

COL380

Introduction to
Parallel & Distributed Programming

Agenda

- Synchronizing for Sorted List
- Parallel and Distributed Mutual exclusion
- Programming Models

Mutex with Registers

- Atomic reads and writes can be implemented from nonatomic reads and writes without some pre-built facility for “mutual exclusion”
 - ➔ Eliminates circular argument

Linearizable Registers: Read ‘most recent’ write
Determined by linearization point

Can be built from:

Single Reader, Single Writer Safe Bit —

Overlapping reader sees ‘any’ value

Non-overlapping reader sees most recent write

Peterson's Mutex Algorithm

Initially: want = {false, false}

Thread 0

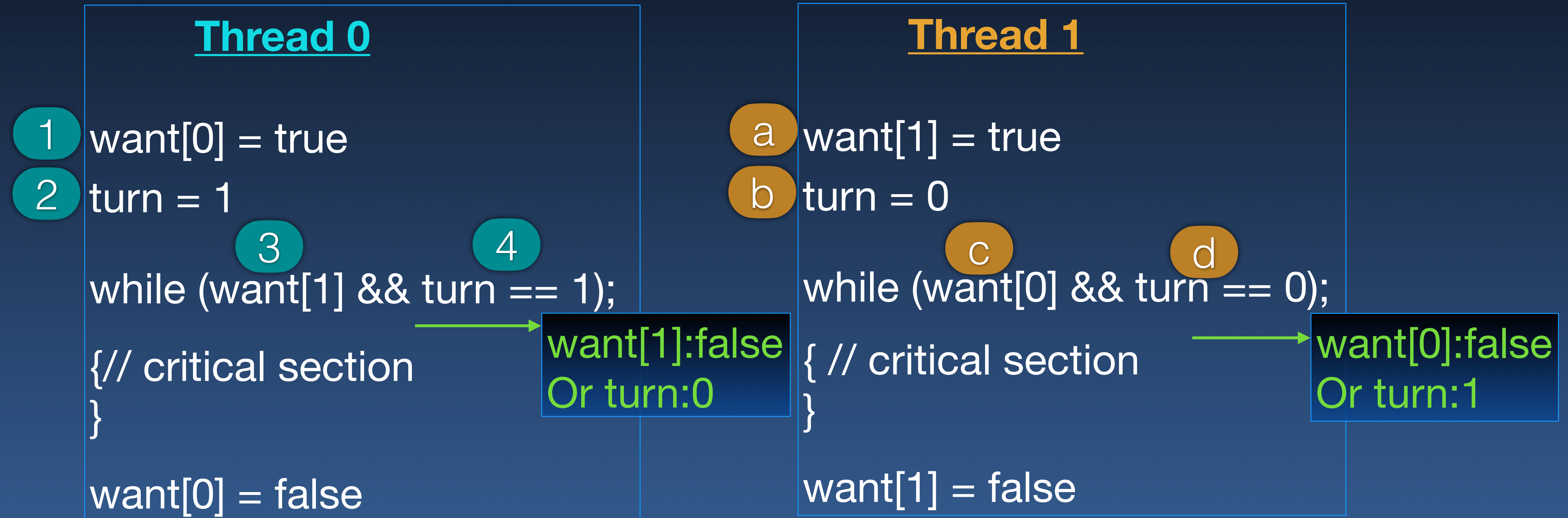
```
1 want[0] = true
2 turn = 1
   3           4
while (want[1] && turn == 1);
{ // critical section
}
want[0] = false
```

Thread 1

```
a want[1] = true
b turn = 0
   c           d
while (want[0] && turn == 0);
{ // critical section
}
want[1] = false
```

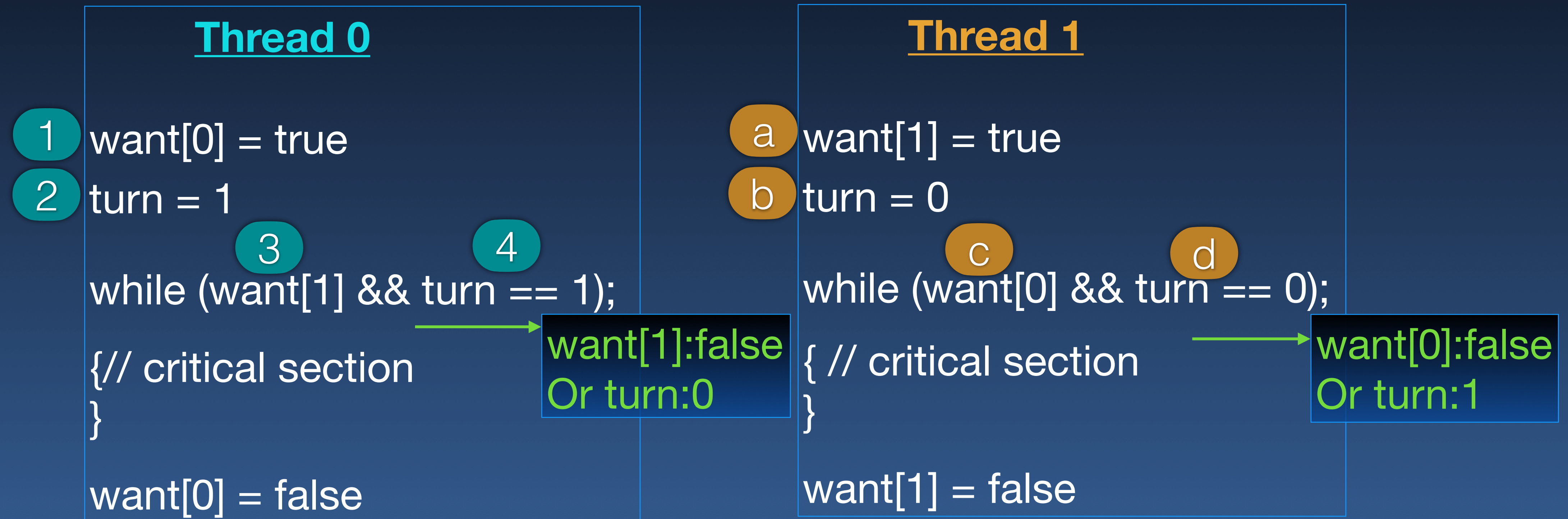

Peterson's Mutex Algorithm

Initially: want = {false, false}



Peterson's Mutex Algorithm

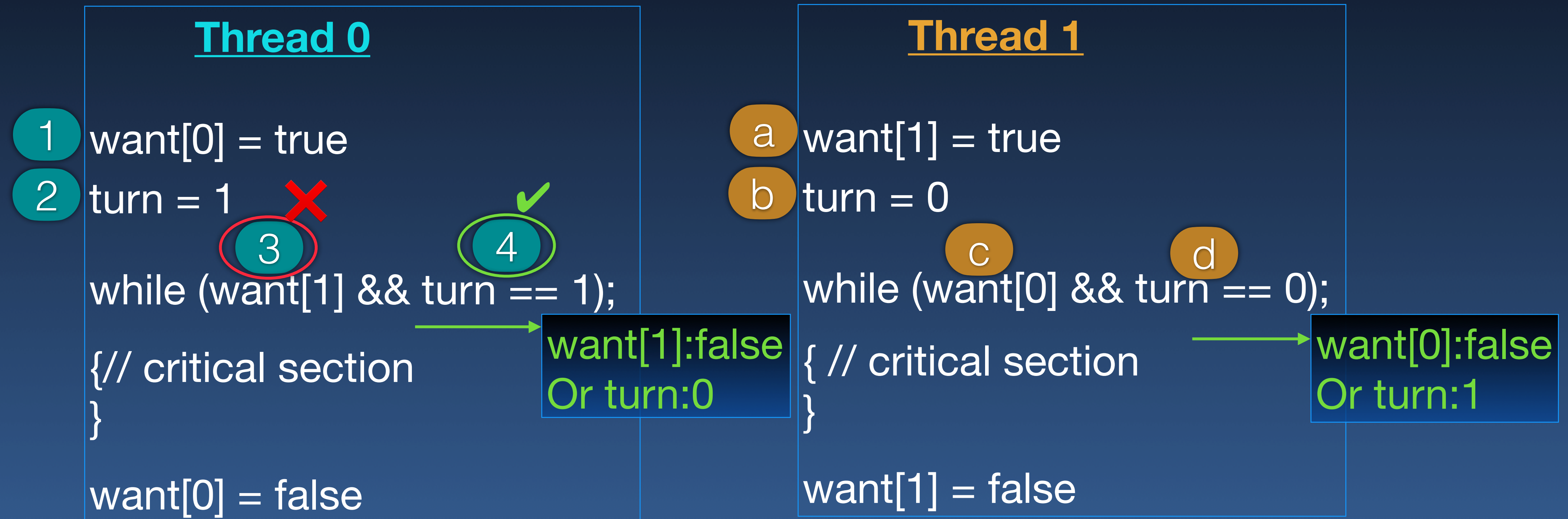
Initially: want = {false, false}



Suppose: b → 2

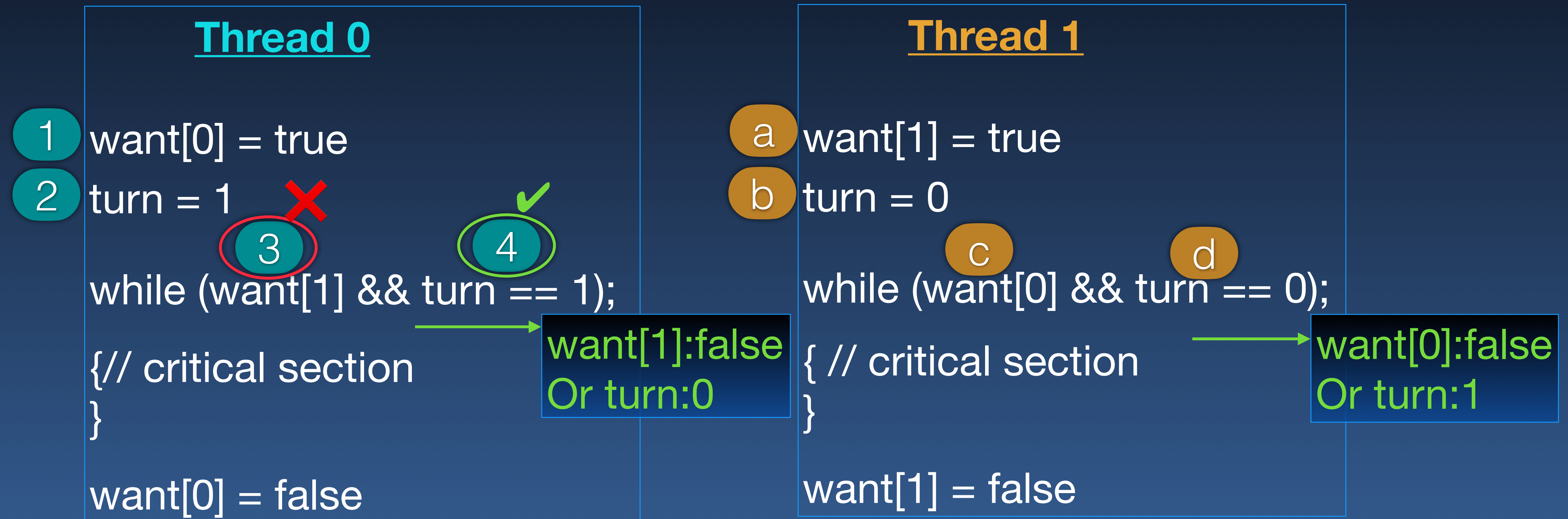
Peterson's Mutex Algorithm

Initially: want = {false, false}



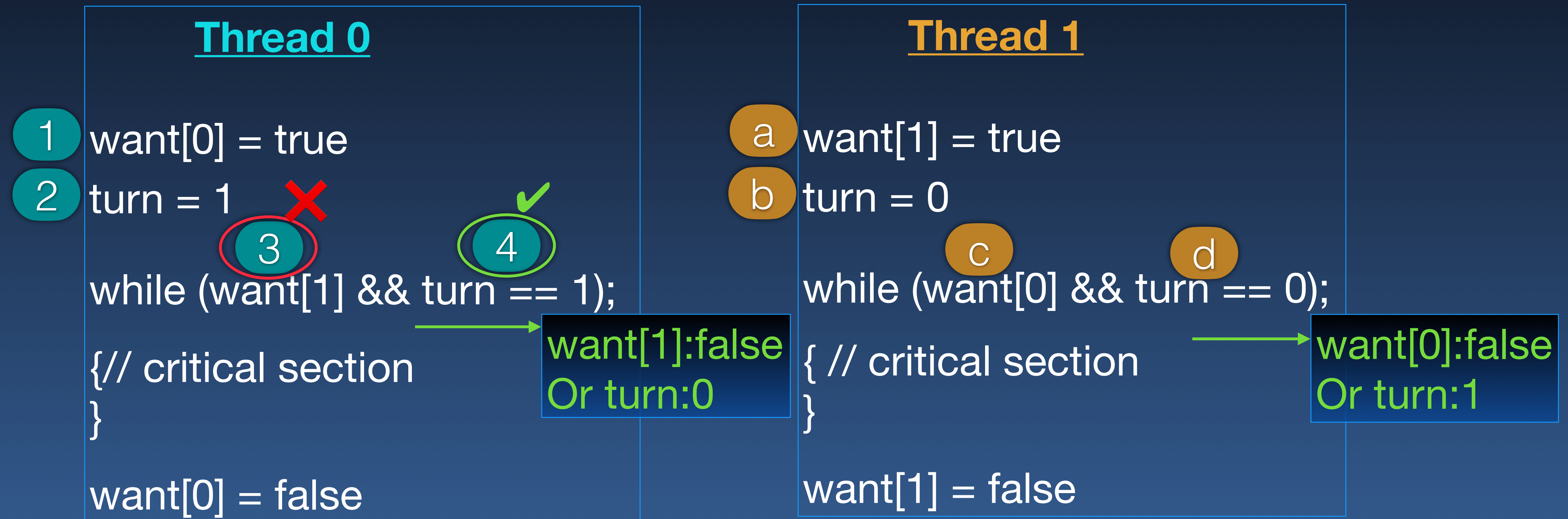
Peterson's Mutex Algorithm

Initially: want = {false, false}



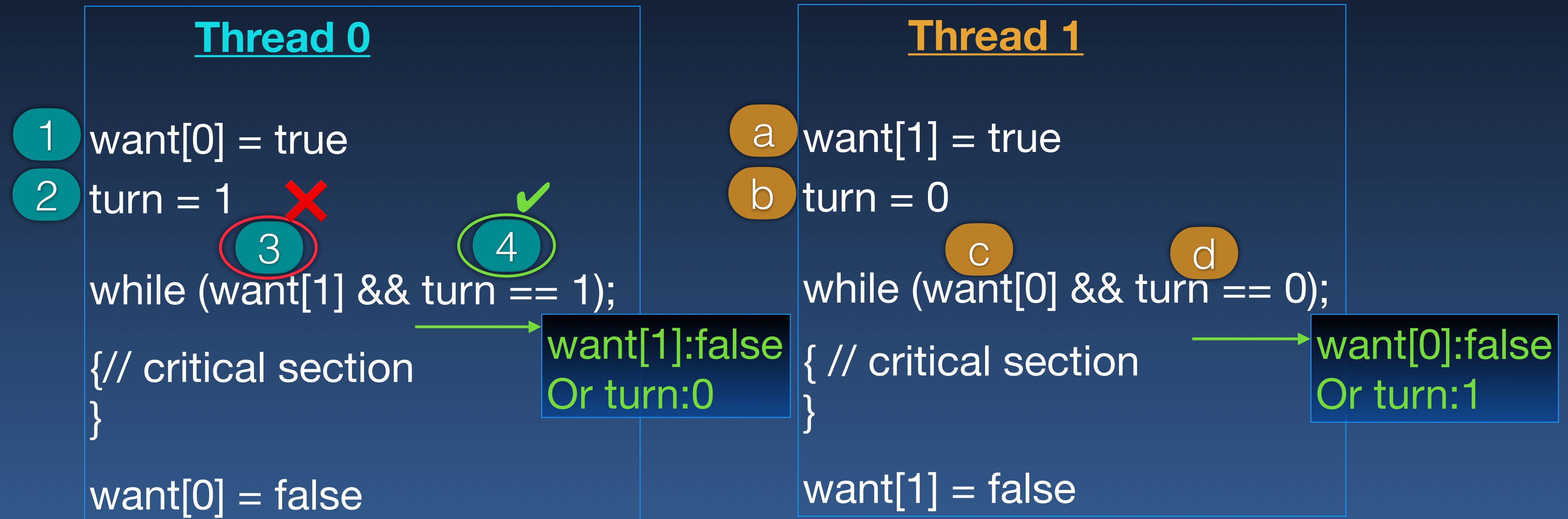
Peterson's Mutex Algorithm

Initially: want = {false, false}



Peterson's Mutex Algorithm

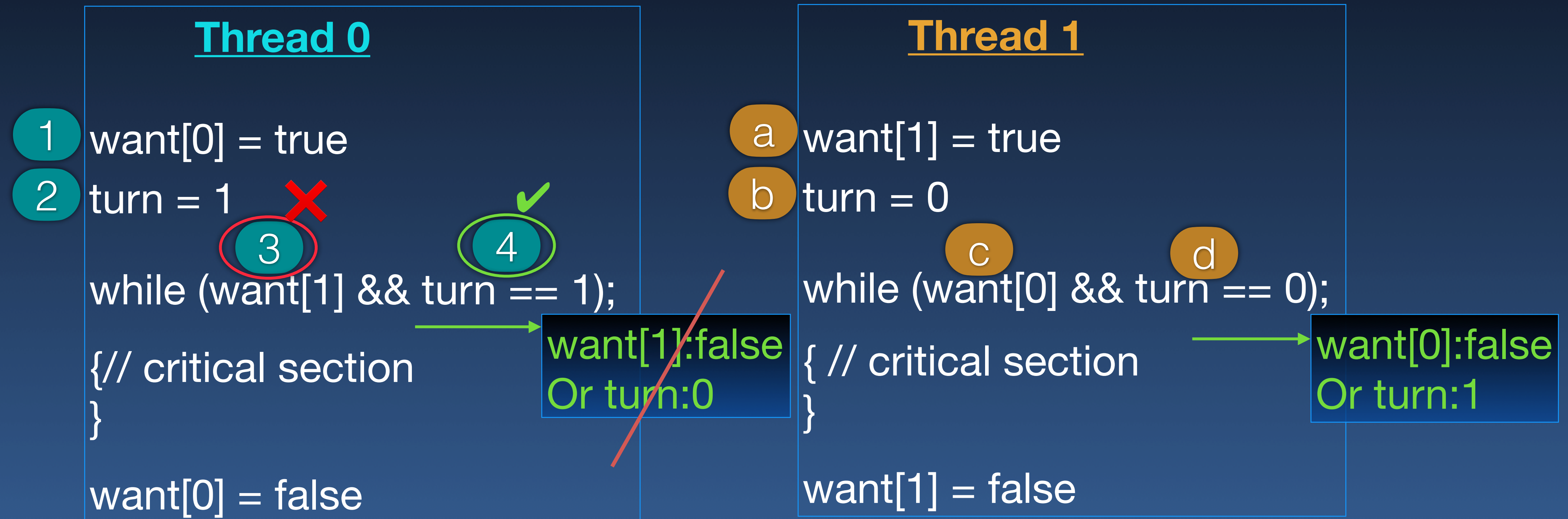
Initially: want = {false, false}



Suppose: b → 2 → 3 → a

Peterson's Mutex Algorithm

Initially: want = {false, false}



Suppose: b → 2 → 3 → a

Mutex w/Registers

— Not Critical Section —

```
1: want[ID] = 1;  
2: token[ID] = 1 + max(token)  
3: want[ID] = 0;  
4: for other != ID {  
5:     while(want[other] == 1);  
6:     while(token[other] > 0 && (token[other]#other) < (token[ID]#ID));  
7: }
```

— Critical Section —

```
8: token[ID] = 0
```

Bakery

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers

— Not Critical Section —

```
1: want[ID] = 1;  
2: token[ID] = 1 + max(token)  
3: want[ID] = 0;  
4: for other != ID {  
5:     while(want[other] == 1);  
6:     while(token[other] > 0 && (token[other]#other) < (token[ID]#ID));  
7: }
```

— Critical Section —

```
8: token[ID] = 0
```

Bakery

- Mutual exclusion does not require hardware synchronization
- Peterson and Bakery use minimal number of registers
 - Still too many? (and ever increasing counter values)

— Not Critical Section —

1: want[ID] = 1;

2: token[ID] = 1 + max(token)

3: want[ID] = 0;

4: for other != ID {

5: while(want[other] == 1);

6: while(token[other] > 0 && (token[other]#other) < (token[ID]#ID);

7: }

— Critical Section —

8: token[ID] = 0

Bakery

Non-shared Logical Clock

- Each entity maintains a counter
 - ➔ increments every *step*, at its own pace
- Interaction between entities is through messages
 - ➔ Data + counter
- On message receipt:
 - ➔ If recipient counter < received count
 - ▶ Increase local counter to received count
 - ▶ Receive is also a '*step*,' so increment by one

[Lamport's Timestamp algorithm]

Request Critical Section:

Broadcast **R** = <request, time(entity)>

Add **R** to local-queue(entity)

Enter Critical section (**R**)

R has the lowest timestamp in local-queue. AND.

Have received some **m** from every other entity with **m**.Time > **R**.time

Exit Critical section (**R**):

Remove **R** from local-queue

Broadcast <release> message to all

Distributed Mutex

Request Critical Section:

Broadcast **R** = <request, time(entity)>

Add **R** to local-queue(entity)

Enter Critical section (**R**)

R has the lowest timestamp in local-queue. AND.

Have received some **m** from every other entity with **m**.Time > **R**.time

Exit Critical section (**R**):

Remove **R** from local-queue

Broadcast <release> message to all

Receive **R**

update(time(entity))

if(type == request)

 Add **R** to local-queue

 Reply <ack, time(entity)>

if(type == release)

 Remove **R** from local-queue

- Shared memory distributed synchronization
- Message passing distributed synchronization