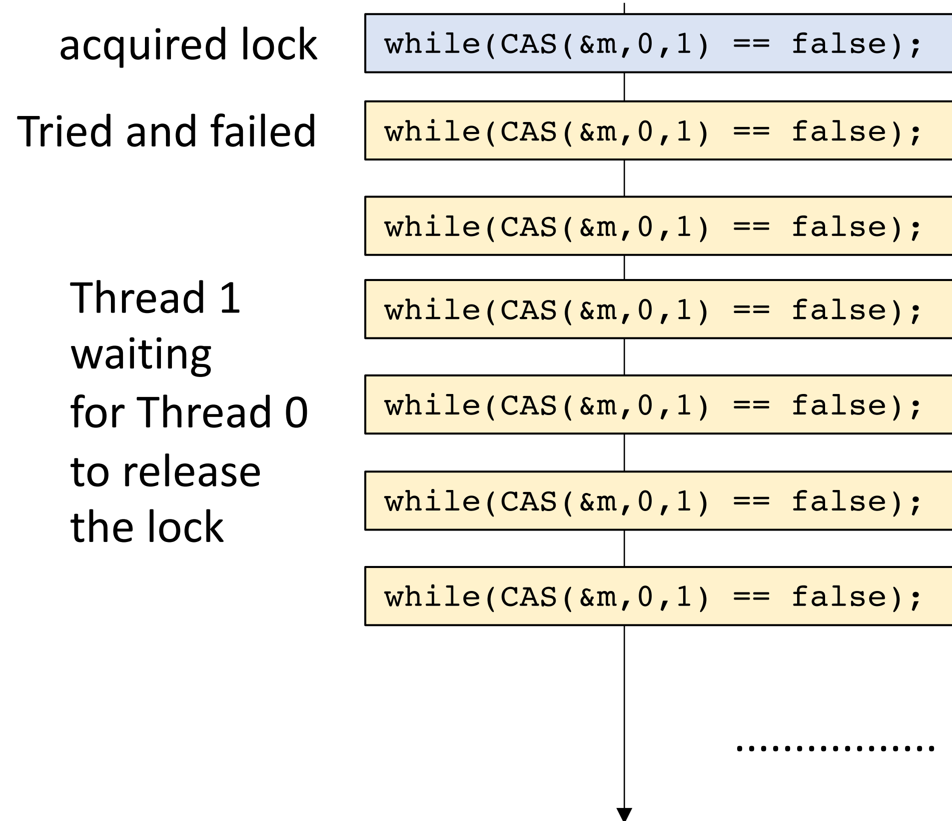
global timeline

*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
```




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

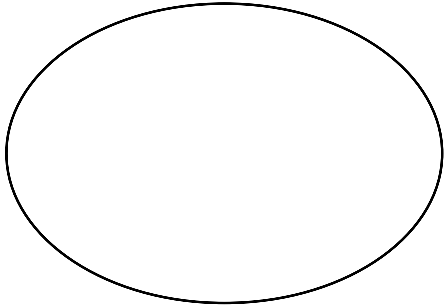
*Thread 0:*

```
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

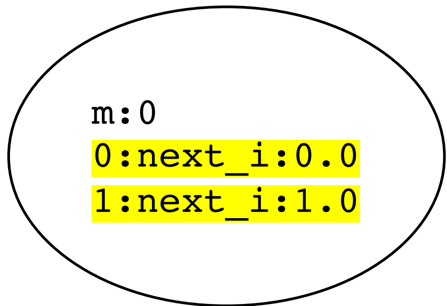
*Thread 0:*

```
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:




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

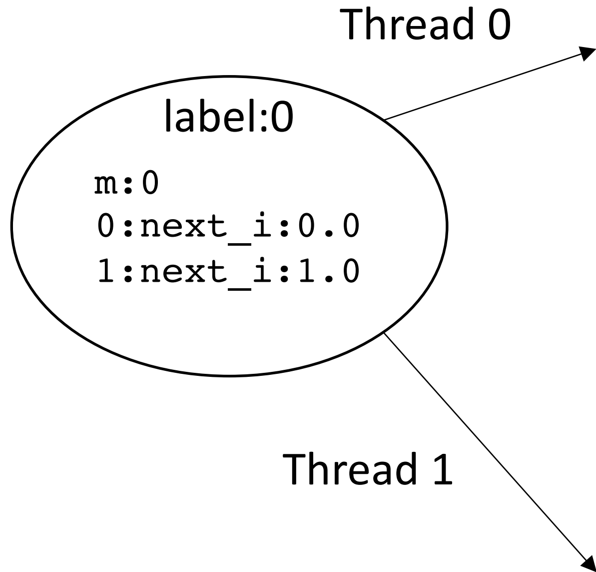
```
m:0
0:next_i:0.0
1:next_i:1.0
```

*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
```



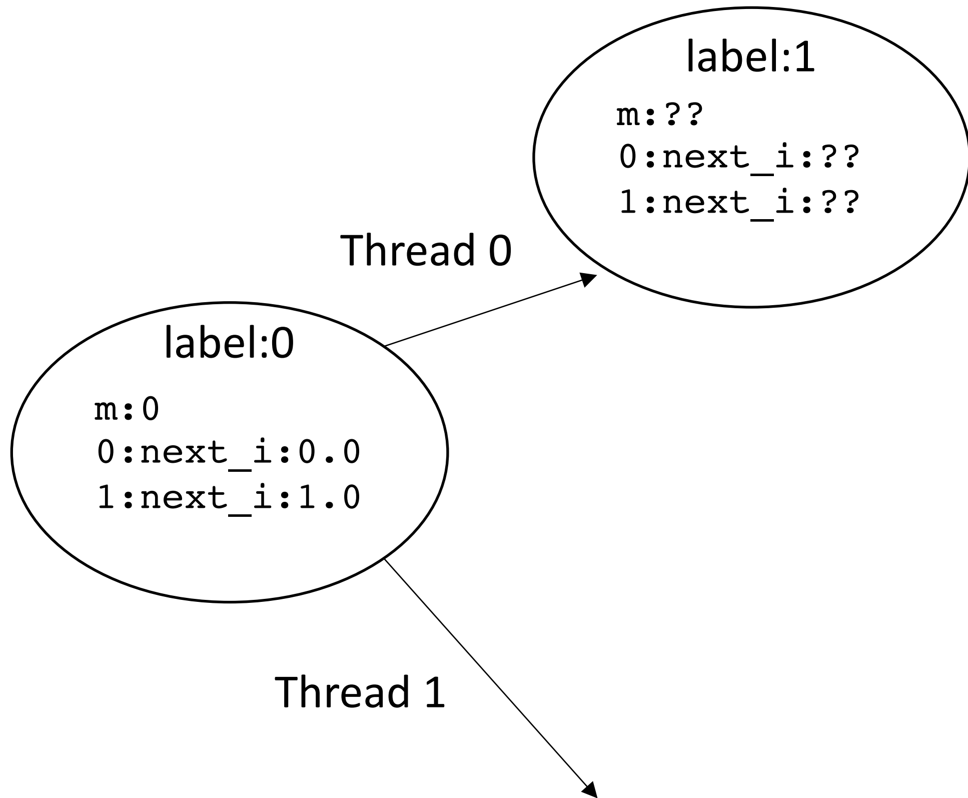

two choices:
thread 0 executes, or thread 1 executes

*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
```

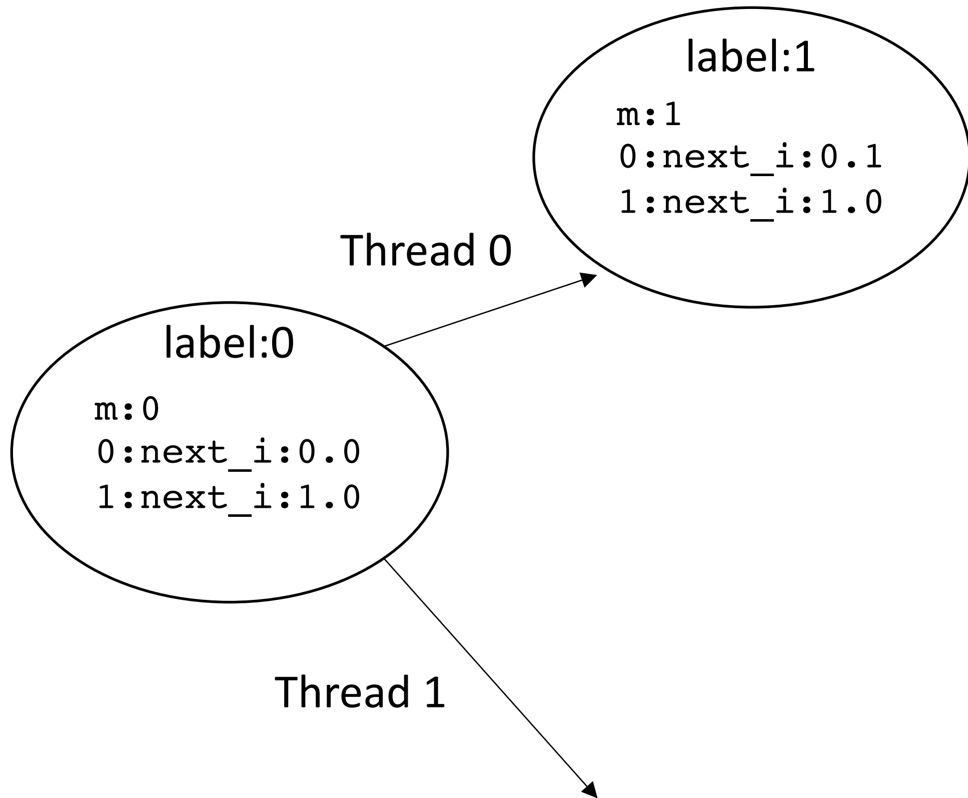


*Thread 0:*

```
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
```

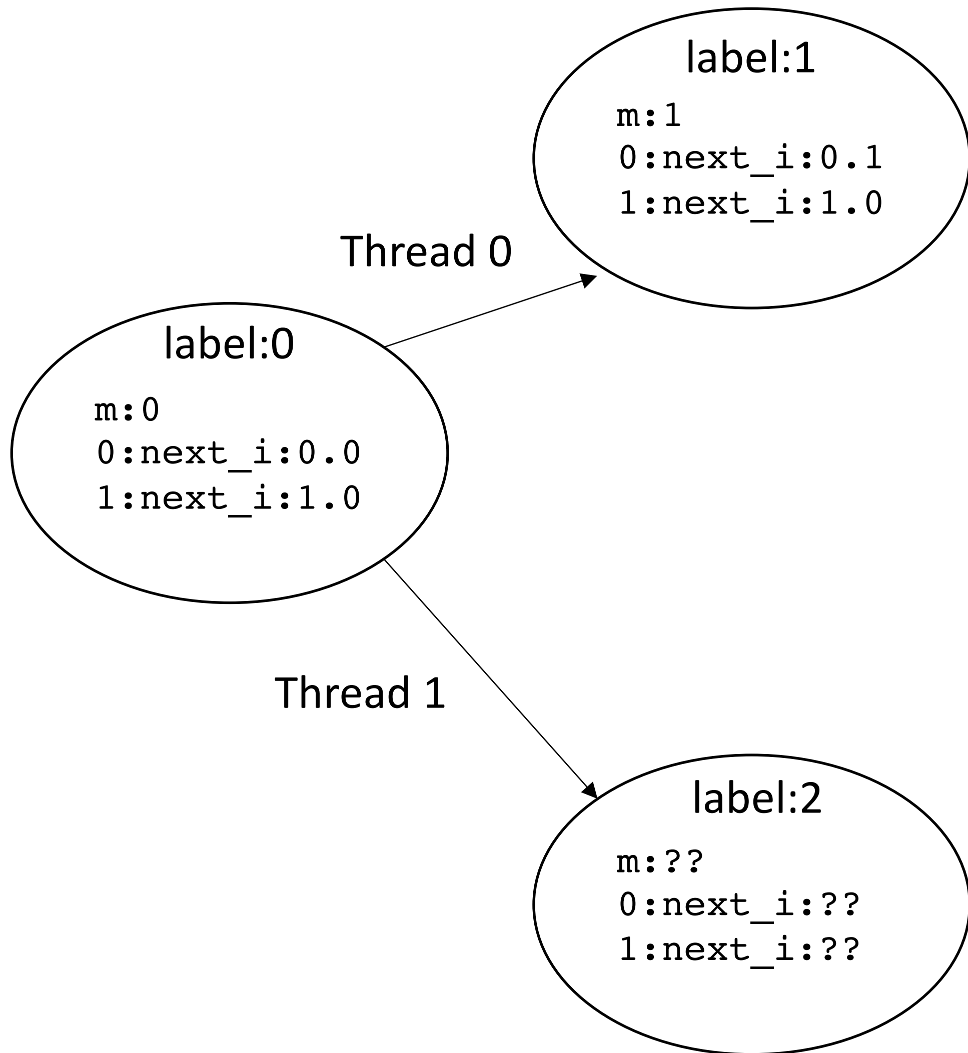


*Thread 0:*

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

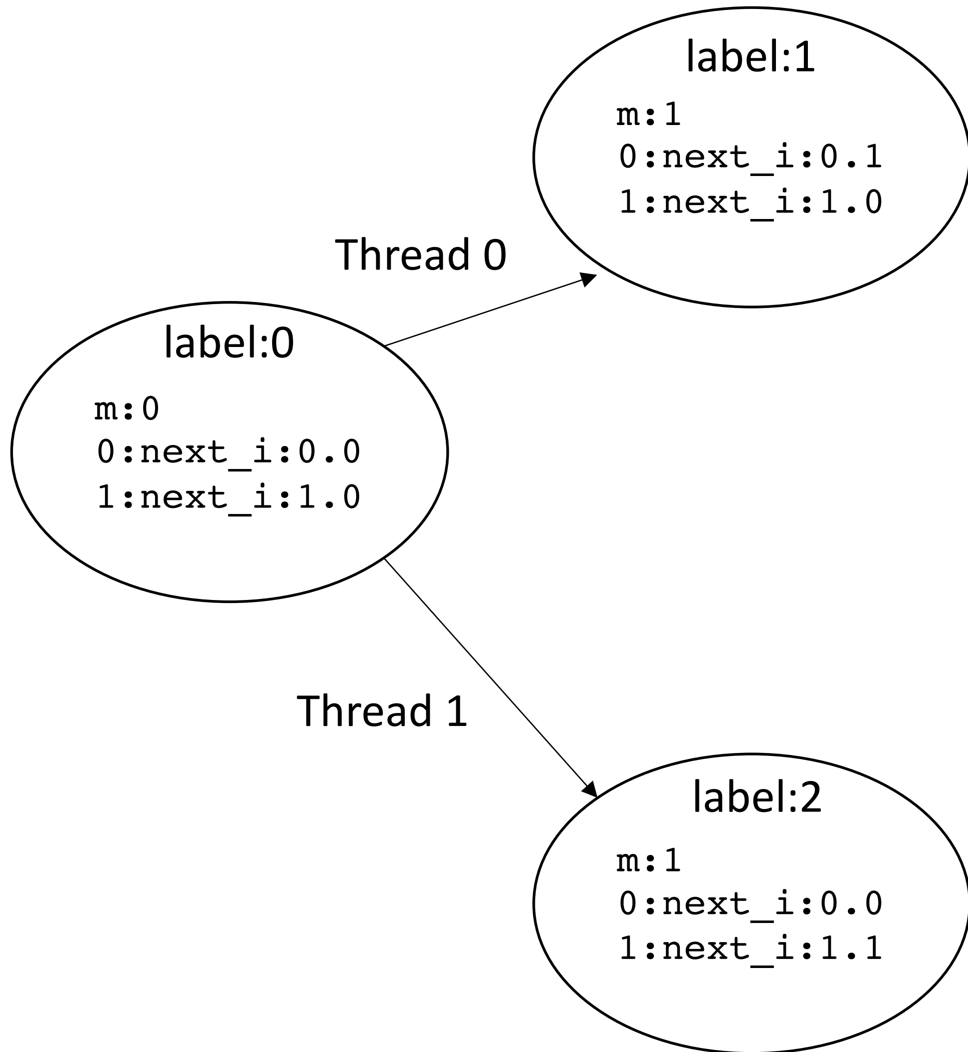
*Thread 1:*

```
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
```

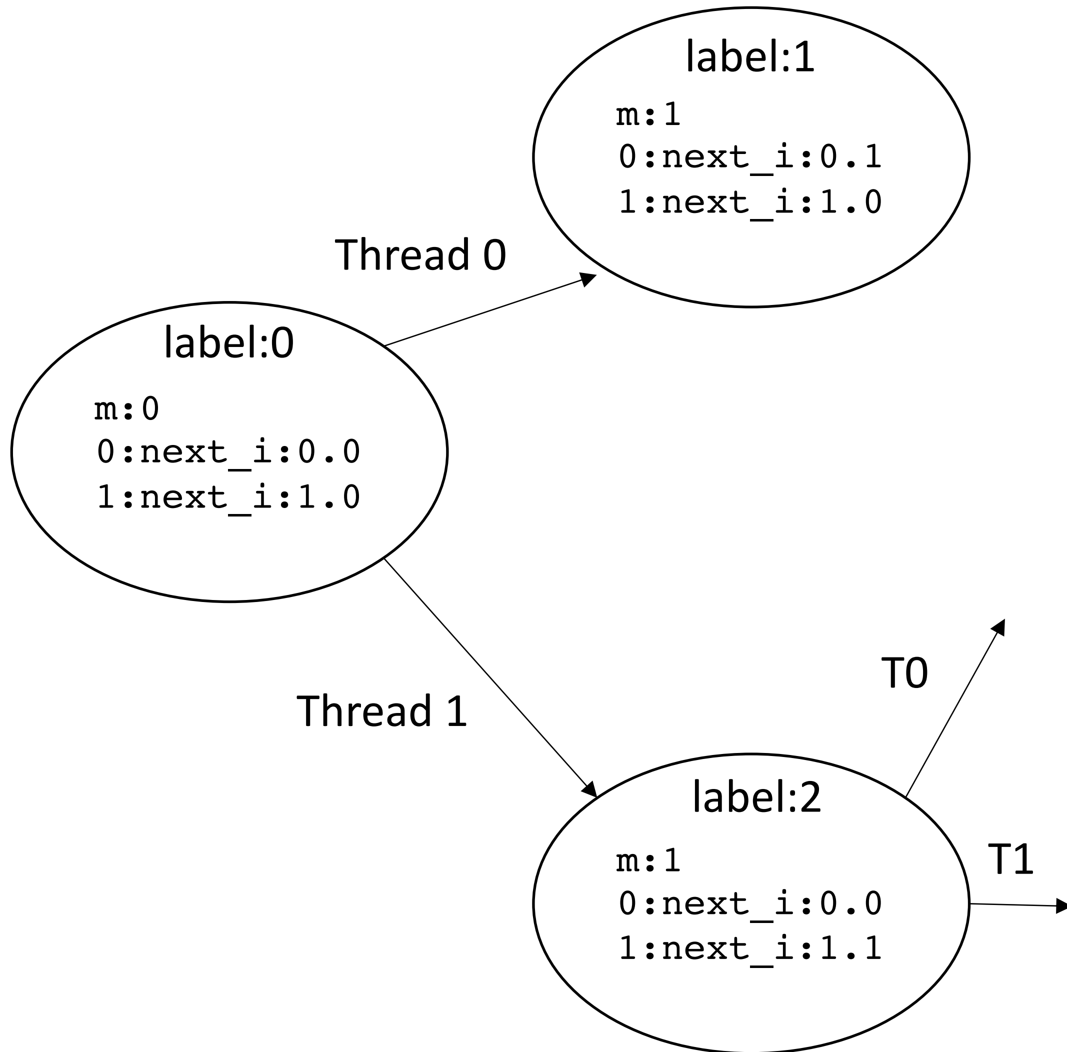


*Thread 0:*

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

*Thread 1:*

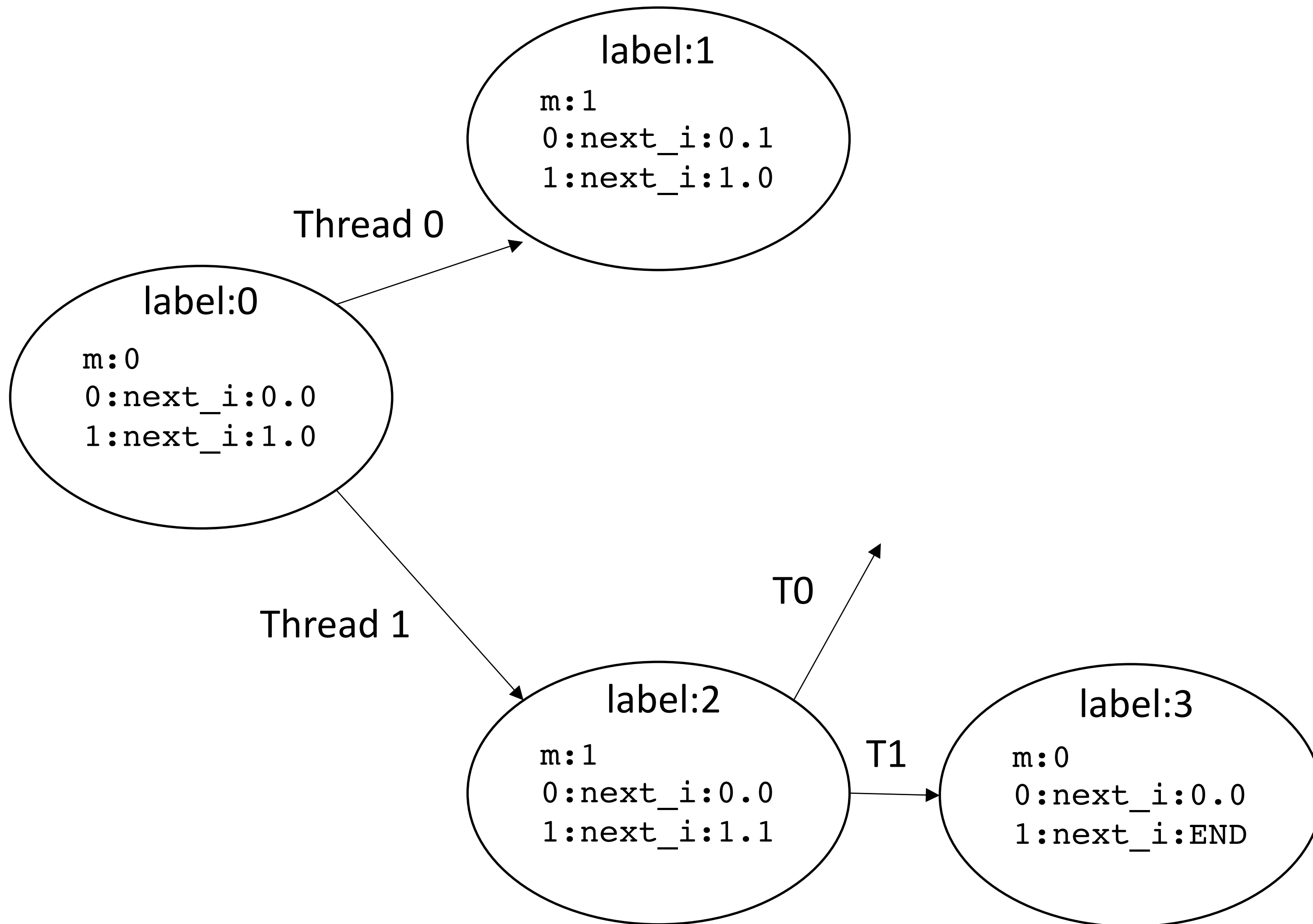
```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```




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
```



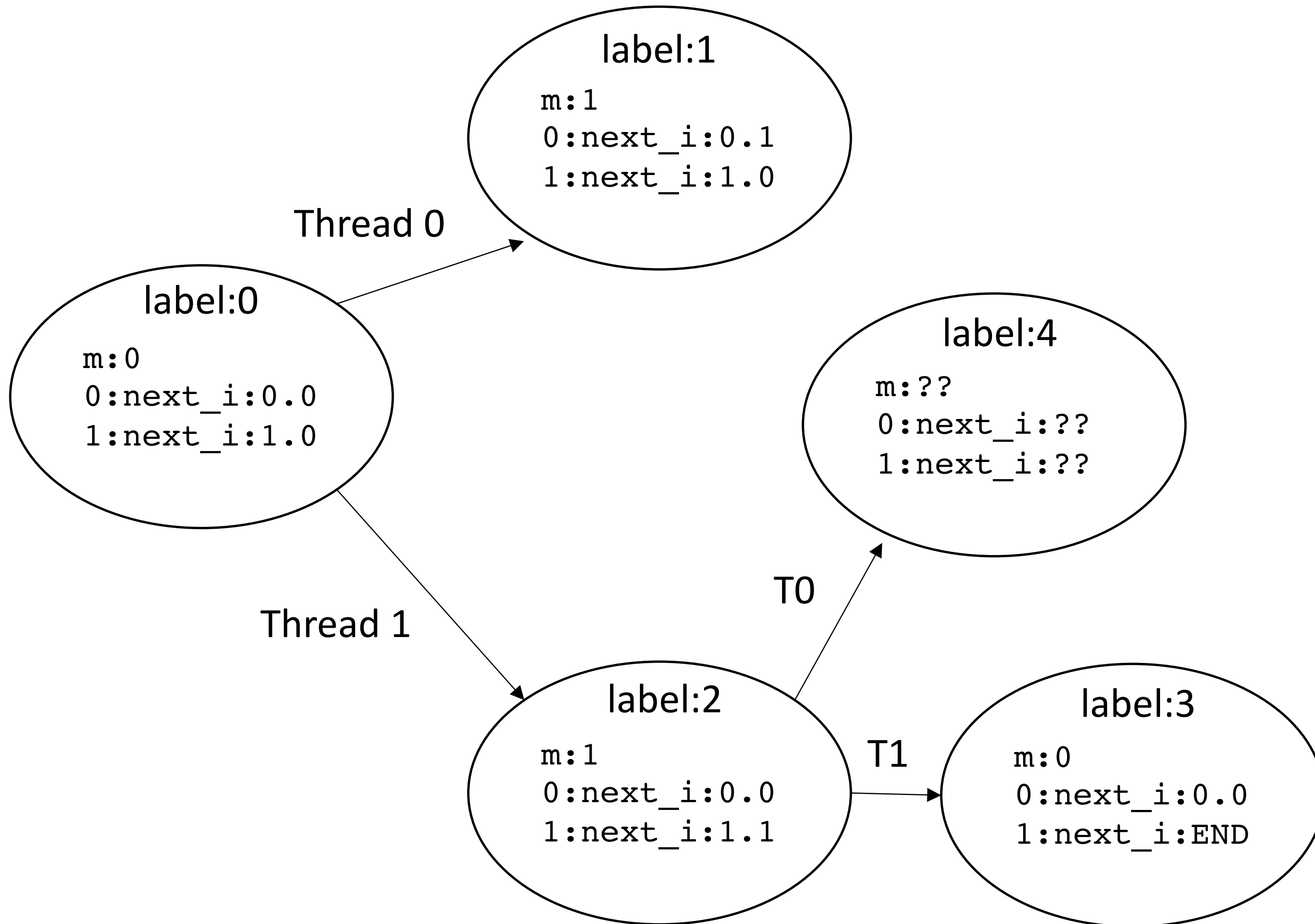

Lets do the states out of label 2

*Thread 0:*

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

*Thread 1:*

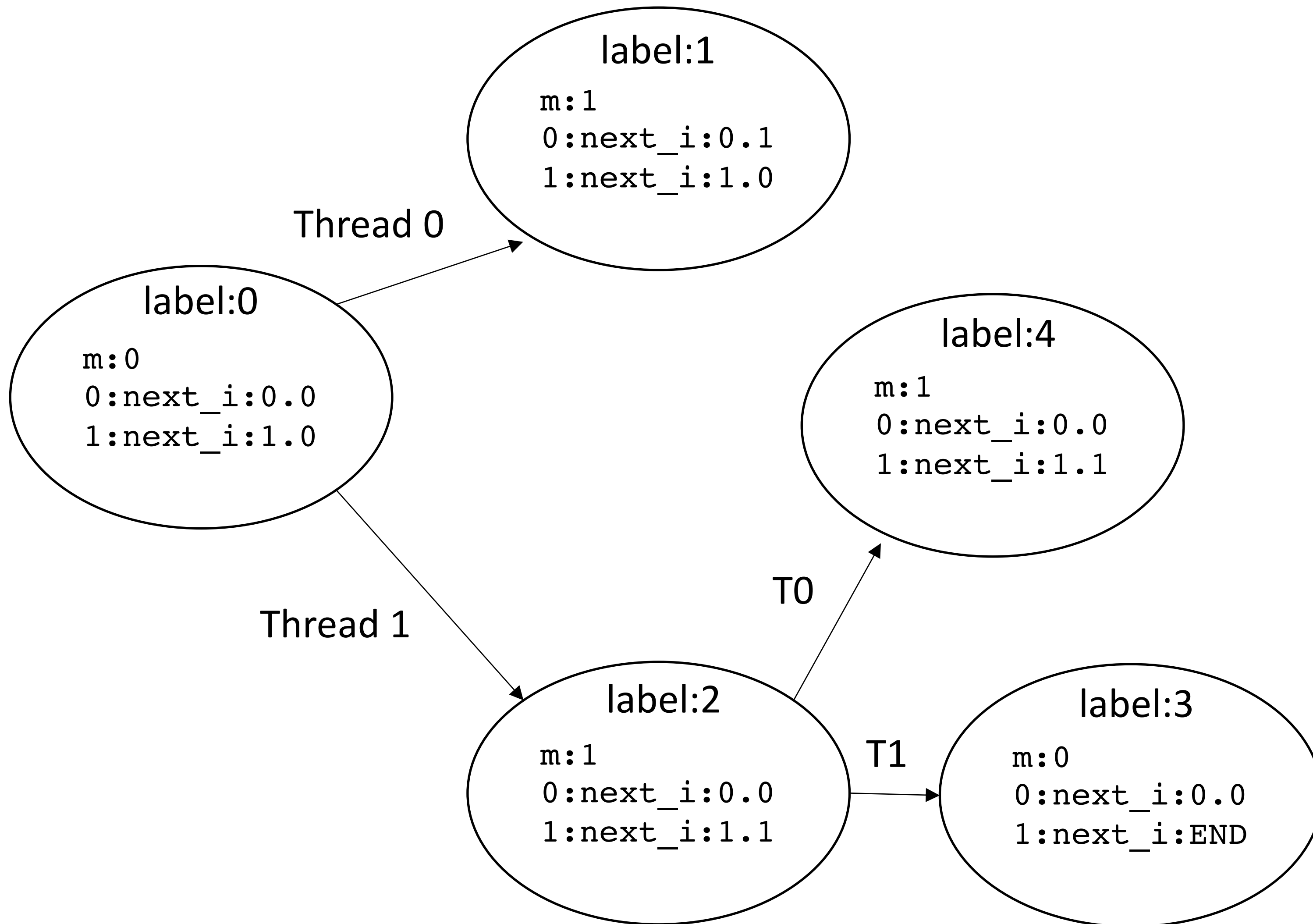
```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```




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
```



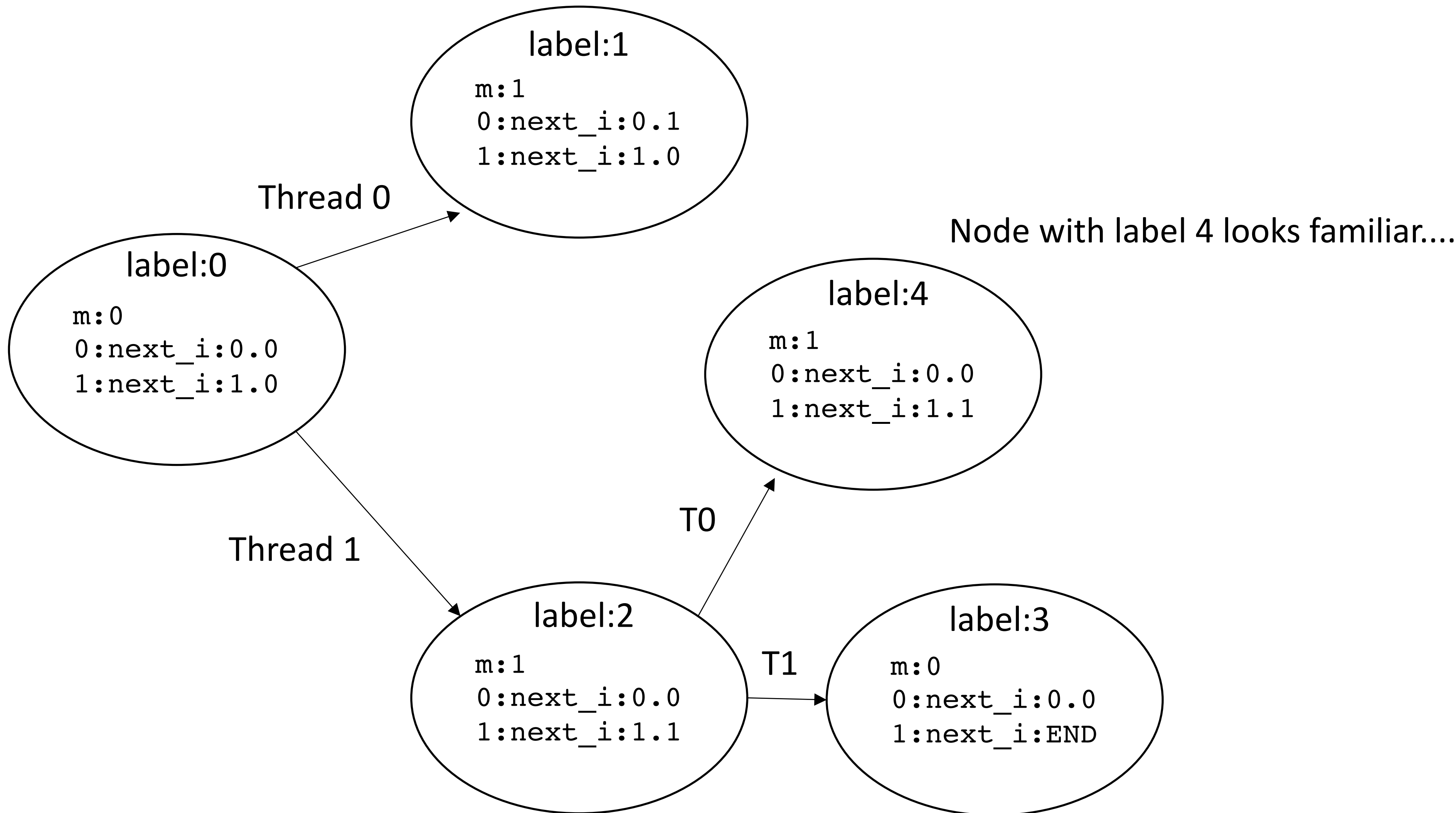

Lets do the states out of label 2

*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
```

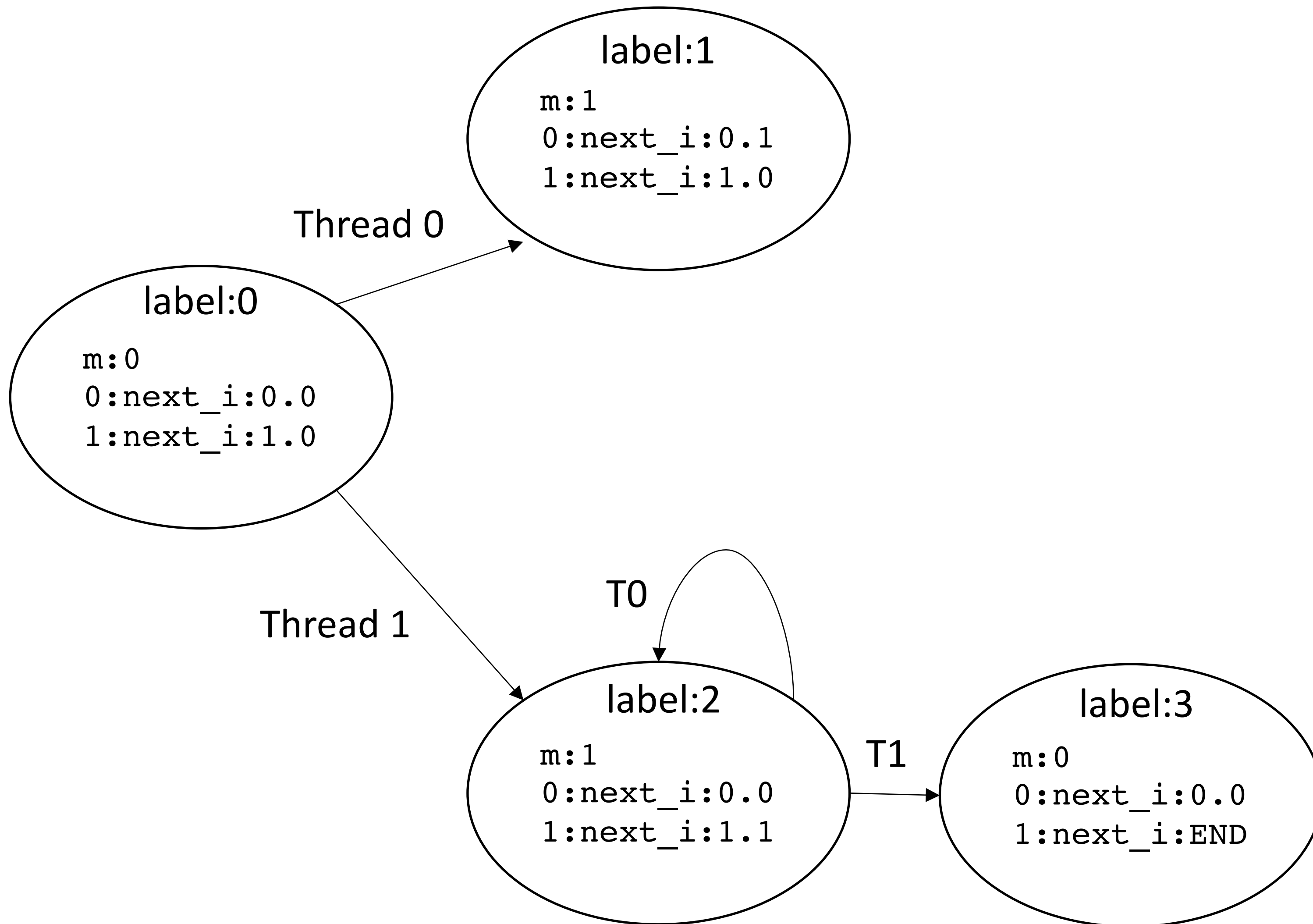


*Thread 0:*

```
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
```

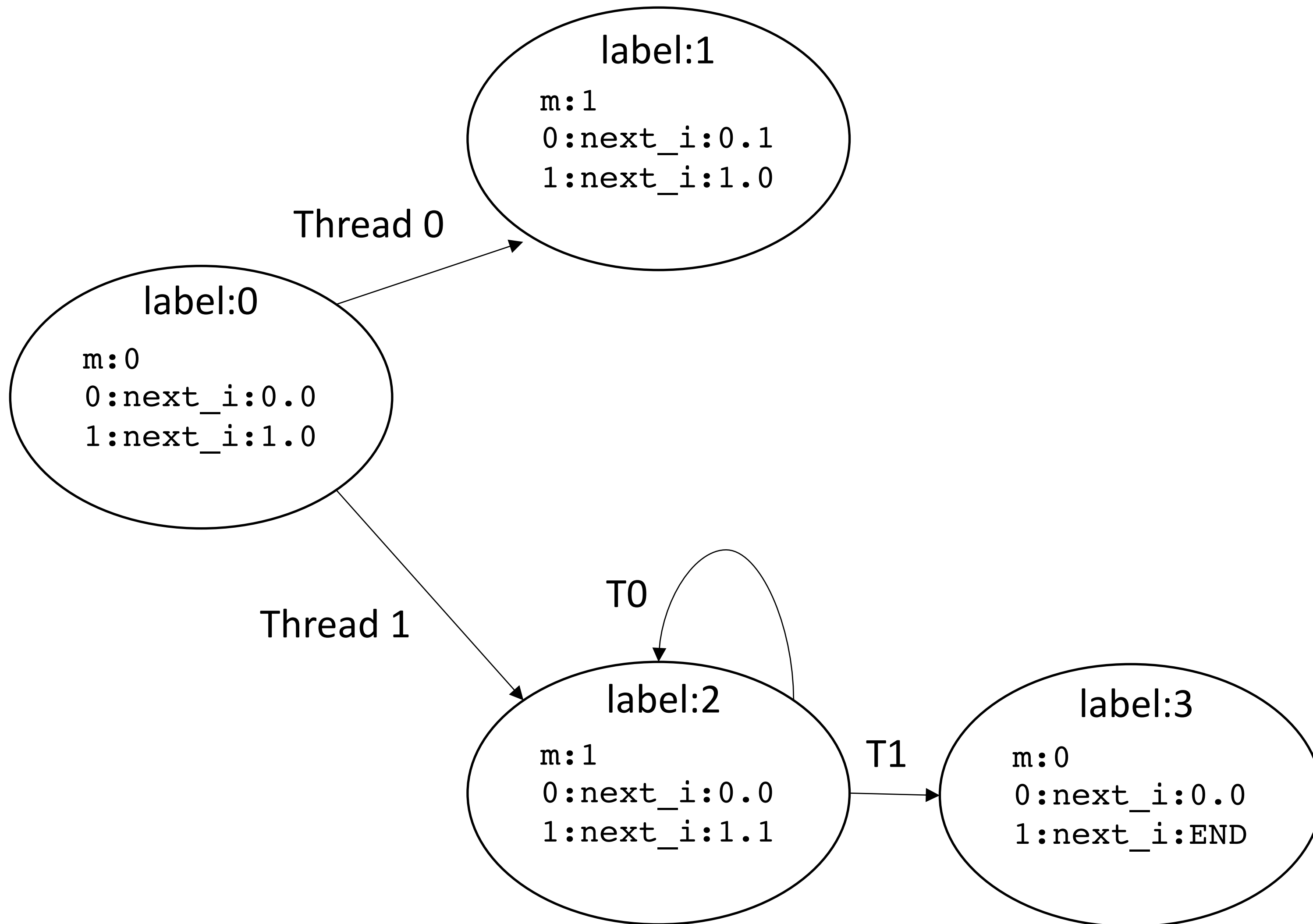


*Thread 0:*

```
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
```

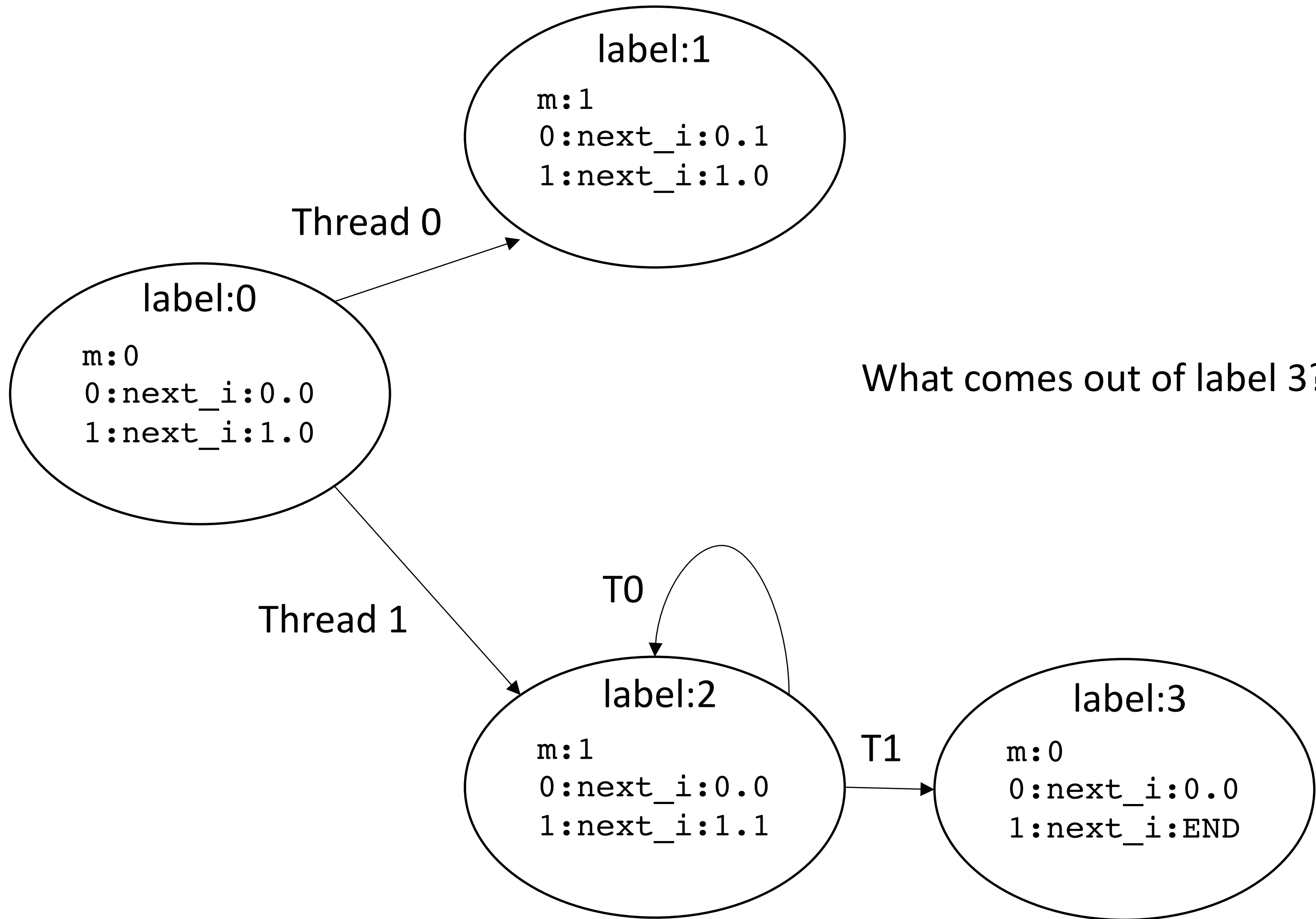


*Thread 0:*

```
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
```



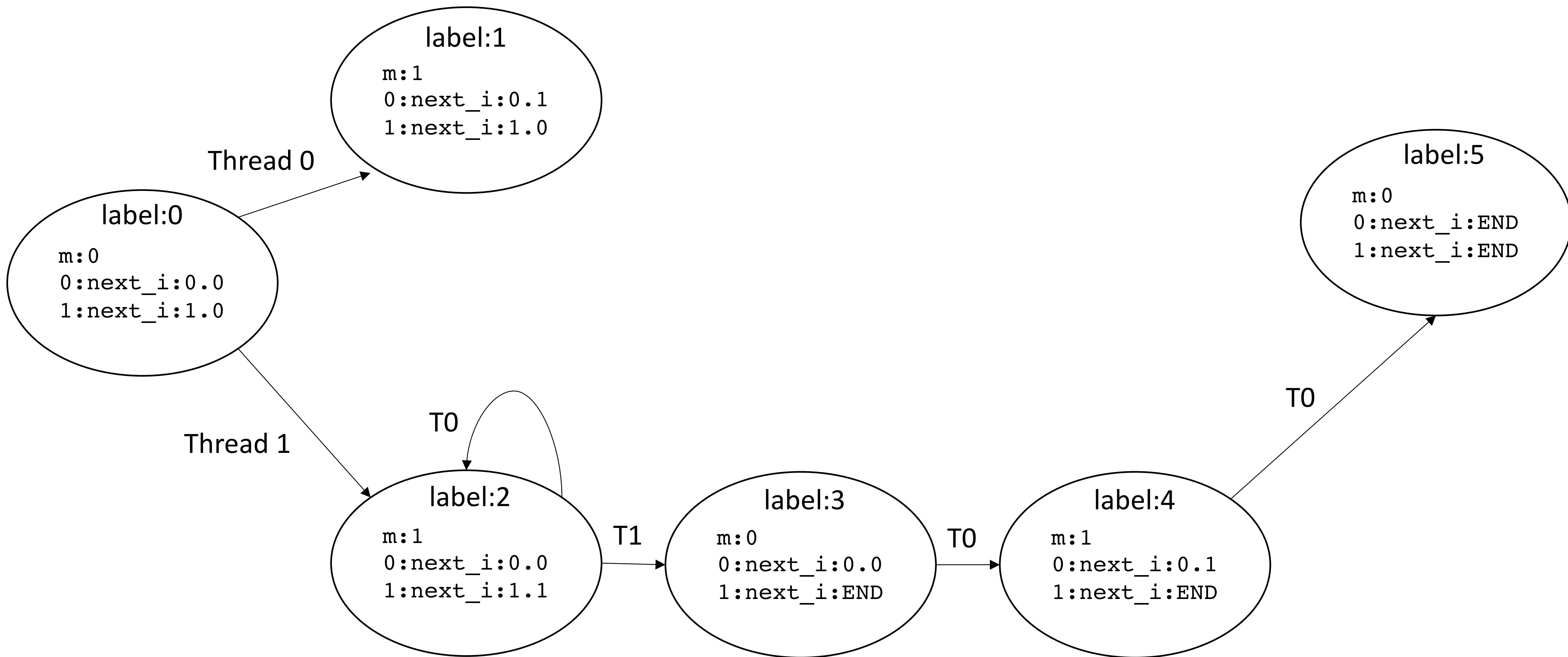

What comes out of label 3?

*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
```

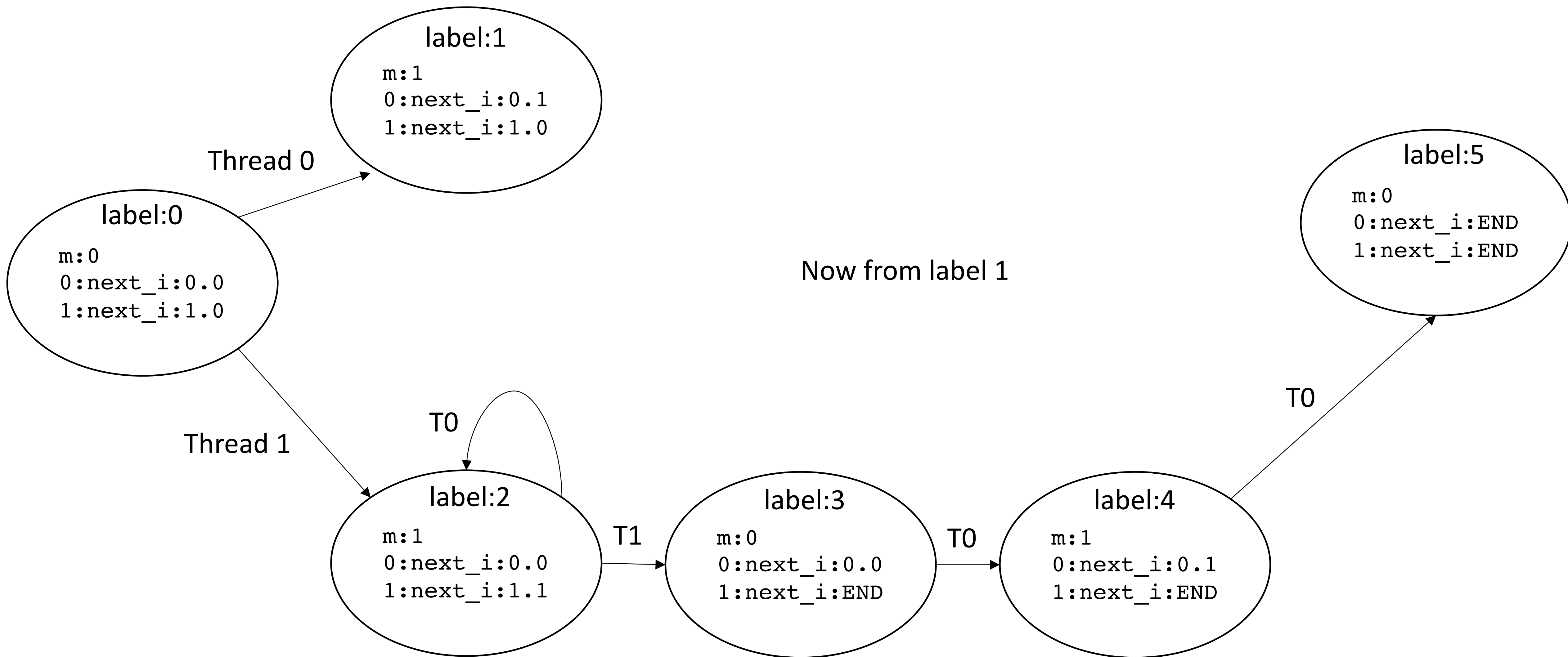


*Thread 0:*

```
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
```

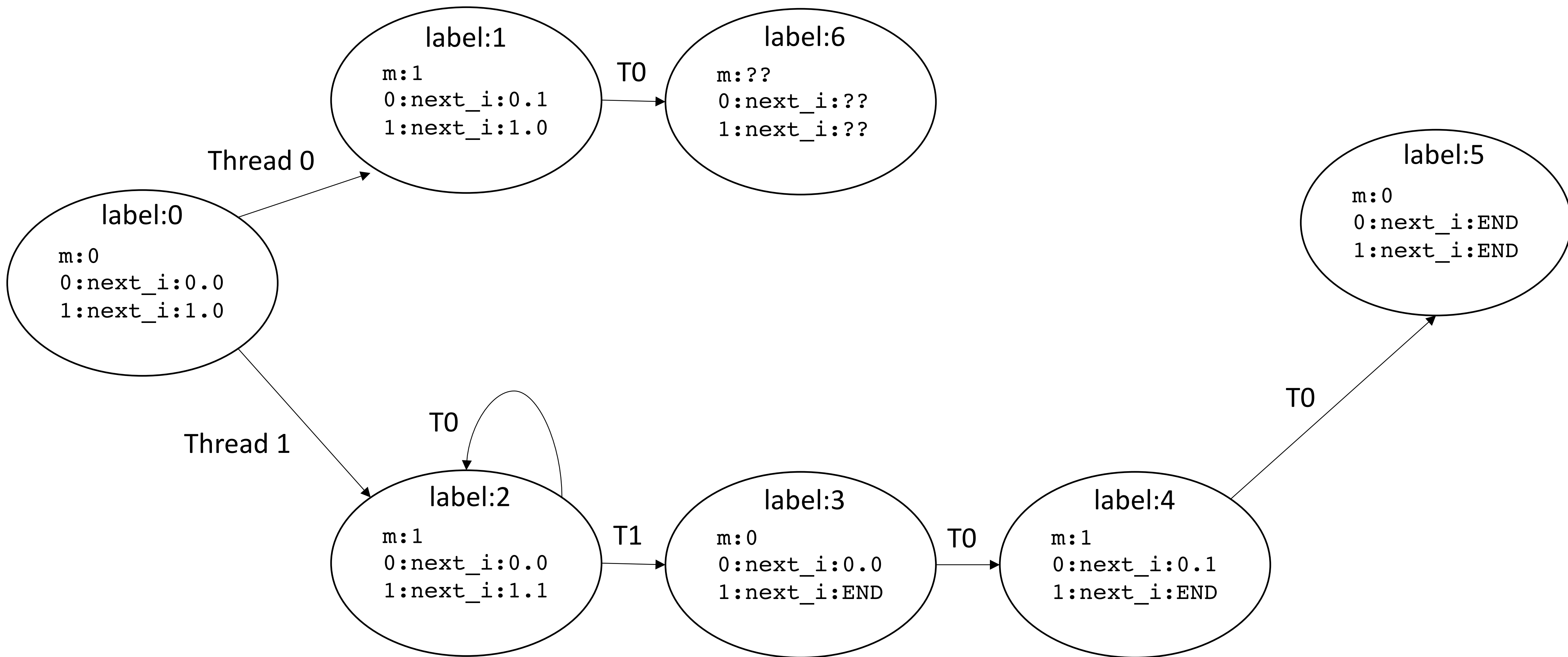


*Thread 0:*

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

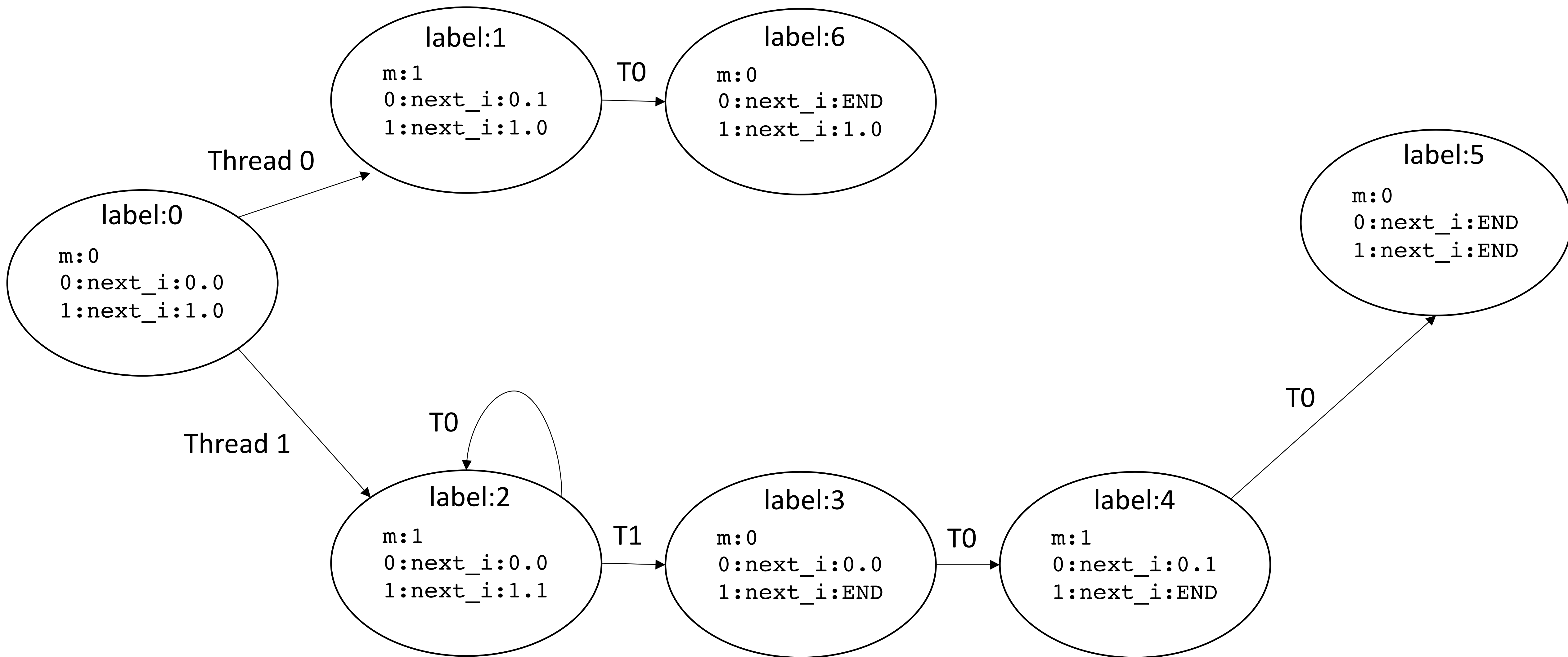
*Thread 1:*

```
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
```

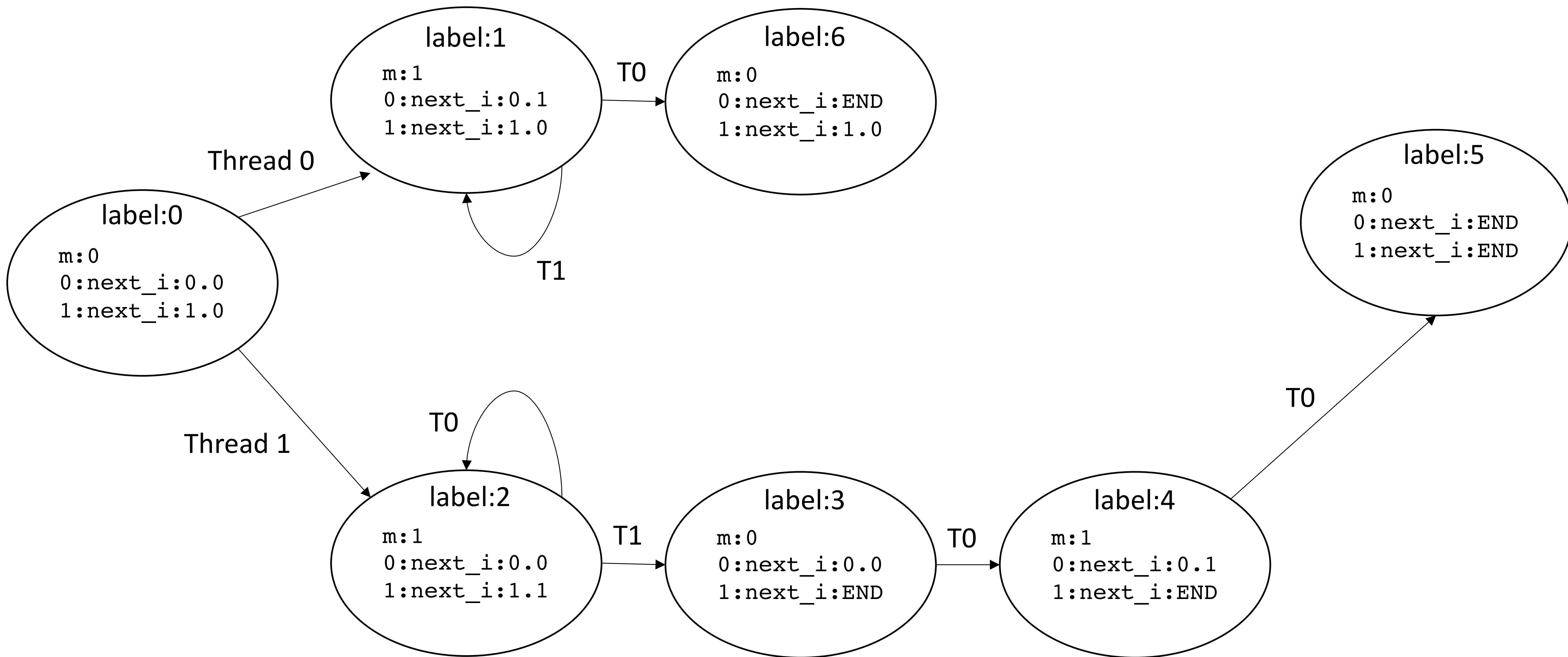


*Thread 0:*

```
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
```

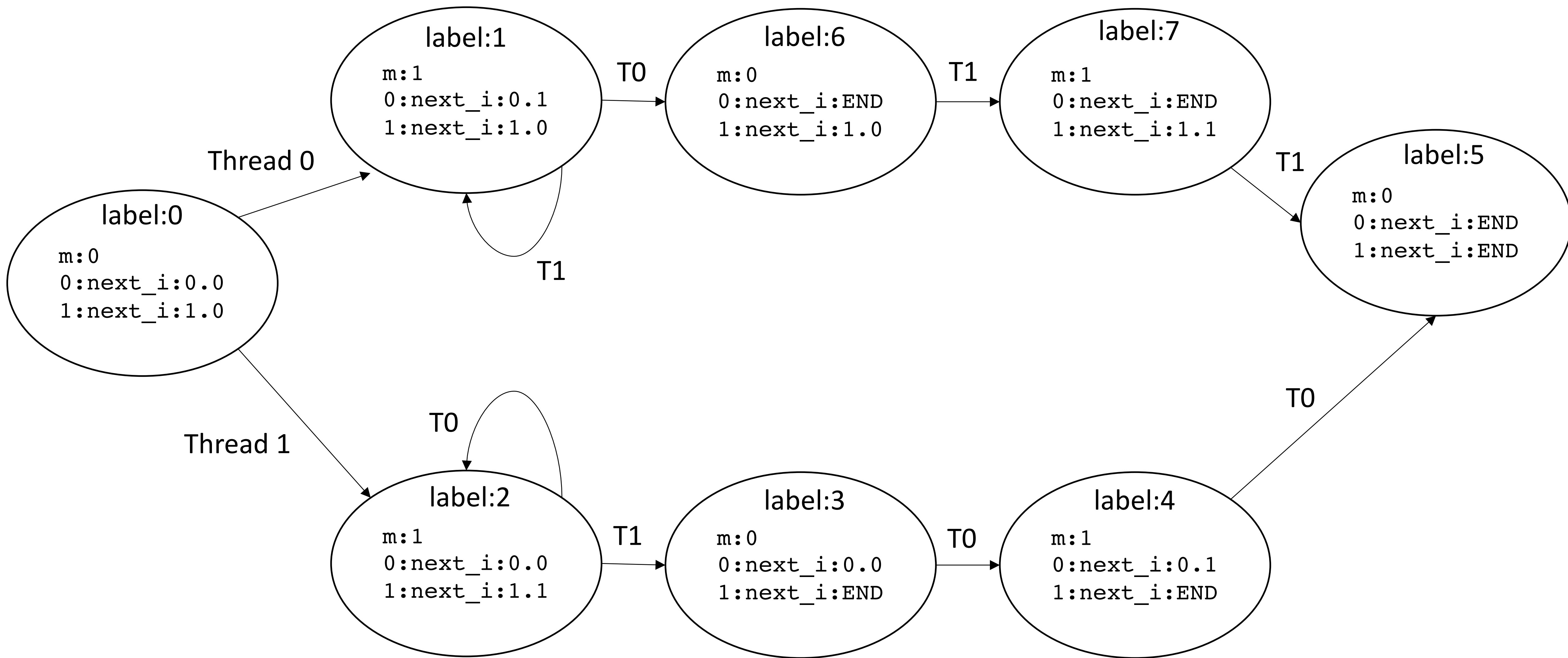


*Thread 0:*

```
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

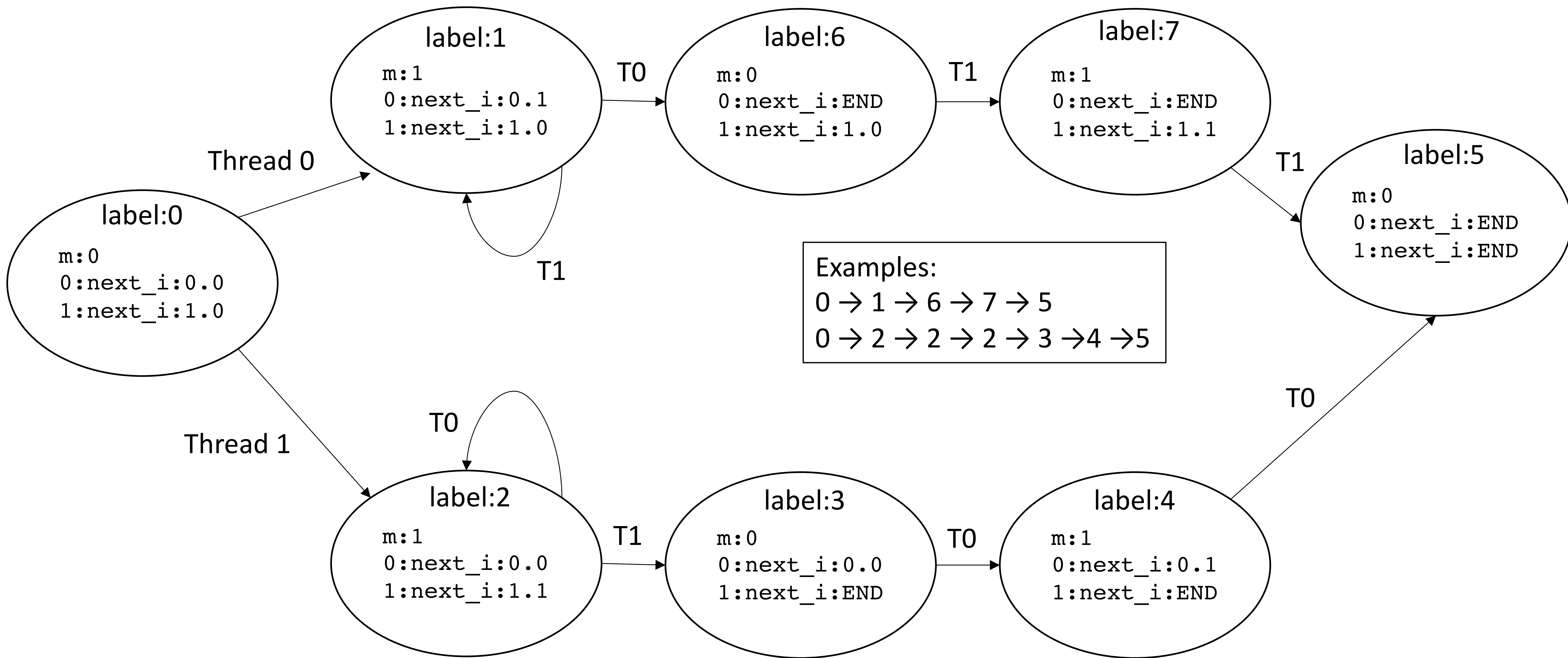
- A graph:
  - Each state encodes all variables/values and what the next instruction to execute is
  - Each edge out of a node is the different threads that can execute
  - A concurrent execution is any path through the LTS

*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
```



# What is this good for?

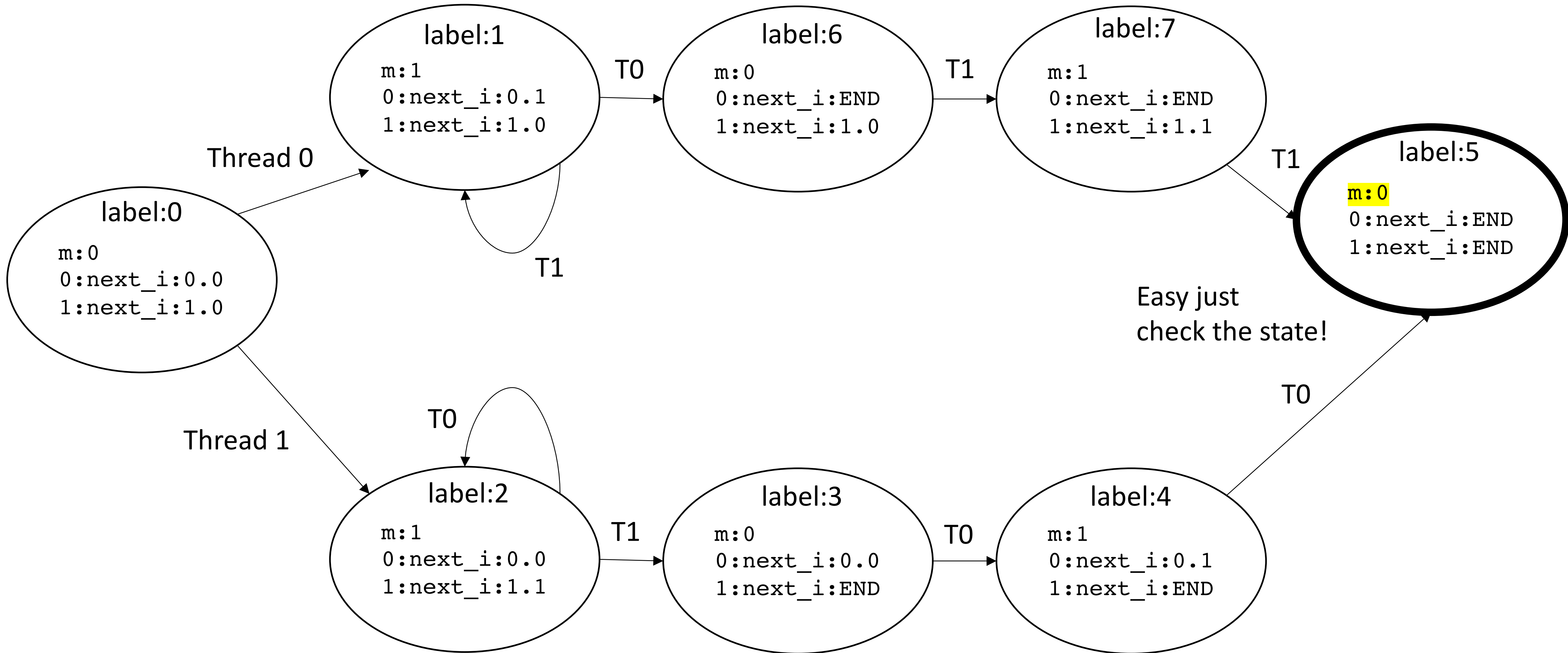
- Given this LTS, what kind of questions can we ask?
- Example:
  - At the end of the program, I want to prove that the mutex will not be taken

*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
```



# 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*

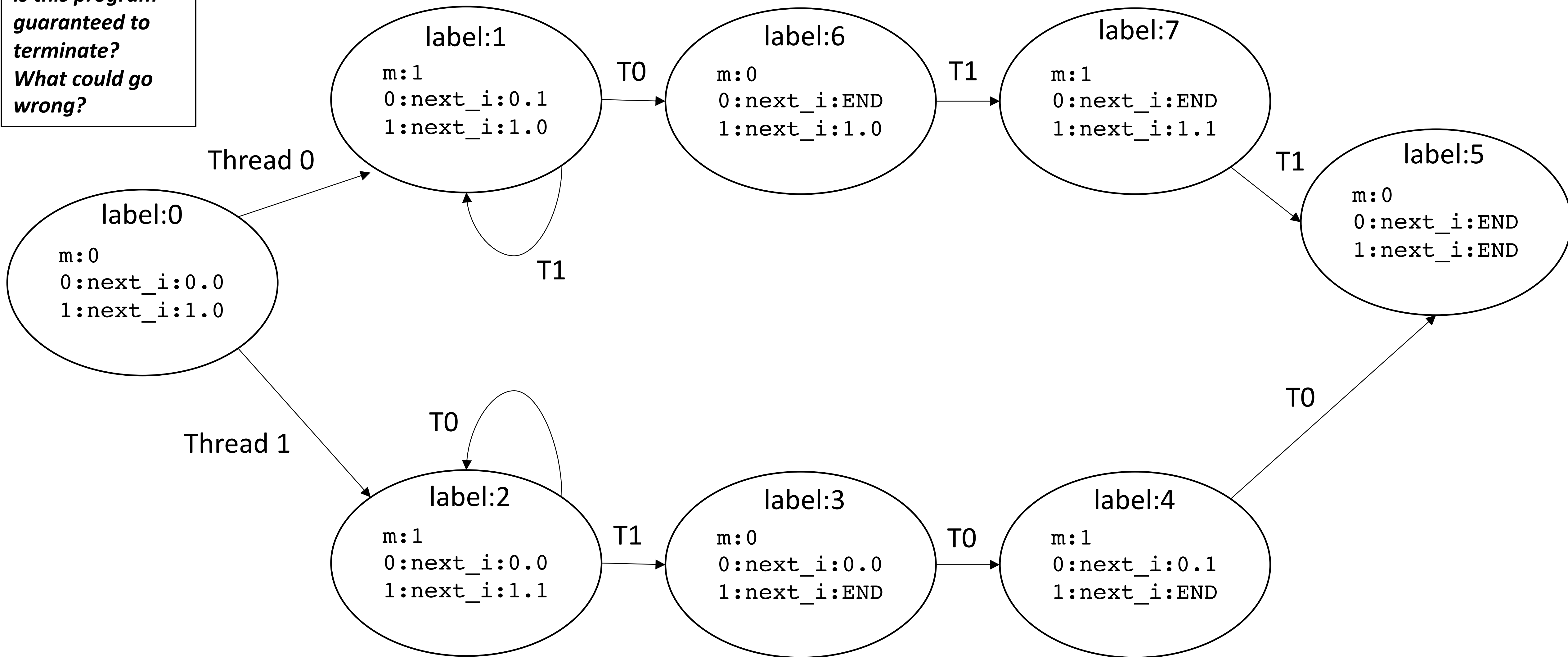
- More difficult to reason about that safety properties

*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
```

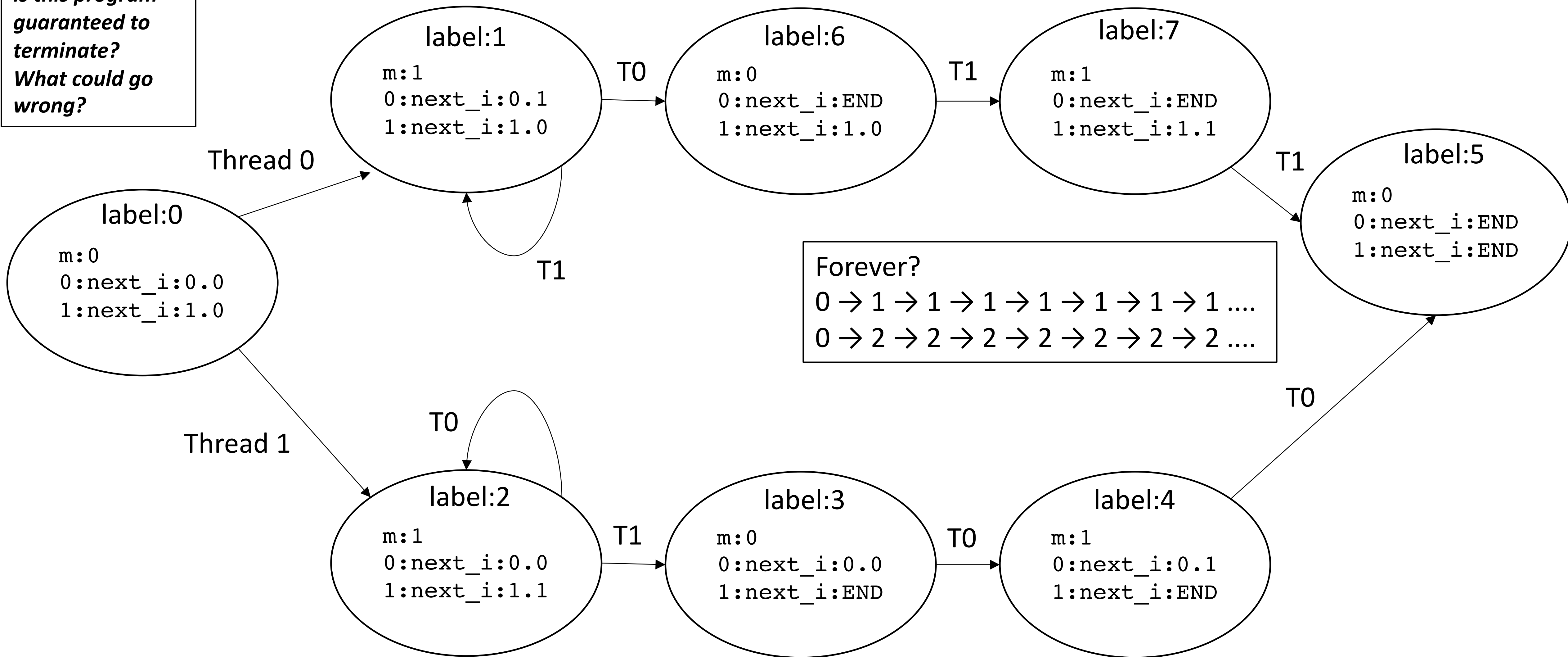


*Thread 0:*

```
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
```



# 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)

# Schedule

- Labeled Transition Systems
- **Scheduler specifications**

*5 minute break*

# The fair scheduler

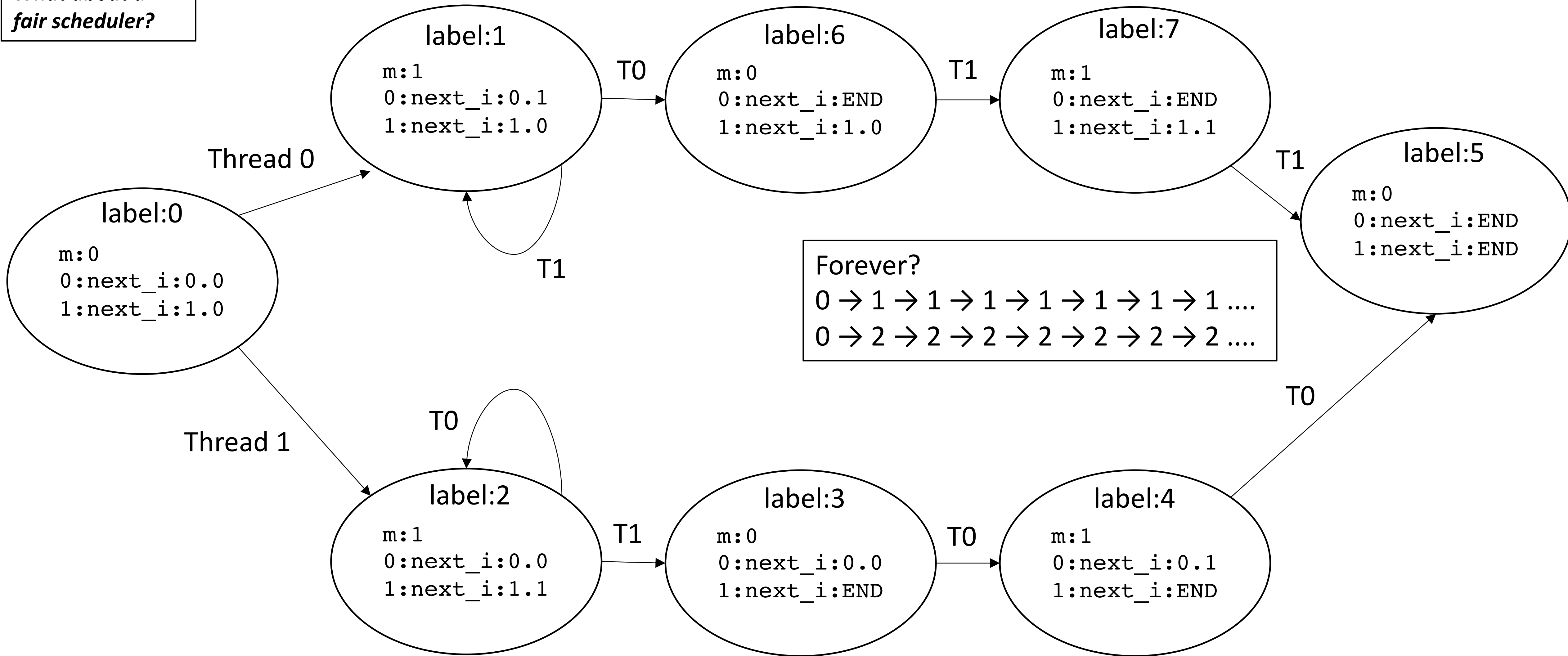
- every thread that has not terminated will “eventually” get a chance to execute.
  - “concurrent forward progress”: defined by C++
    not guaranteed, but encouraged (and likely what you will observe)
  - “weakly fair scheduler”: defined by classic concurrency textbooks
- The fair scheduler disallows 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
```

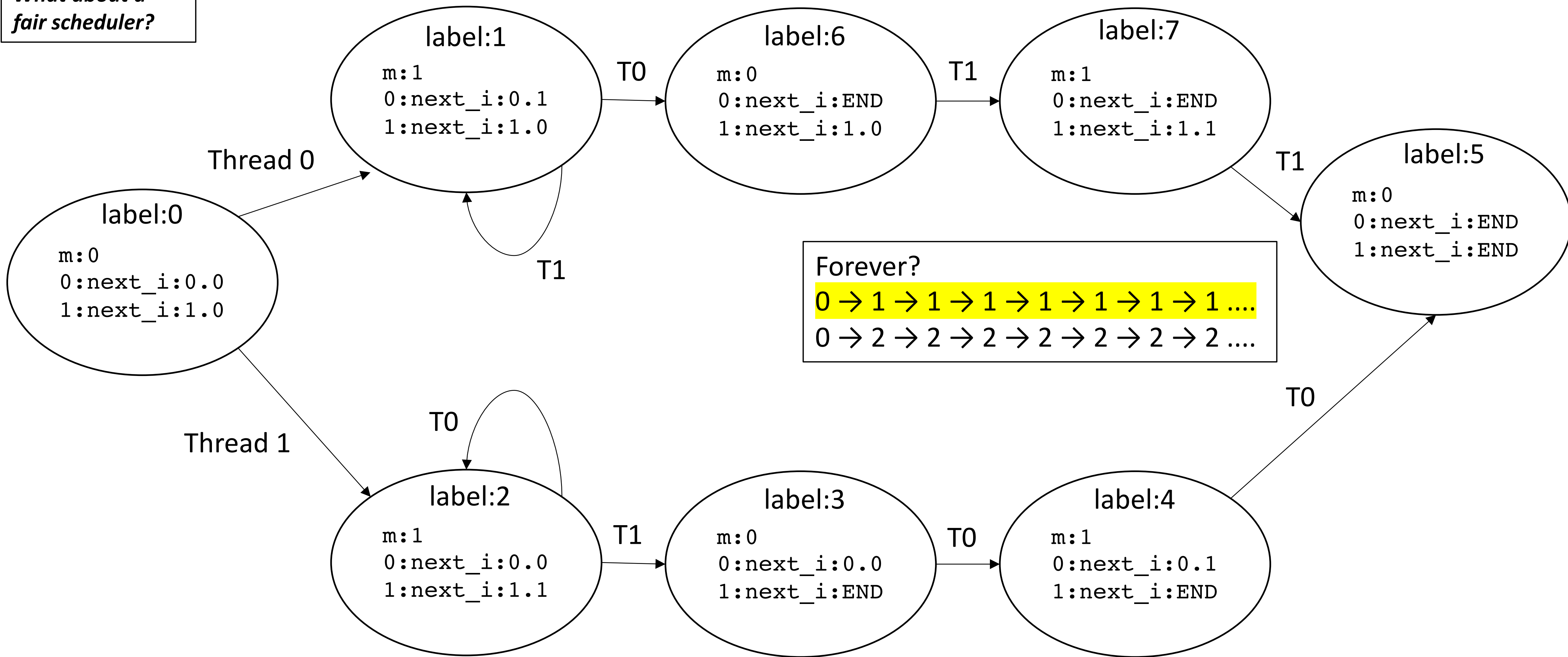


*Thread 0:*

```
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
```

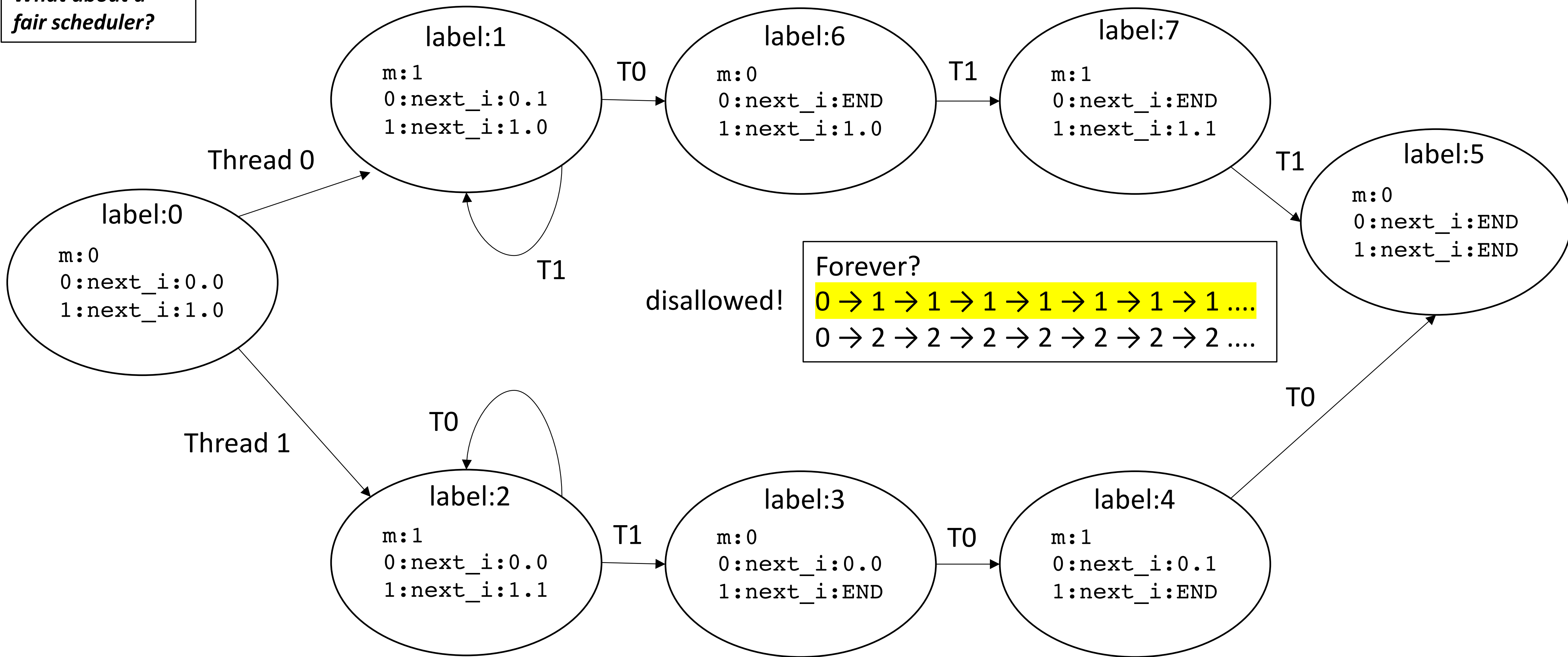


*Thread 0:*

```
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
```

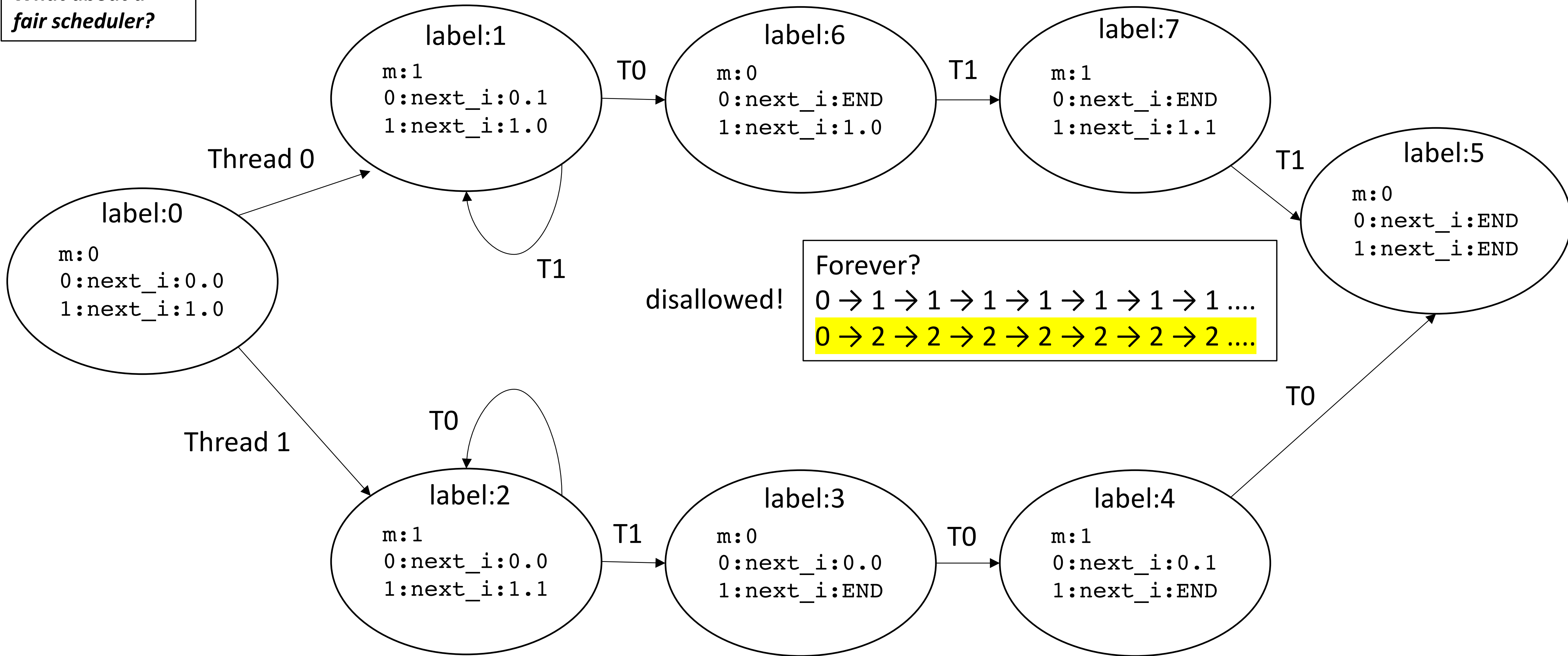


*Thread 0:*

```
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
```

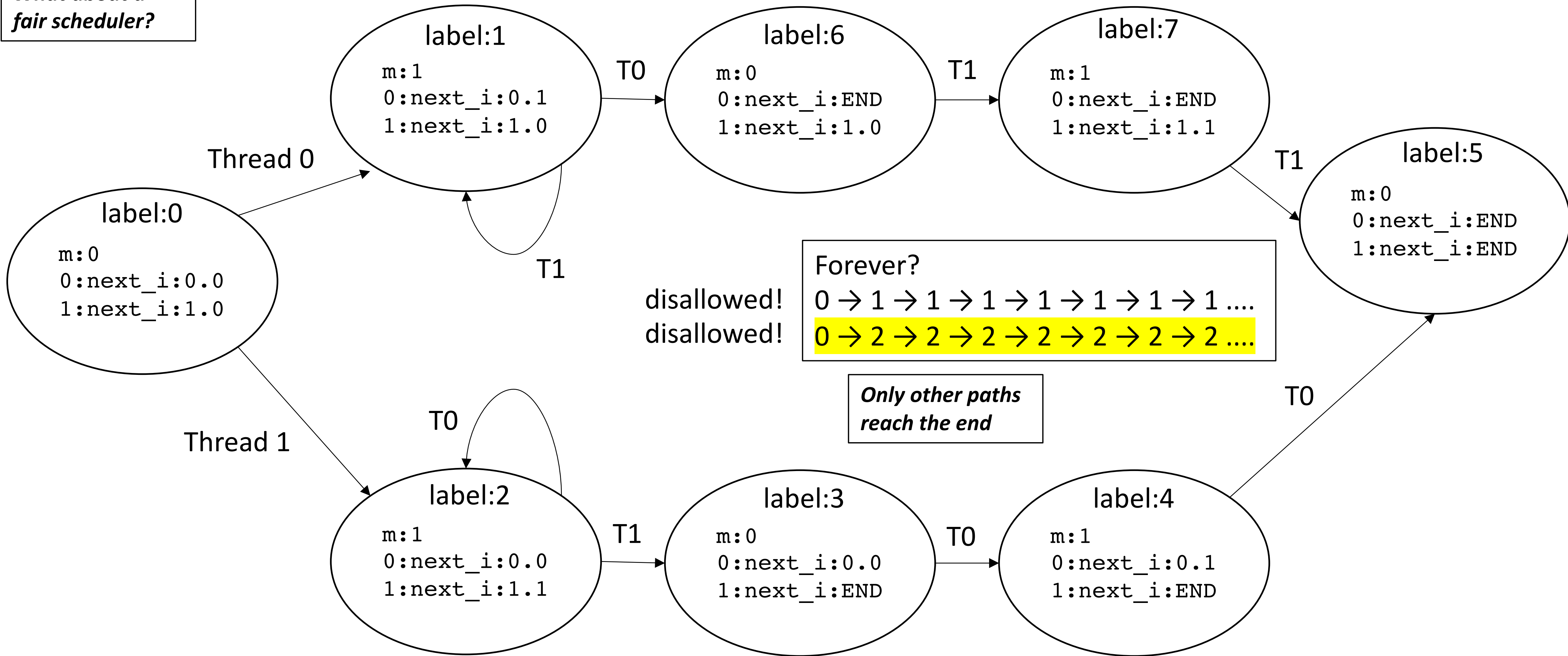


*Thread 0:*

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

*Thread 1:*

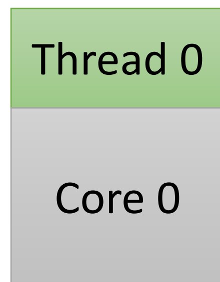
```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```



# Schedulers

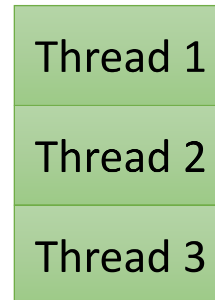
- A fair scheduler typically requires preemption

Thread list



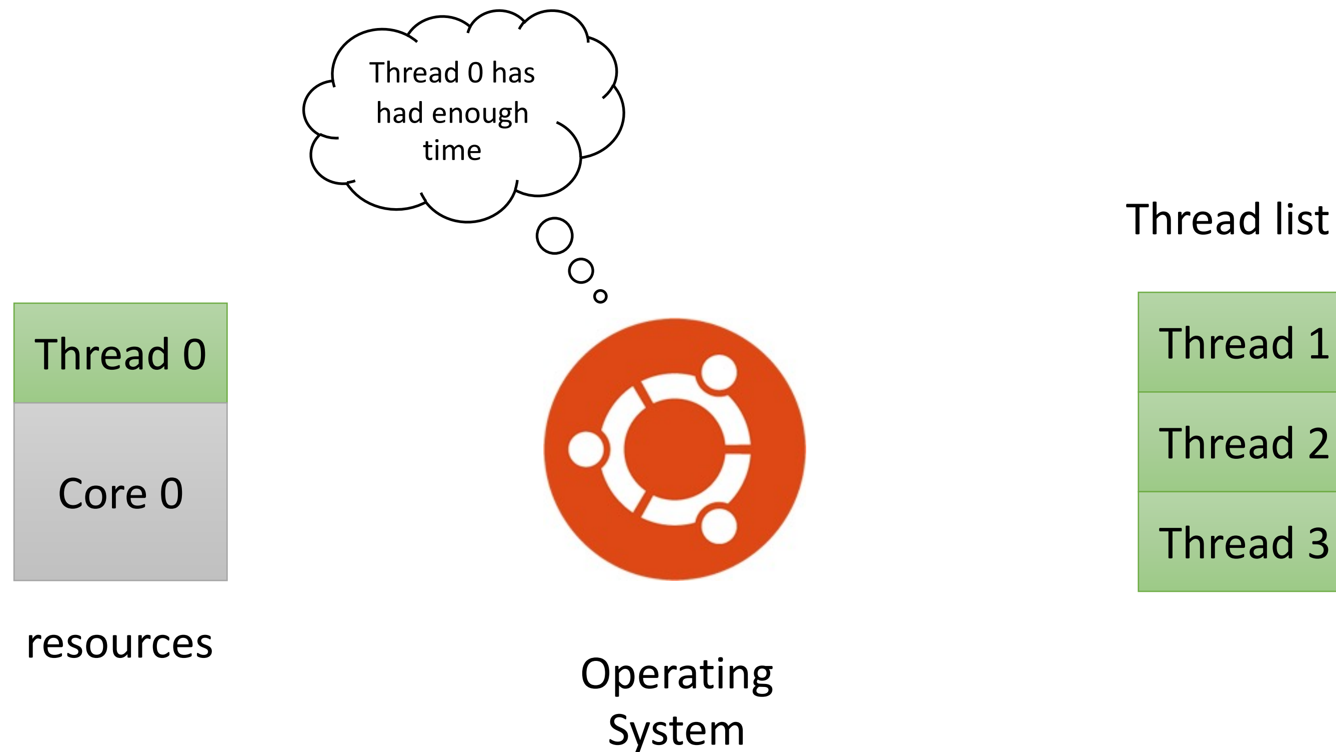

resources

Operating System



# Schedulers

- A fair scheduler typically requires preemption



# Schedulers

- A fair scheduler typically requires preemption

# Schedulers

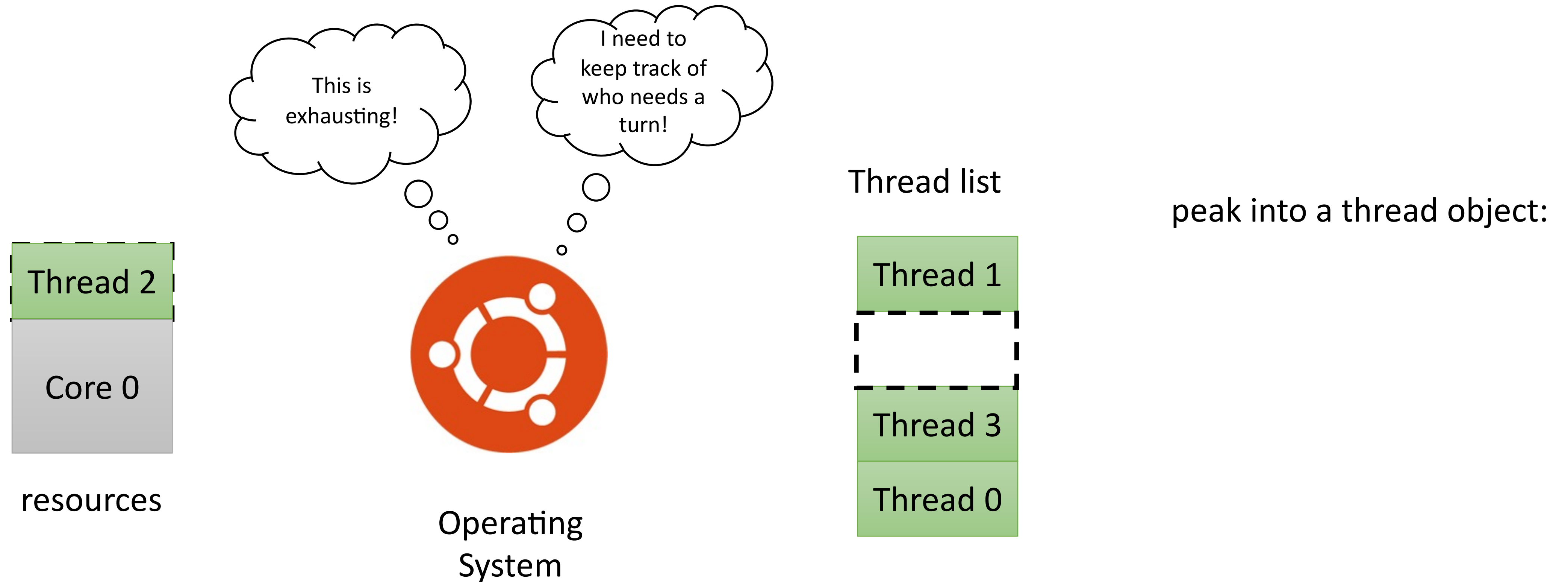
- A fair scheduler typically requires preemption

# Schedulers

- A fair scheduler typically requires preemption

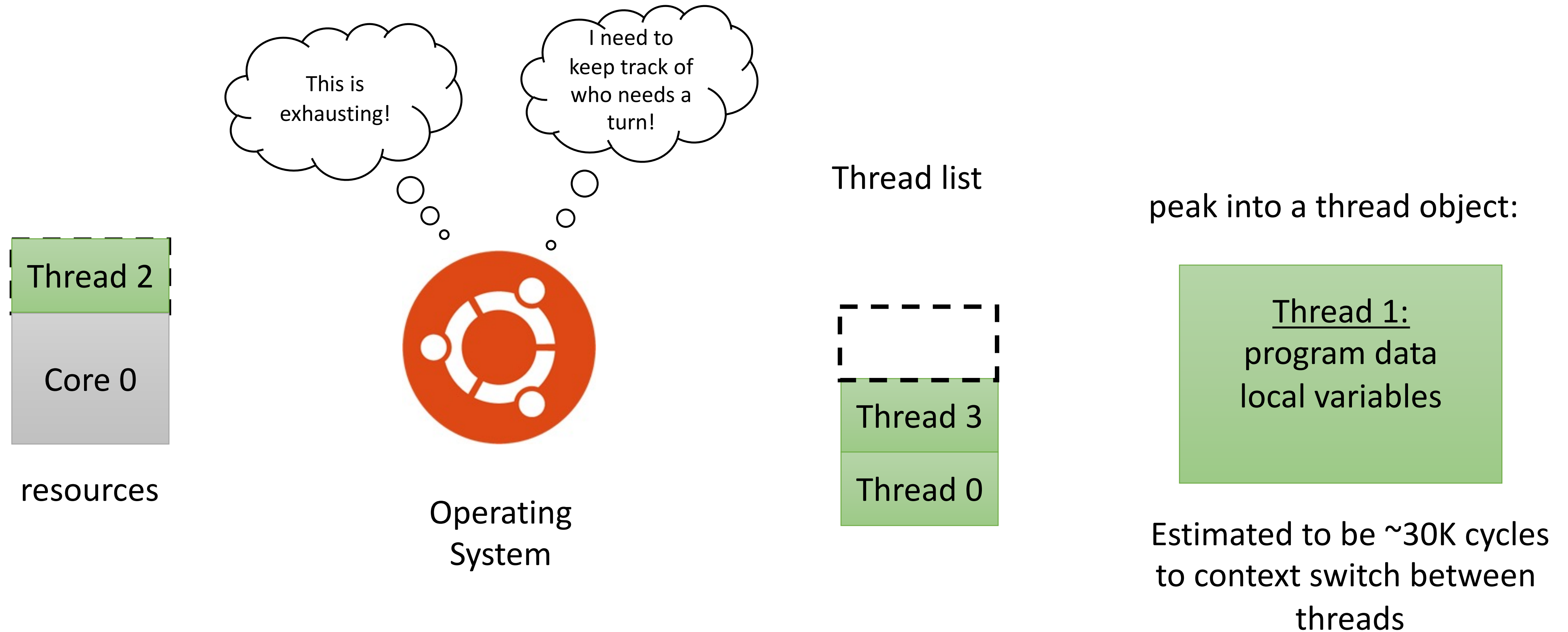
# Schedulers

- A fair scheduler typically requires preemption



# Schedulers

- A fair scheduler typically requires preemption

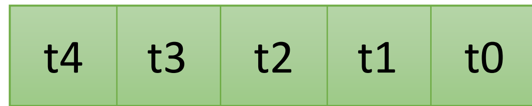


# Schedulers

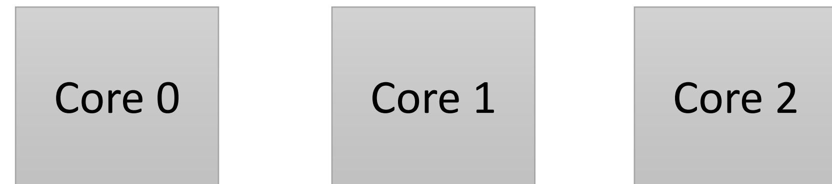
- Systems might not support preemption: e.g. GPUs

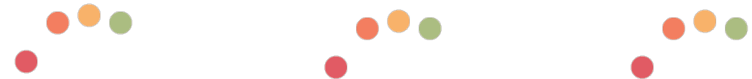
# simplified execution model

Program with 5 threads




*thread pool*



Device with 3 Cores

finished threads

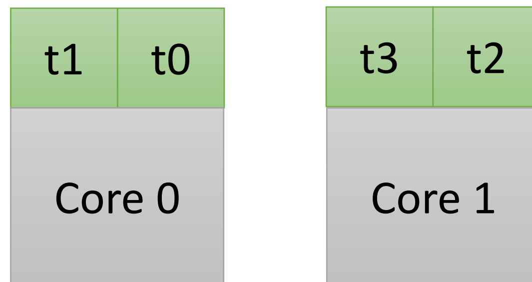
# Solutions?

- I have N cores, only run N threads?

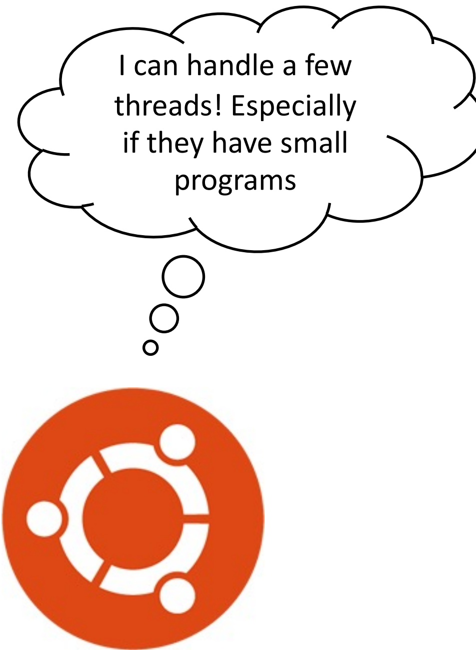
# Solutions?

- I have N cores, only run N threads?

sometimes concurrency can help hide latency! Don't want to completely disallow it!




Device with 2 cores



# Solutions?

- I have N cores, only run N threads?

- GPU examples:
  - Depending on program size Nvidia GPUs support
    - 32 threads per core for small programs
    - 2 threads per core for big programs

- We need a better specification

# Parallel Forward Progress

- “Any thread that has executed at least 1 instruction, is guaranteed to continue to be fairly executed”

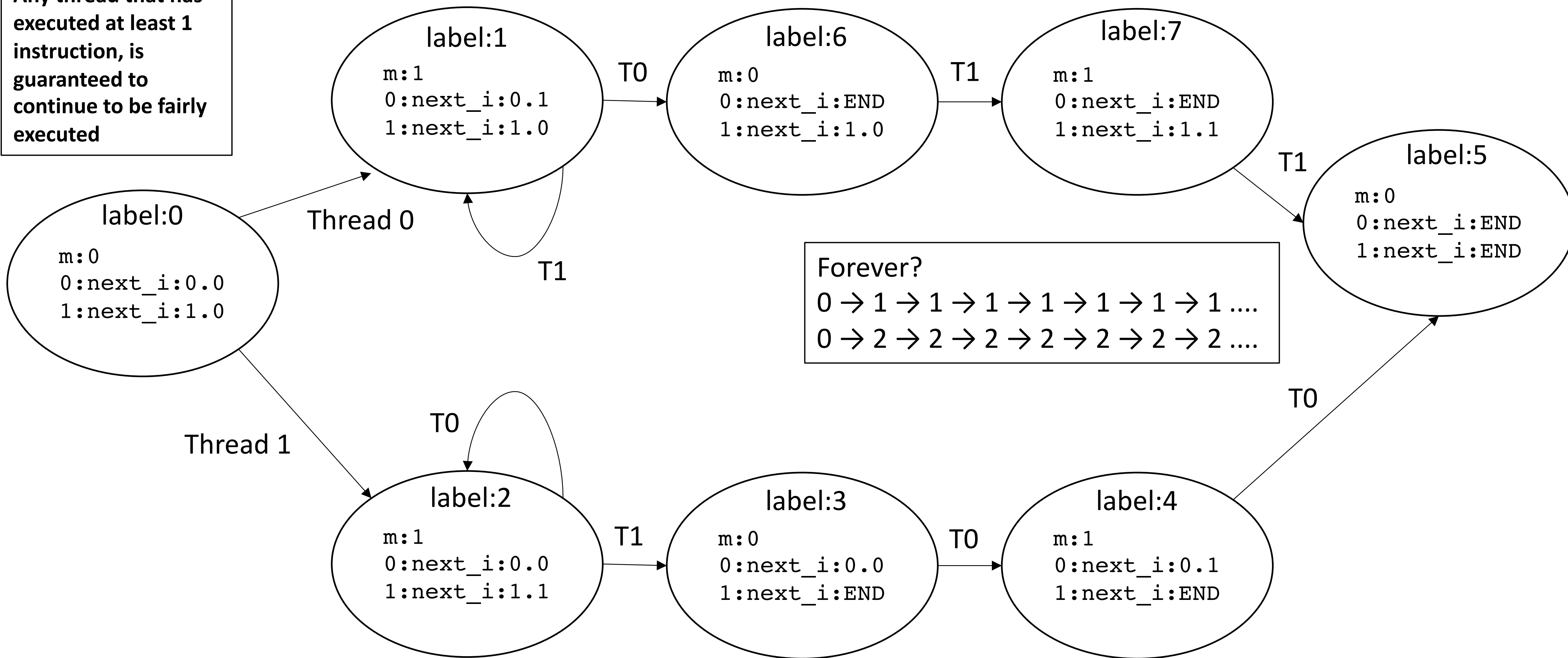
- Also called:
  - “Parallel Forward Progress”: by C++
  - “Persistent Thread Model”: by GPU programmers
  - ”Occupancy Bound Execution Model”: in some of my papers

*Thread 0:*

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

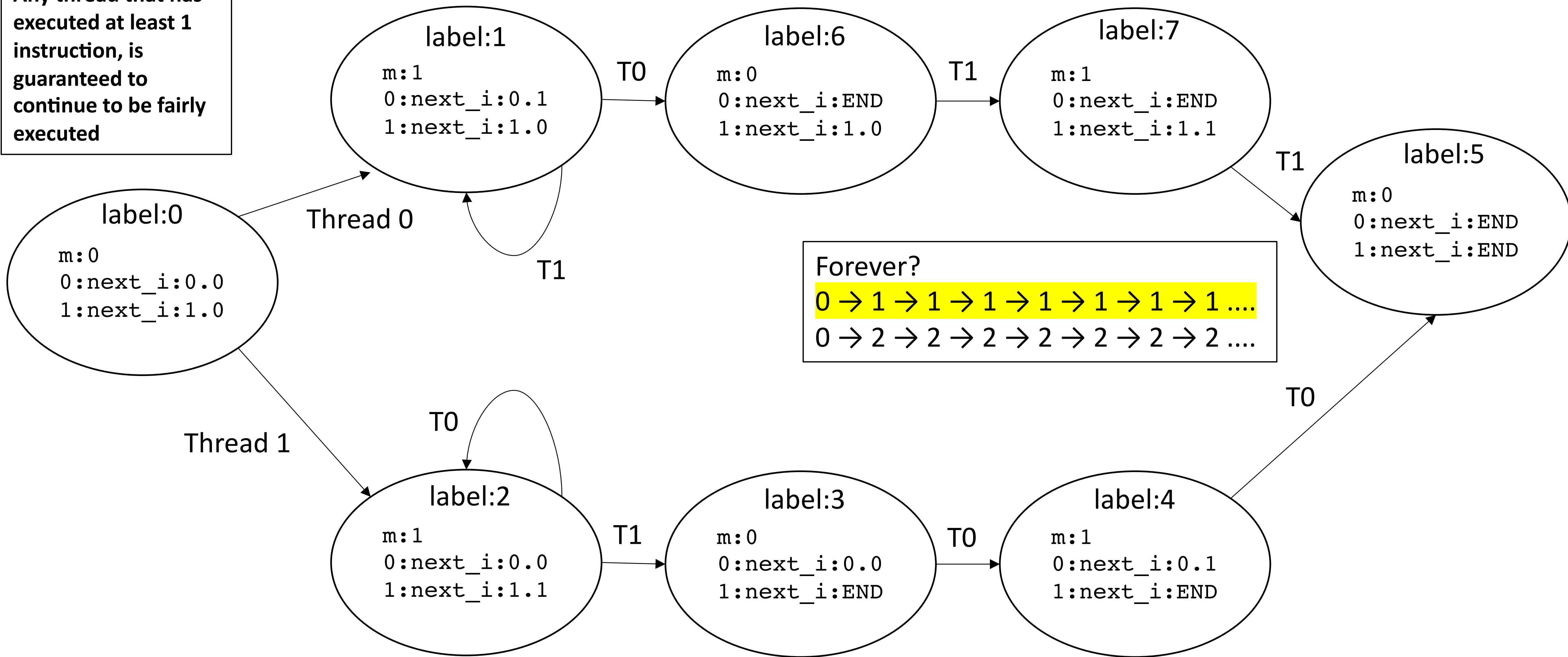
*Thread 1:*

```
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
```

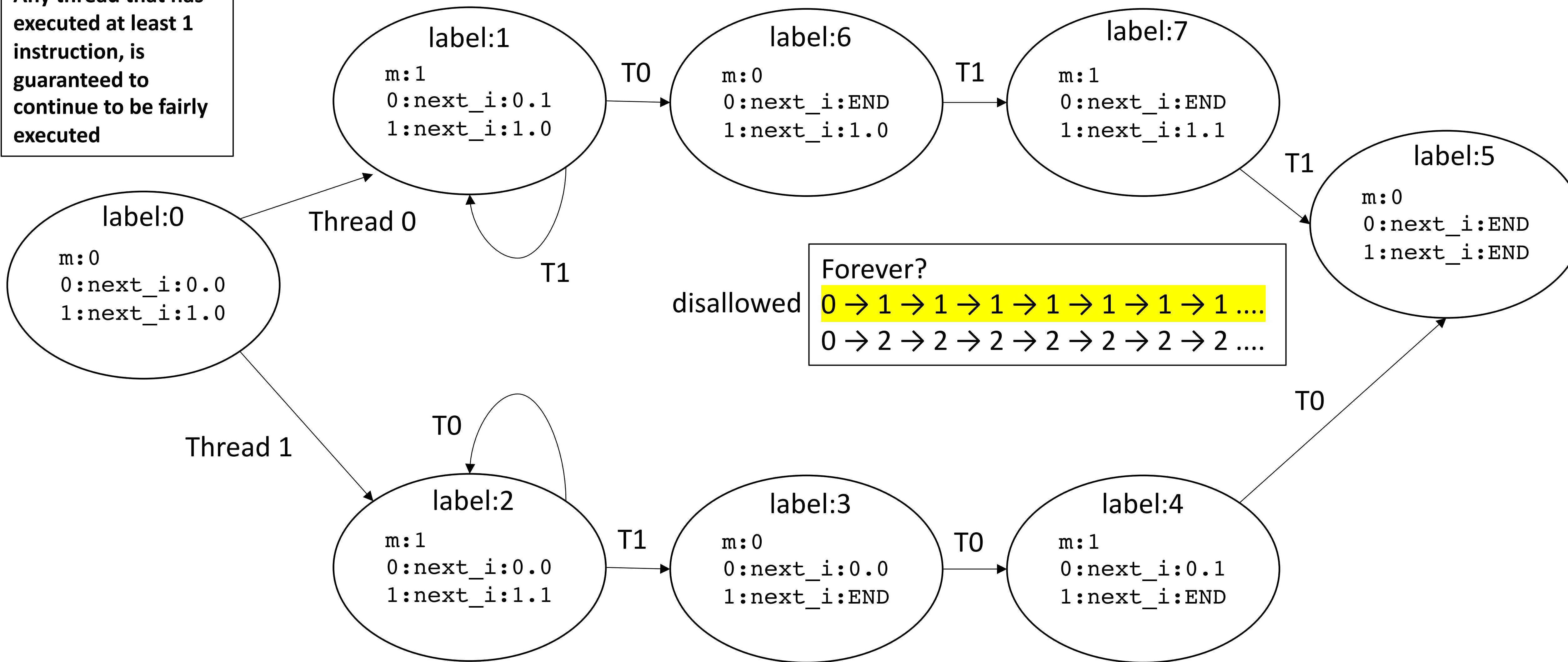


*Thread 0:*

```
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
```

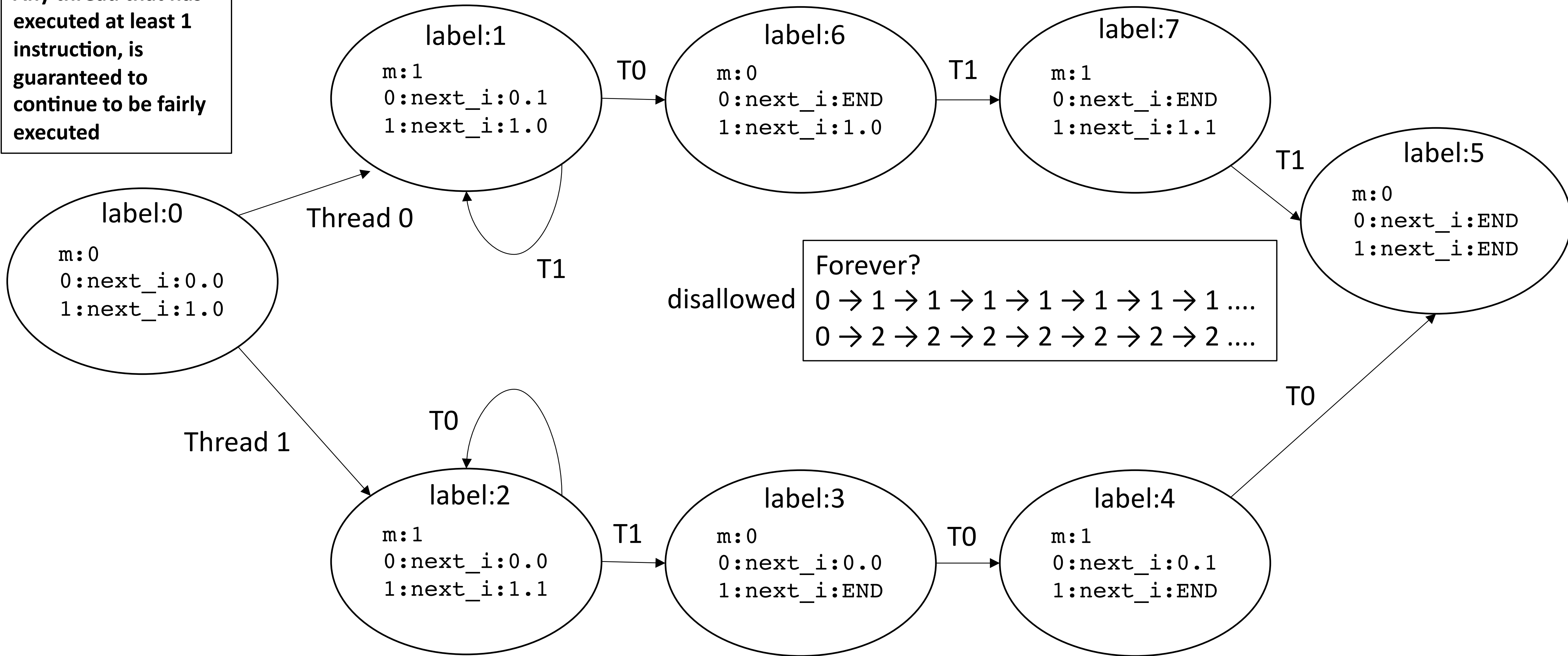


*Thread 0:*

```
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
```

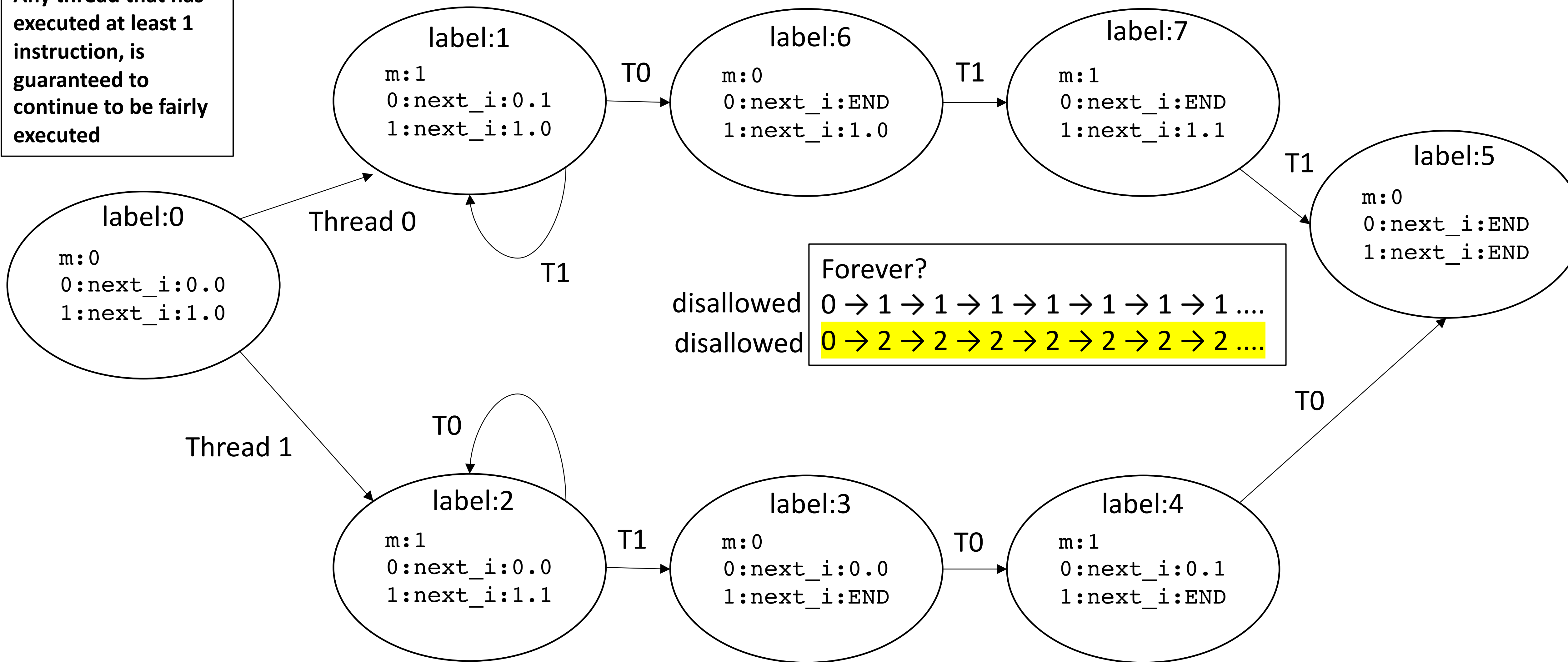


*Thread 0:*

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

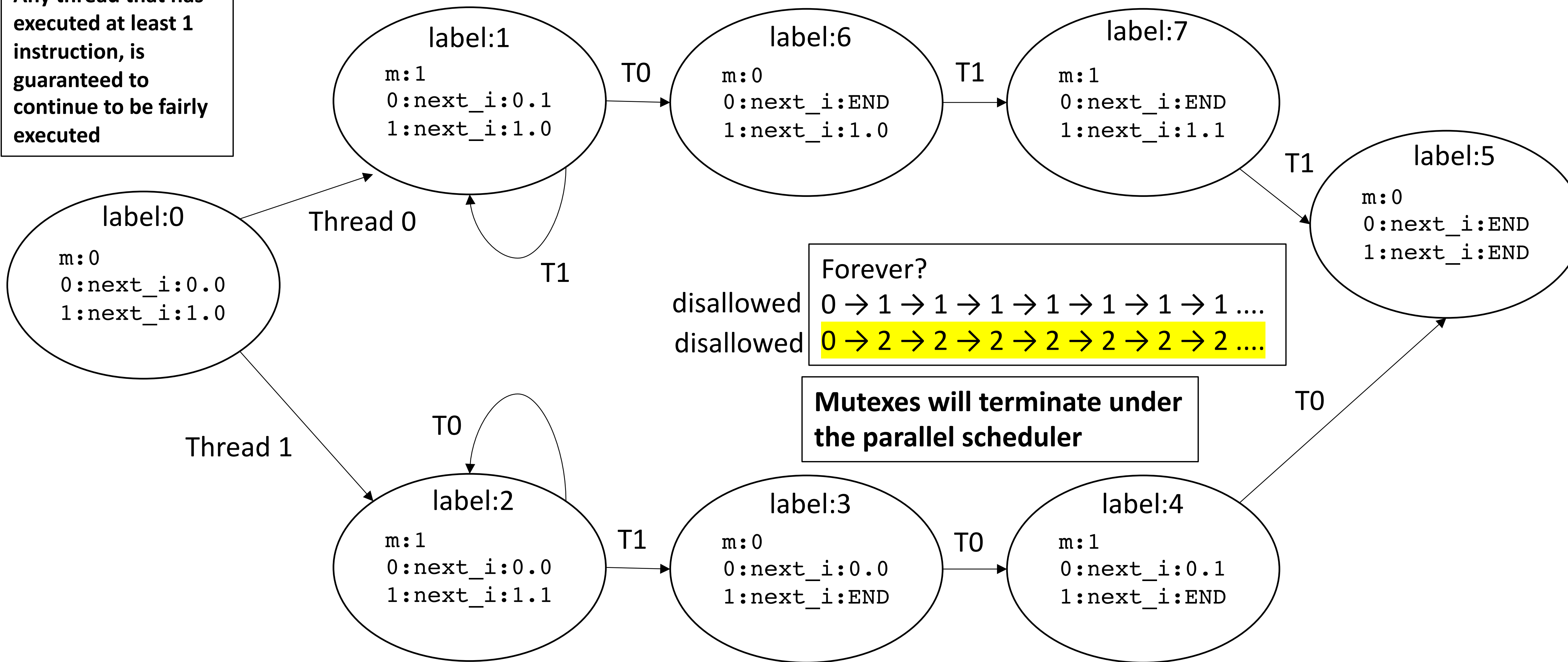
*Thread 1:*

```
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
```



# Another example

- Producer - consumer
  - Thread 0 waits for Thread 1 to write a flag

*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

start with initial node

label:0

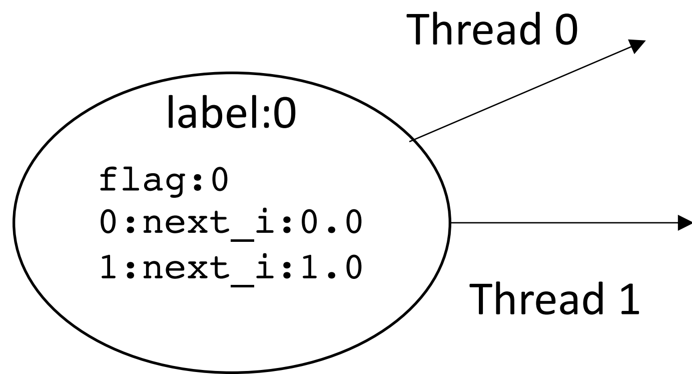
```
flag:0
0:next_i:0.0
1:next_i:1.0
```

*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

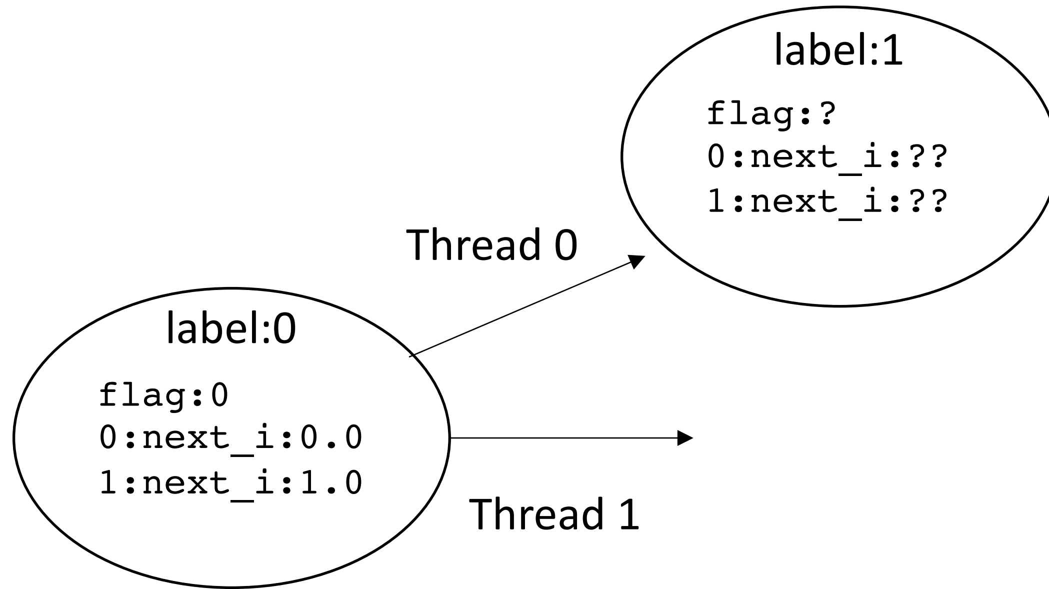


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

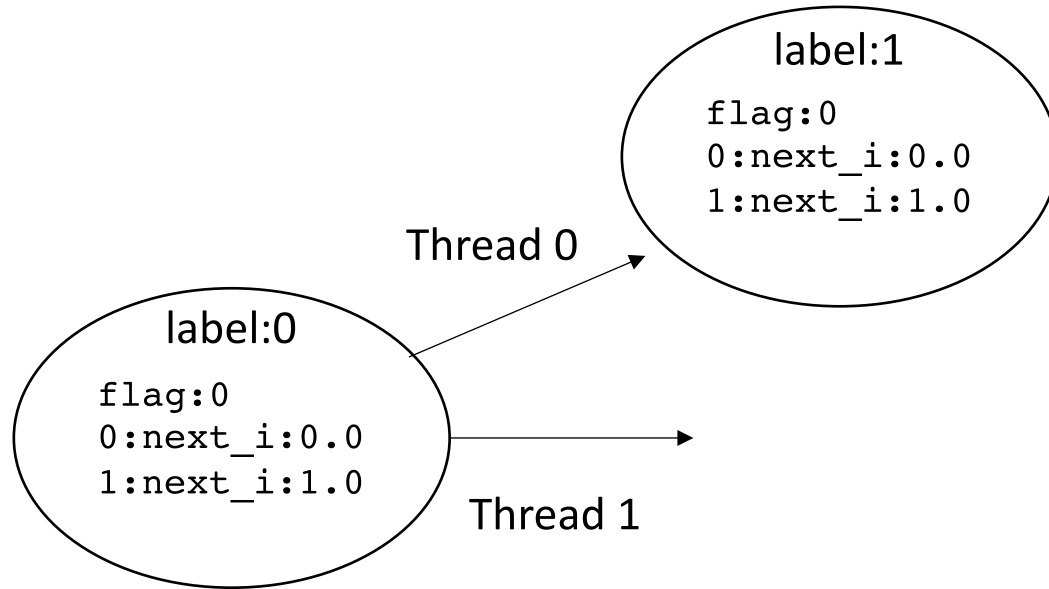


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

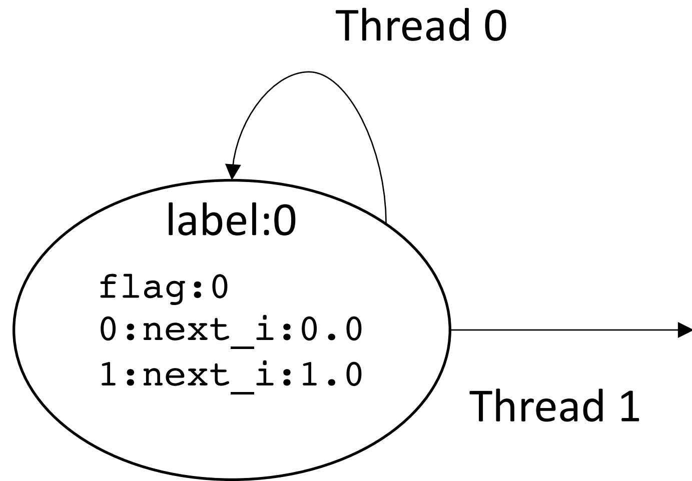


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

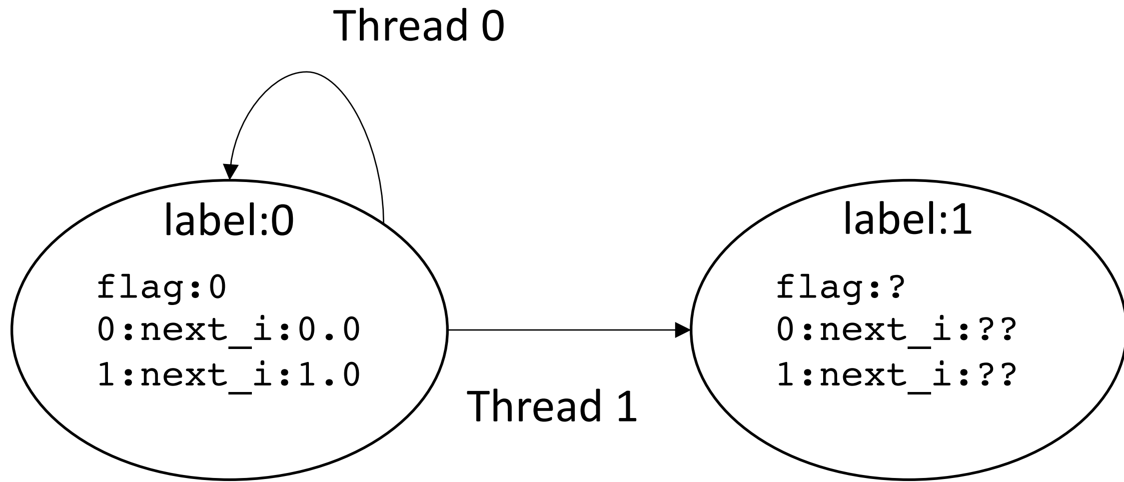


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

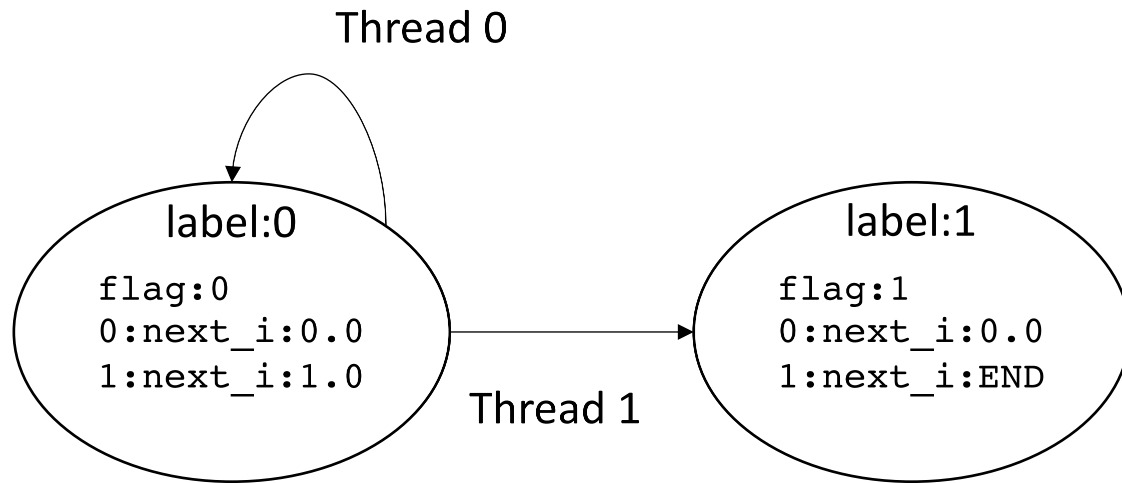


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

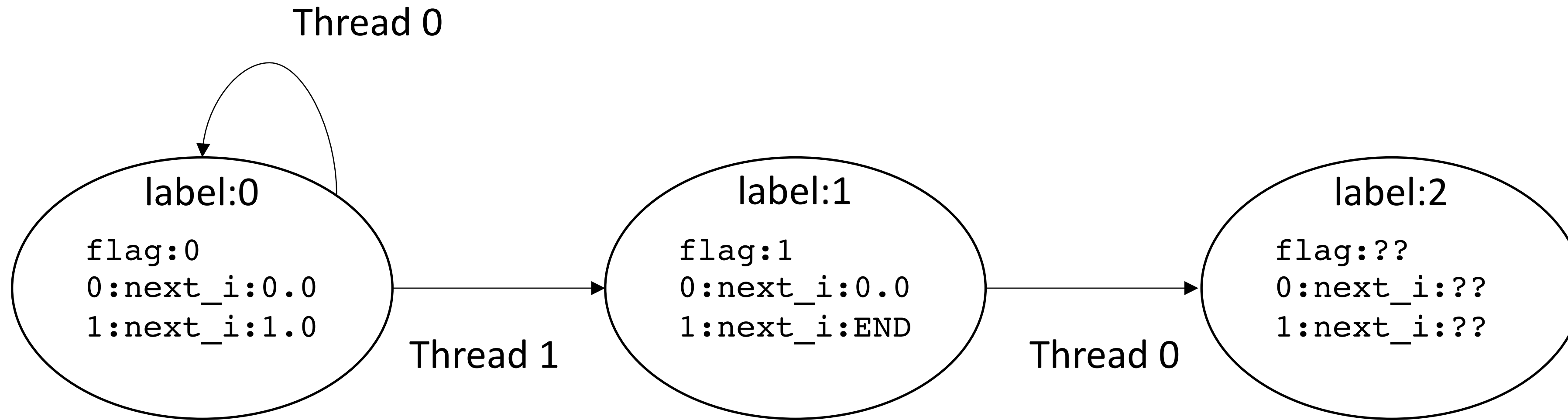


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

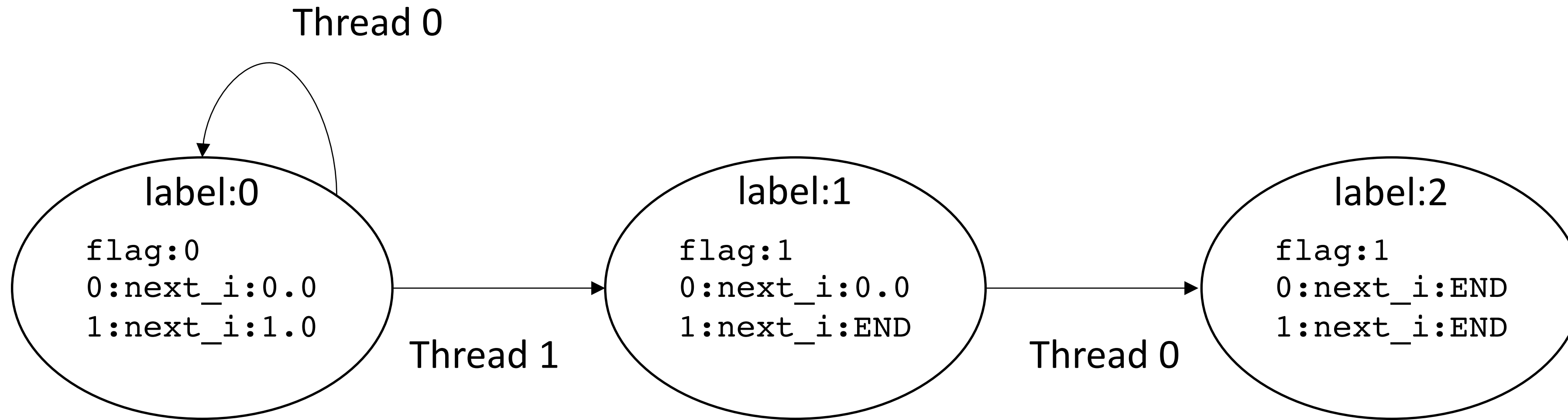


*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```



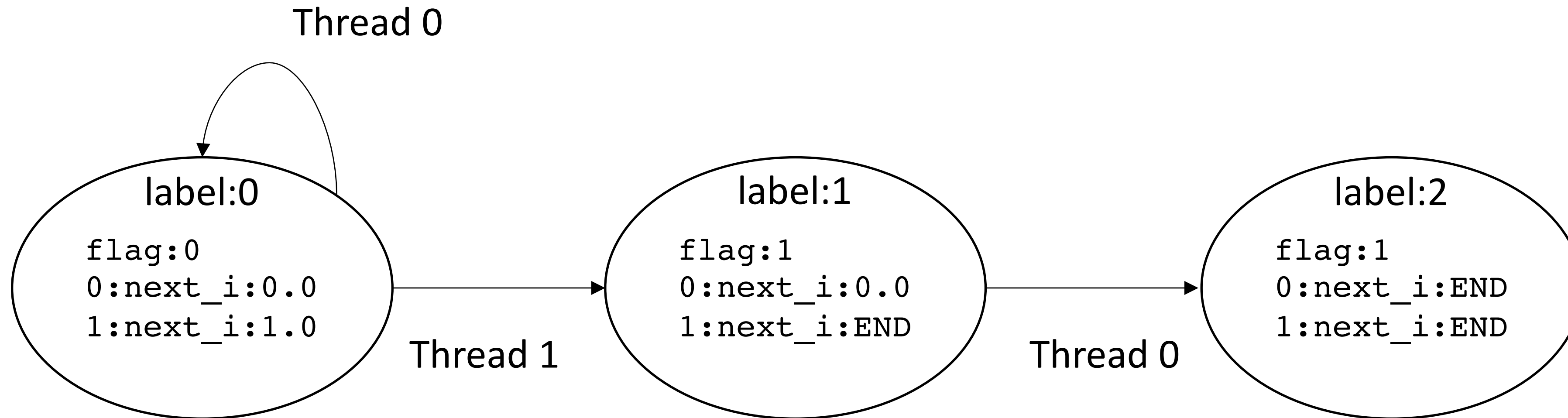
*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the fair scheduler?*



*Thread 0:*

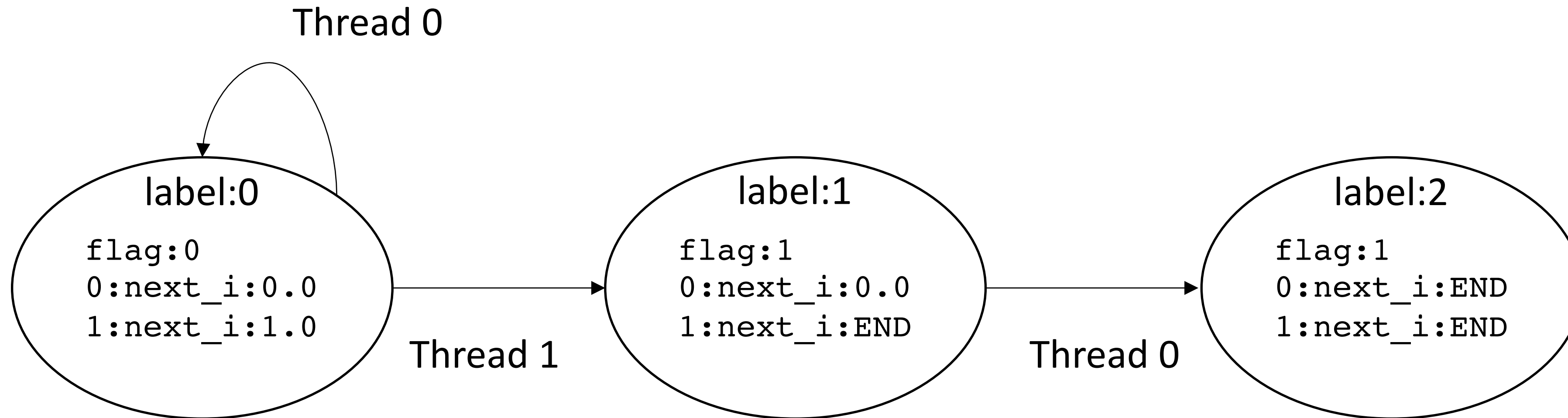
```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the fair scheduler?*

*Is this program guaranteed to terminate under the parallel scheduler?*



*Thread 0:*
```
0.0: while(flag.load() == 0);
```

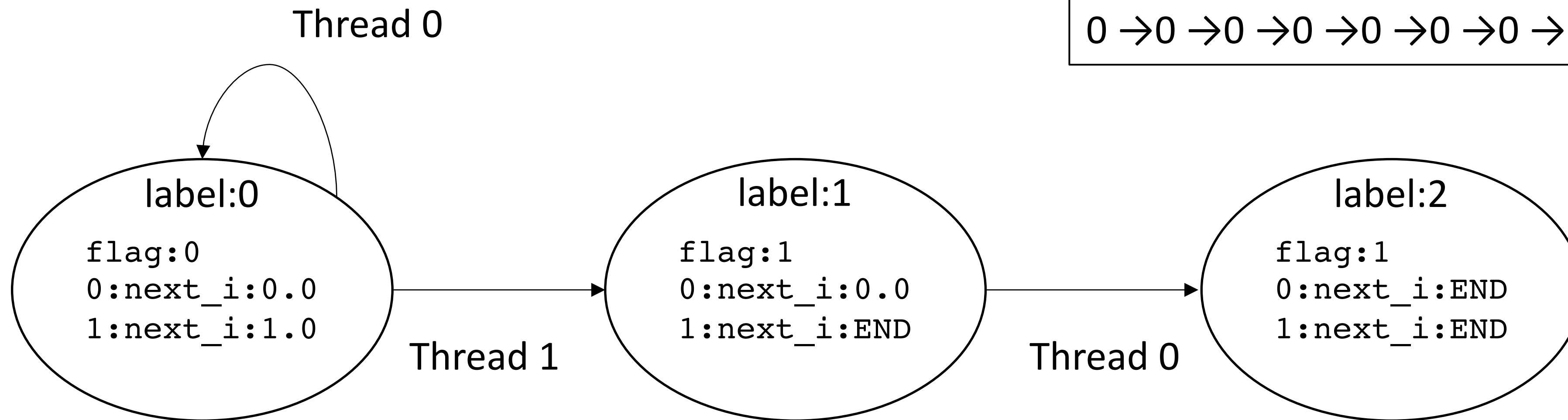
*Thread 1:*
```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the fair scheduler?*

*Is this program guaranteed to terminate under the parallel scheduler?*

**Any thread that has executed at least 1 instruction, is guaranteed to continue to be fairly executed**

Forever?
0 →0 →0 →0 →0 →0 →0 →0....



*Thread 0:*
```
0.0: while(flag.load() == 0);
```

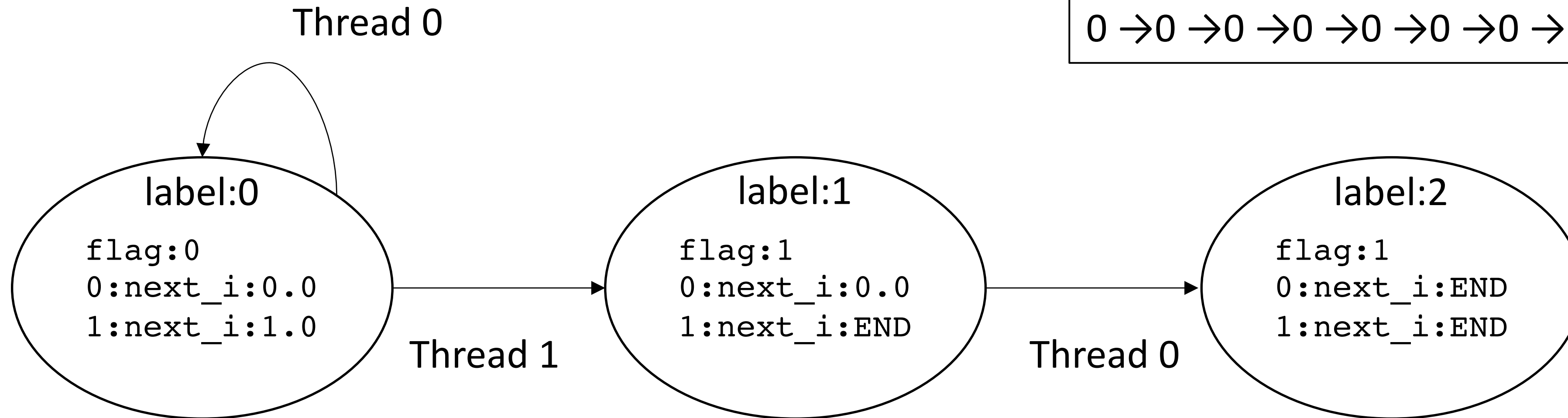
*Thread 1:*
```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the fair scheduler?*

*Is this program guaranteed to terminate under the parallel scheduler?*

**Any thread that has executed at least 1 instruction, is guaranteed to continue to be fairly executed**

Forever?
0 →0 →0 →0 →0 →0 →0 →0....




*allowed to spin forever in the parallel scheduler!*

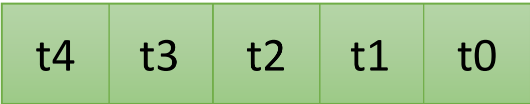
*Thread 0 could be scheduled on the only core while thread 1 spins*

# Schedulers

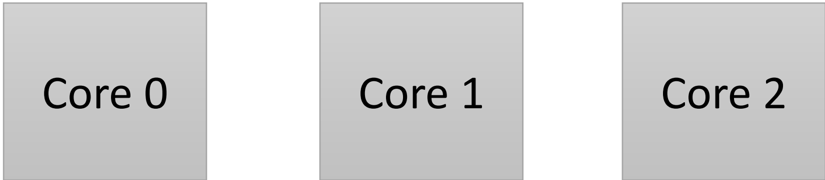
- In some cases the Parallel scheduler might be too strong
- For example dynamic power management on mobile devices

# A power-saving scheduler

Program with 5 threads




*thread pool*



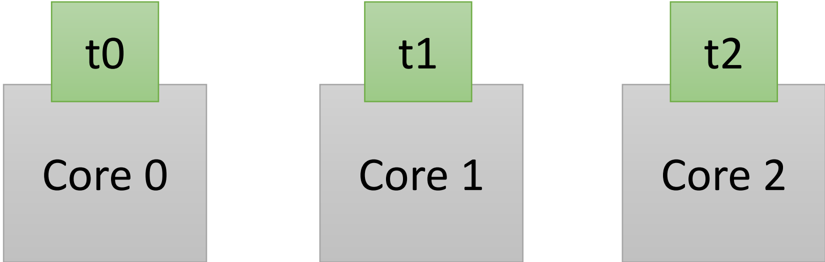

Device with 3 Cores

finished threads

# A power-saving scheduler

Program with 5 threads

*thread pool*



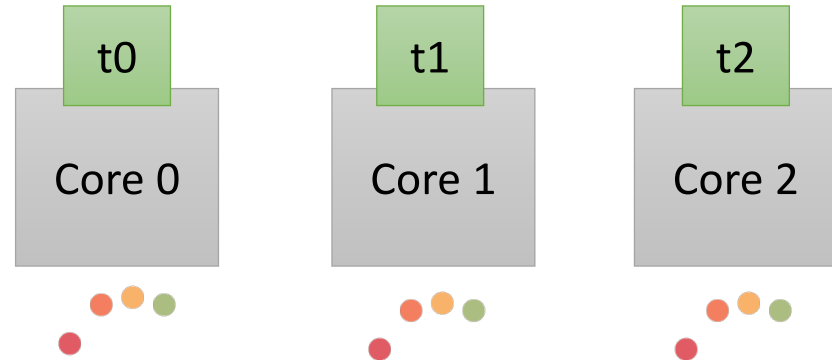

Device with 3 Cores

finished threads

# A power-saving scheduler

Program with 5 threads

*thread pool*



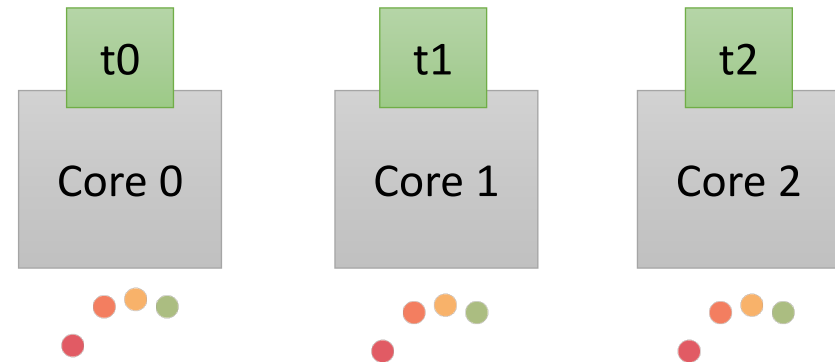

Device with 3 Cores

finished threads

# A power-saving scheduler

Program with 5 threads

*thread pool*




Device with 3 Cores

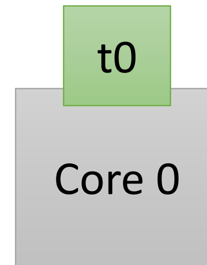
finished threads

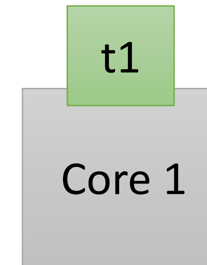
# A power-saving scheduler

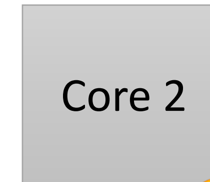
Program with 5 threads

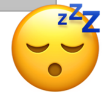
*thread pool*

preempted



Device with 3 Cores

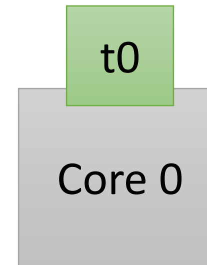
finished threads

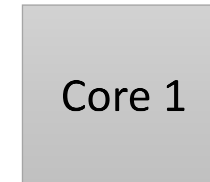
# A power-saving scheduler

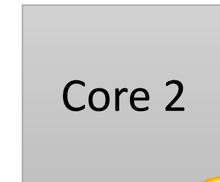
Program with 5 threads

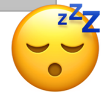
*thread pool*

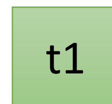
preempted





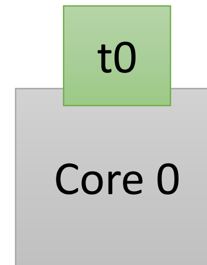

finished threads

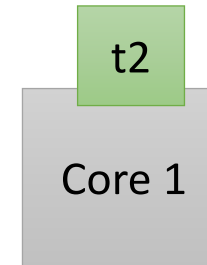
Device with 3 Cores

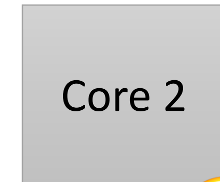
# A power-saving scheduler

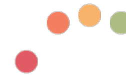
Program with 5 threads

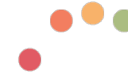
*thread pool*

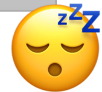


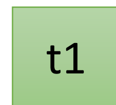










Device with 3 Cores

finished threads

# Schedulers

- This power-saving optimization messes up the Parallel Scheduler guarantees

- Can we do anything interesting with a scheduler like this?

# Schedulers

- This power-saving optimization messes up the Parallel Scheduler guarantees
- Can we do anything interesting with a scheduler like this?
- The OS can give guarantees about the threads that it preempts for energy savings.

# Schedulers

- This power-saving optimization messes up the Parallel Scheduler guarantees

- Can we do anything interesting with a scheduler like this?

- The OS can give guarantees about the threads that it preempts for energy savings.

- The OS could target threads with higher ids and give priority with threads with the lower id.

# The HSA schduler

- The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.

- Called:
  - "HSA" - Heterogeneous System Architecture, programming language proposed by AMD for new systems.

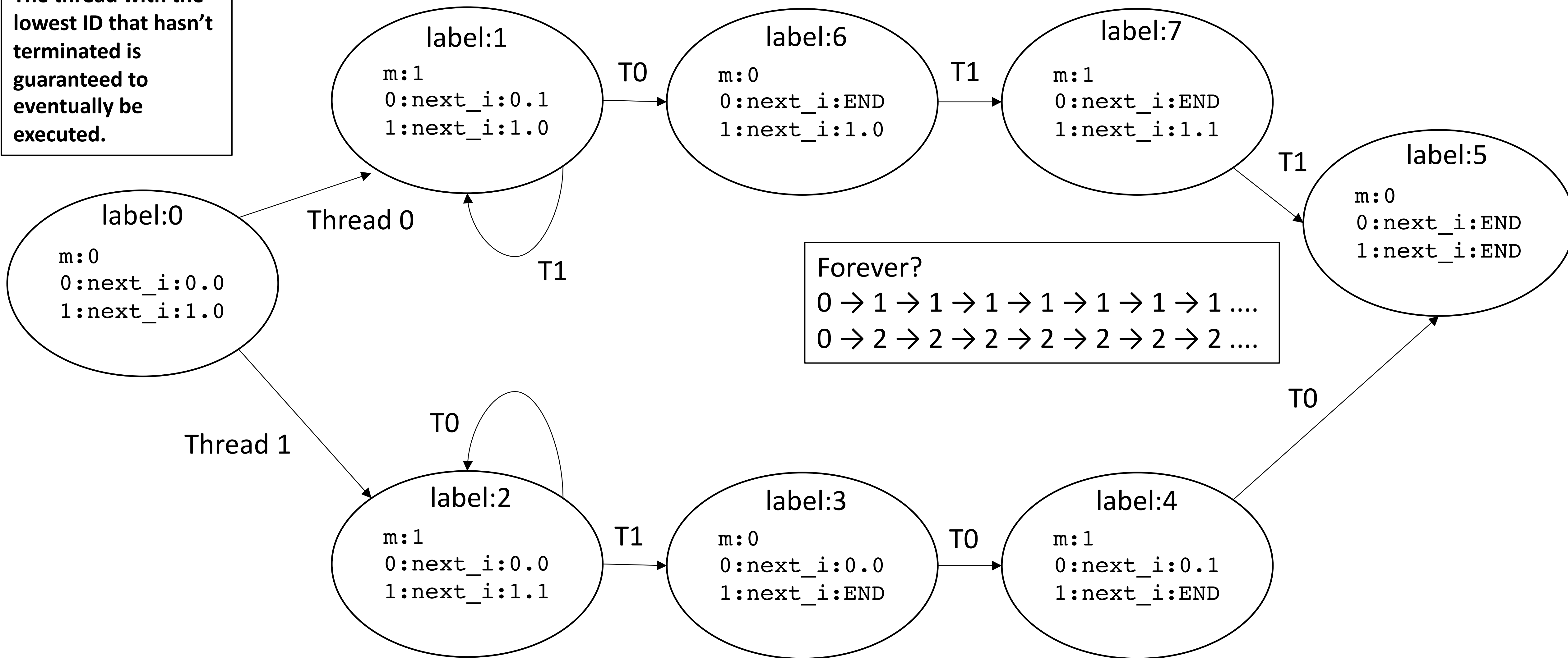
  - The HSA language appears to be defunct now, but the scheduler is a good fit for mobile devices (esp. mobile GPUs).

*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
```

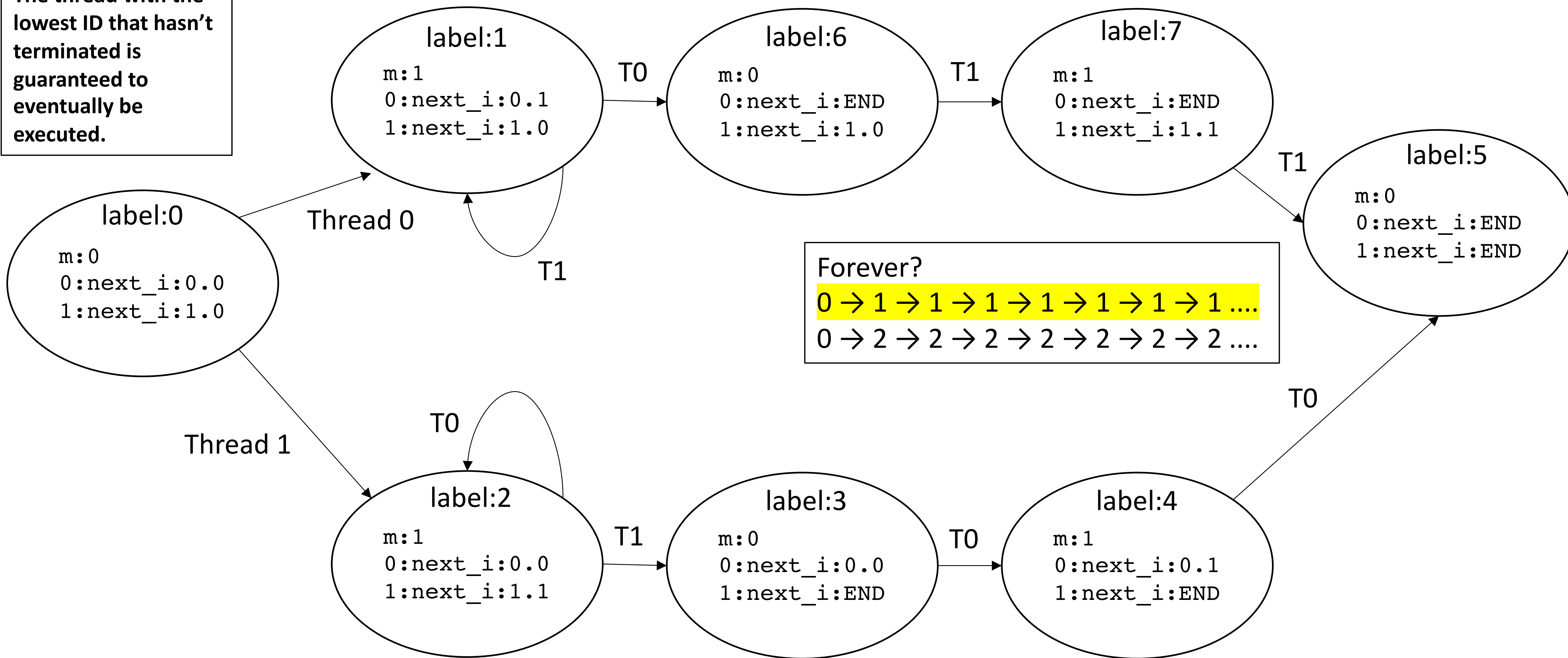


*Thread 0:*

```
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
```

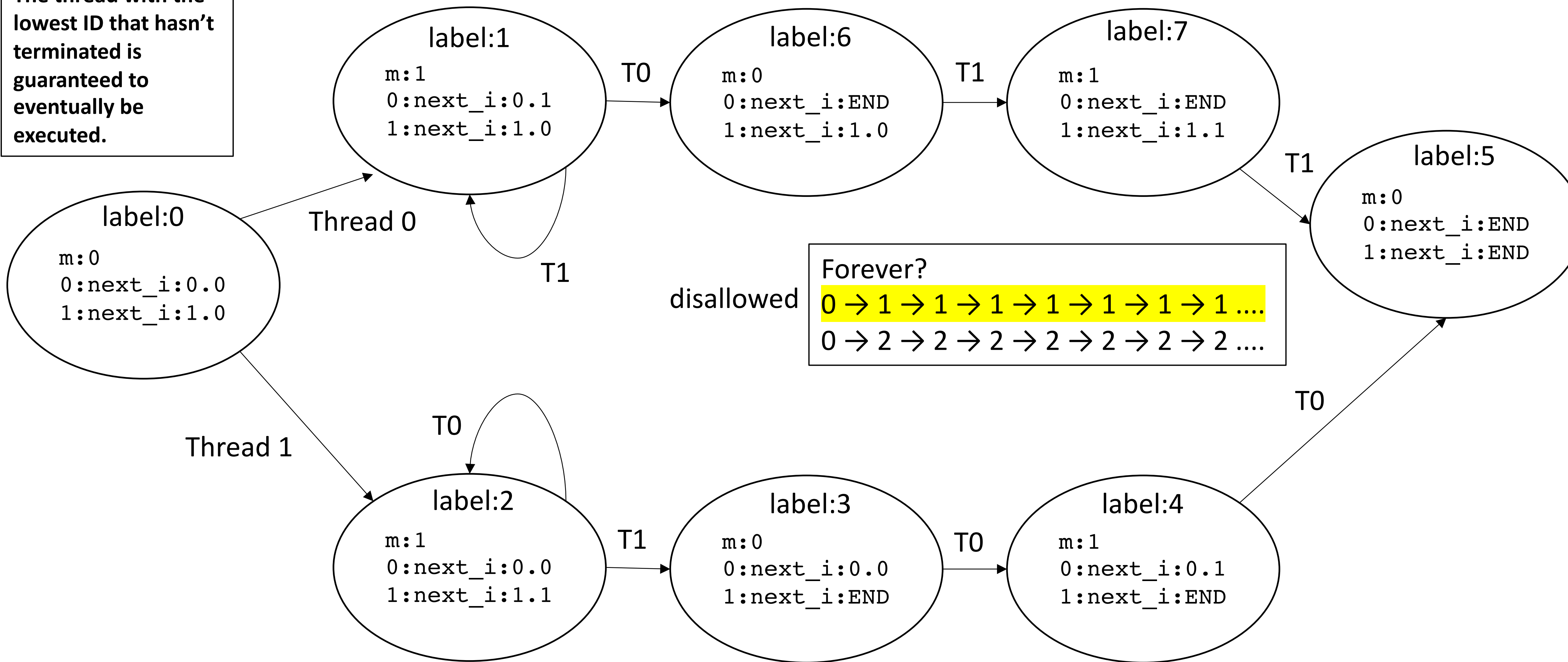


*Thread 0:*

```
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
```

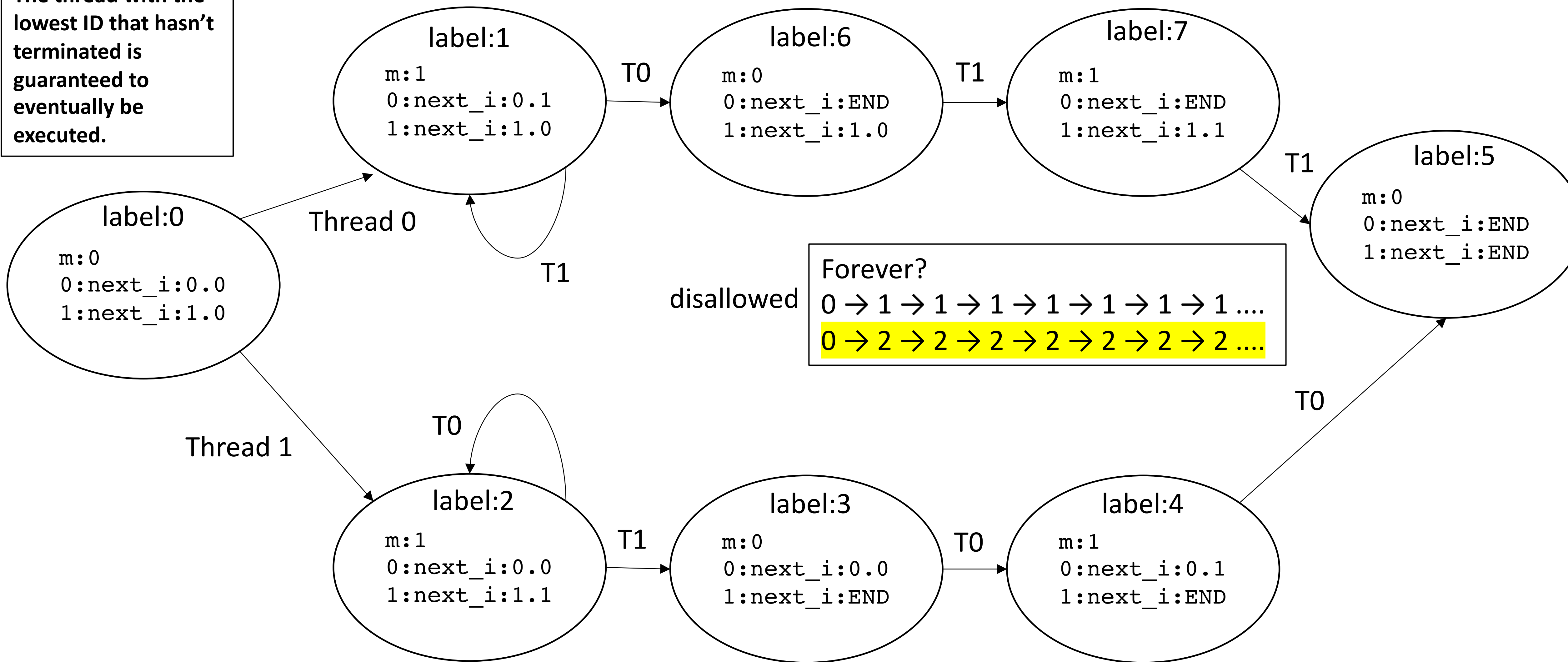


*Thread 0:*

```
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
```



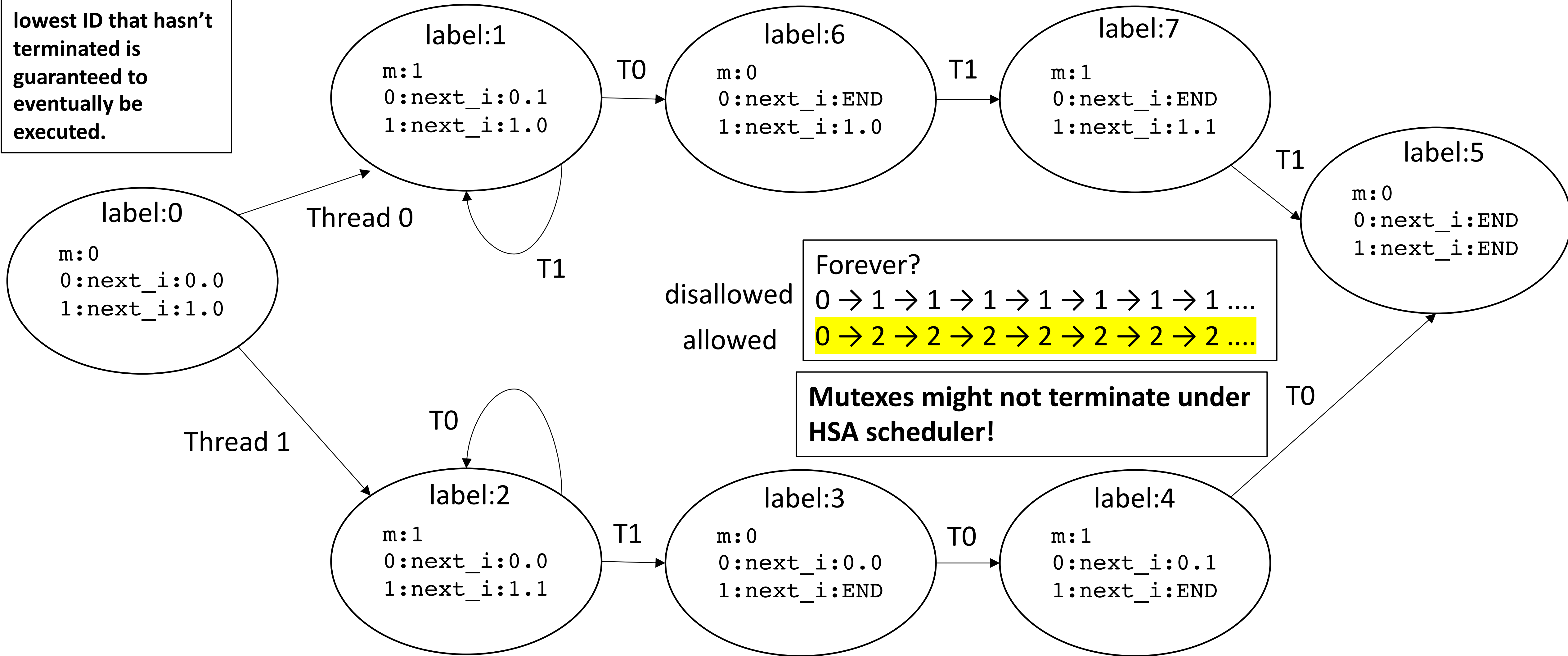
*Thread 0:*

```
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
```

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**

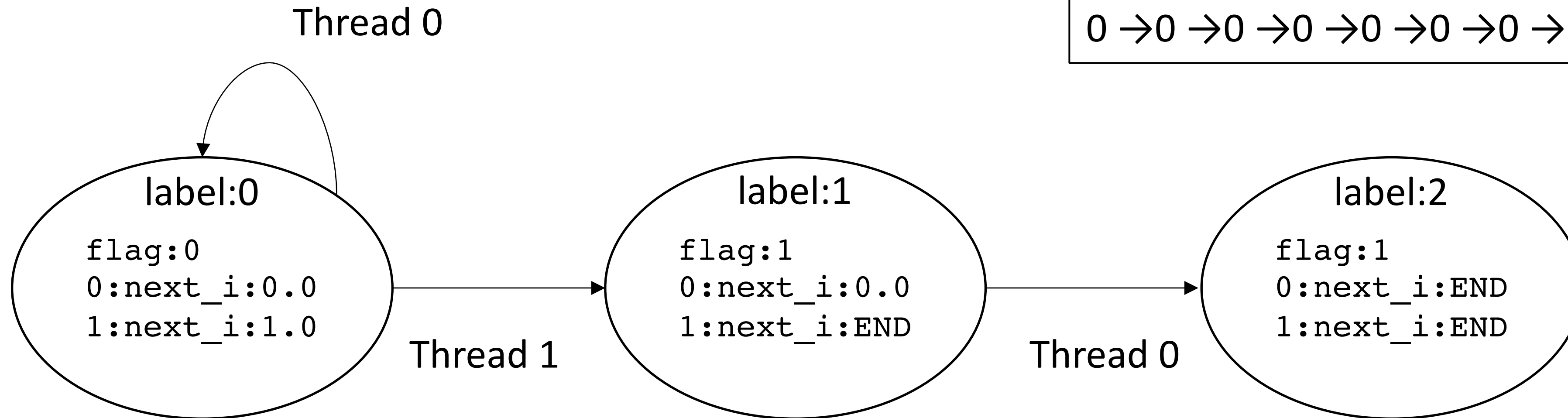


*Thread 0:*
```
0.0: while(flag.load() == 0);
```

*Thread 1:*
```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the HSA scheduler*

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**




Forever?
0 →0 →0 →0 →0 →0 →0 →0....

*Thread 0:*

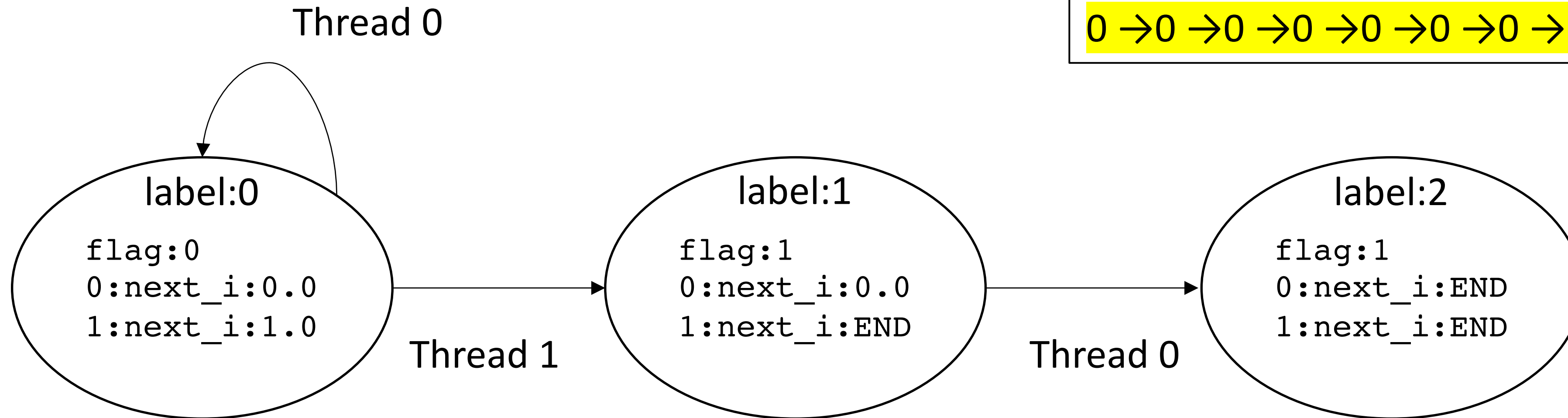
```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the HSA scheduler*

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**




Forever?

0 →0 →0 →0 →0 →0 →0 →0….

*Thread 0:*

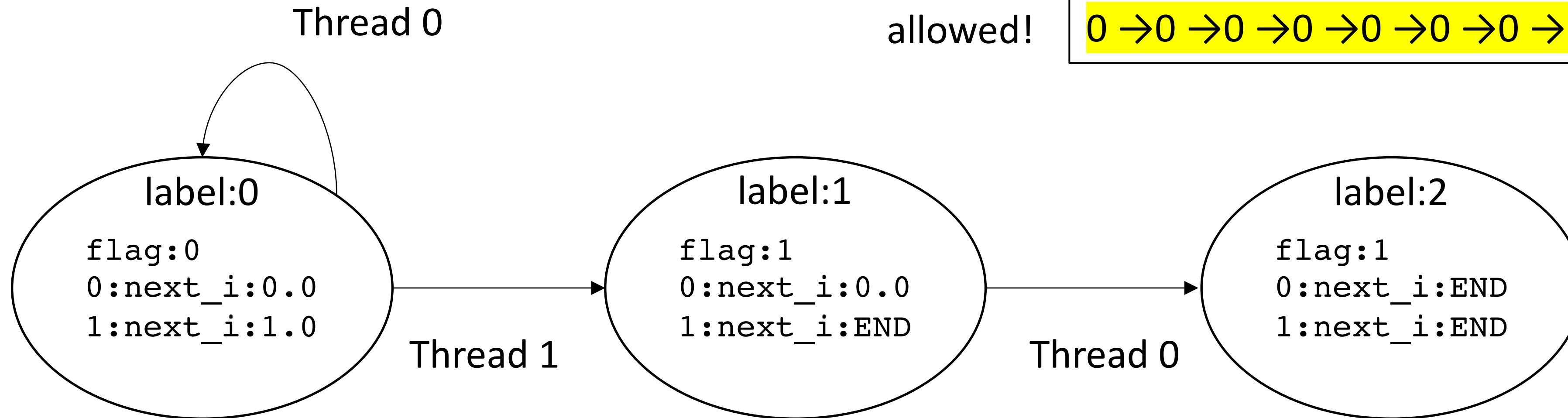
```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

*Is this program guaranteed to terminate under the HSA scheduler*

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**




allowed!

Forever?
0 →0 →0 →0 →0 →0 →0 →0....

*allowed to spin forever in the HSA scheduler!*

*Thread 0 is guaranteed to be executed because it has the lowest id. Thread 1 is not!*

*Thread 0:*

```
0.0: while(flag.load() == 0);
```

*Thread 1:*

```
1.0: flag.store(1);
```

What if we switch the threads?
Thread 1 waits for Thread 0?

*Thread 0:*

```
0.0: flag.store(1);
```

*Thread 1:*

```
1.0: while(flag.load() == 0);
```

What if we switch the threads?
Thread 1 waits for Thread 0?

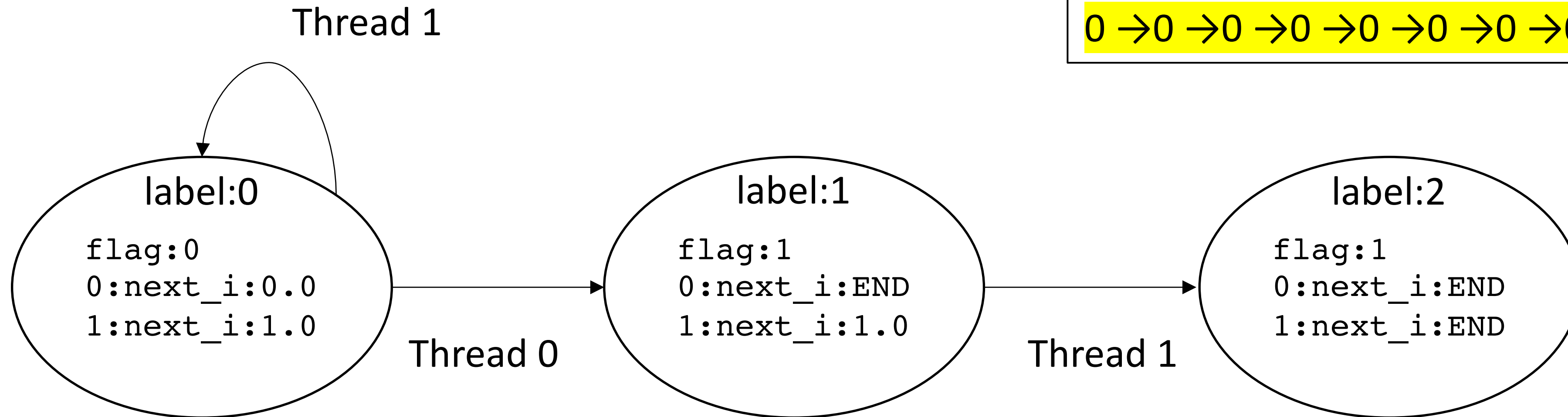
*Thread 0:*
```
0.0: flag.store(1);
```

*Thread 1:*
```
1.0: while(flag.load() == 0);
```

What if we switch the threads?
Thread 1 waits for Thread 0?

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**

Forever?
0 →0 →0 →0 →0 →0 →0 →0....




*thread 0 has the lowest id so it is guaranteed to eventually be executed*

*Thread 0:*

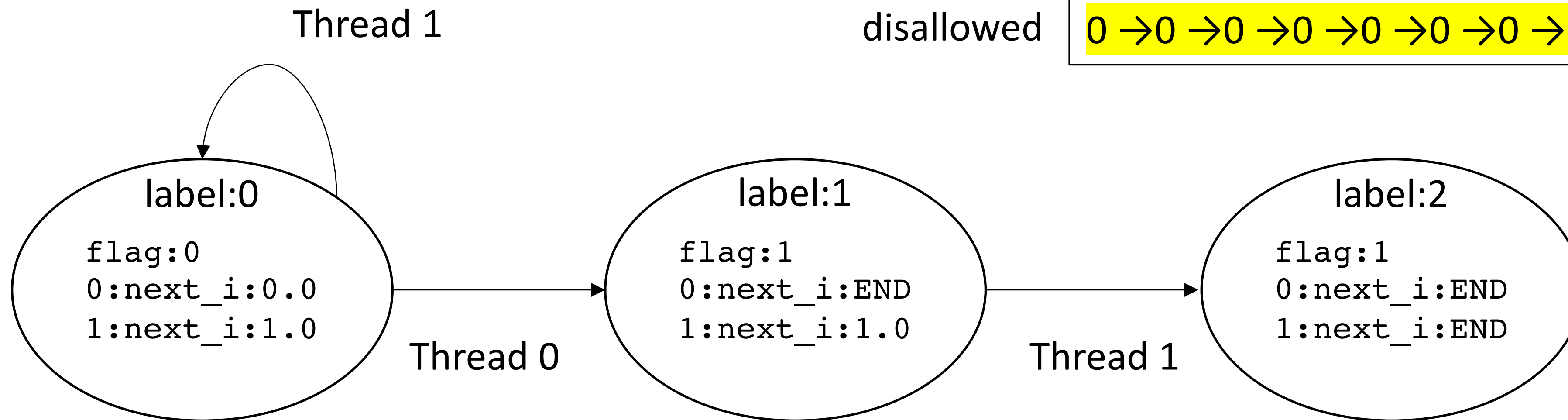
```
0.0: flag.store(1);
```

*Thread 1:*

```
1.0: while(flag.load() == 0);
```

What if we switch the threads?
Thread 1 waits for Thread 0?

**The thread with the lowest ID that hasn't terminated is guaranteed to eventually be executed.**




*thread 0 has the lowest id so it is guaranteed to eventually be executed*

# Liveness

- Combining HSA and Parallel Execution?

- Threads are scheduled in the order of their thread IDs and are guaranteed fair execution once they start executing.

- Most modern GPUs seem to support this:
  - With the exception of ARM and Apple GPUs

# Liveness

- So where are we now?

- C++ gives 3 degrees of progress guarantees:
  - Concurrent scheduler
    - what you will likely see on your machine; fair scheduler!
  - Parallel scheduler
    - Threads that start executing will continue to be fairly executed. Allows mutexes!
  - Weakly parallel scheduler
    - No guarantees. Any cycle in the LTS can potentially execute forever!

# Liveness

- So where are we now?

- GPU schedulers:
  - Nvidia provides Parallel Forward Progress
    - Allows mutexes, concurrent data structures, etc.

  - OpenCL, Vulkan, and Metal provide no documentation on scheduler behaviors.
    - In practice, many assume parallel forward progress
    - This is not portable (esp. to ARM and Apple)
    - Working with specification groups to try and provide these

# Conclusion

- Schedulers are becoming more aggressive
  - Preemption is expensive
  - Power saving shut downs are possible
- Concurrent objects require different amounts of fairness
  - Mutexes require parallel forward progress
  - Producer Consumer requires HSA forward progress
- Be careful that the programs you are writing make the correct assumptions about the underlying scheduler!

# Conclusion

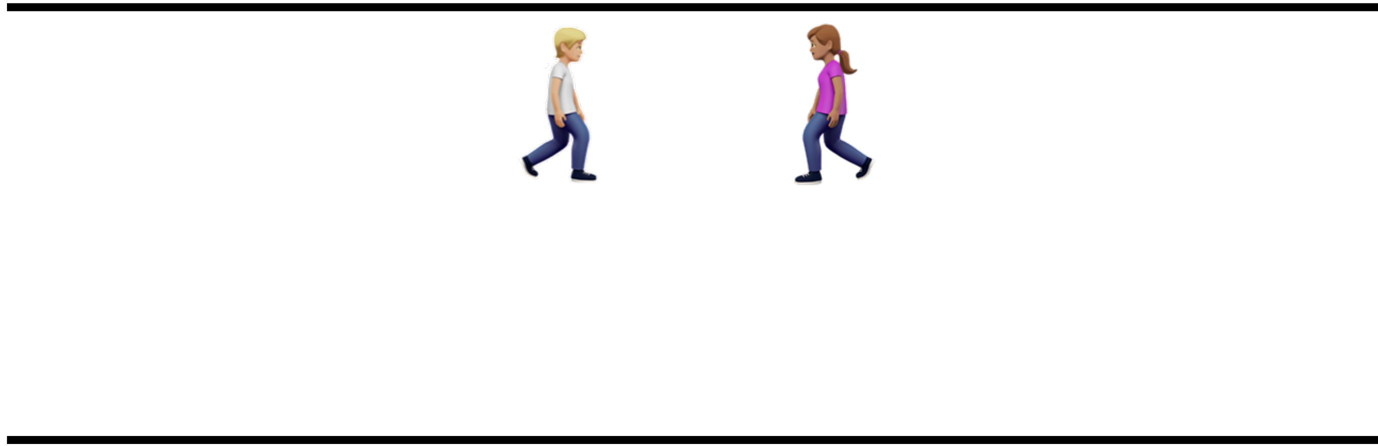
- Demo about how things can go wrong on Ipad.

# A different type of non-termination

Hallway problem

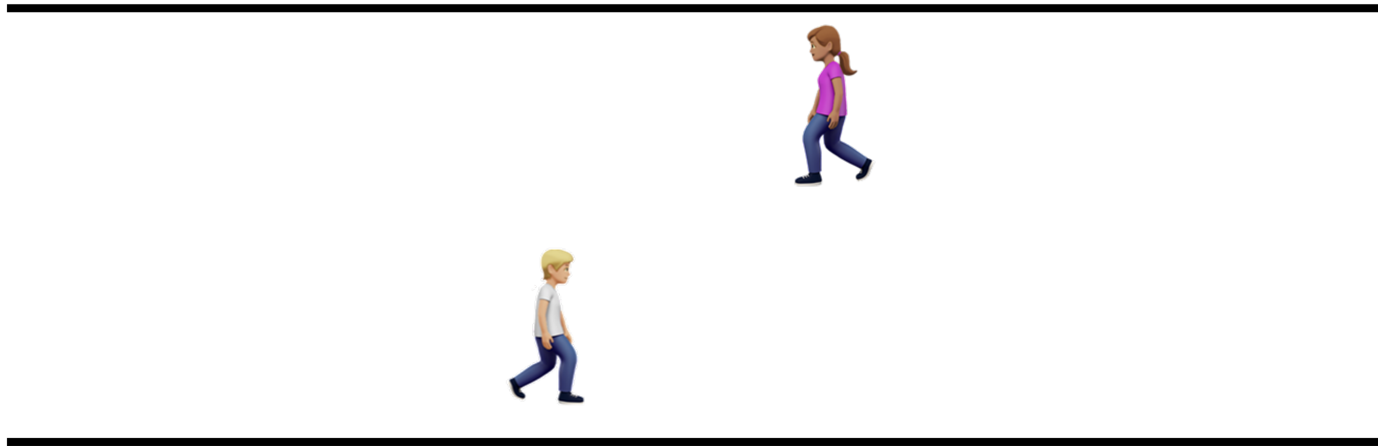
# A different type of non-termination

Hallway problem



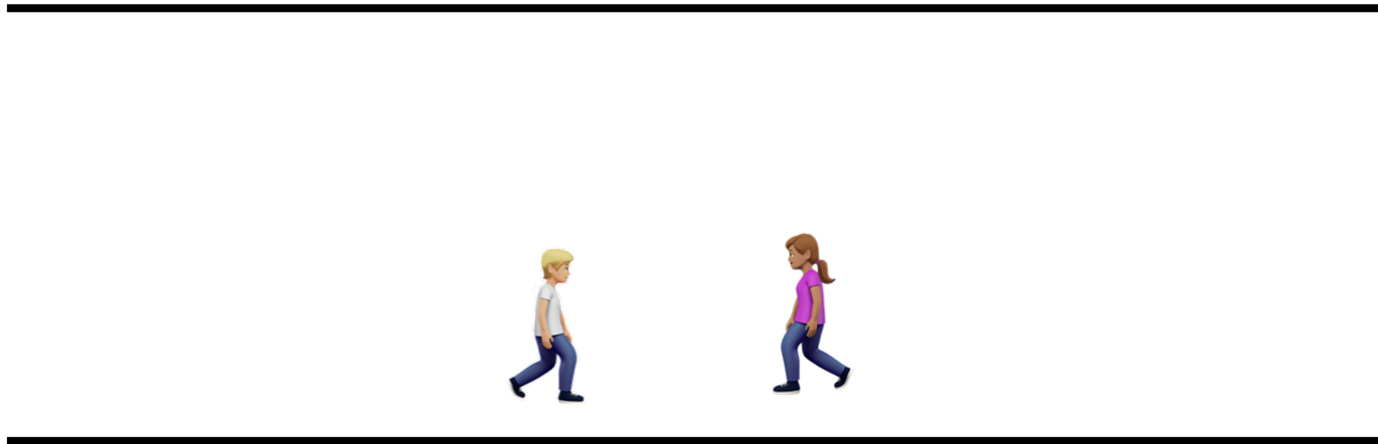
# A different type of non-termination

Hallway problem



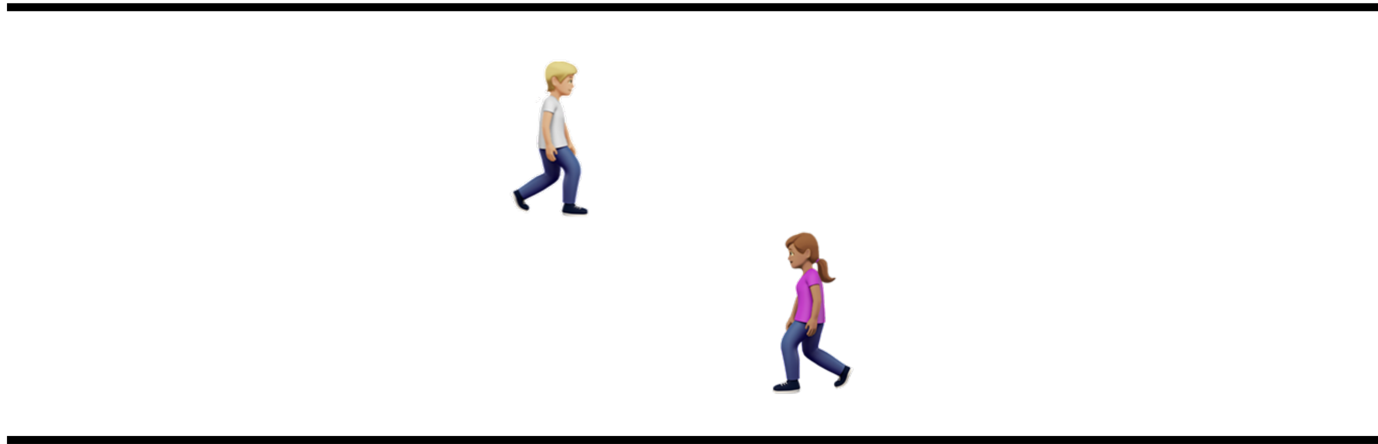
# A different type of non-termination

Hallway problem



# A different type of non-termination

Hallway problem



# A different type of non-termination

Hallway problem

Can they dance around each other forever?

*Thread 0:*

```
... do {
0.0   x.store(0);
0.1 } while (x.load() != 0)
```

*Thread 1:*

```
... do {
1.0   x.store(1);
1.1 } while (x.load() != 1)
```

Each thread stores their thread id,
and then loads the thread id. It loops while
it doesn’t see its id

Each thread gets a chance to execute, but they
get in each others way.

This is called a livelock

We don’t have time to get into it deeply here,
but there are lots of interesting research challenges
around these types of behaviors!

# Thanks!

- See you on Thursday!

- We will start a 2 part lecture on GPUs