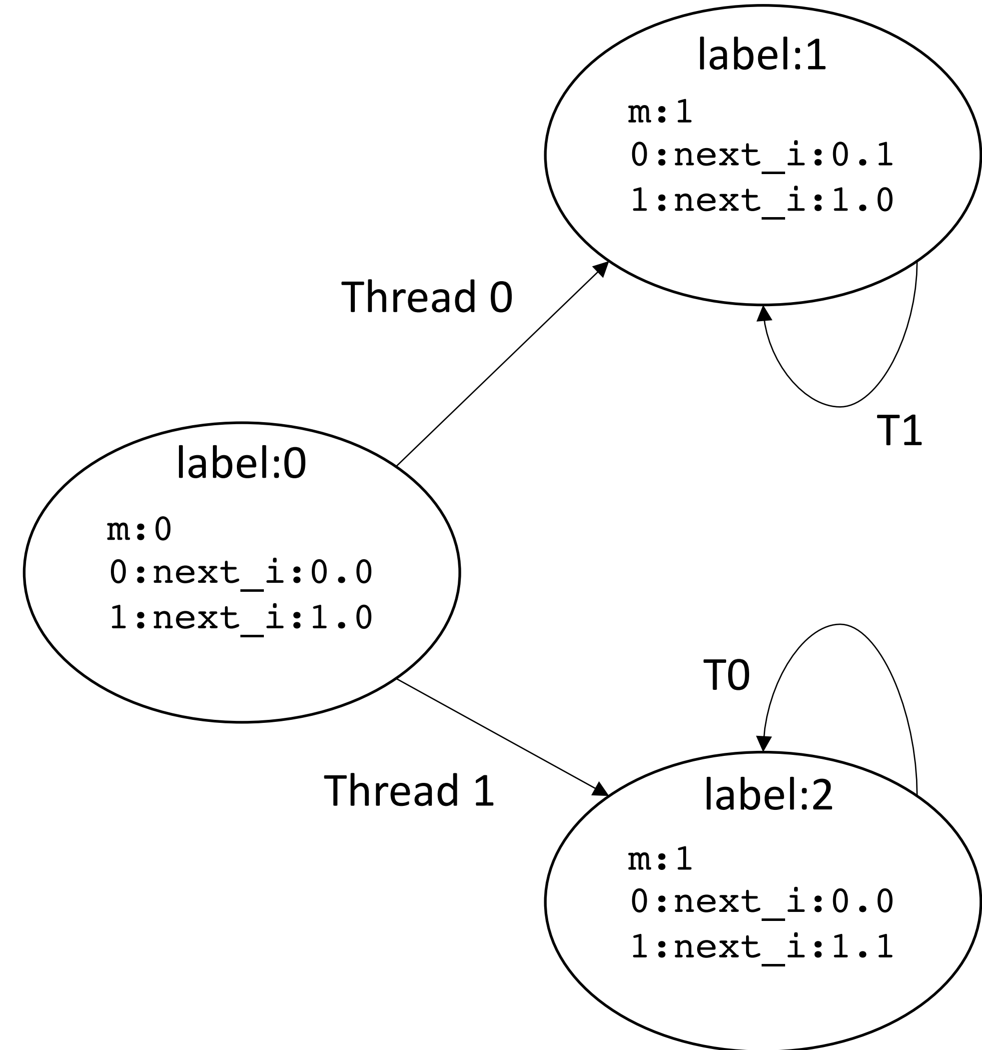
CSE113: Parallel Programming

Feb. 28, 2022

- **Topics**:
  - Forward progress
    - scheduler specifications



Announcements

- HW 4 is out
  - Due on Friday
  - You can share timing results
- Grades for HW 2 are out
  - last day to raise issues
- Grades for Midterm are just wrapping up
  - Expect them by the end of the day

Today’s Quiz

- Due Tomorrow by midnight; please do it!

Previous quiz

It is safe to run a program written for RMO on a TSO system?

- ○ True
- ○ False

Previous quiz

It is safe to run a program written for TSO on an RMO system

- ○ True
- ○ False

Previous quiz

The edges in a labeled transition system encode what?

Previous quiz

A cycle in the labeled transition system (LTS) indicates:

- ◯ a weak memory behavior has occurred
- ◯ a bug and the program must abort and retry
- ◯ a potential source of non-termination

Review

Relaxed memory models and mutexes

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

<u>Thread 0:</u>

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

<u>Thread 1:</u>

```
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)

<u>Thread 0:</u>

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

<u>Thread 1:</u>

```
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

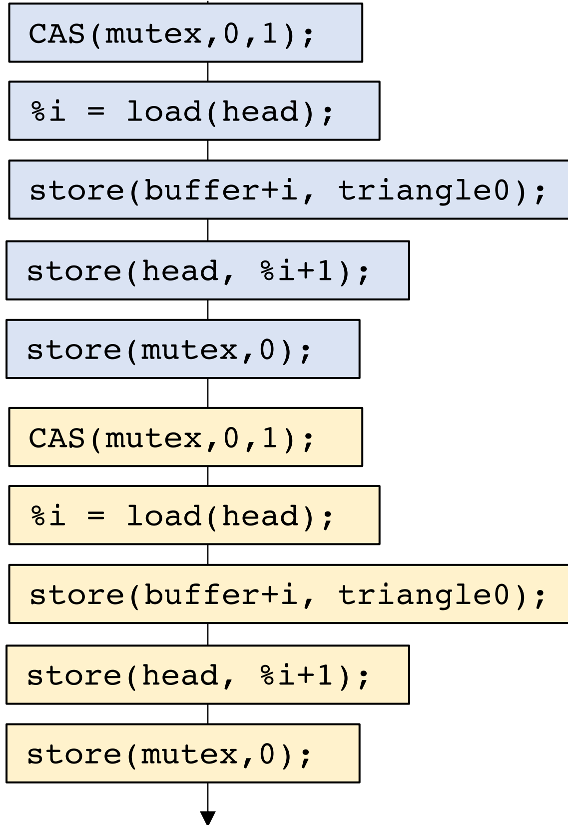
Thread 0:

```
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);
```

now yellow gets a change to go



Thread 0:

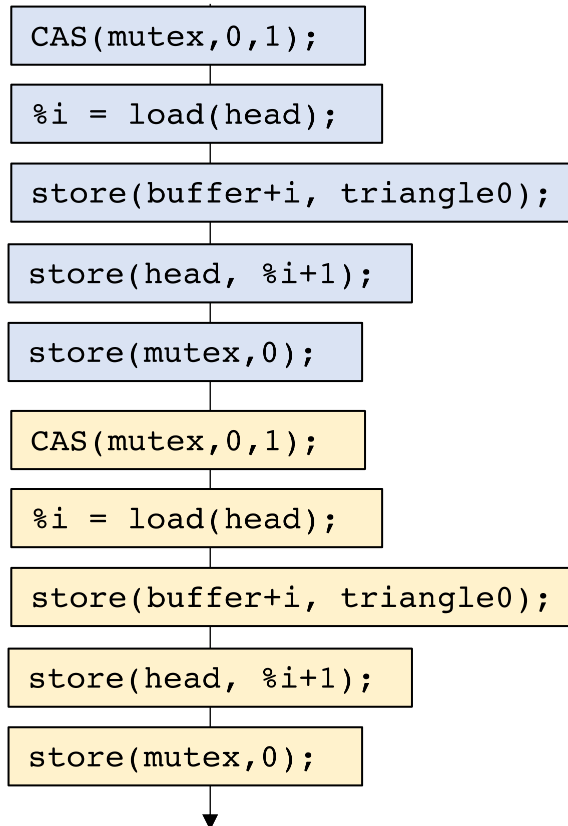
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



Thread 0:

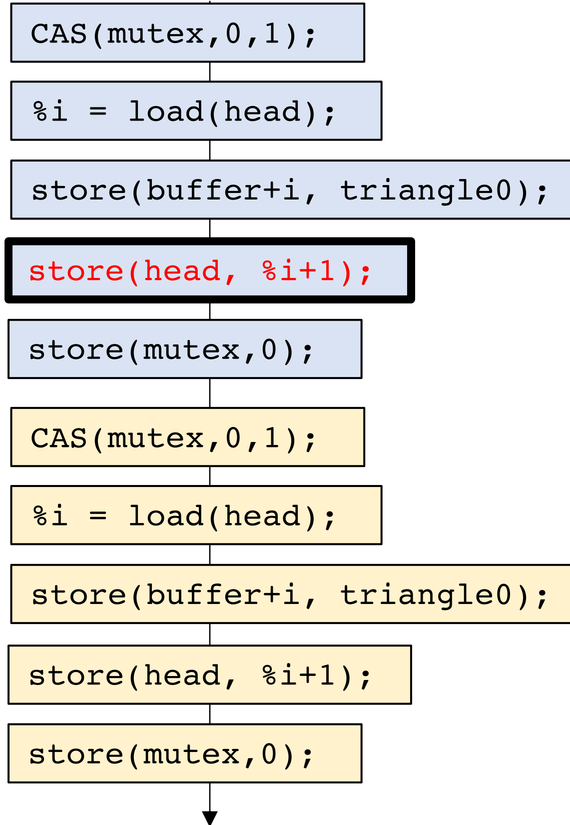
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



Thread 0:

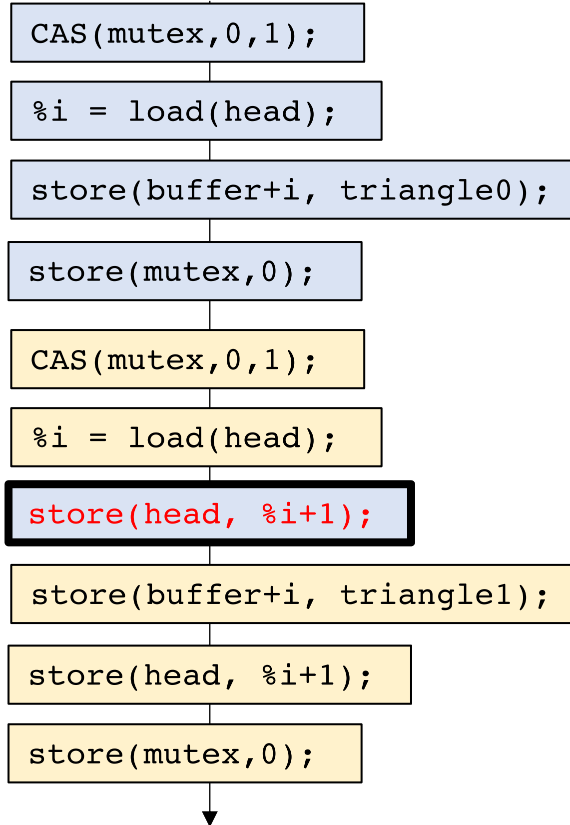
```
SPIN:CAS(mutex,0,1);
%i = load(head);
store(buffer+i, triangle0);
store(head, %i+1);
store(mutex,0);
```

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



What just happened if this store moves?

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!

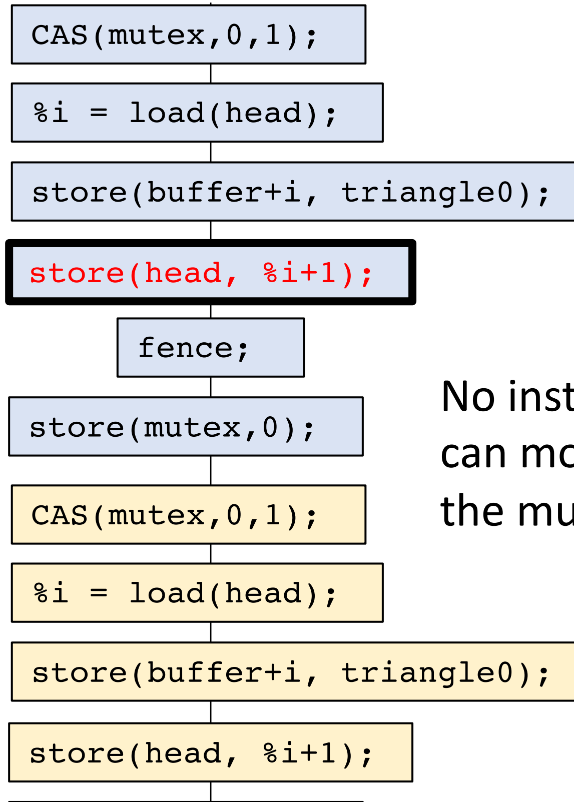
Thread 1:

```
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



No instructions can move after the mutex store!

How to fix the issue?

your unlock function should contain a fence!

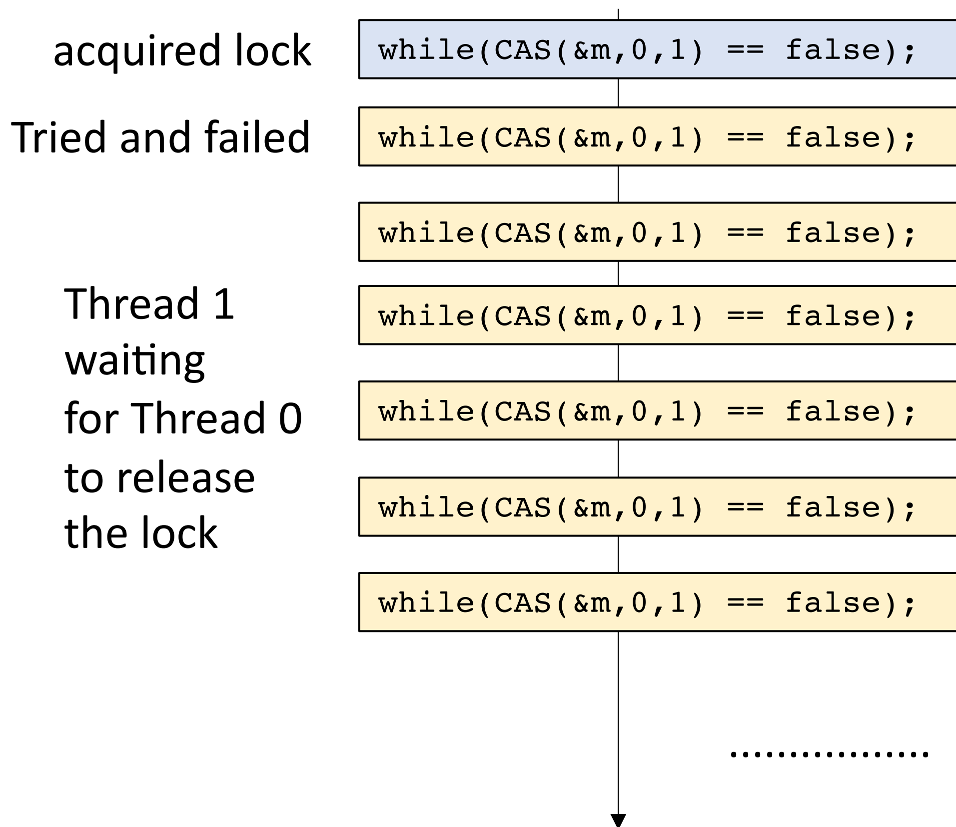
Scheduler specifications

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?

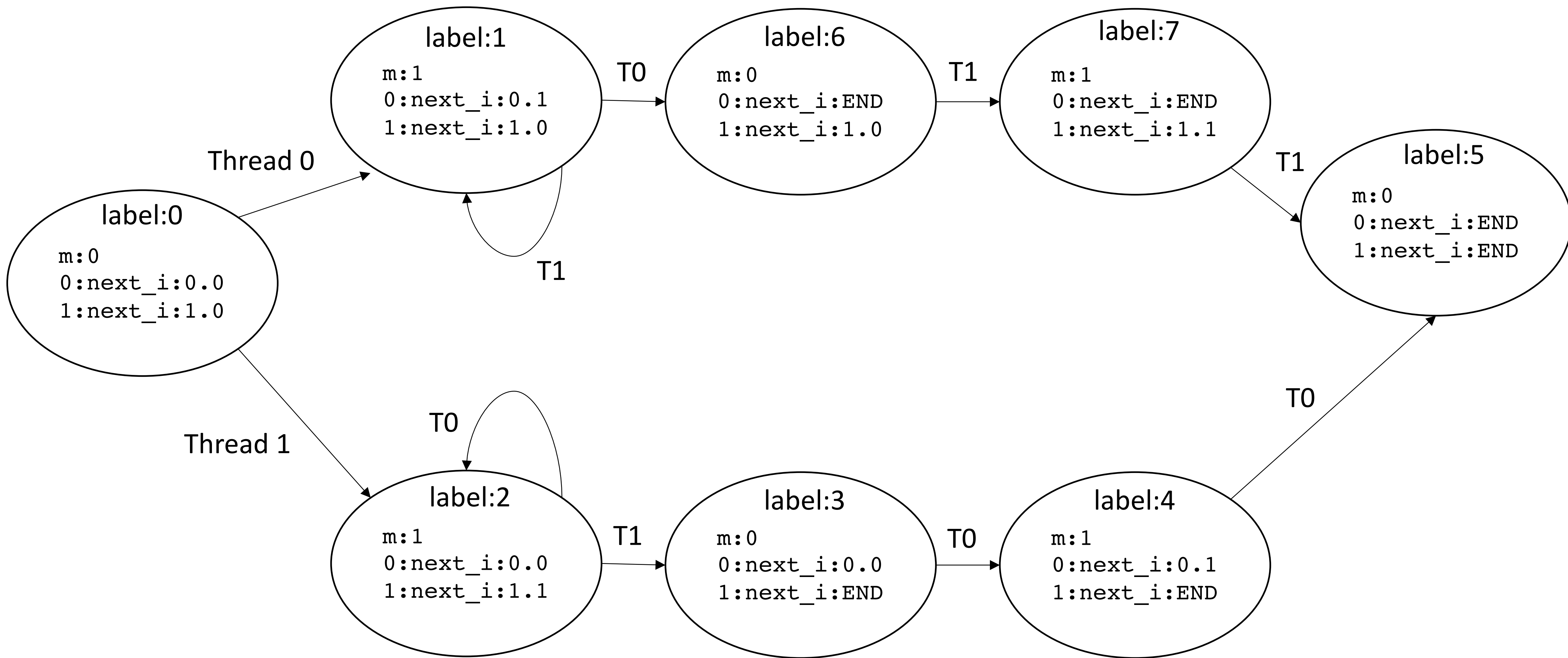
assuming sequential consistency

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

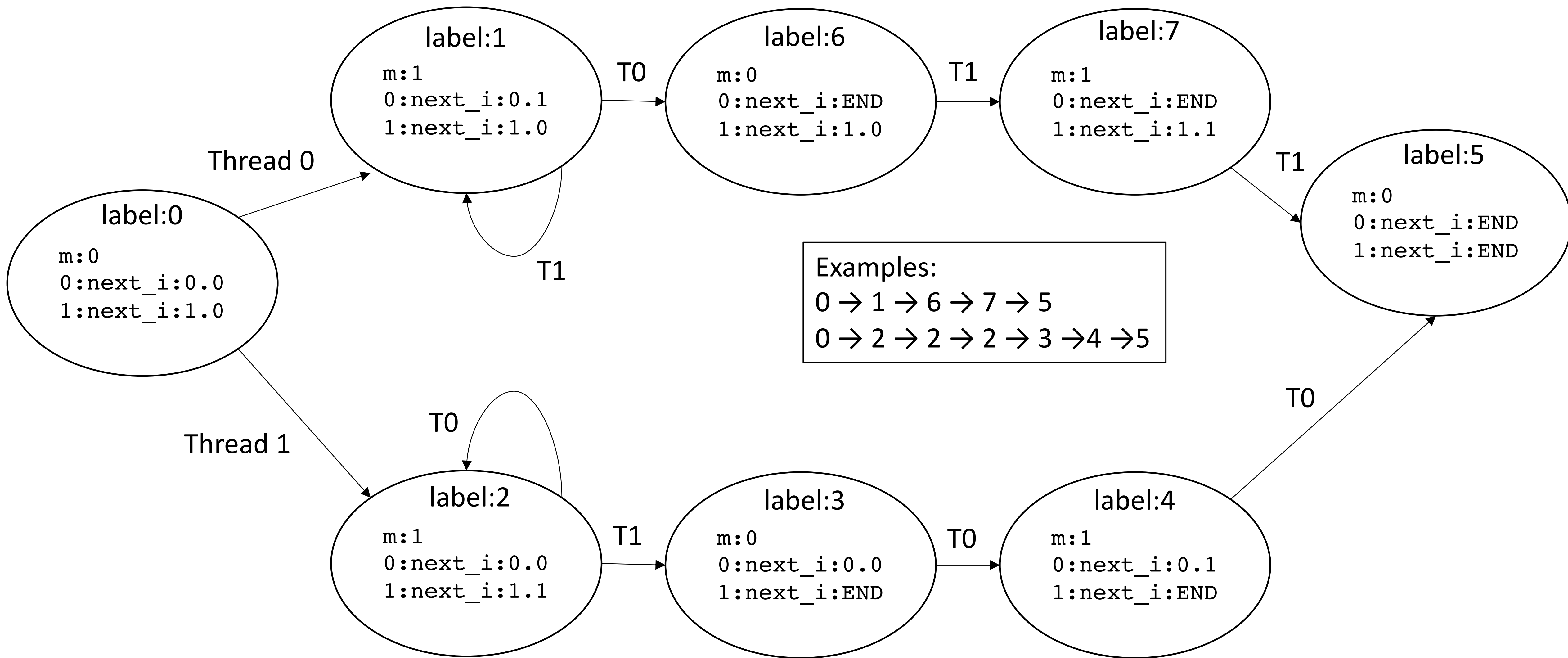
  - A concurrent execution is any path through the LTS

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
```



Liveness property

- Something good will eventually happen

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

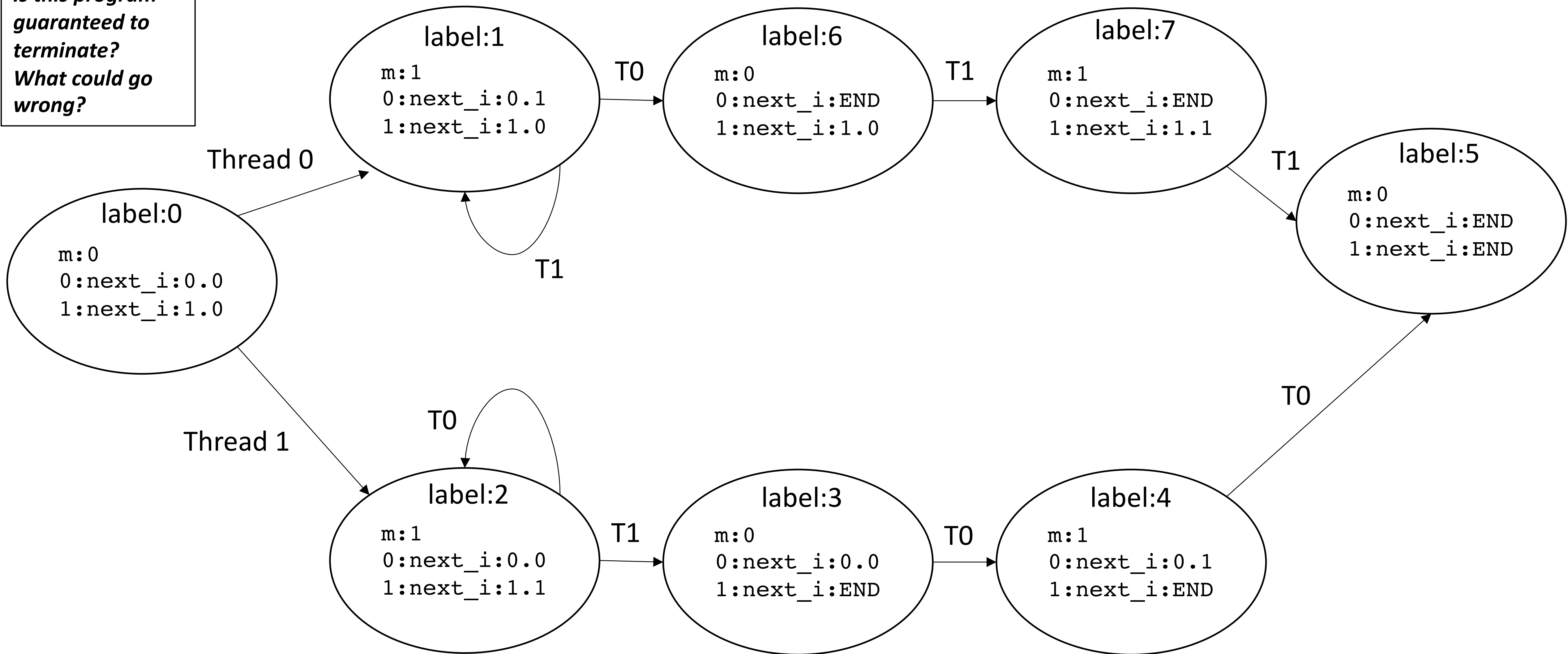
- More difficult to reason about that safety properties

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
```

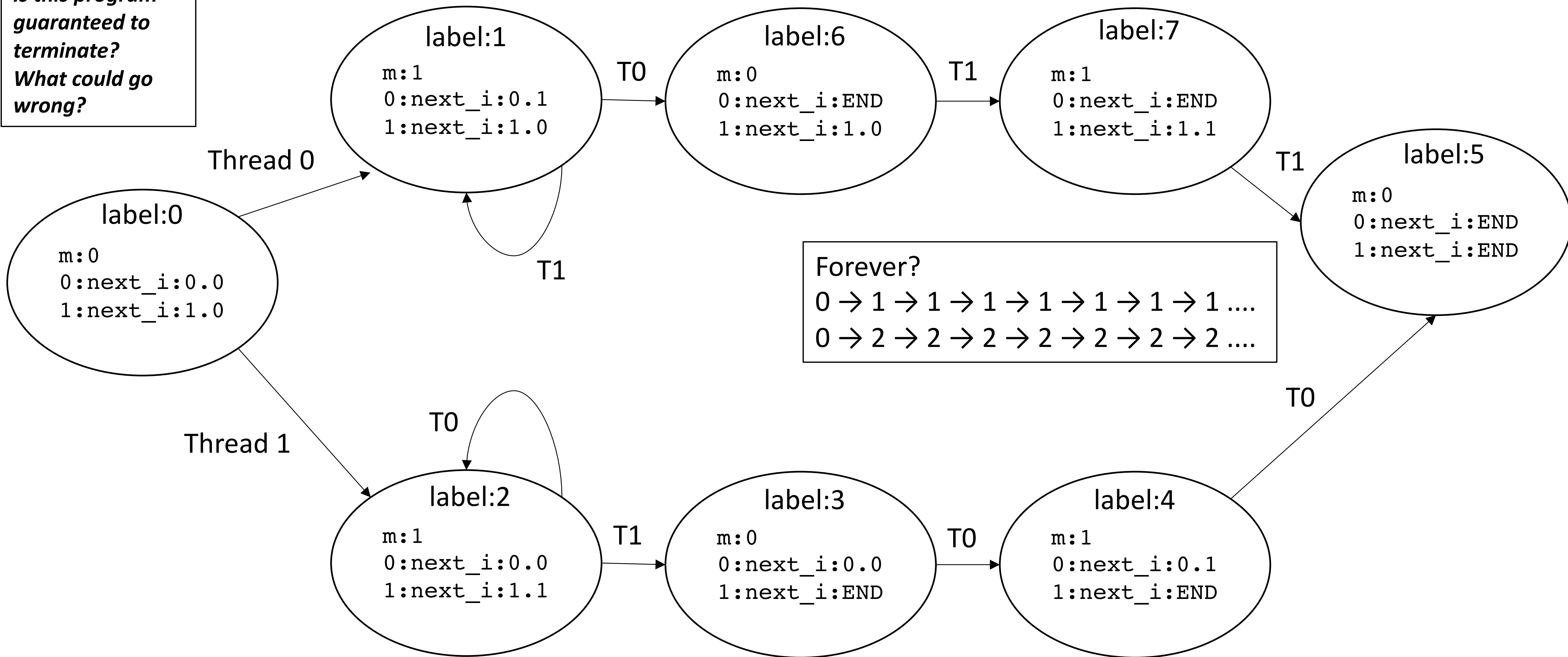


Thread 0:

```
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)

On to new material

Schedule

- Schedule specifications
  - Fair schedule

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)

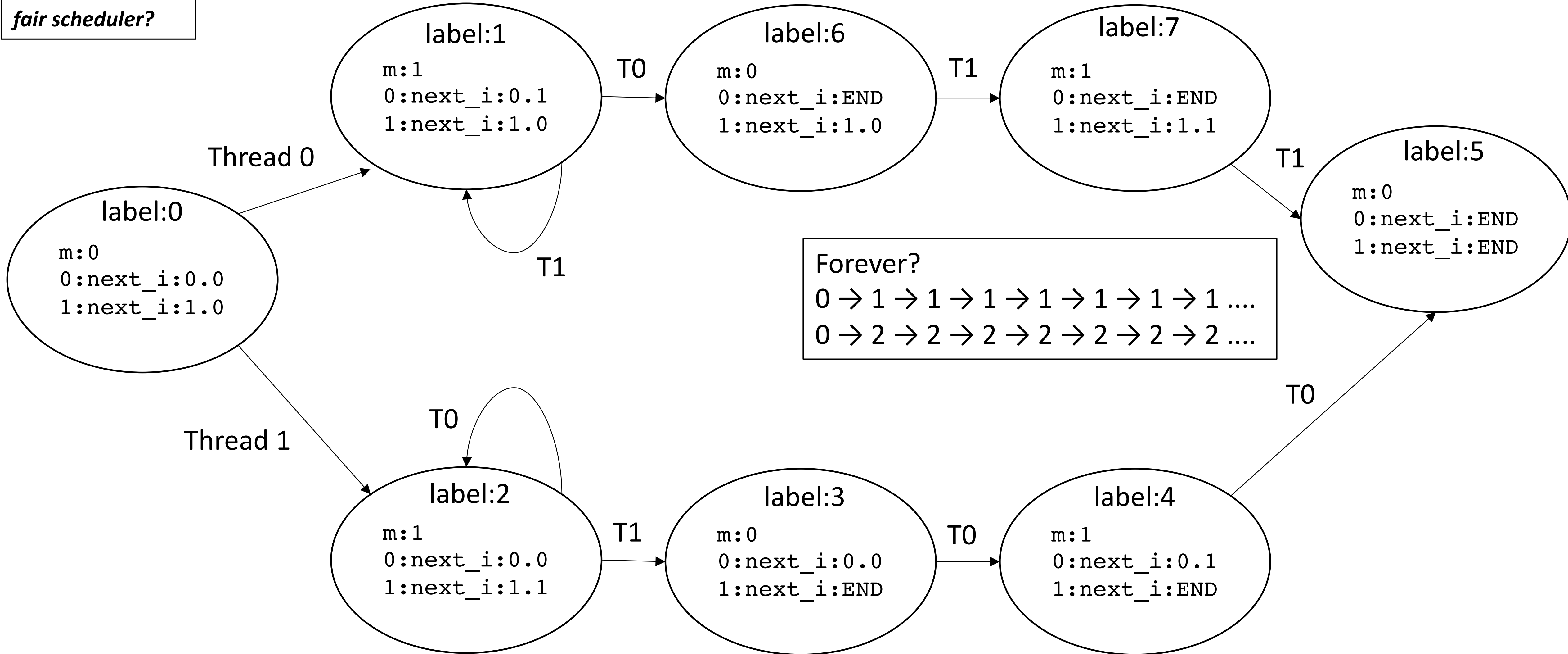
Thread 0:

```
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?

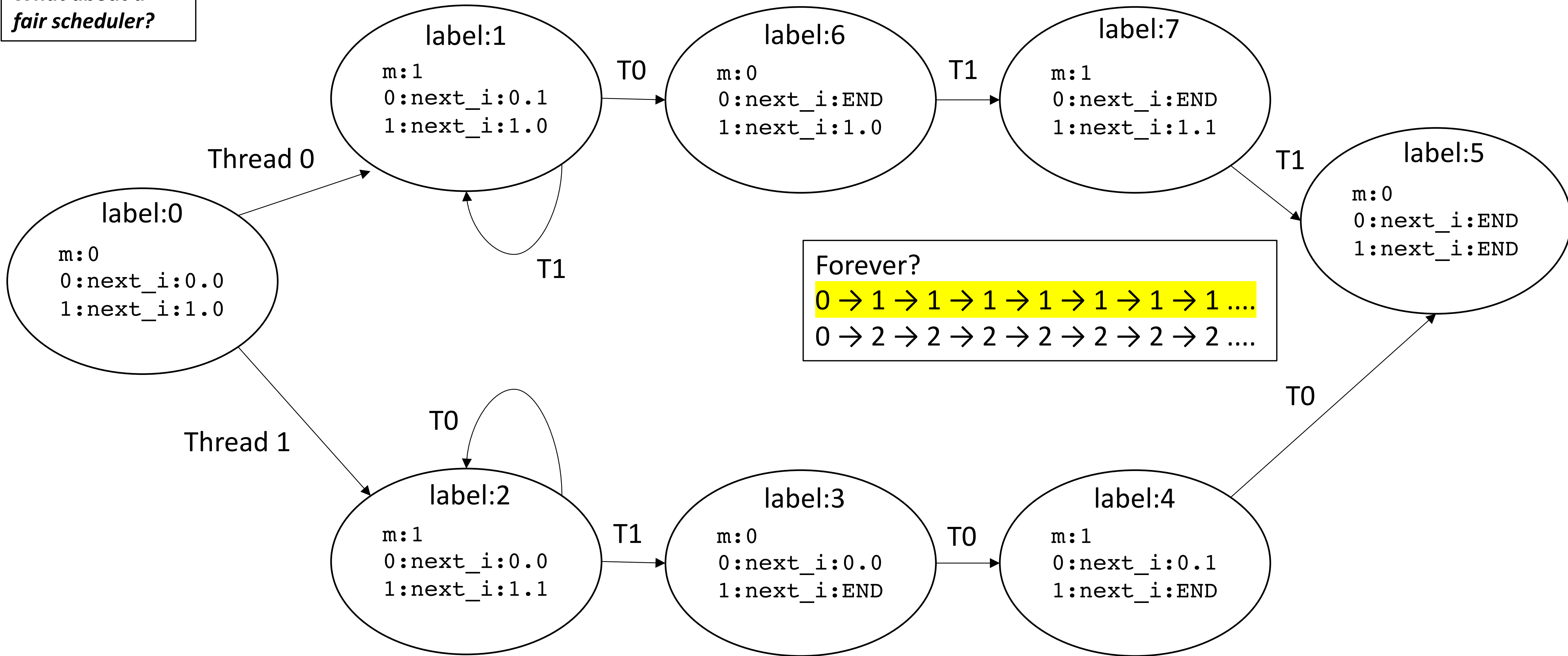


Thread 0:

```
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
```

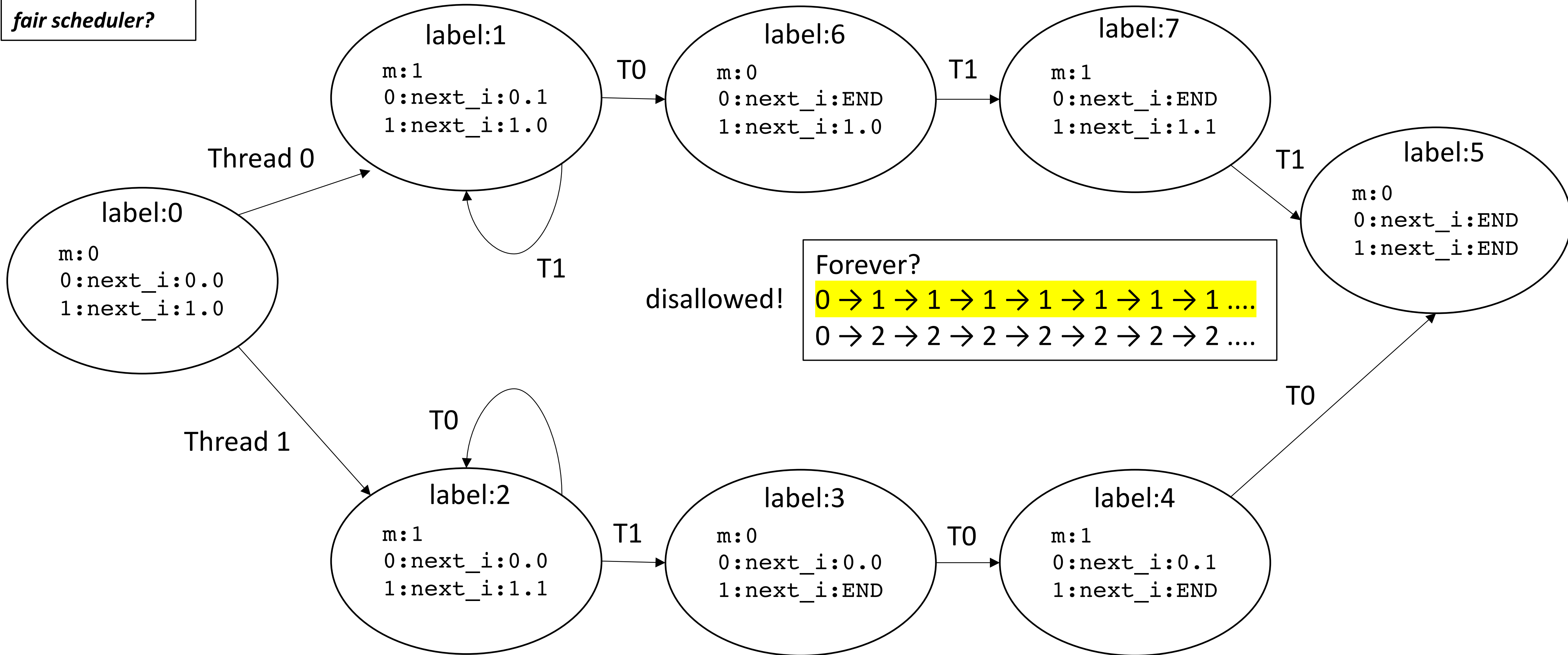


Thread 0:

```
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
```



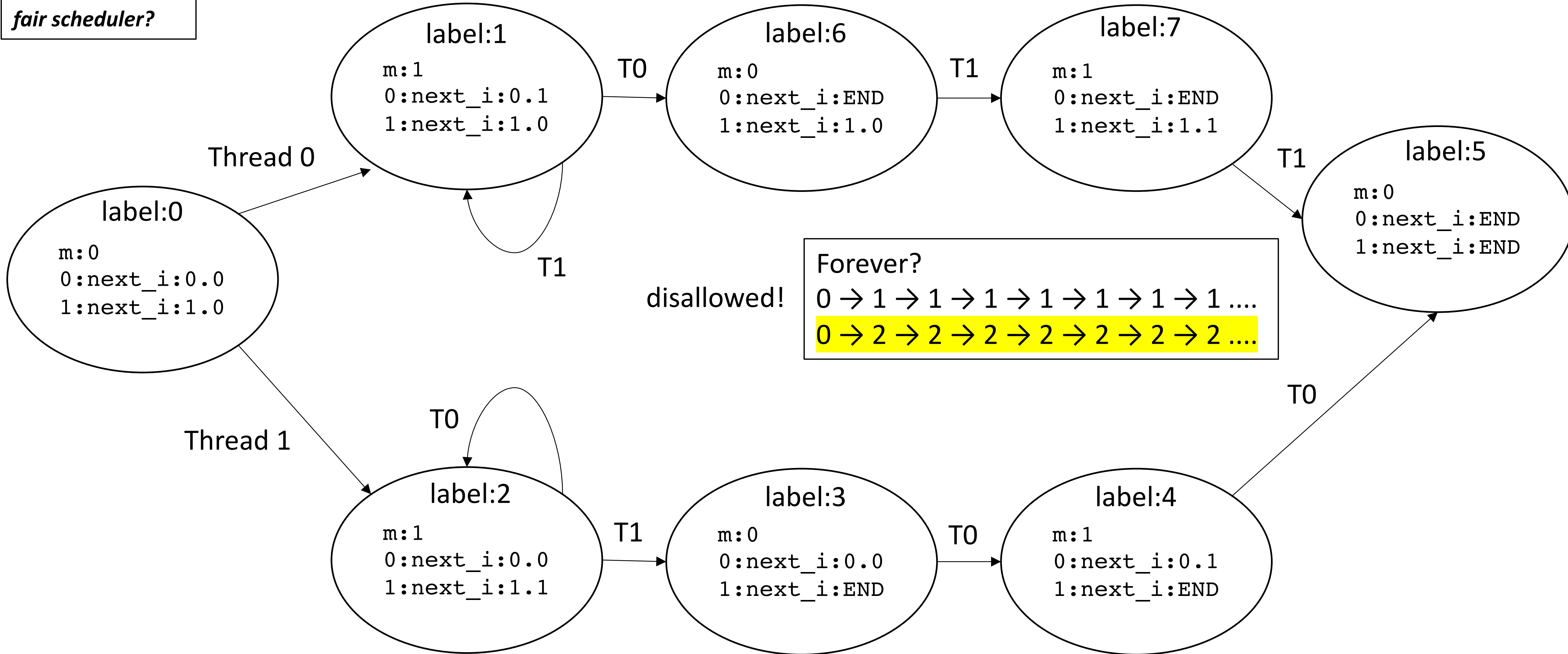
Thread 0:

```
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 about a fair scheduler?



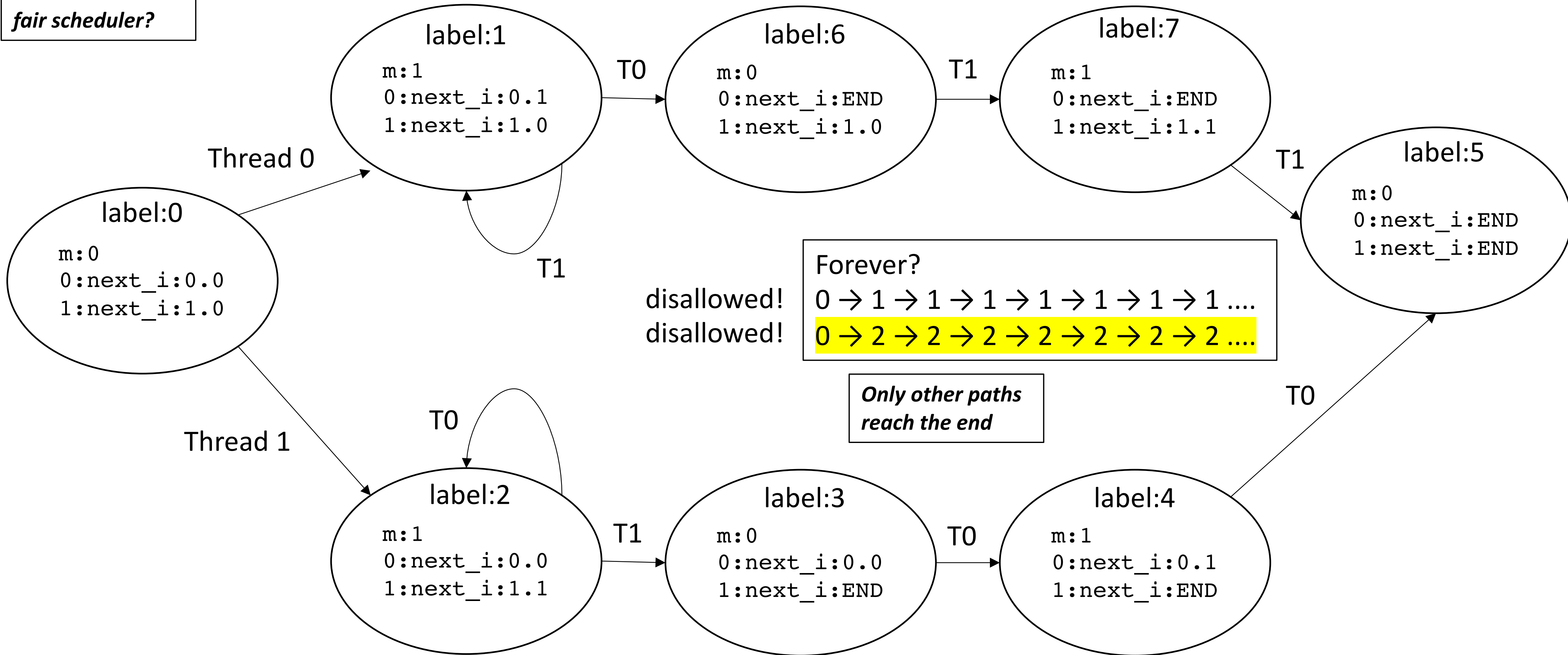
Thread 0:

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

Thread 1:

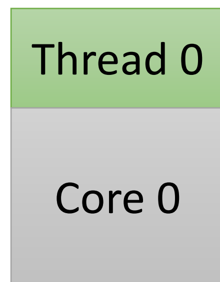
```
1.0: while(CAS(&m,0,1) == false); //lock
     // critical section
1.1: m.store(0); //unlock
```

What about a fair scheduler?



Schedulers

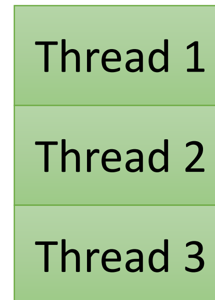
- A fair scheduler typically requires preemption



resources

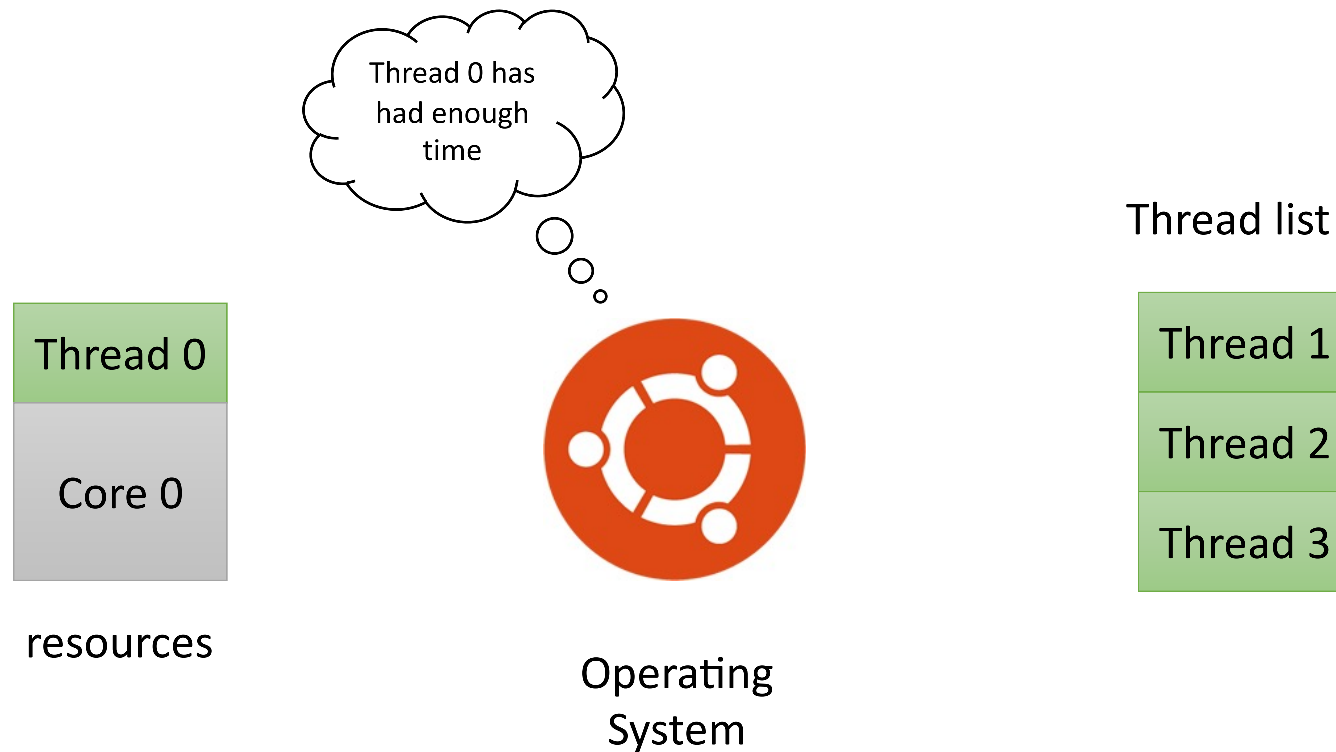
Operating System

Thread list



Schedulers

- A fair scheduler typically requires preemption



Schedulers

- A fair scheduler typically requires preemption

Schedulers

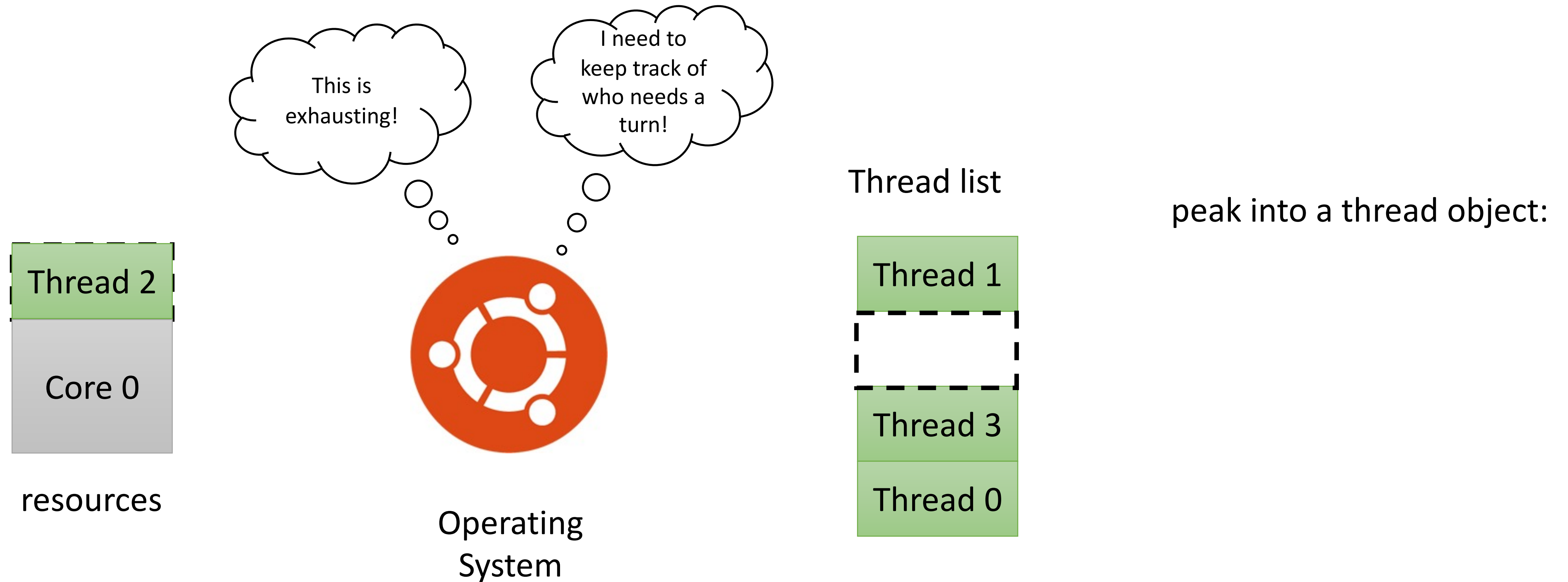
- A fair scheduler typically requires preemption

Schedulers

- A fair scheduler typically requires preemption

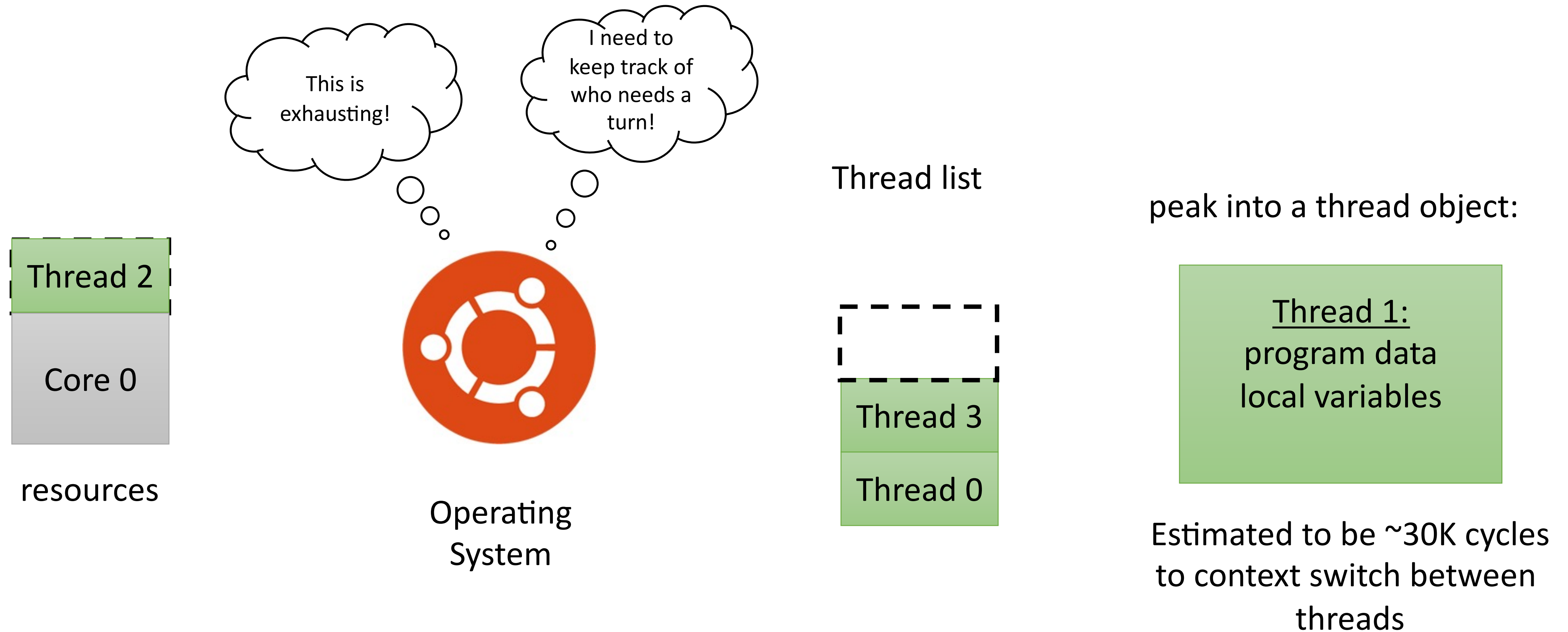
Schedulers

- A fair scheduler typically requires preemption



Schedulers

- A fair scheduler typically requires preemption

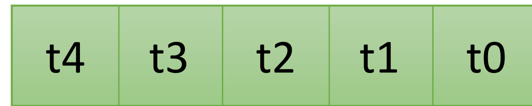


Schedulers

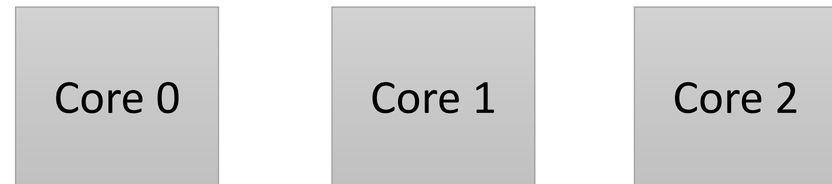
- Systems might not support preemption: e.g. GPUs

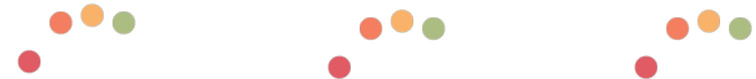
simplified execution model

Program with 5 threads



thread pool





Device with 3 Cores

finished threads

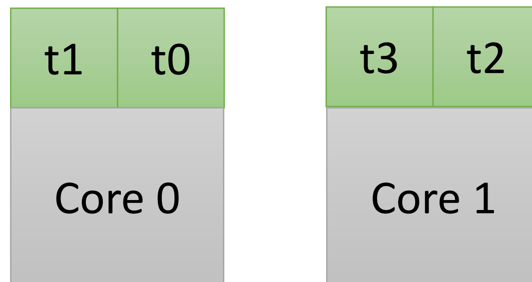
Solutions?

- I have N cores, only run N threads?

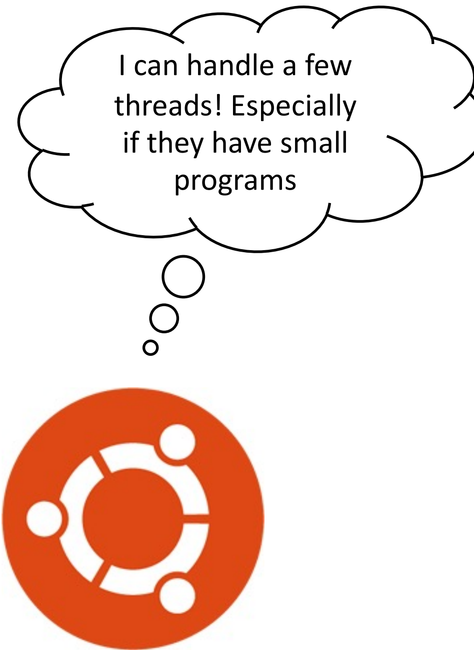
Solutions?

- 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”

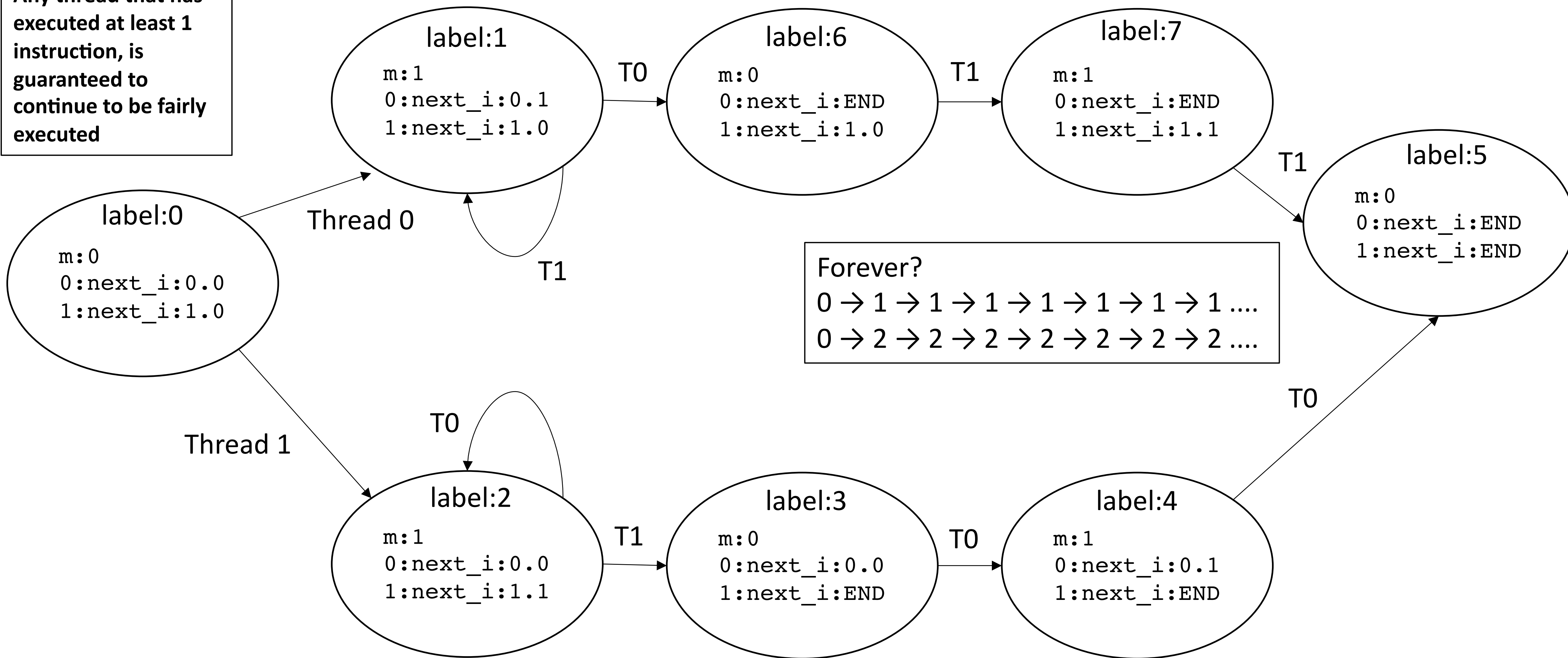
- Also called:
  - “Parallel Forward Progress”: by C++
  - “Persistent Thread Model”: by GPU programmers
  - ”Occupancy Bound Execution Model”: in some of my papers

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
```

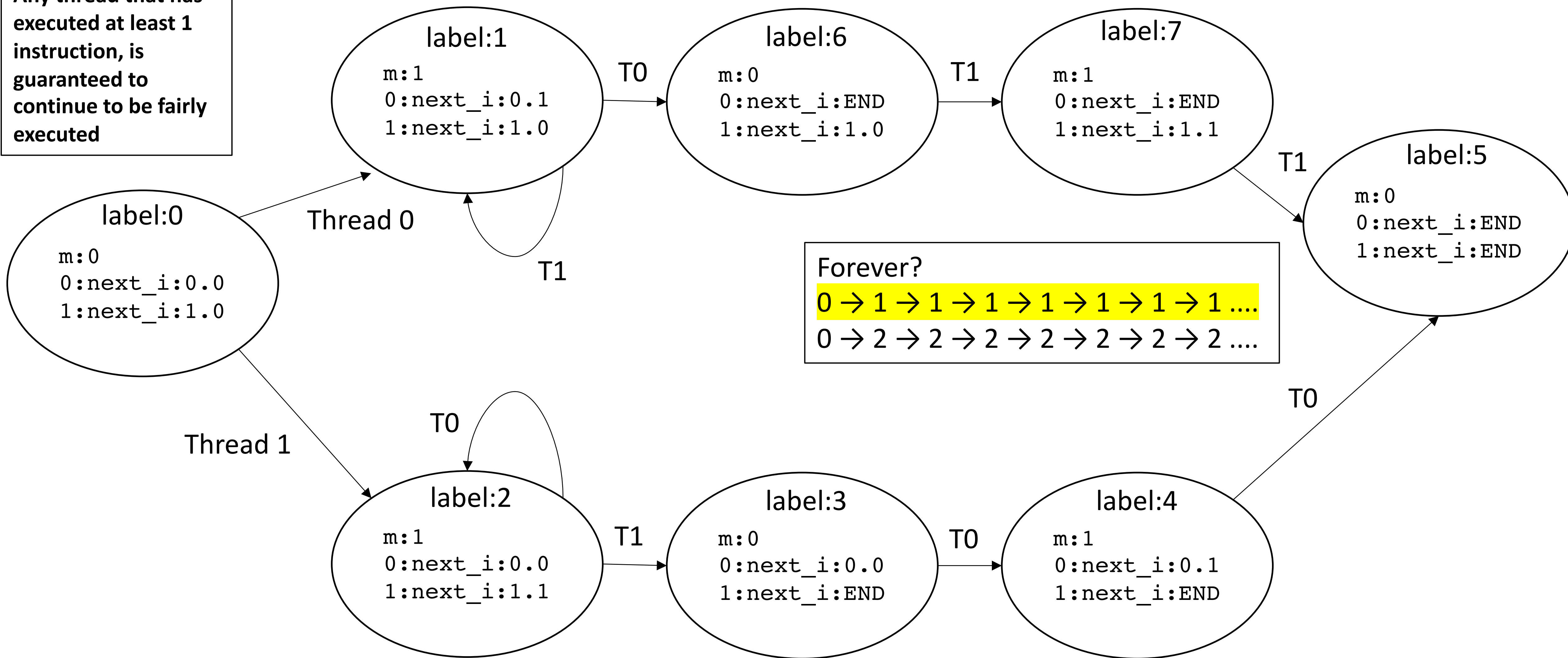


Thread 0:

```
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
```

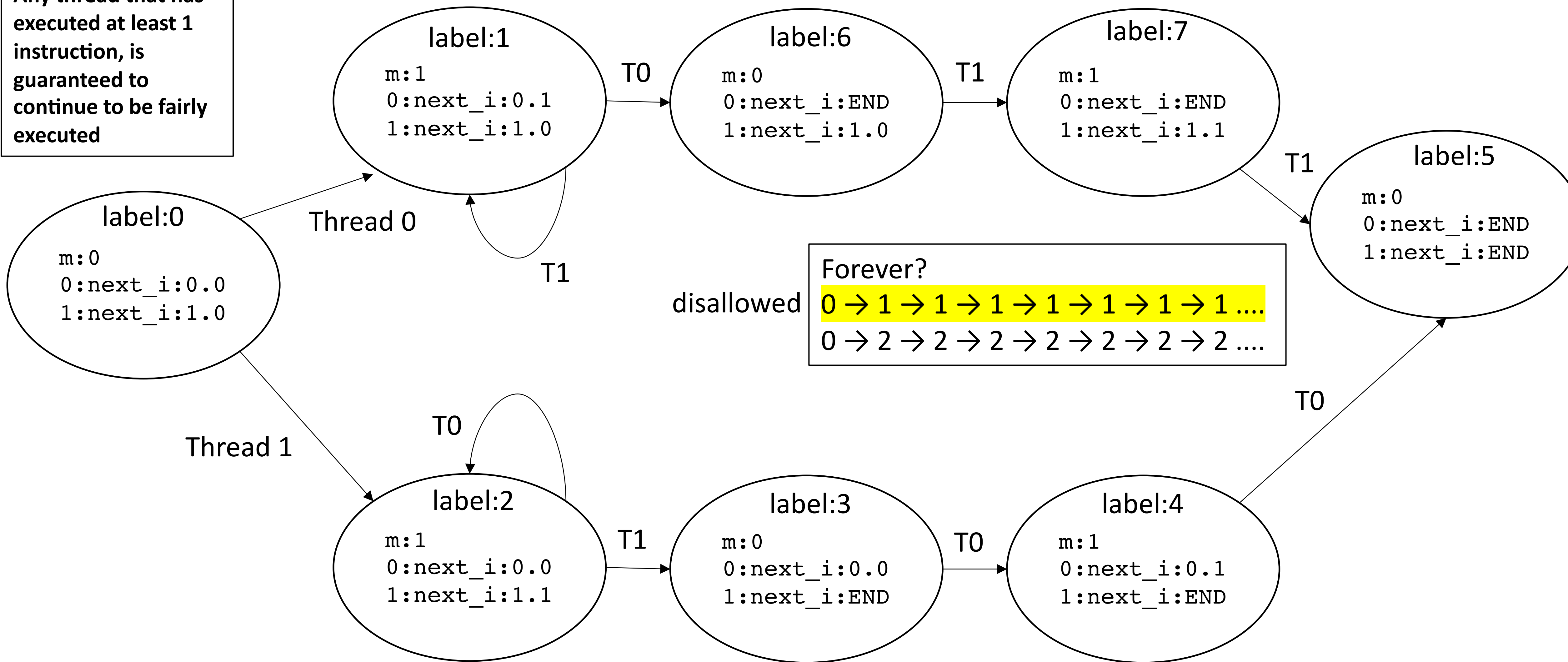


Thread 0:

```
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
```

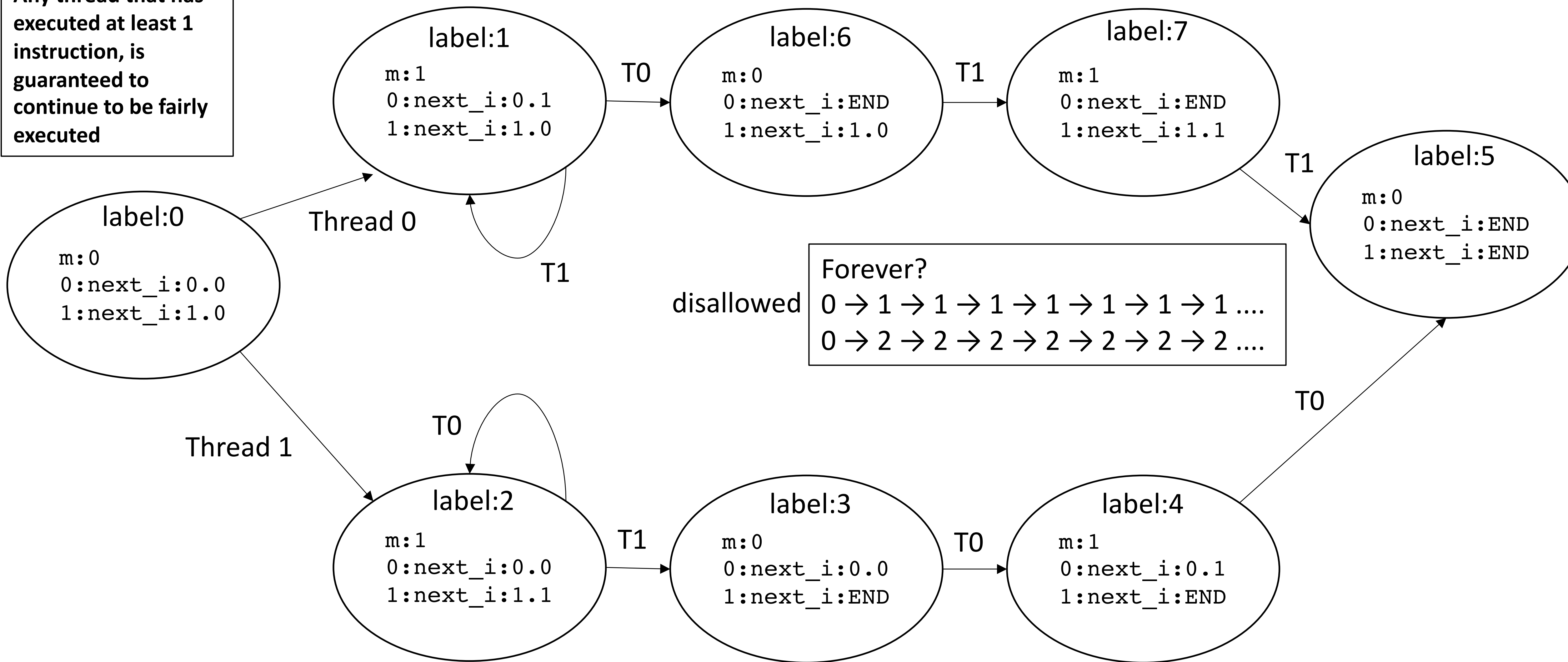


Thread 0:

```
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
```

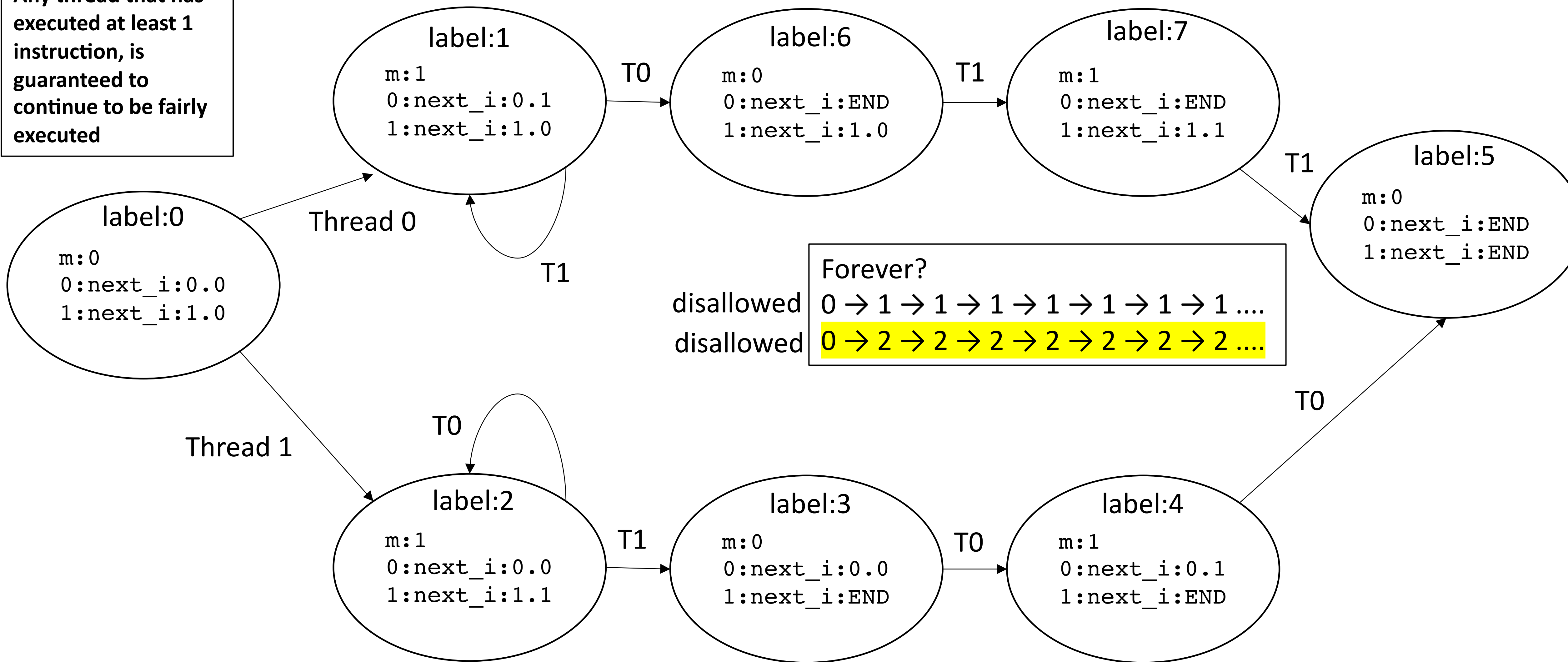


Thread 0:

```
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
```

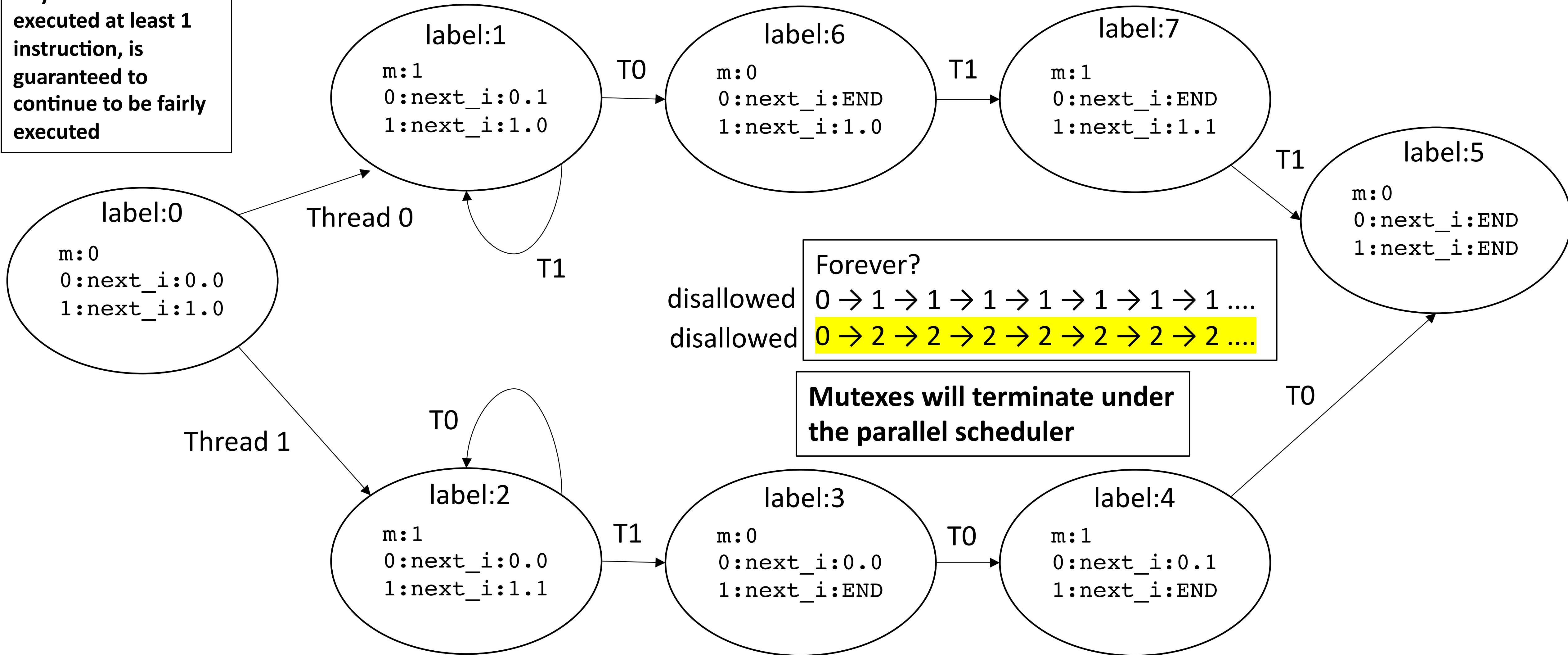


Thread 0:

```
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

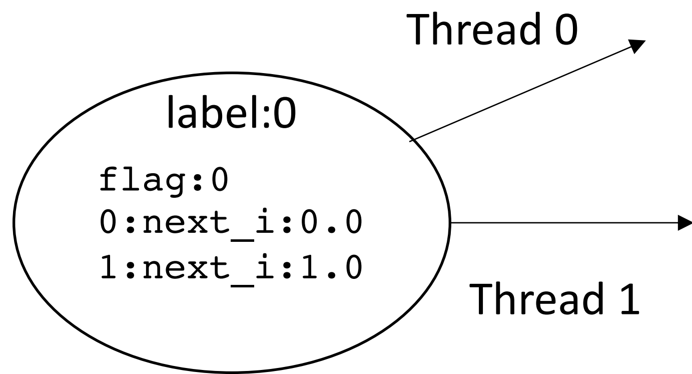
```
flag:0
0:next_i:0.0
1:next_i:1.0
```

Thread 0:

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

Thread 1:

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

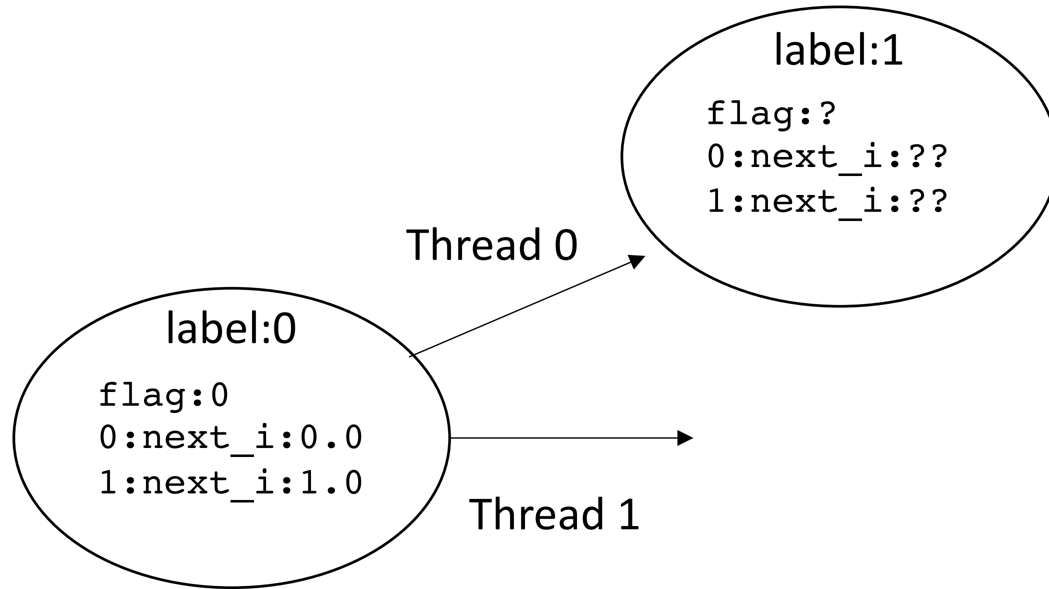


Thread 0:

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

Thread 1:

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

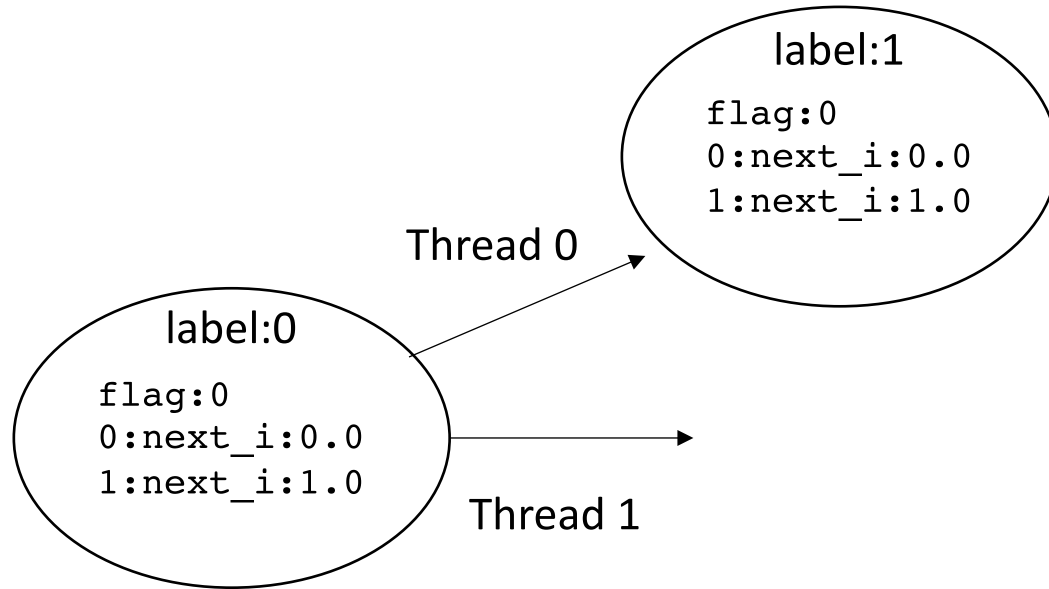


Thread 0:

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

Thread 1:

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

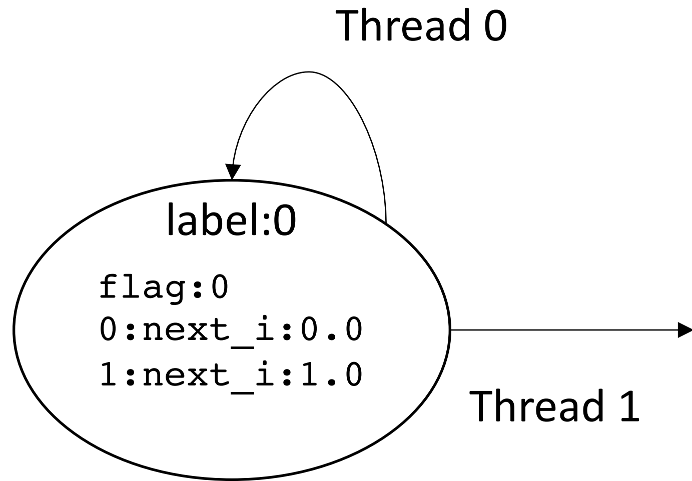


Thread 0:

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

Thread 1:

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

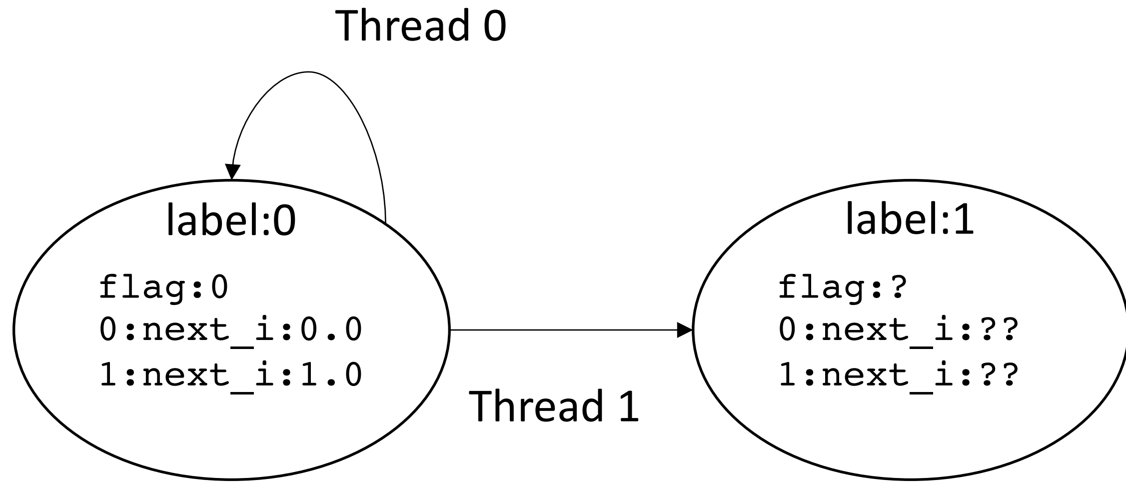


Thread 0:

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

Thread 1:

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

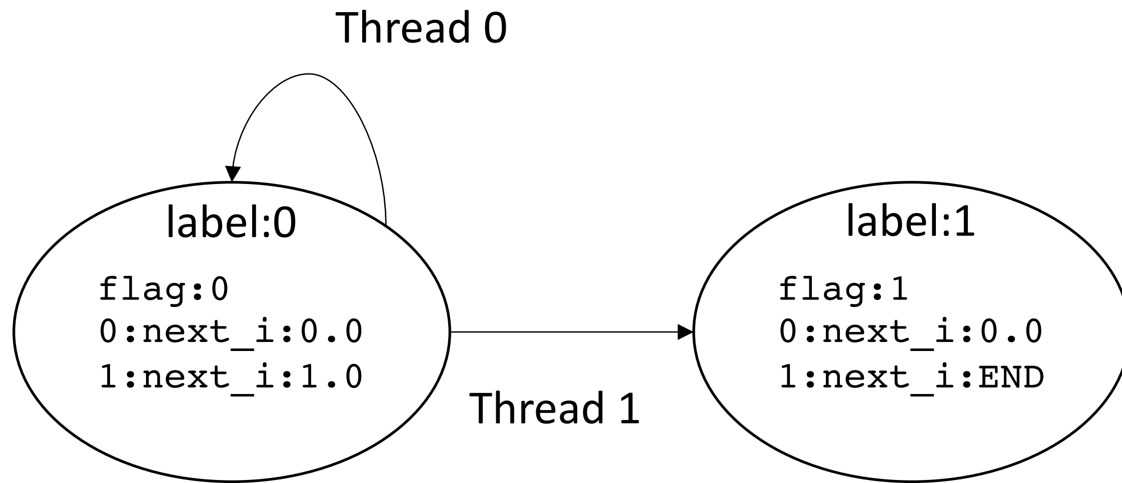


Thread 0:

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

Thread 1:

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

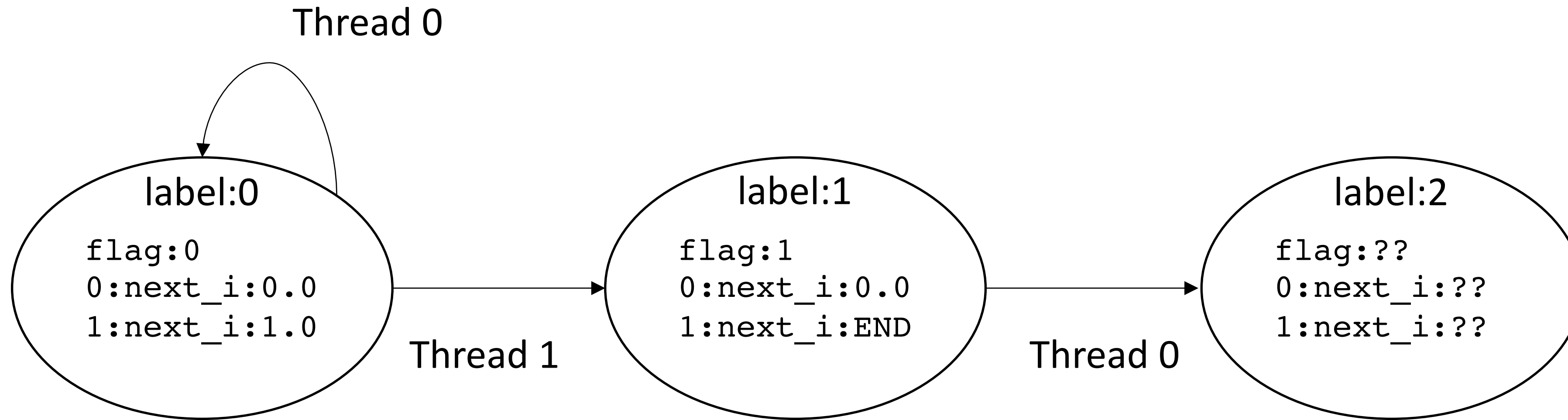


Thread 0:

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

Thread 1:

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

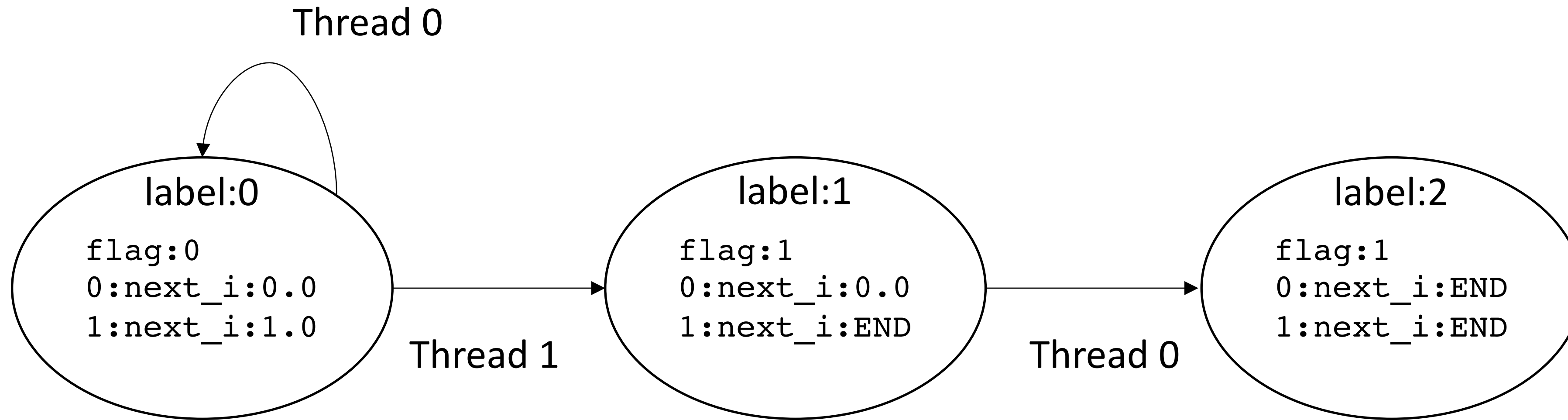


Thread 0:

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

Thread 1:

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



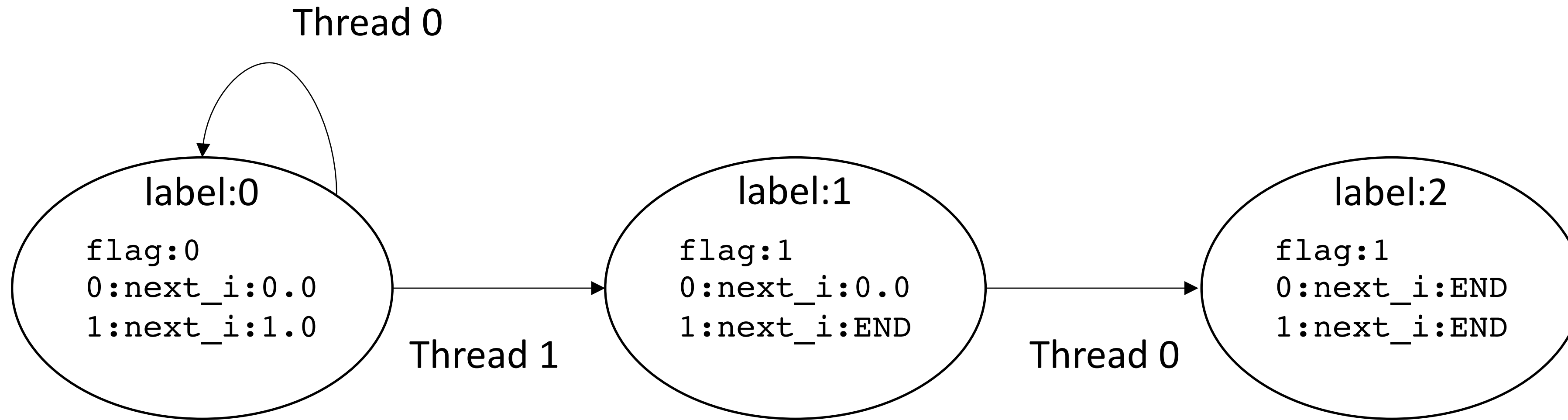
Thread 0:

```
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:

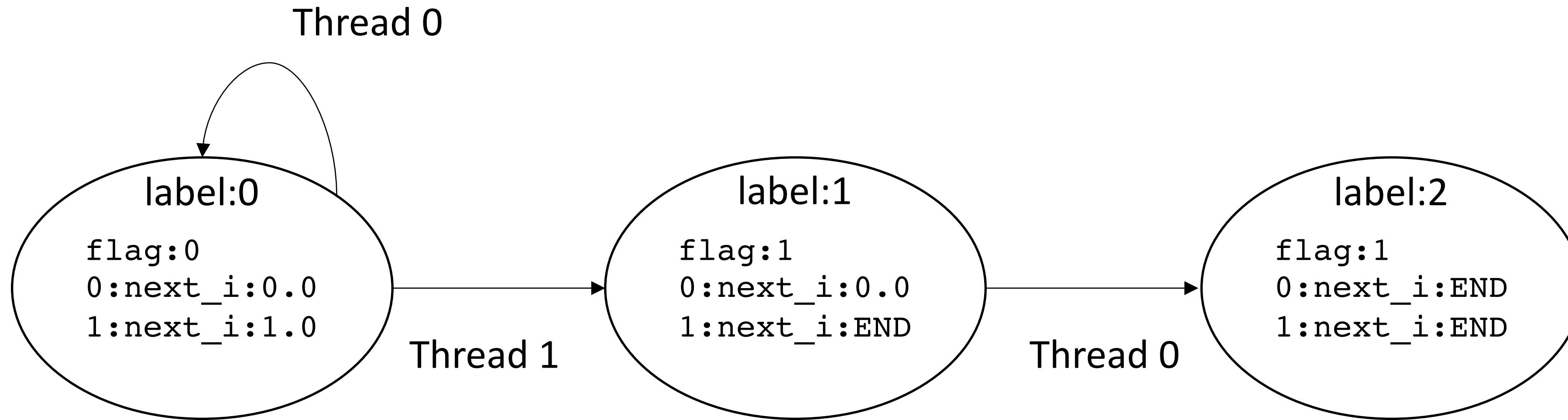
```
0.0: while(flag.load() == 0);
```

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);
```

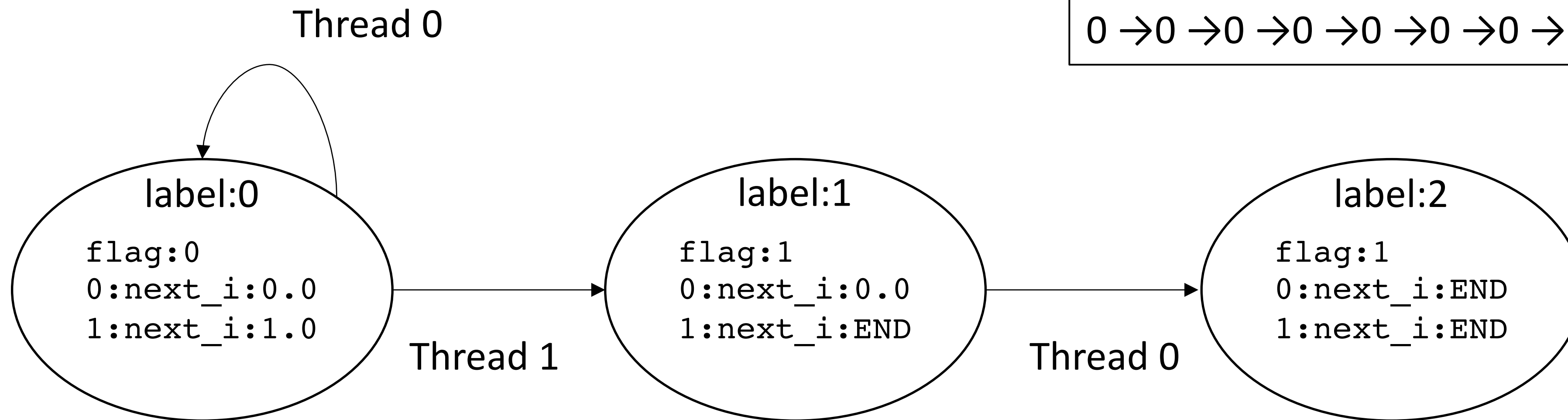
Is this program guaranteed to terminate under the fair scheduler?

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);
```

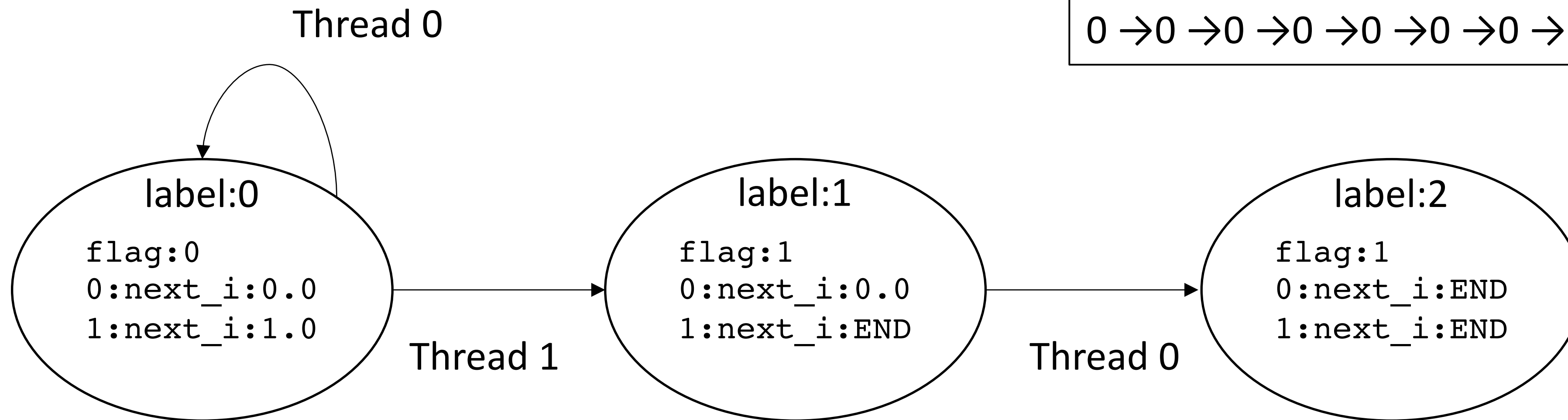
Thread 1:

```
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!

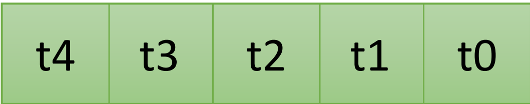
Thread 0 could be scheduled on the only core while thread 1 spins

Schedulers

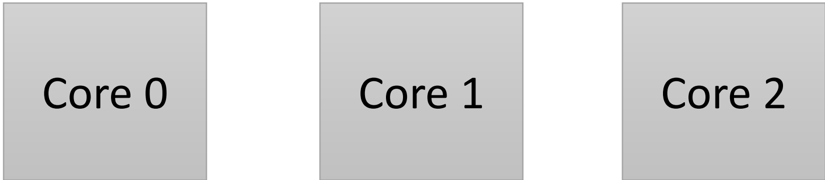
- In some cases the Parallel scheduler might be too strong
- For example dynamic power management on mobile devices

A power-saving scheduler

Program with 5 threads



thread pool



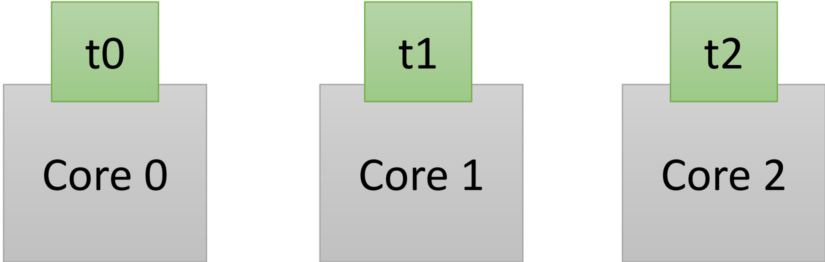
Device with 3 Cores

finished threads

A power-saving scheduler

Program with 5 threads

thread pool



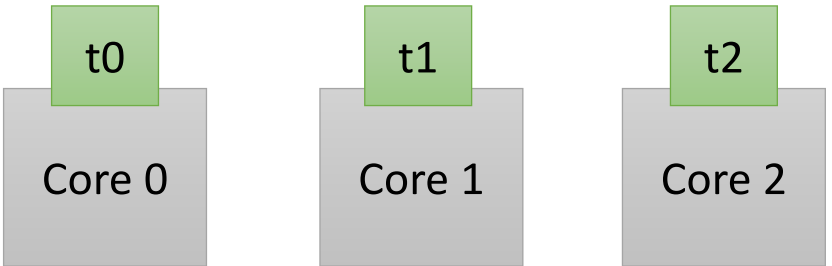
Device with 3 Cores

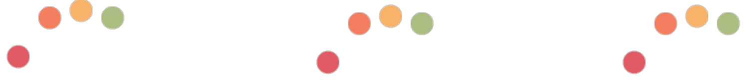
finished threads

A power-saving scheduler

Program with 5 threads

thread pool





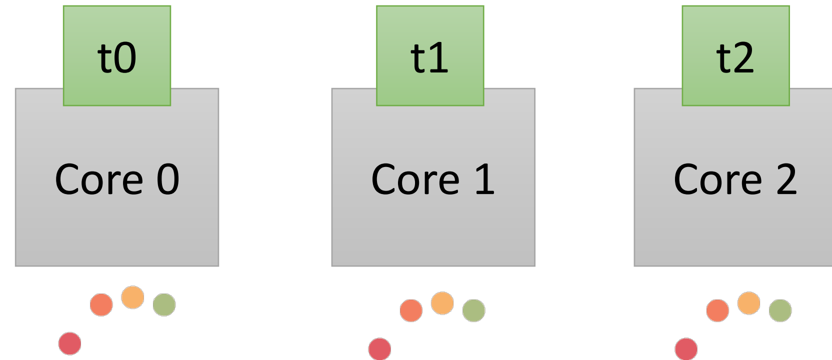
Device with 3 Cores

finished threads

A power-saving scheduler

Program with 5 threads

thread pool



Device with 3 Cores

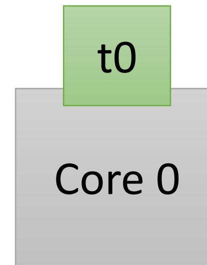
finished threads

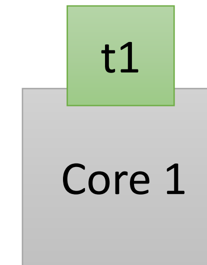
A power-saving scheduler

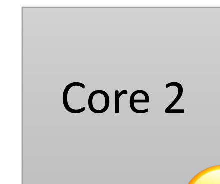
Program with 5 threads

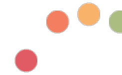
thread pool

preempted









Device with 3 Cores

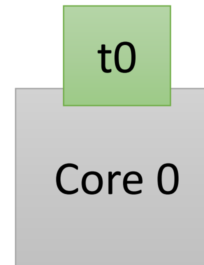
finished threads

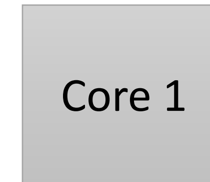
A power-saving scheduler

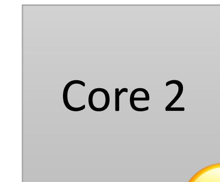
Program with 5 threads

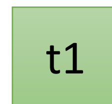
thread pool

preempted









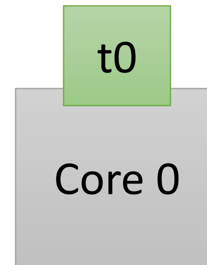
finished threads

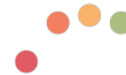
Device with 3 Cores

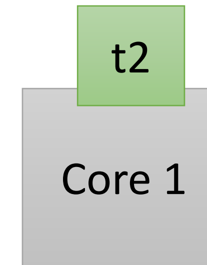
A power-saving scheduler

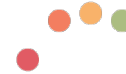
Program with 5 threads

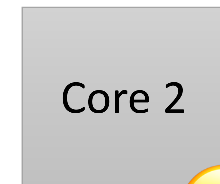
thread pool

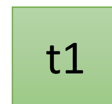












Device with 3 Cores

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 scheduler

- 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.

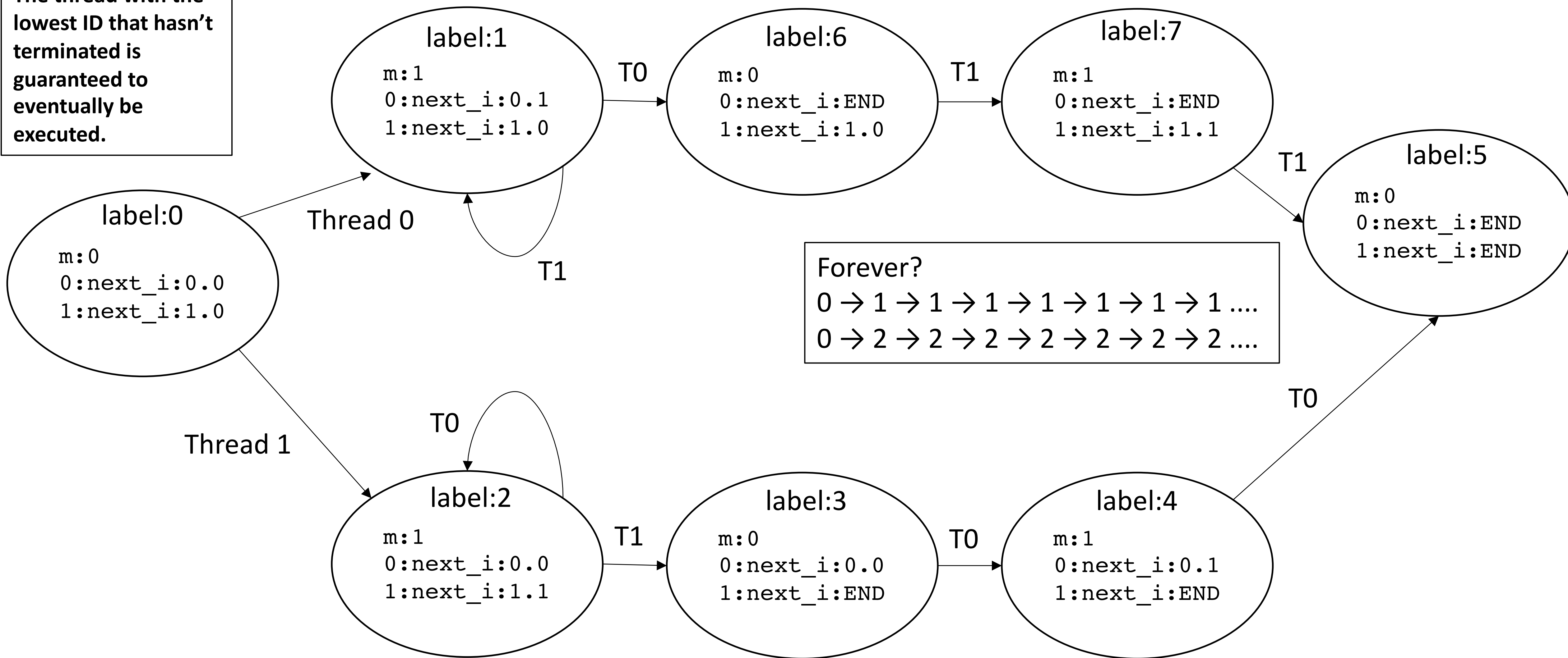
    - The HSA language appears to be defunct now, but the scheduler is a good fit for mobile devices (esp. mobile GPUs).

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
```

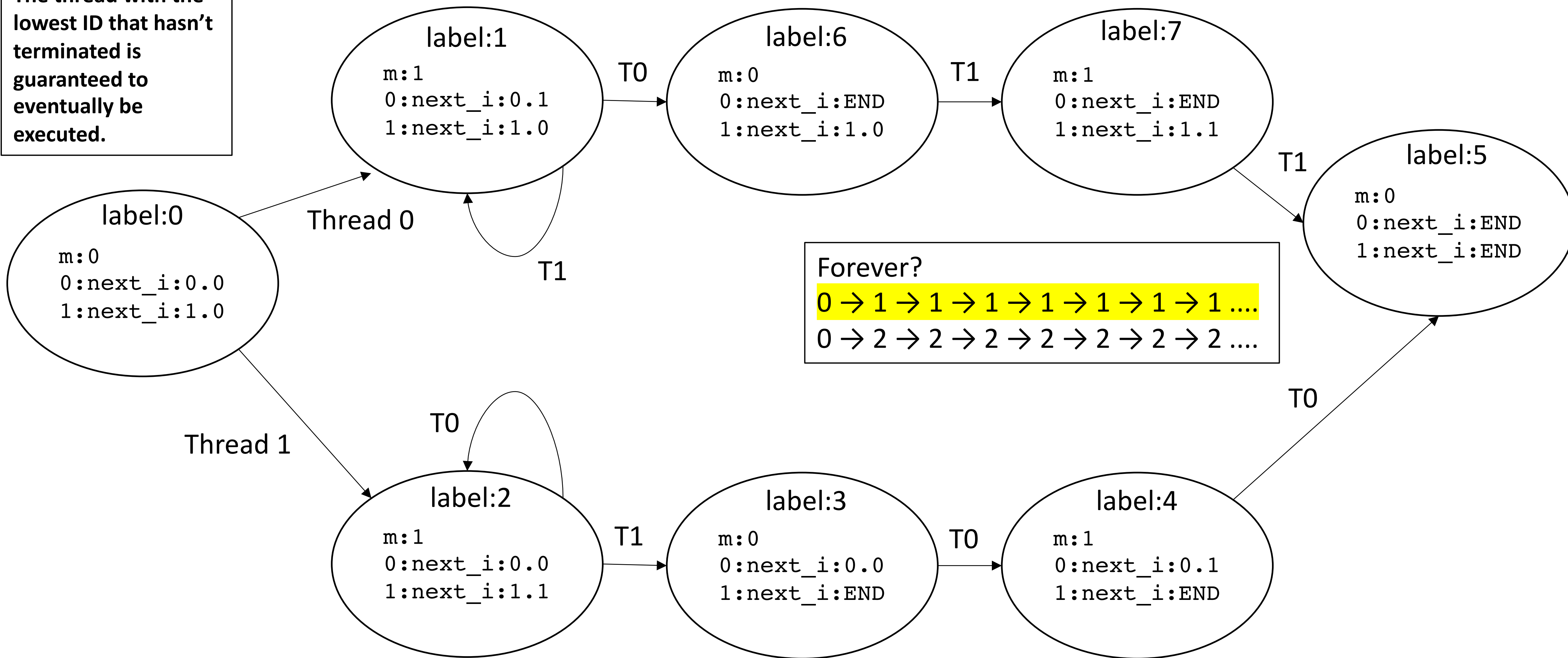


Thread 0:

```
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
```

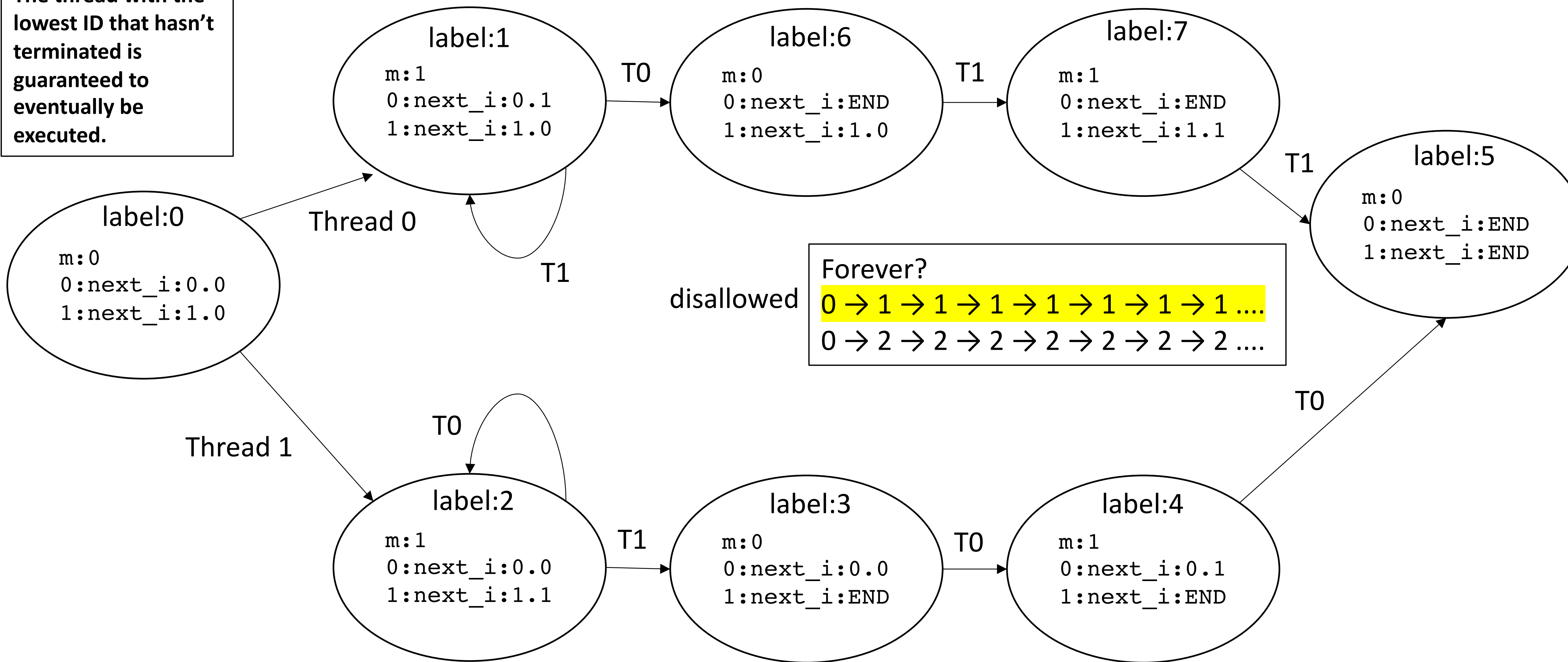


Thread 0:

```
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
```

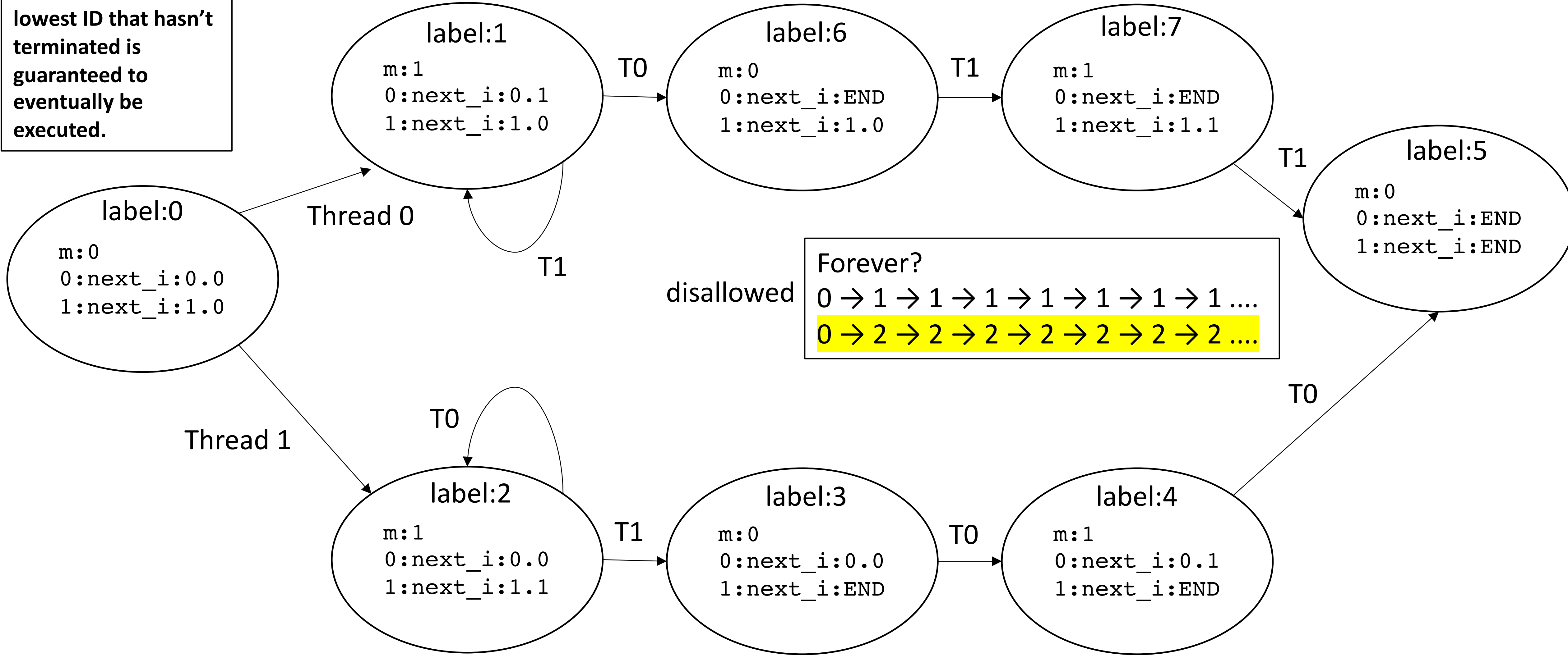


Thread 0:

```
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
```

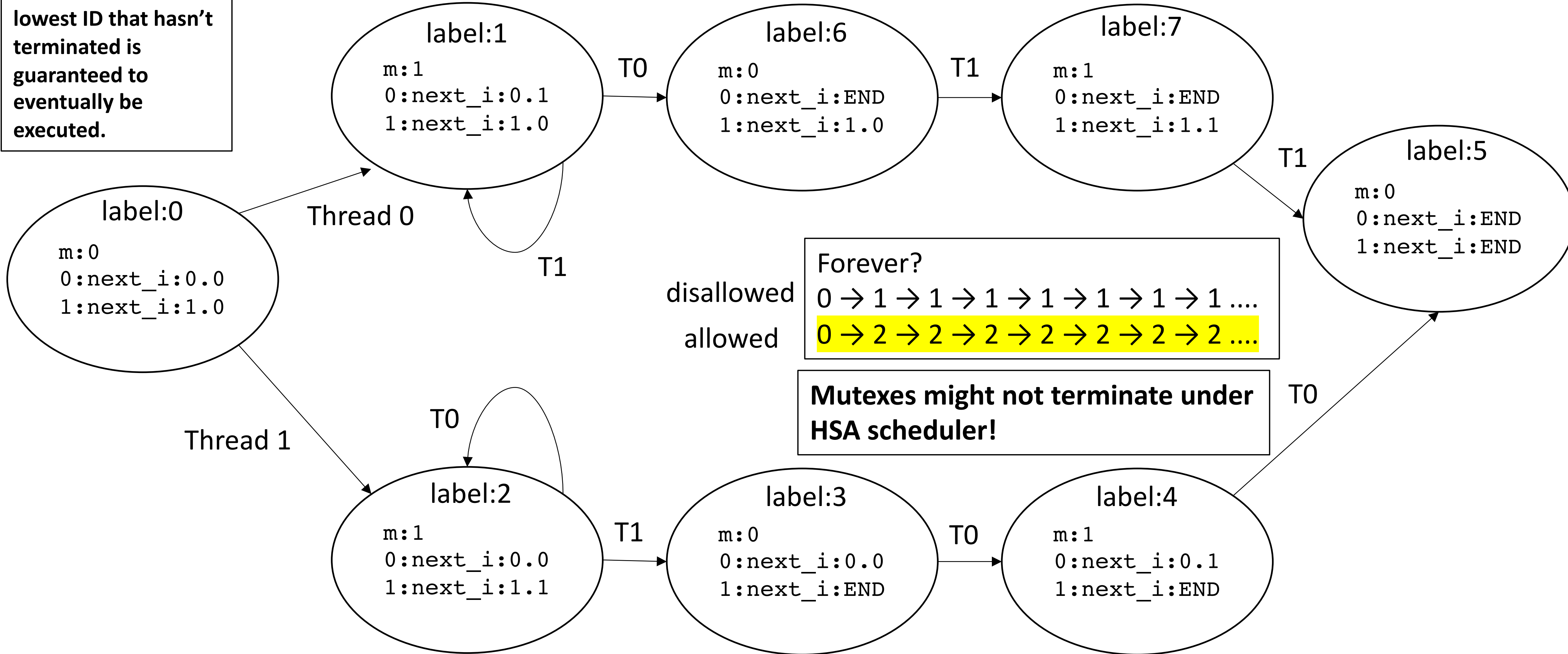


Thread 0:

```
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
```



Thread 0:

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

Thread 1:

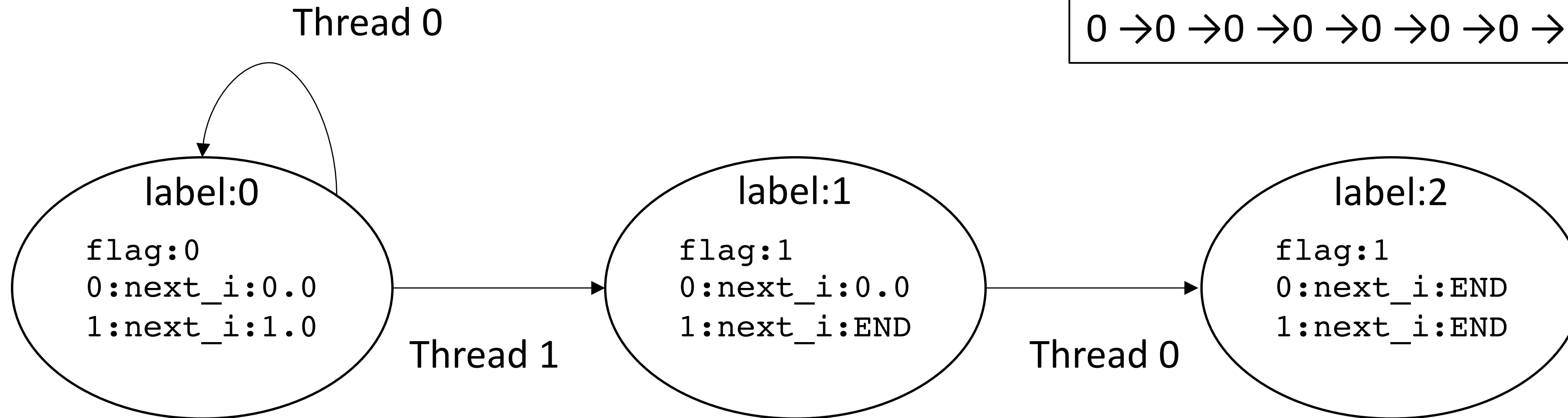
```
1.0: flag.store(1);
```

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:

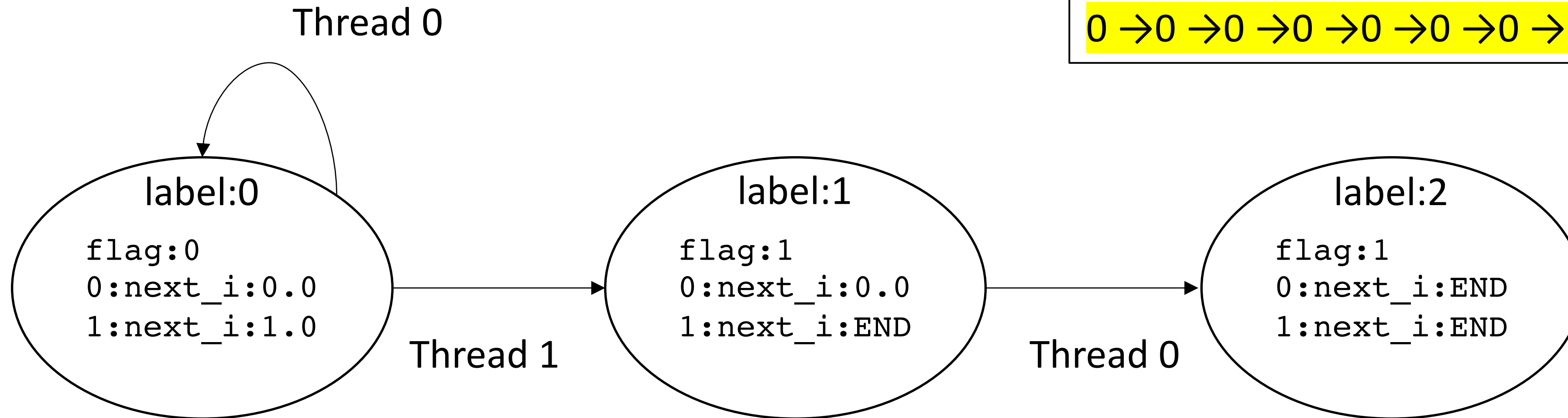
```
0.0: while(flag.load() == 0);
```

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);
```

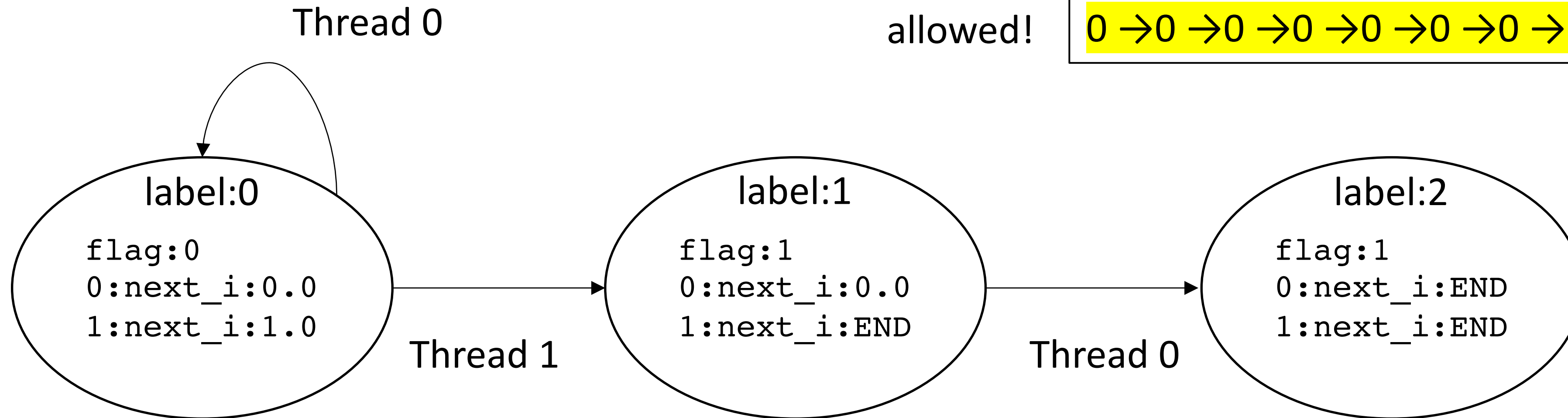
Thread 1:
```
1.0: flag.store(1);
```

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);
```

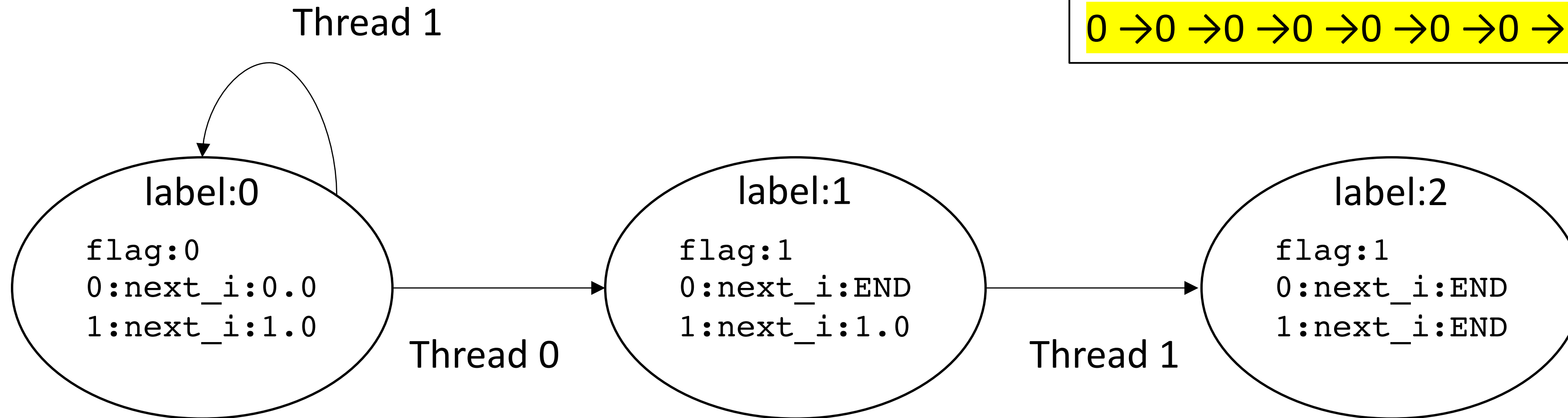
Thread 1:

```
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:

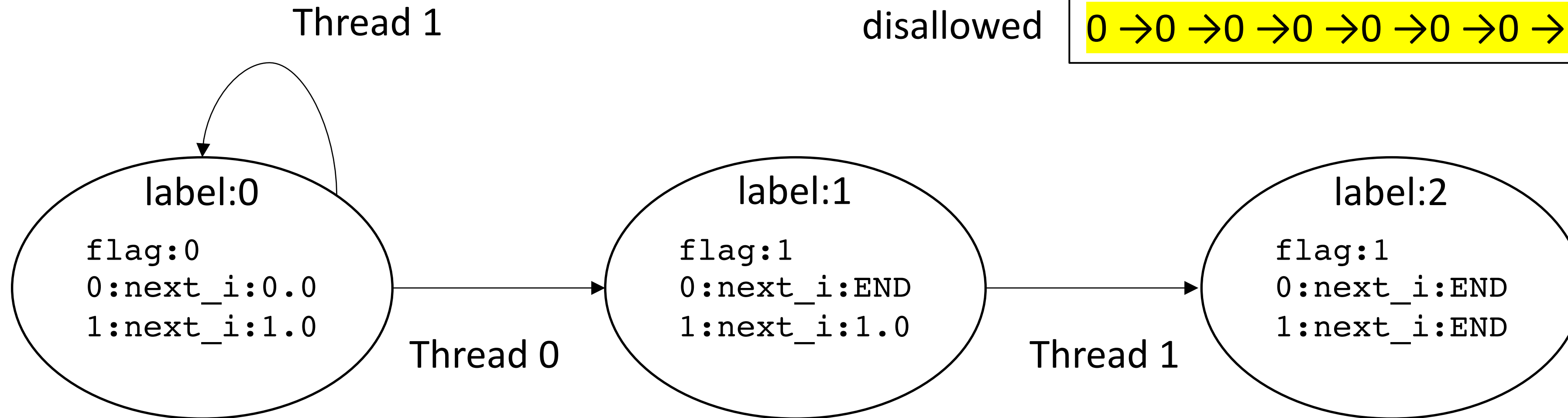
```
0.0: flag.store(1);
```

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.



Forever?
0 →0 →0 →0 →0 →0 →0 →0....

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

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

Demo

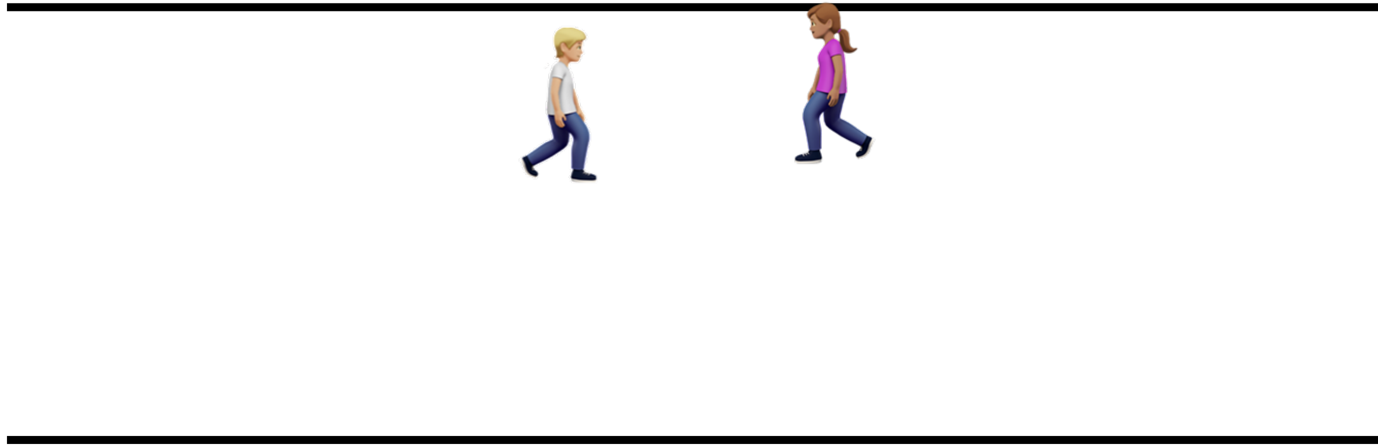
- Demo about how things can go wrong on Ipad.

A different type of non-termination

Hallway problem

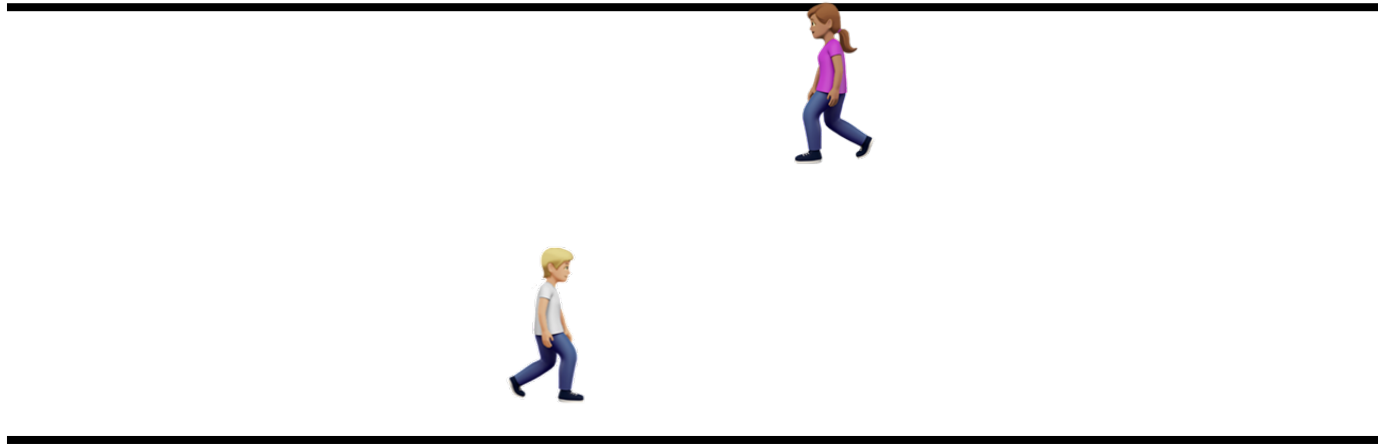
A different type of non-termination

Hallway problem



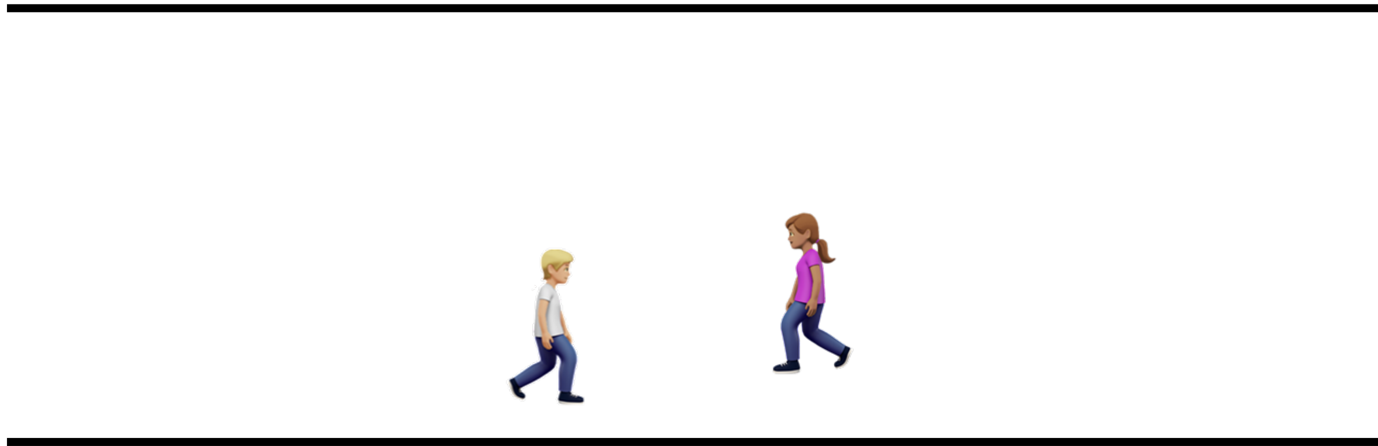
A different type of non-termination

Hallway problem



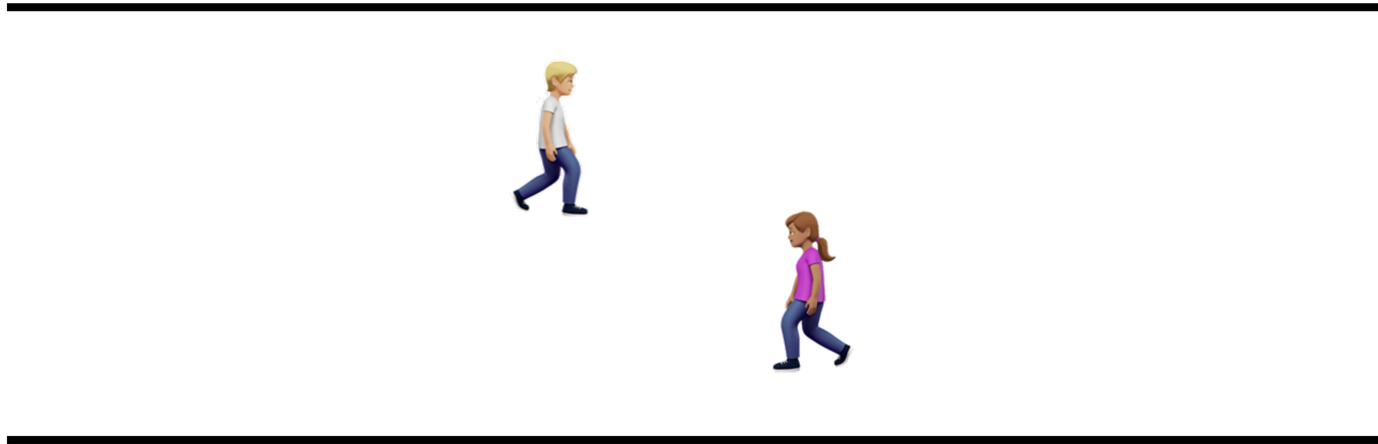
A different type of non-termination

Hallway problem



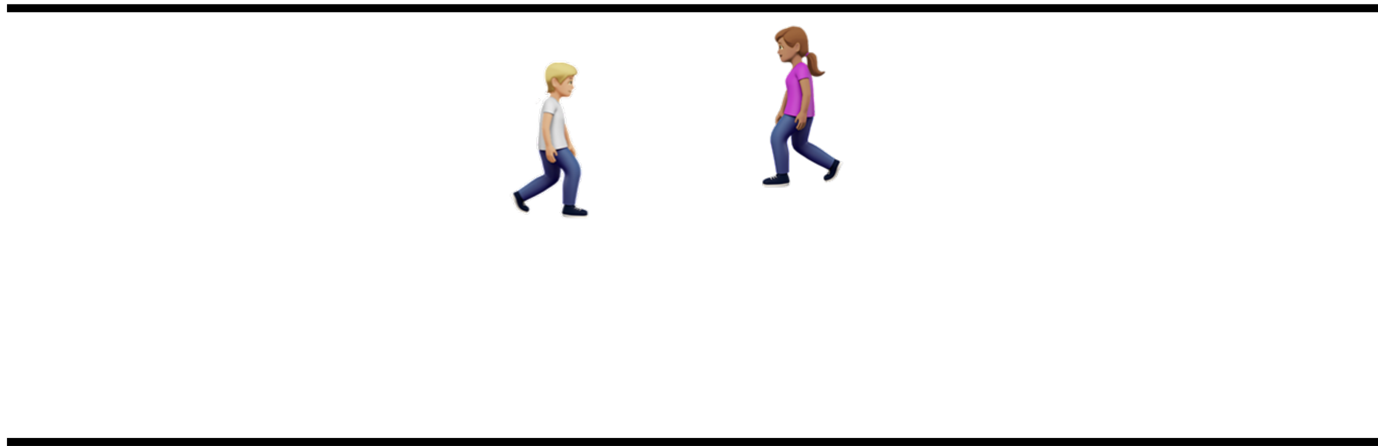
A different type of non-termination

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.

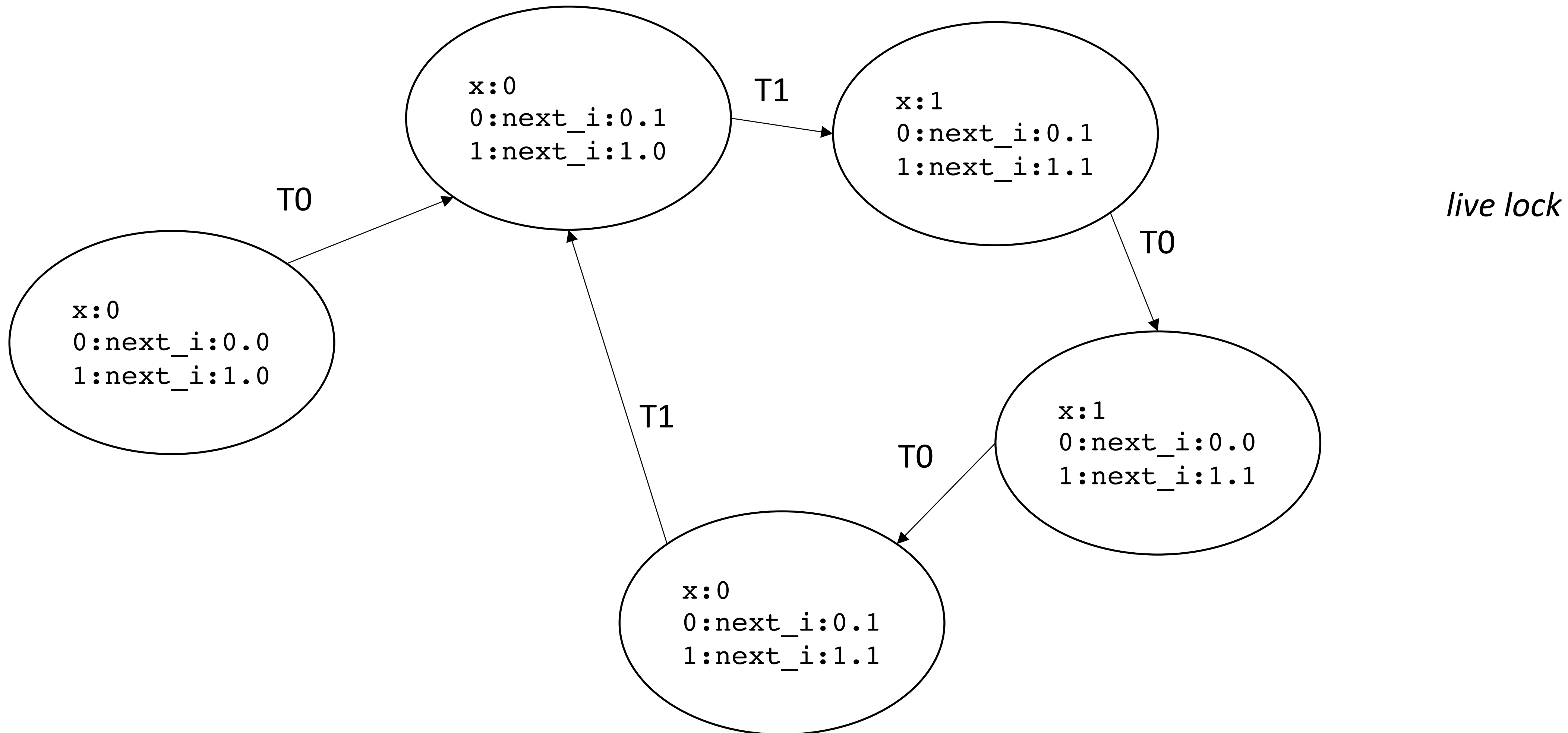
This is called a livelock

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)
```



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!

See you on Wednesday!

- Homework 4 due on Friday
- Expect midterm grades by end of day.
- Last day of module 4, moving on to last module next week!

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)
```