Concurrency again

Sistemi di Calcolo 2

Instructor: Riccardo Lazzeretti

Software solutions for syncronization

Dijkstra’s Algorithm

```
/* global storage */
boolean interested[N]     = {false, ..., false}
boolean passed[N]   = {false, ..., false}
int k = <any>                 // k ∈ {0, 1, ..., N-1}

/* local info */
int i = <entity ID> // i ∈ {0, 1, ..., N-1}
1.  interested[i] = true
2.  while (k != i) {
3.    passed[i] = false
4.    if (!interested[k]) then k = i
    }
5.  passed[i] = true
6.  for j in 1 ... N except i do
7.    if (passed[j]) then goto 2
8.  <critical section>
9.  passed[i] = false; interested[i] = false
```

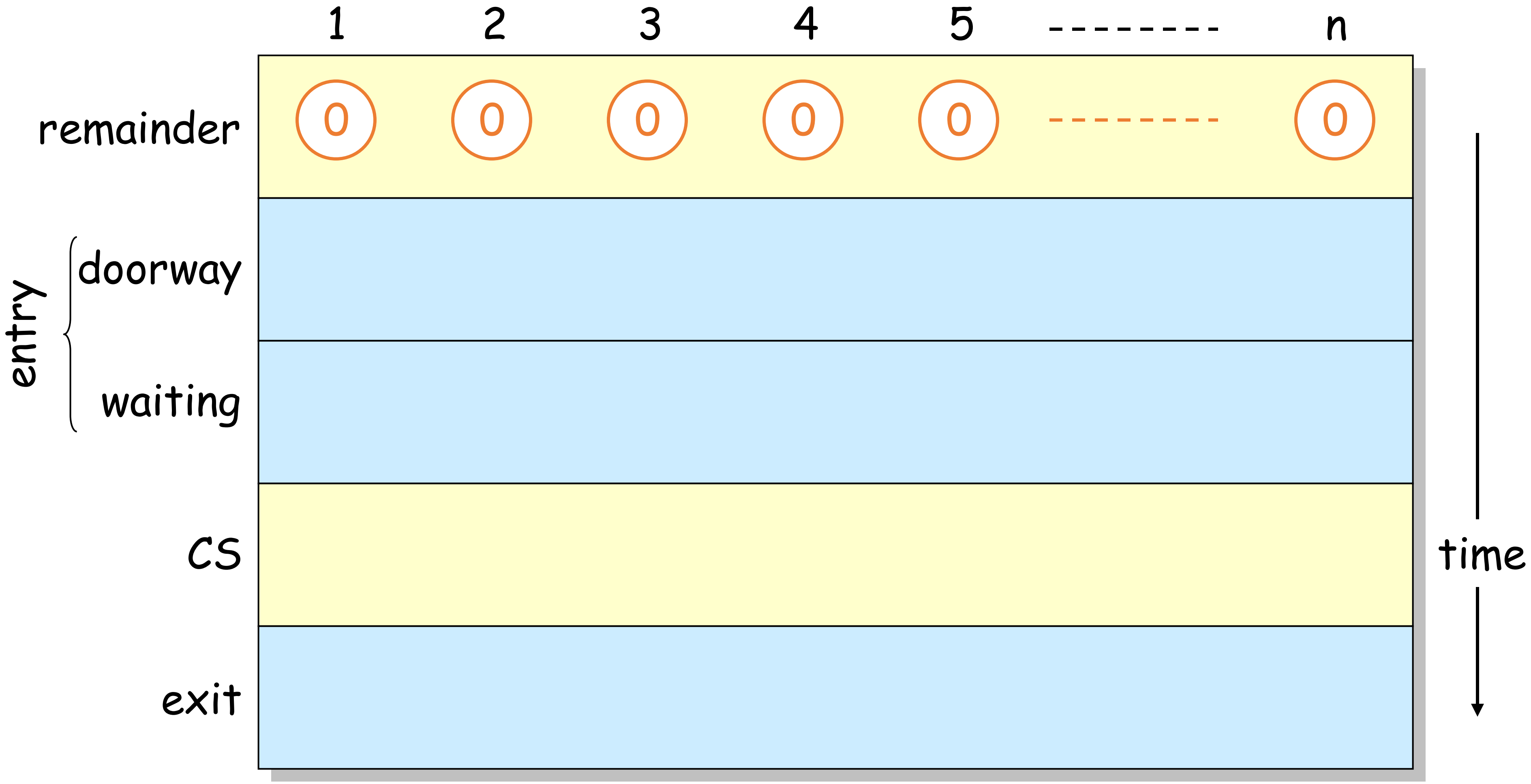
Dijkstra characteristics

- Mutual Exclusion
- No deadlock
- No starvation?
  - Not guaranteed
- Other problems:
  - Needs atomic read/write
  - Needs memory sharing for k

Bakery Algorithm
Lamport (1975)

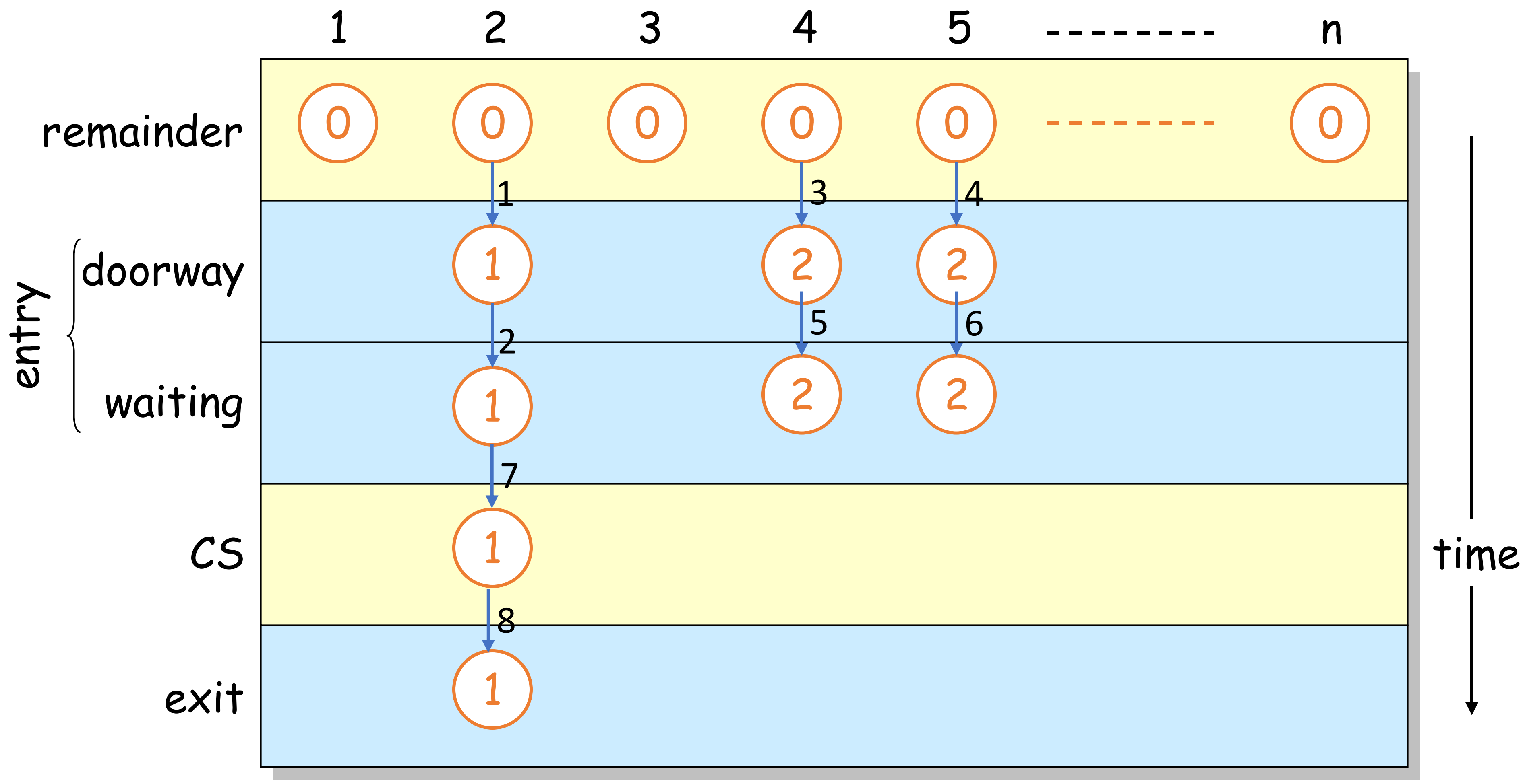
- Concept:
  - Think of a popular store with a crowded counter
    - People take a ticket from a machine
    - If nobody is waiting, tickets don’t matter
    - When several people are waiting, ticket order determines order in which they can make purchases

The Bakery Algorithm



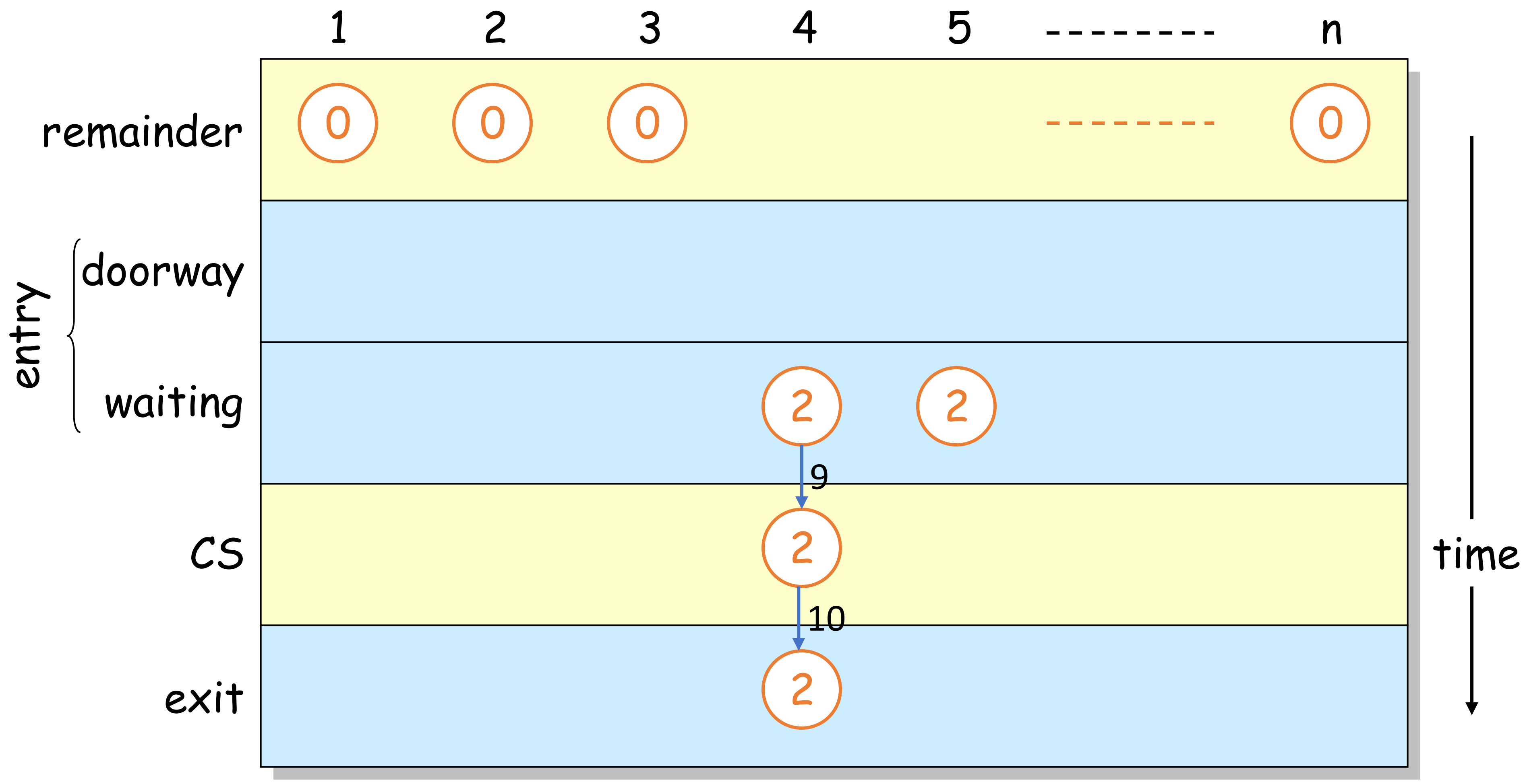
From "Synchronization Algorithms and Concurrent Programming"
Gadi Taubenfeld © 2007

The Bakery Algorithm



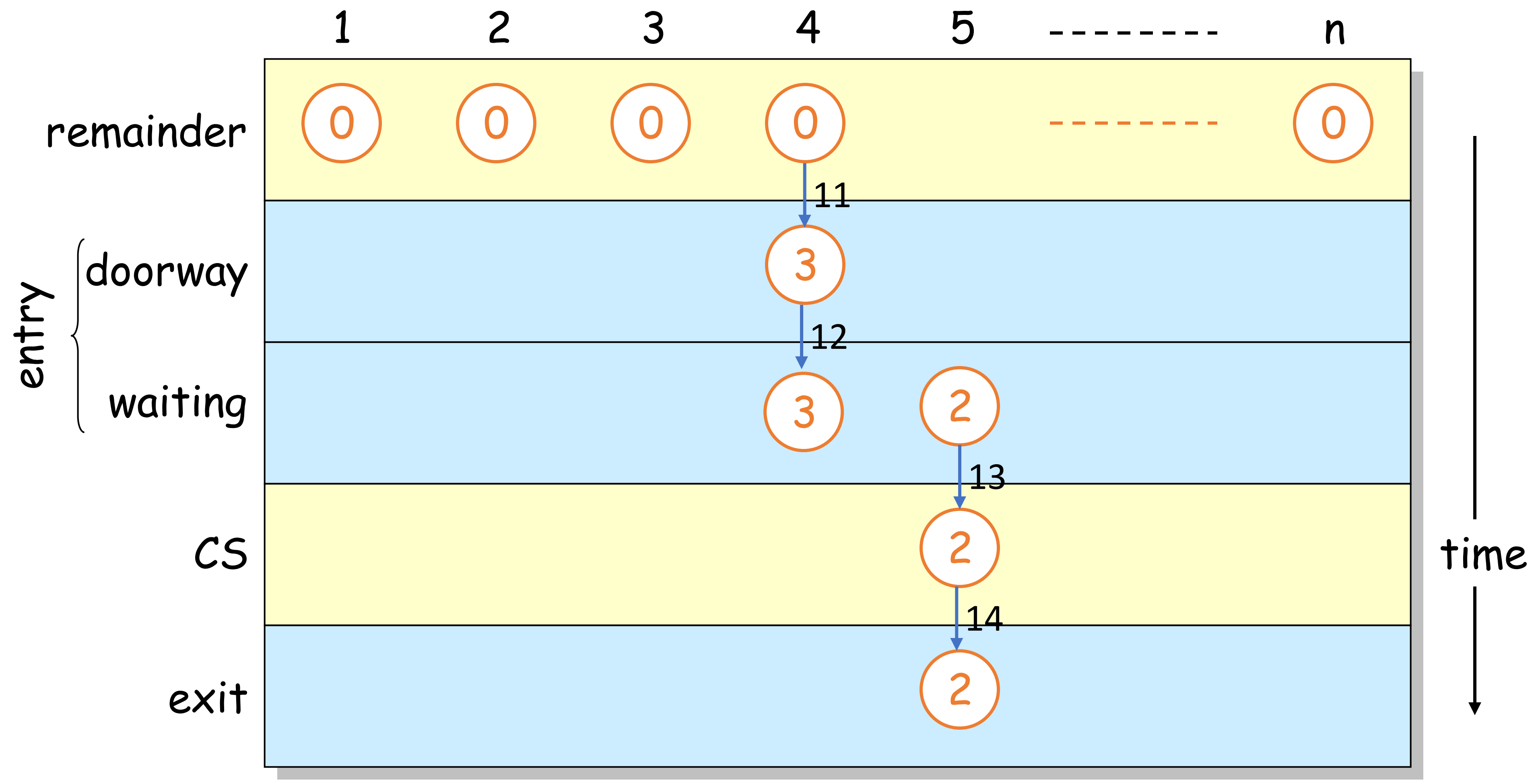
From "Synchronization Algorithms and Concurrent Programming"
Gadi Taubenfeld © 2007

The Bakery Algorithm



From “Synchronization Algorithms and Concurrent Programming”
Gadi Taubenfeld © 2007

The Bakery Algorithm



Implementation 1

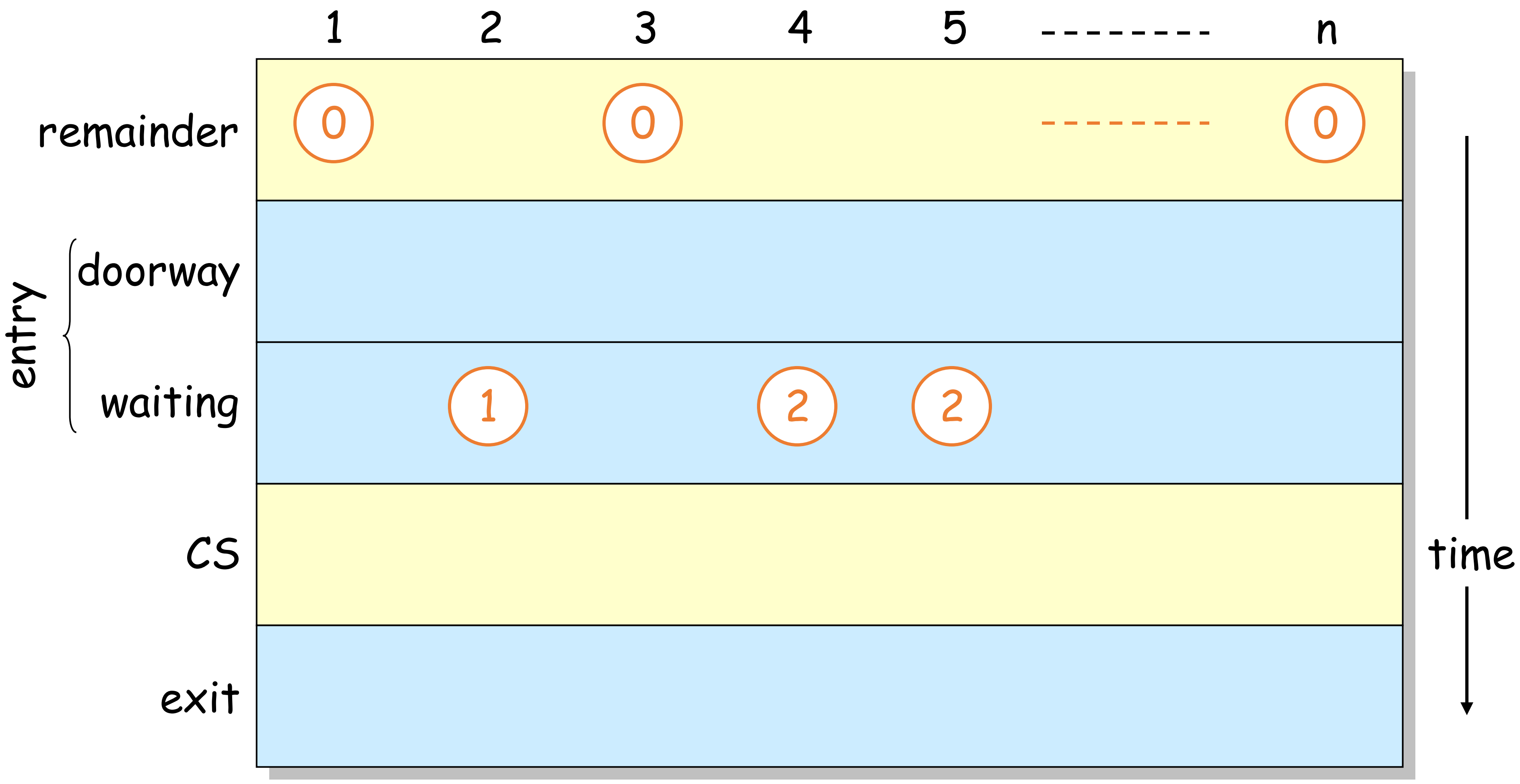
code of process i , i ∈ {1 ,…, n}

```
while (1){
  /*NCS*/                                                    //Doorway
  number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}
  for j in 1 .. N except i {
    while (number[j] != 0 && number[j] < number[i]);
  }
  /*CS*/                                                     //Bakery
  number[i] = 0;
}
```

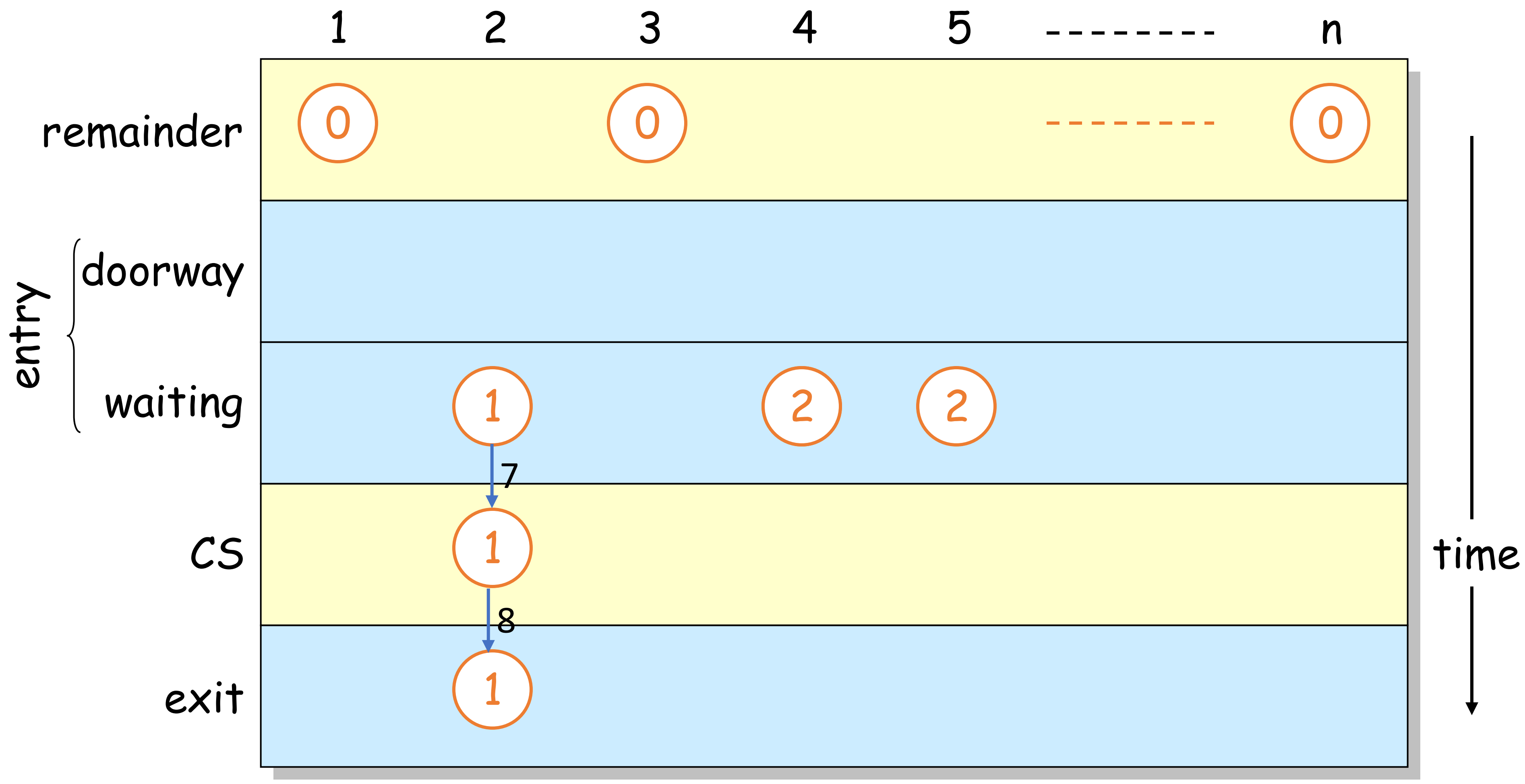
	1	2	3	4	--------	n	
number	0	0	0	0	0	0	integer

Answer: does not satisfy mutual exclusion

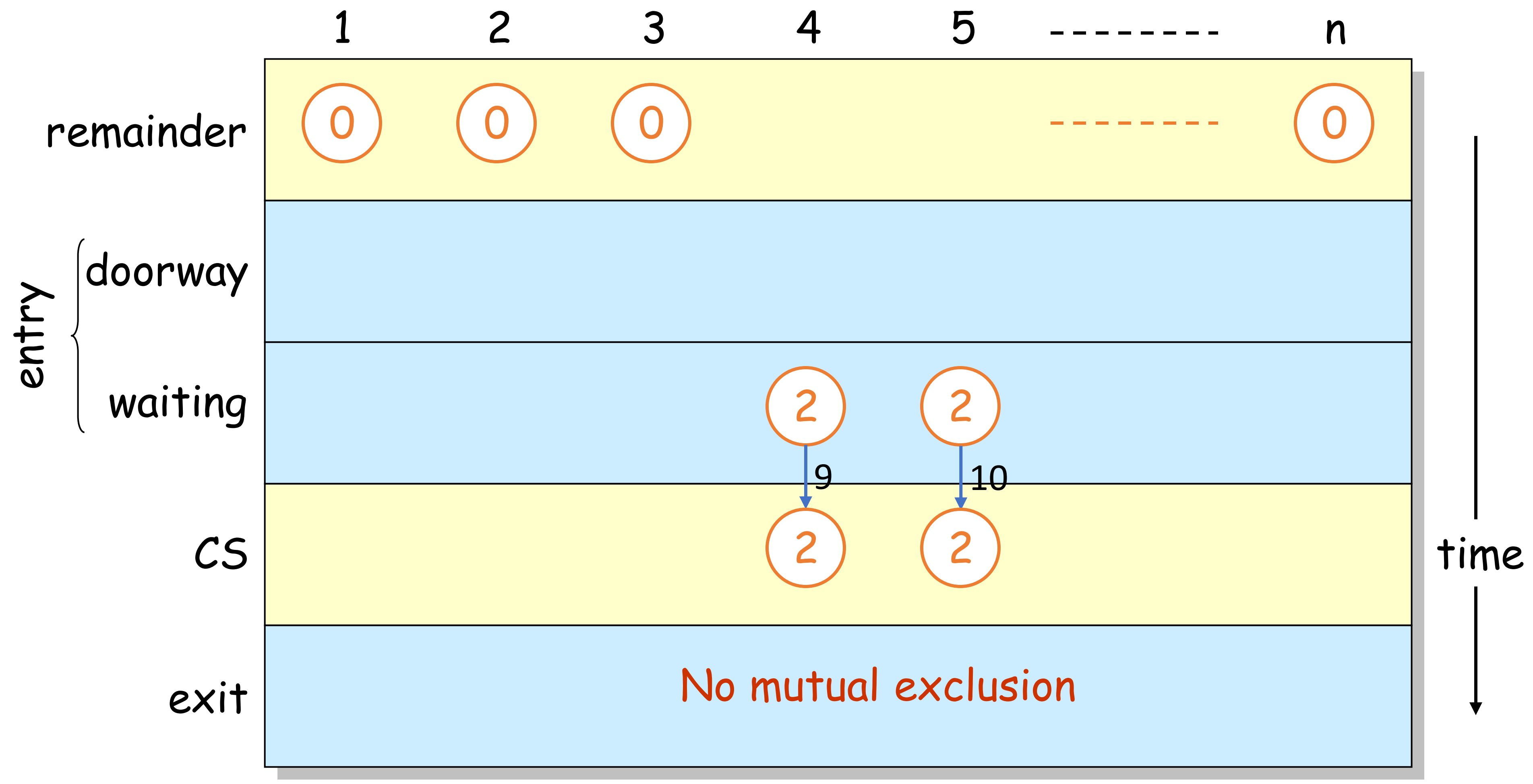
Implementation 1: deadlock



Implementation 1: deadlock



Implementation 1: deadlock



Implementation 1

code of process i , i ∈ {1 ,…, n}

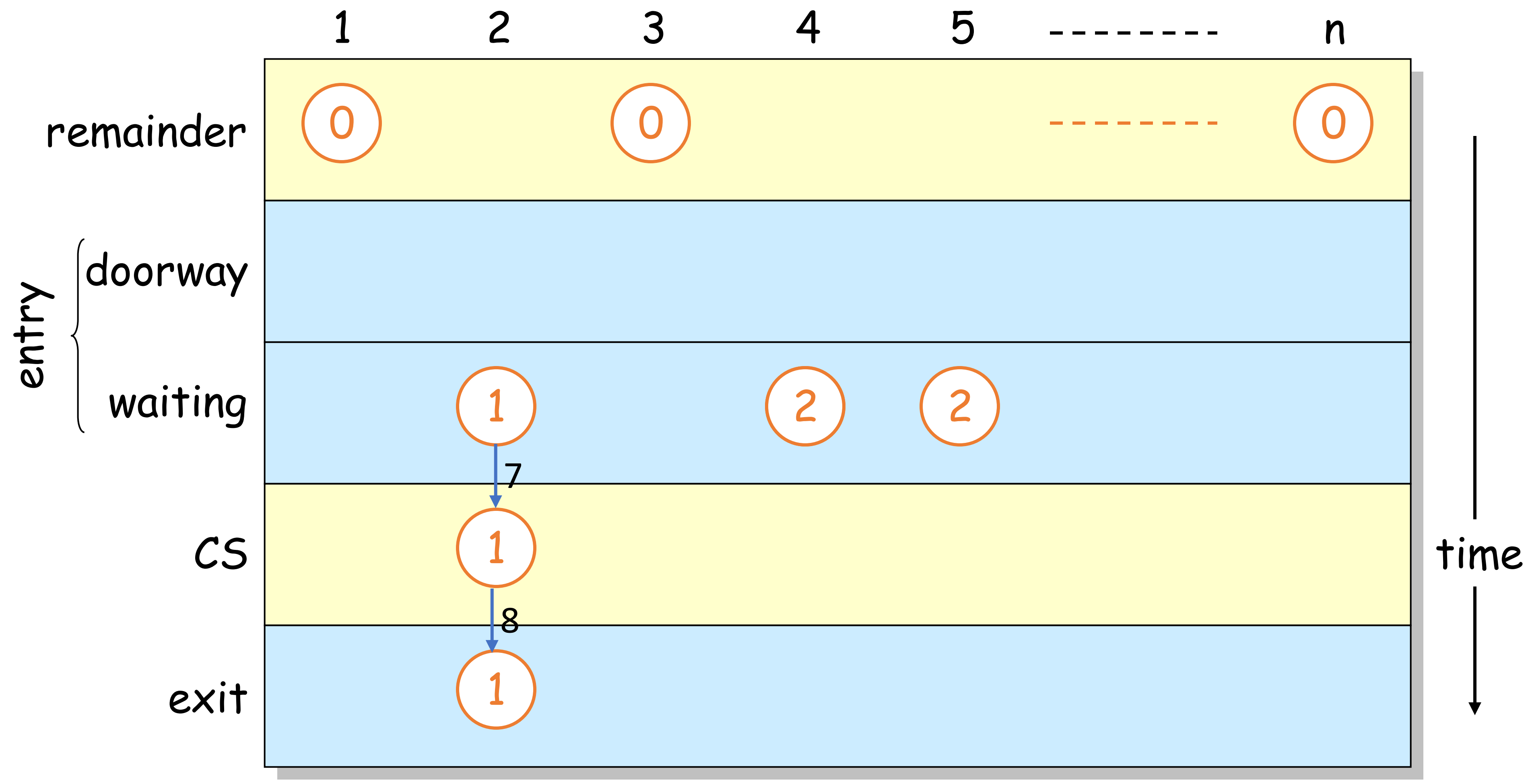
```
while (1){
  /*NCS*/
  number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}
  for j in 1 .. N except i {
    while (number[j] != 0 && number[j] < number[i]);
  }
  /*CS*/
  number[i] = 0;
}
```

What if we replace < with ≤ ?

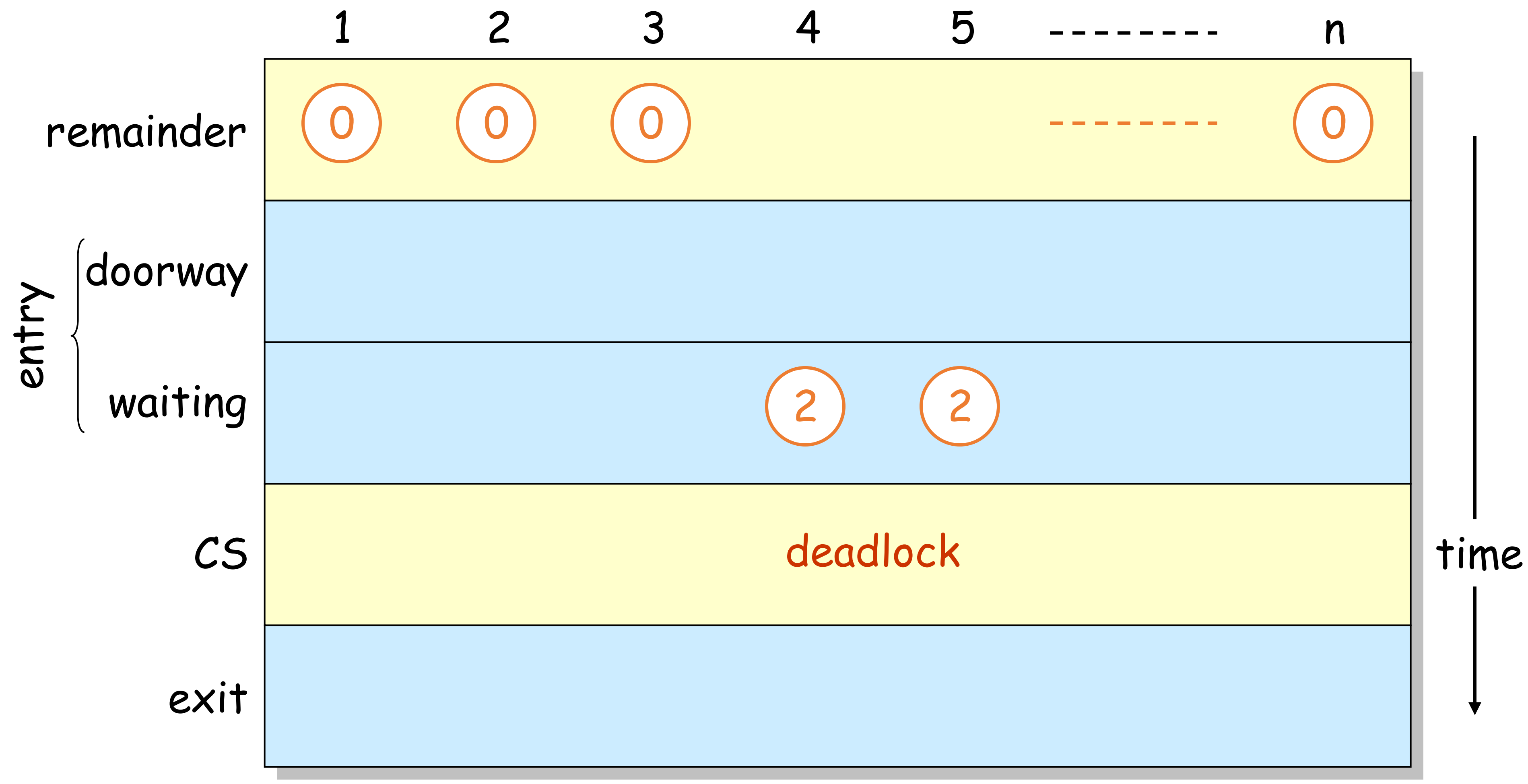
	1	2	3	4	--------	n	
number	0	0	0	0	0	0	integer

Answer: No! can deadlock

Implementation 1: deadlock



Implementation 1: deadlock



Implementation 2

code of process i , i ∈ {1 ,…, n}

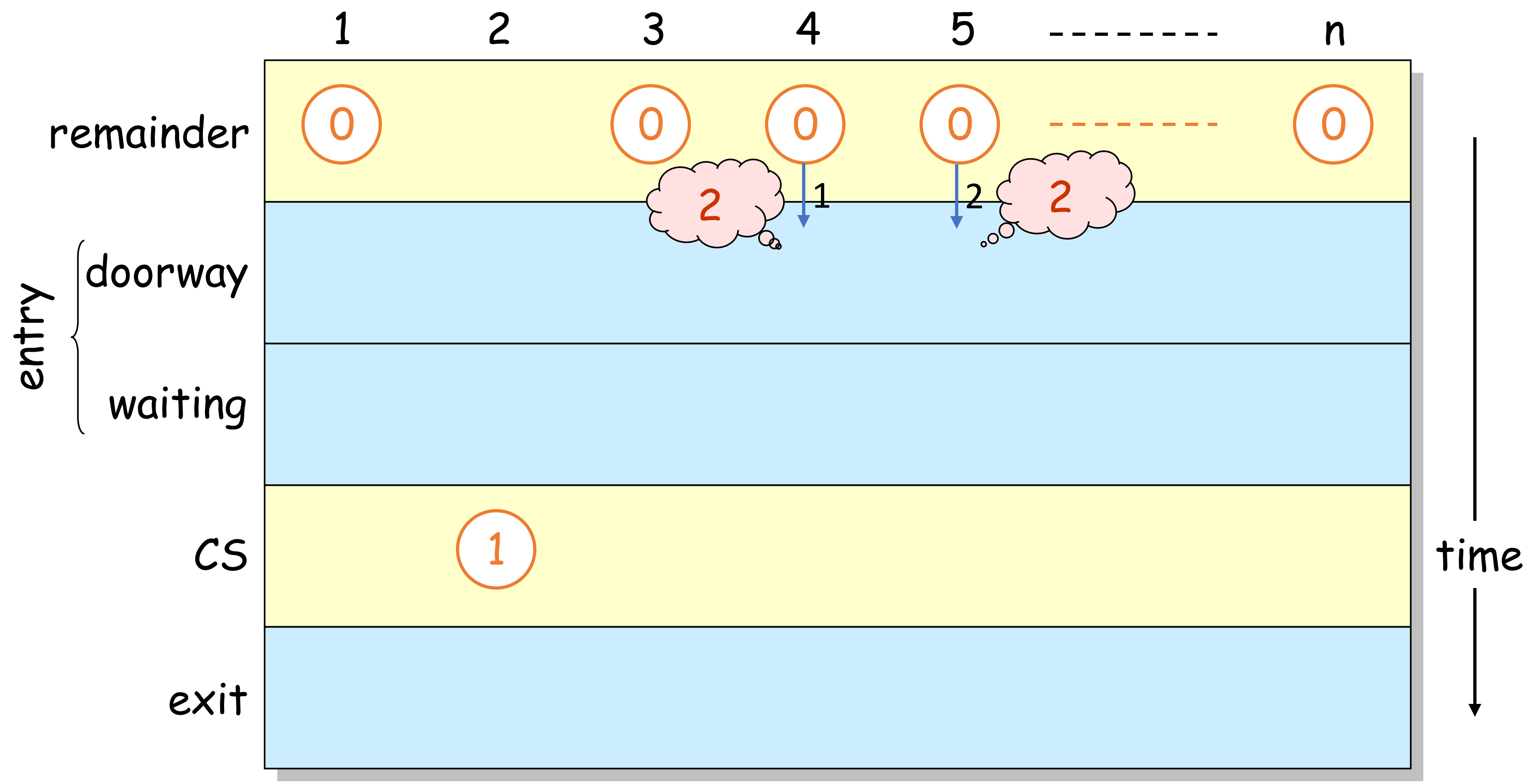
```
while (1){
  /*NCS*/
  number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}
  for j in 1 .. N except i {
    while (number[j] != 0 && (number[j],j) < (number[i],i));
  }
  /*CS*/
  number[i] = 0;
}
```

```
// lexicographical order: (B,j) < (A,i) means (B<A || (B==A && j<i))
```

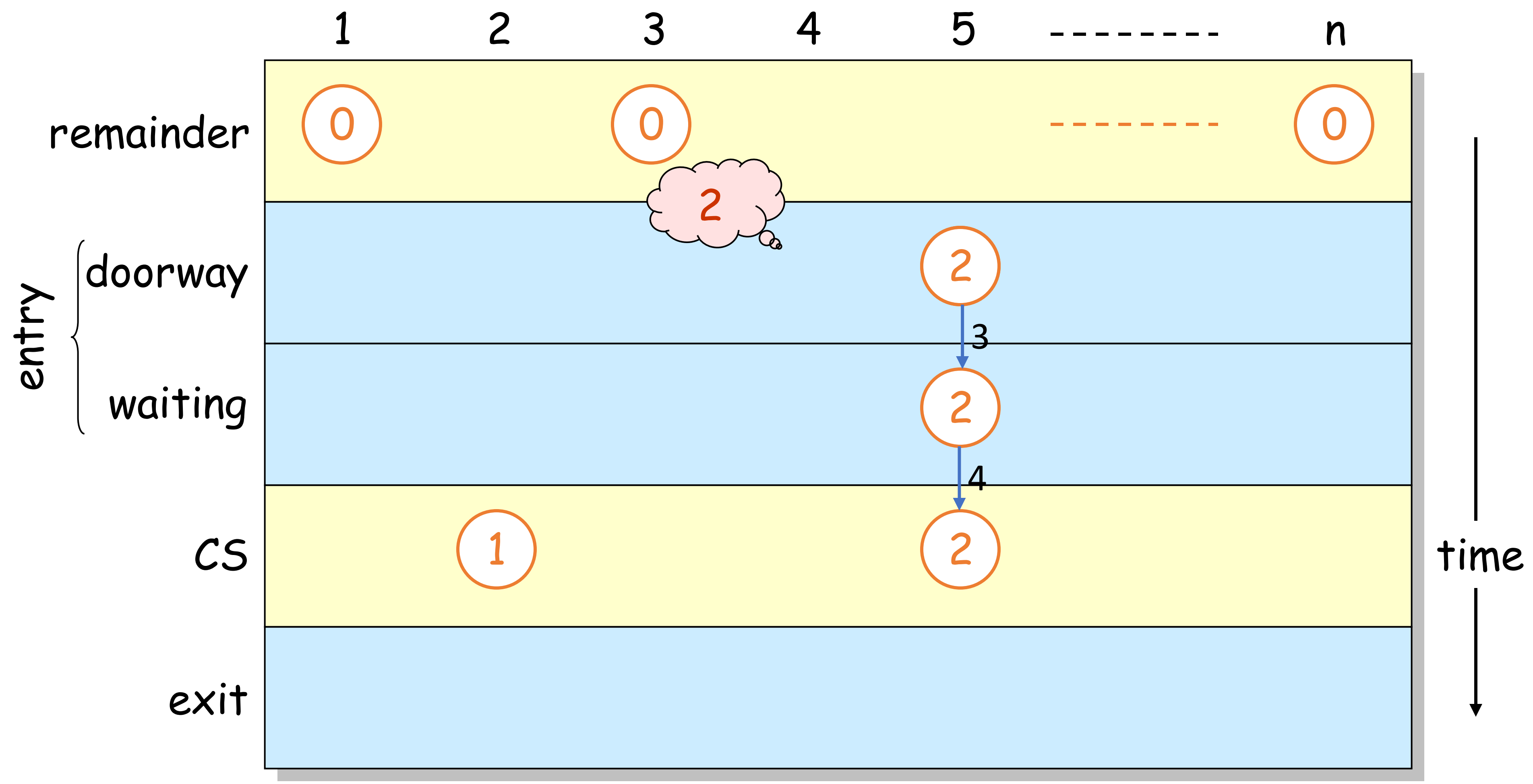
	1	2	3	4	--------	n	
number	0	0	0	0	0	0	integer

Answer: does not satisfy mutual exclusion

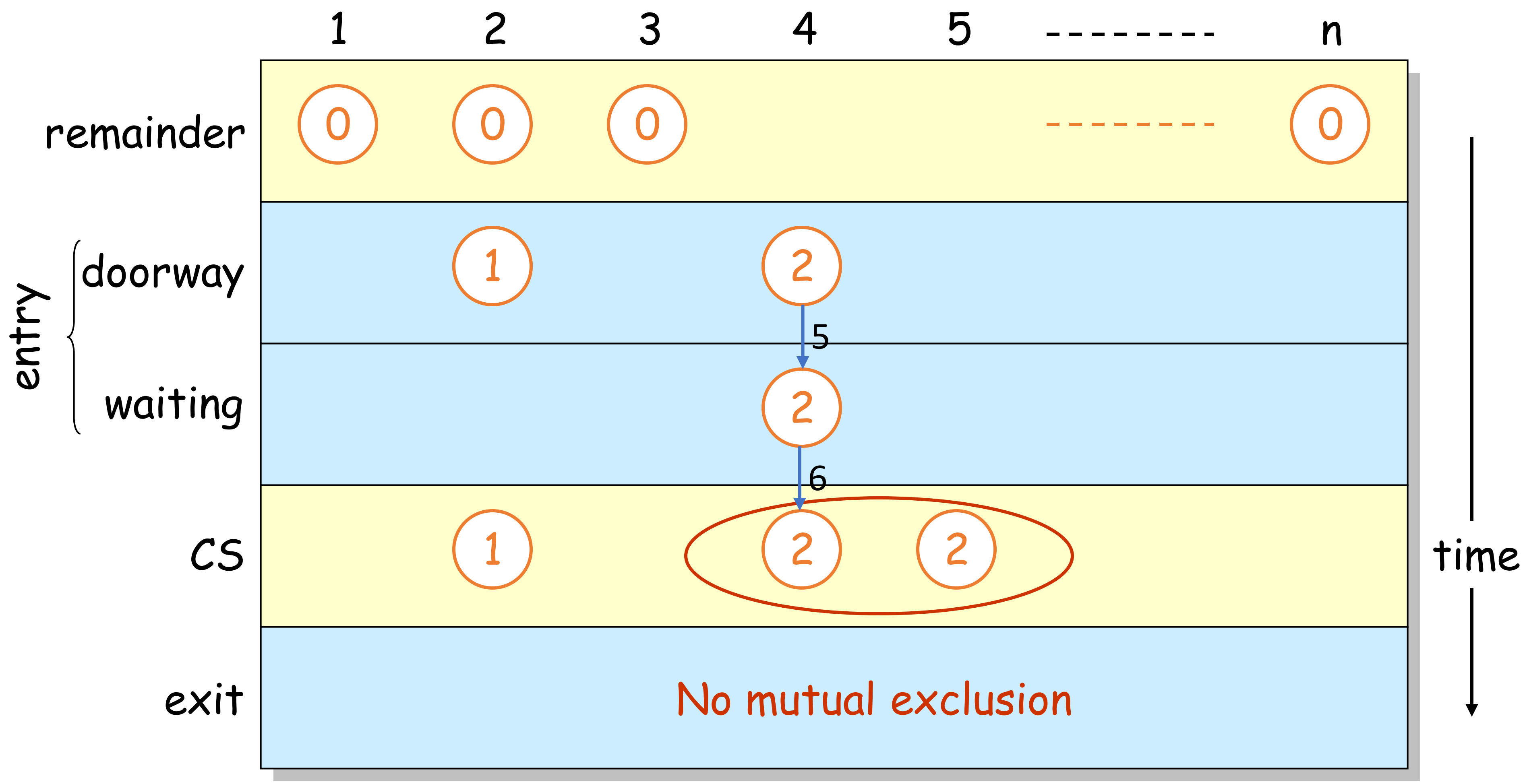
Implementation 2: no mutual exclusion



Implementation 2: no mutual exclusion



Implementation 2: no mutual exclusion



The Bakery Algorithm

code of process i , i ∈ {1 ,…, n}

```
while (1){
  /*NCS*/
  choosing[i] = true;
  number[i] = 1 + max {number[j] | (1 ≤ j ≤ N) except i}
  choosing[i] = false;
  for j in 1 .. N except i {
    while (choosing[j] == true);
    while (number[j] != 0 && (number[j],j) < (number[i],i));
  }
  /*CS*/
  number[i] = 0;
}
```

	1	2	3	4	--------	n	
choosing	false	false	false	false	false	false	bits
number	0	0	0	0	0	0	integer

Computing the Maximum

code of process i , i ∈ {1 ,…, n}

Correct implementation

```
number[i] = 1 + max {number[j] | (1 ≤ j ≤ N)}
```

```
local1 = 0;
for local2 in 1 .. N {
       local3 = number[local2];
       if (local1 < local3)
              local1 = local3;
}
number[i] = 1 + local1
```

	1	2	3	4	--------	n	
number	0	0	0	0	0	0	integer

The Bakery Algorithm with bounded numbers

code of process i , i ∈ {1 ,…, n}

```
while (1){
  /*NCS*/
 while(number[i] == 0){
    choosing[i] = true;
    number[i] = (1 + max {number[j] | (1 ≤ j ≤ N) except i}) % MAXIMUM
    choosing[i] = false;
  }
  for j in 1 .. N except i {
    while (choosing[j] == true);
    while (number[j] != 0 && (number[j],j) < (number[i],i));
  }
  /*CS*/
  number[i] = 0;
}
```

Bakery algorithm characteristics

- Processes communicate by writing/reading shared variables (as Dijkstra)
- Read/write are not atomic operations
  - Reader can read while writer is writing
  - None receives any notification
- Any shared variable is owned by a process that can write it, others can read it
- No process can perform two concurrent writings
- Execution times are not correlated

The Bakery Algorithm in client/server app.

code of process i , i ∈ {1 ,…, n}

```
while (1){ //client thread
  /*NCS*/
  choosing = true; //doorway
  for j in 1 .. N except i {
    send(Pj,num);
    receive(Pj,v);
    num = max(num,v);
  }
  num = num+1;
  choosing = false;
  for j in 1 .. N except i {  //backery
    do{
      send(Pj,choosing);
      receive(Pj,v);
    }while (v == true);
    do{
      send(Pj,v);
      receive(Pj,v);
    }while (v != 0 && (v,j) < (num,i));
  }
  /*CS*/
  num = 0;
}
```

```
//global variable
//inizialization:
int num = 0;
boolean choosing = false;
// and process ip/ports
```

```
while (1){ //server thread
  receive(Pj,message);
  if (message is a number)
    send(Pj,num);
  else
    send(Pj,choosing);
}
```

Assumptions:

- Finite response time
- Reliable communication channels