Operating Systems

David Hay

SYNCHRONIZATION

Motivation: Two processes (or threads) want to print

- Both processors write their job to the *spooler* – the (**shared**) queue from which the printer extracts job to print.
  - NULL: No job to print
  - OUT: Next job to print
  - IN: End of queue, write new job here

- Code for each process (adding jobs):
  Spooler[IN] = job
  IN++

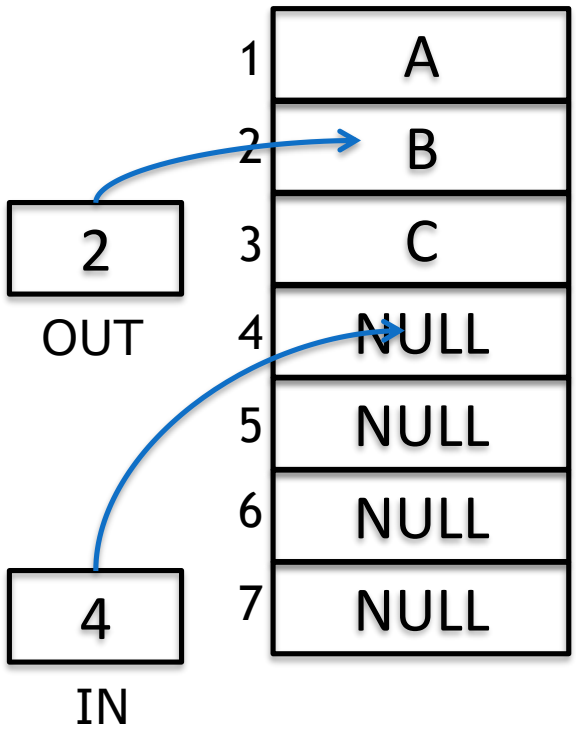
IN and Spooler[] are shared among all processes

Motivation: Two processes (or threads) want to print

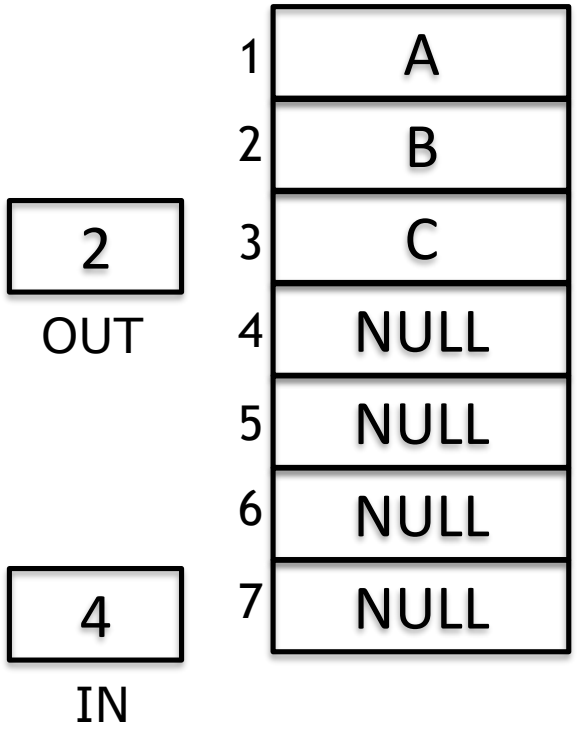
- Both processors write their job to the *spooler* – the (**shared**) queue from which the printer extracts job to print.
  - NULL: No job to print
  - OUT: Next job to print
  - IN: End of queue, write new job here

- Code for each process (adding jobs):
  Spooler[IN] = job
  IN++

IN and Spooler[] are shared among all processes

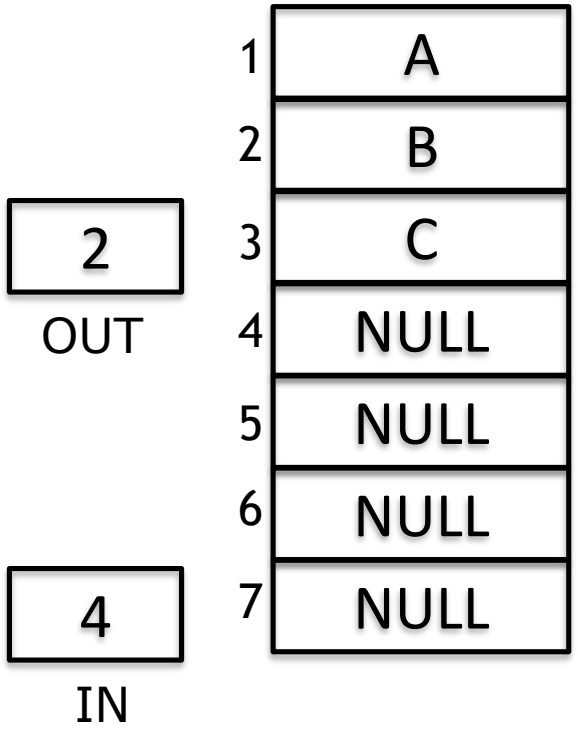


Possible Outcomes 1/2



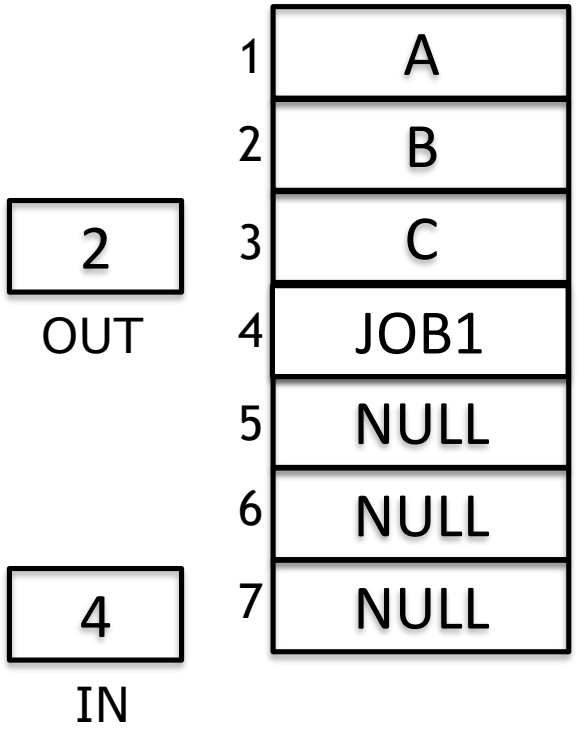
Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1



Possible Outcomes 1/2

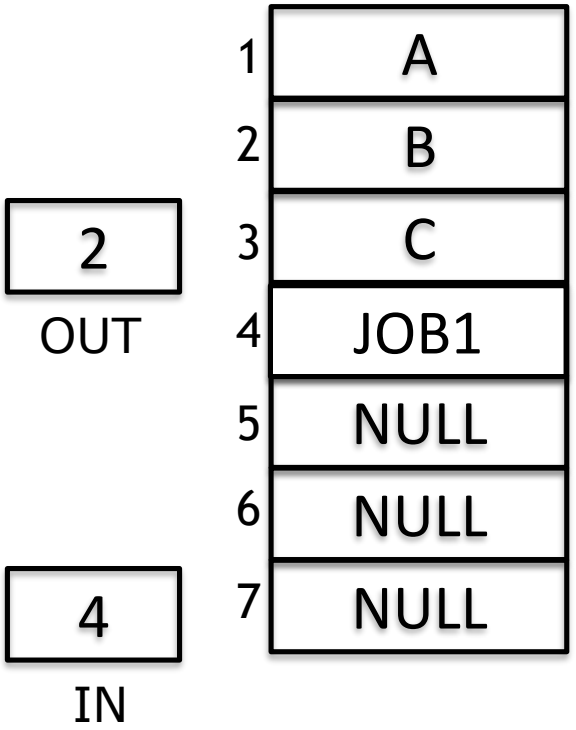
Process 1: Spooler[IN] = JOB1



Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1

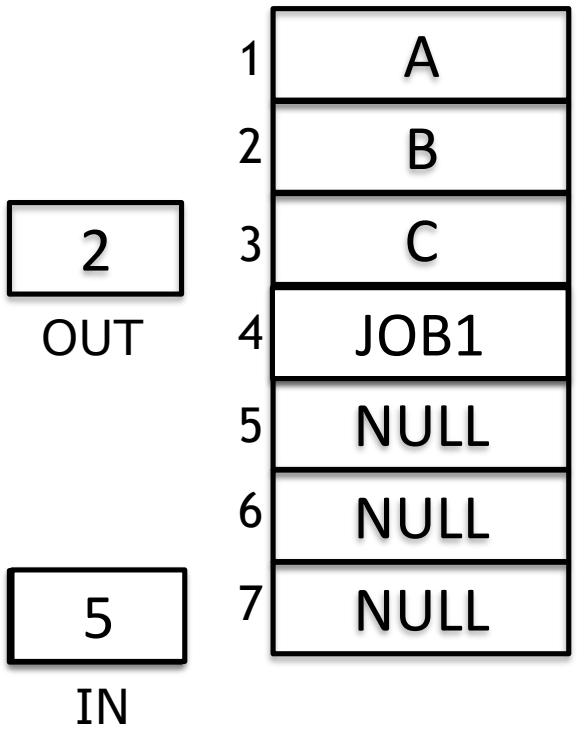
Process 1: IN++



Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1

Process 1: IN++

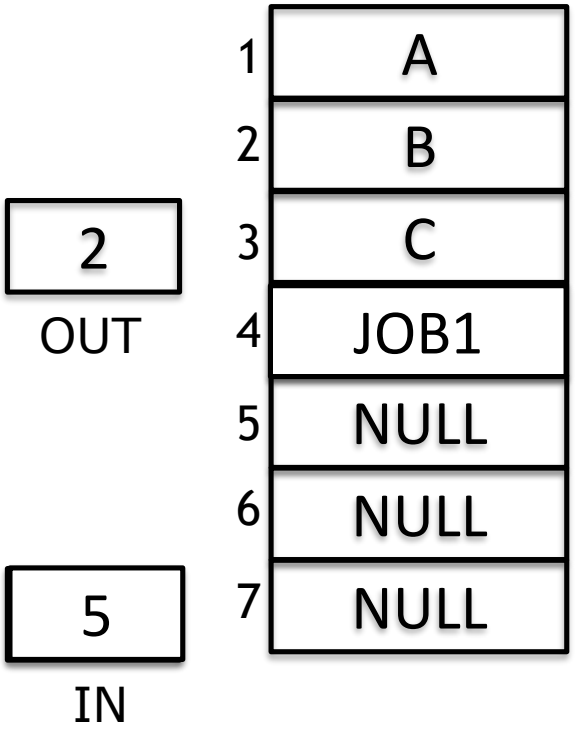


Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1

Process 1: IN++

<Context Switch by OS>



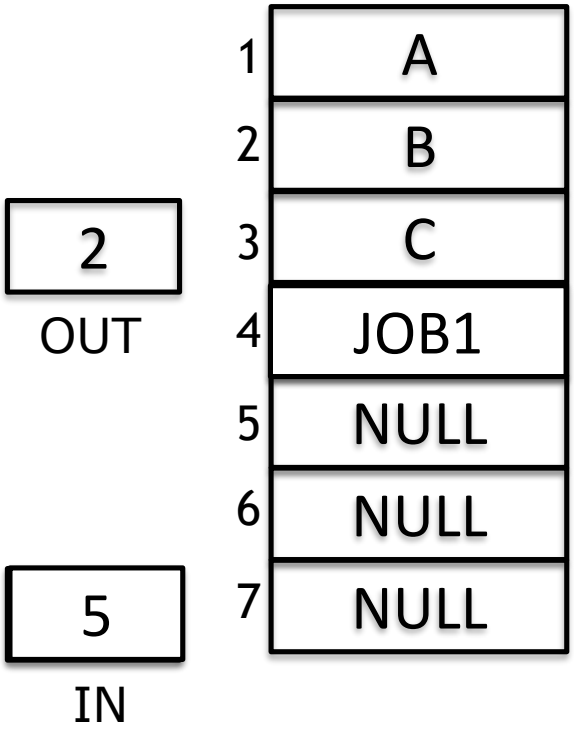
Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1

Process 1: IN++

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2



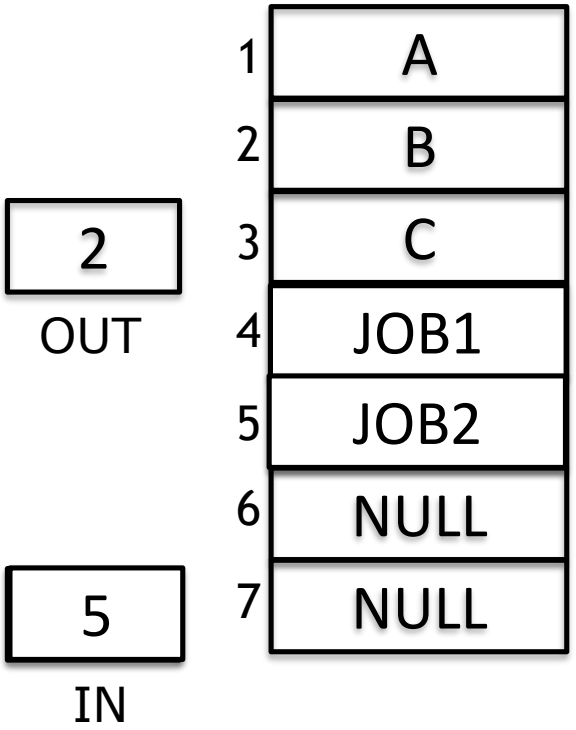
Possible Outcomes 1/2

Process 1: Spooler[IN] = JOB1

Process 1: IN++

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2



Possible Outcomes 1/2

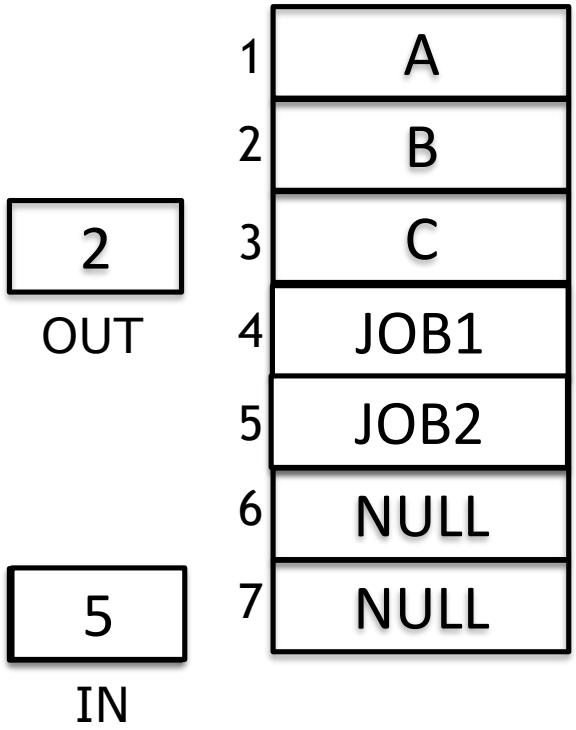
Process 1: Spooler[IN] = JOB1

Process 1: IN++

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

Process 2: IN++



Possible Outcomes 1/2

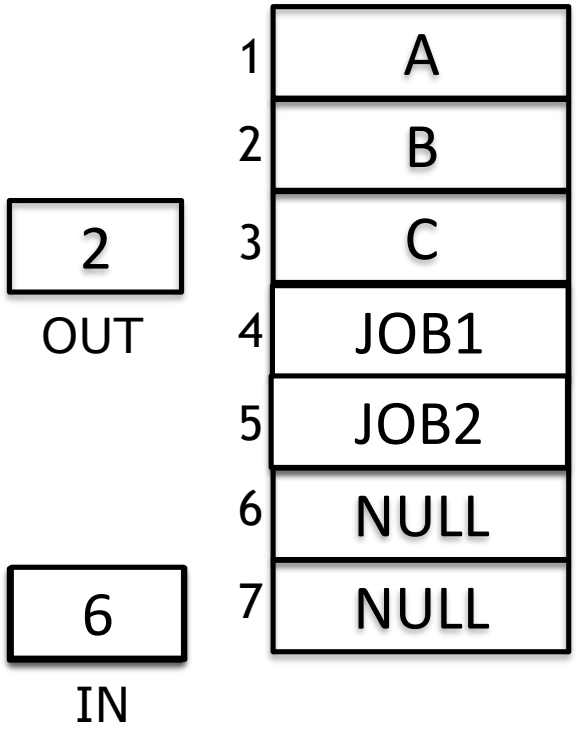
Process 1: Spooler[IN] = JOB1

Process 1: IN++

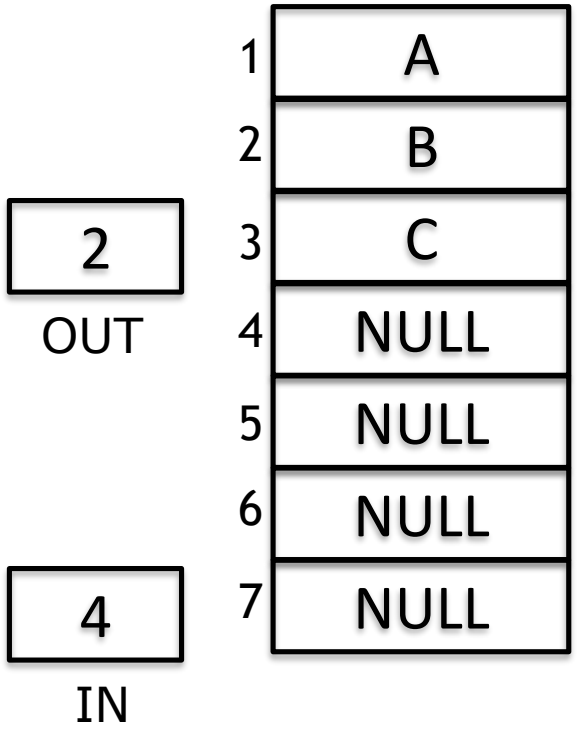
<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

Process 2: IN++

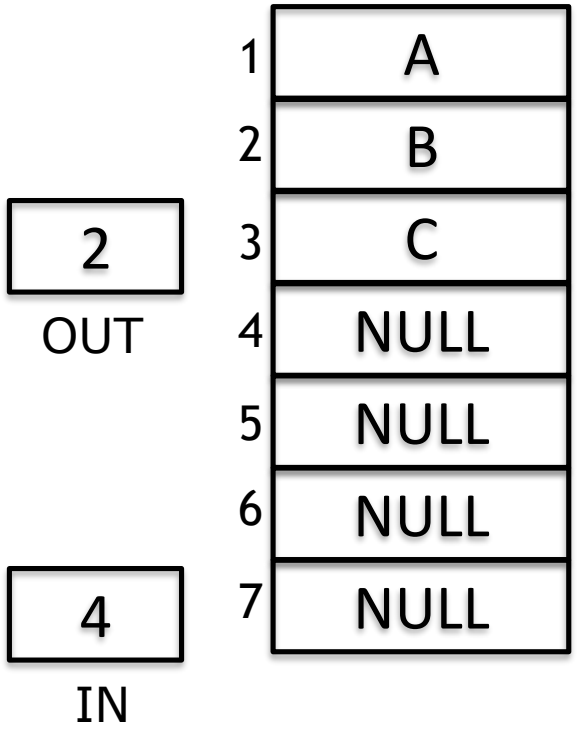


Possible Outcomes 2/2



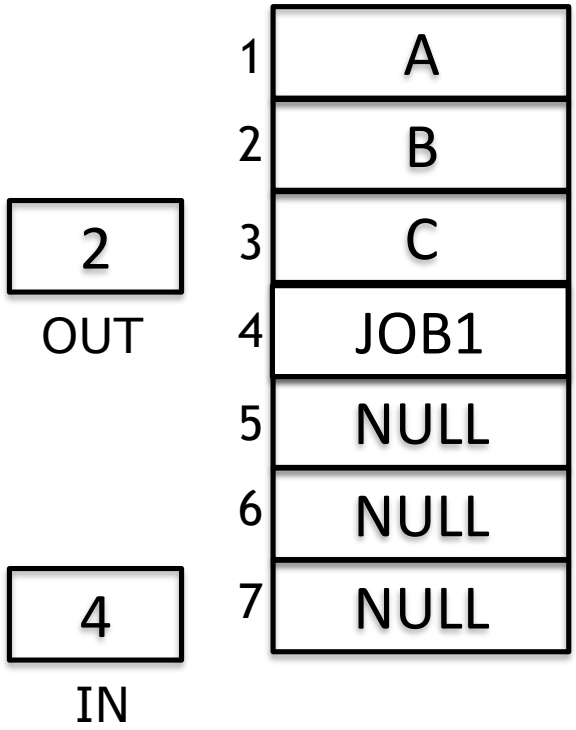
Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1



Possible Outcomes 2/2

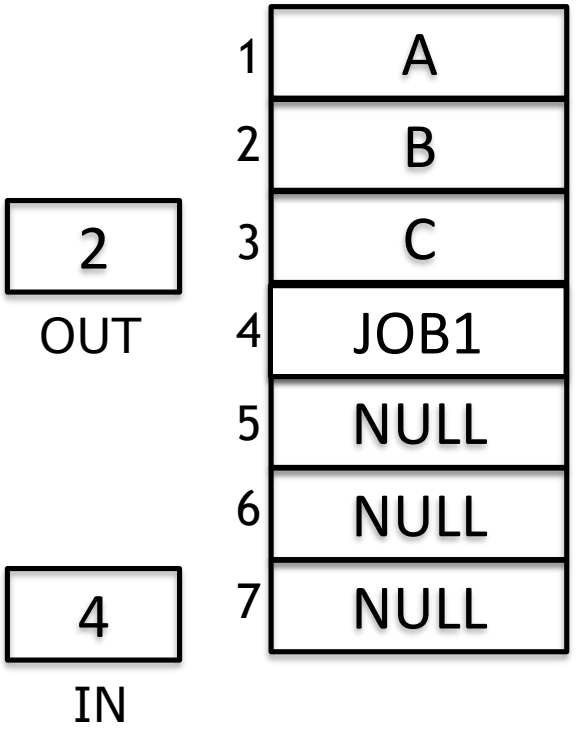
Process 1: Spooler[IN] = JOB1



Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

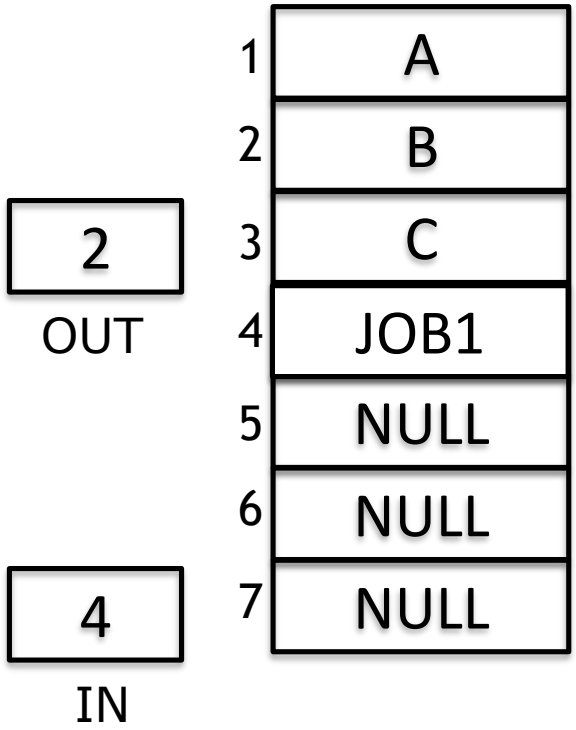


Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

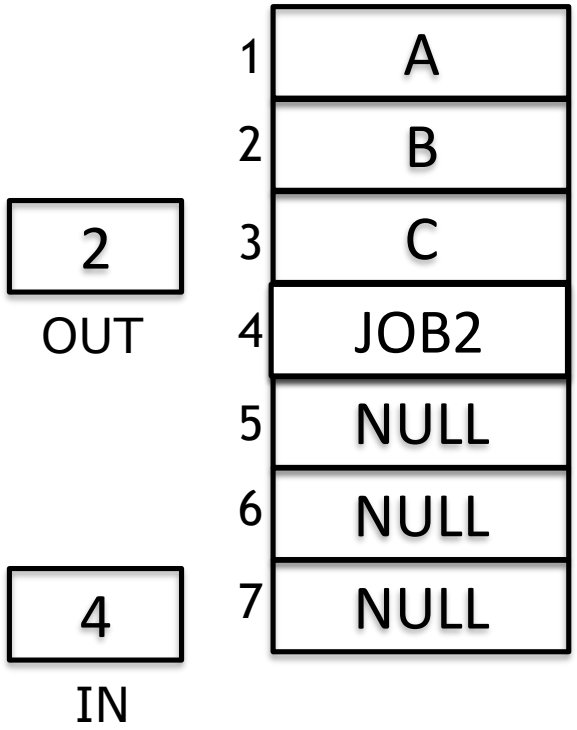


Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2



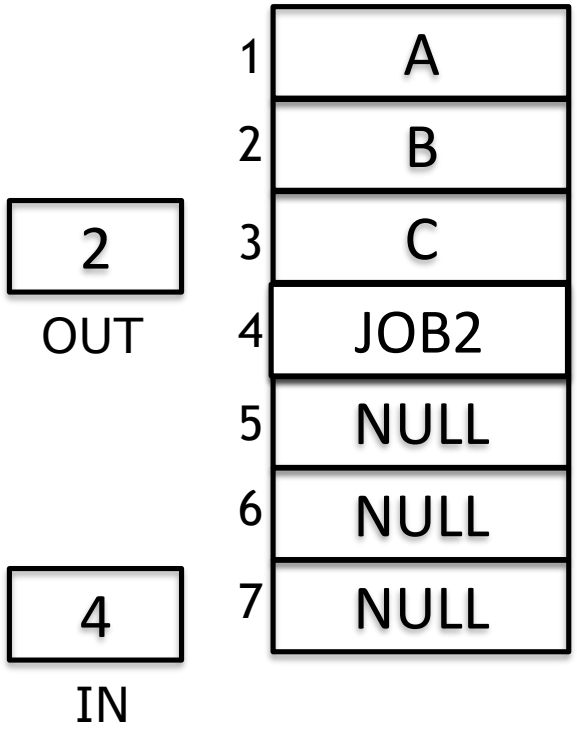
Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>



Possible Outcomes 2/2

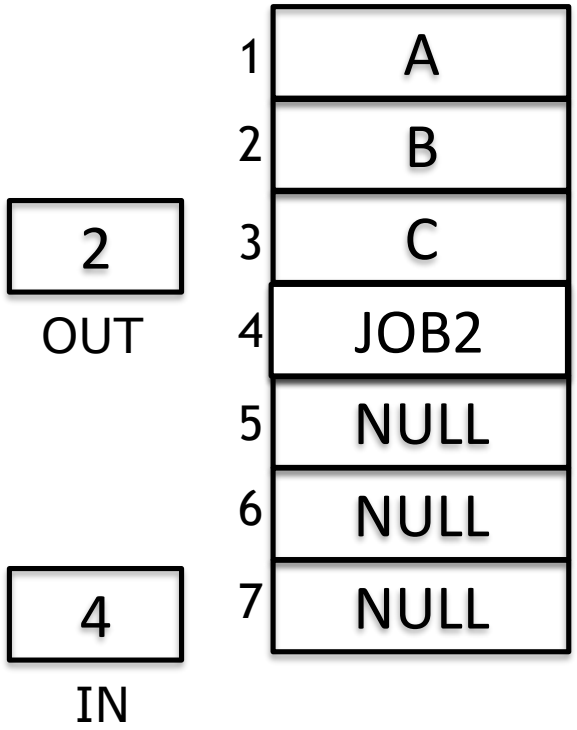
Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++



Possible Outcomes 2/2

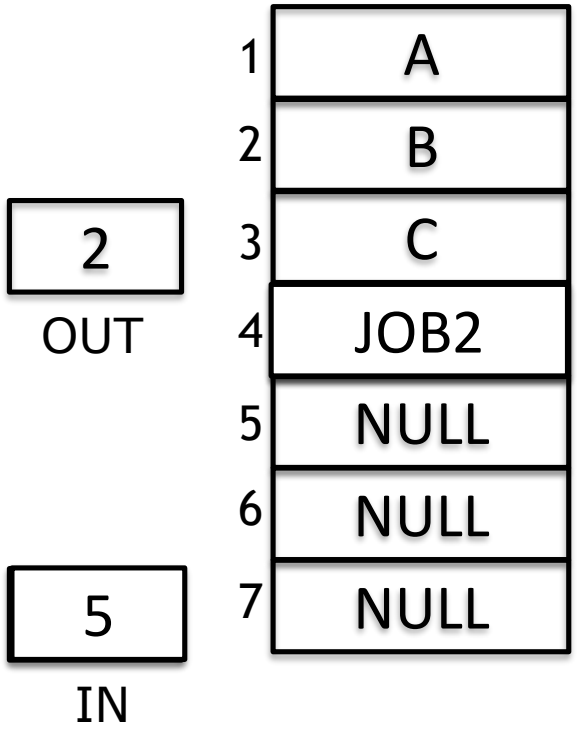
Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++



Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

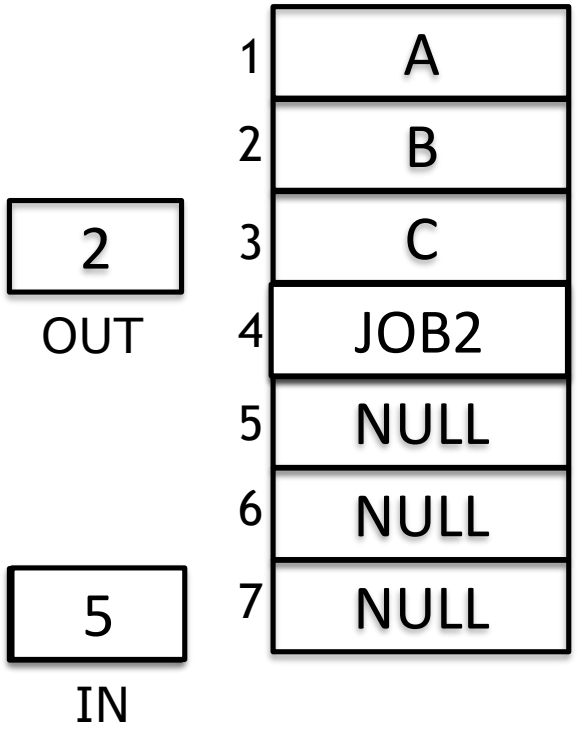
<Context Switch by OS>

Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

<Context Switch by OS>



Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

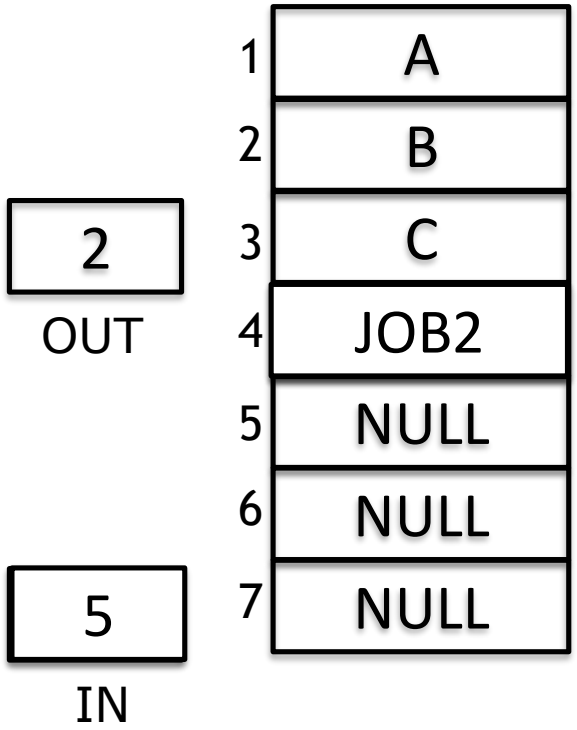
Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

<Context Switch by OS>

Process 2: IN++



Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

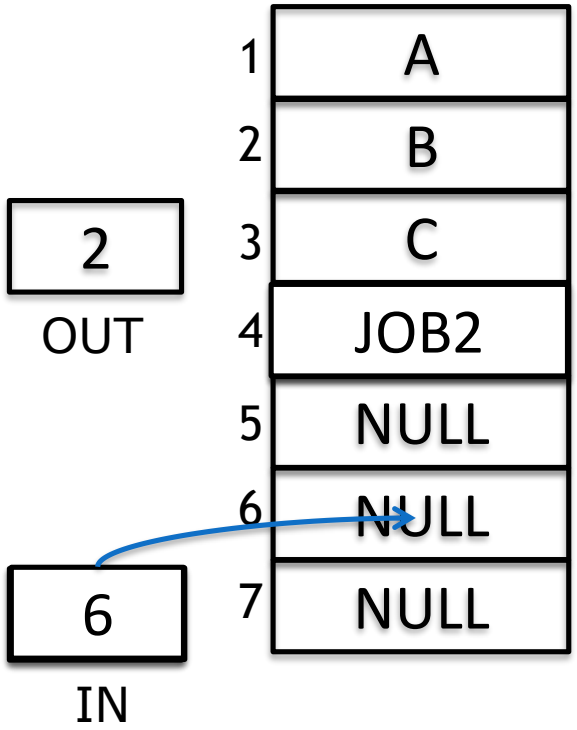
Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

<Context Switch by OS>

Process 2: IN++



Possible Outcomes 2/2

Process 1: Spooler[IN] = JOB1

<Context Switch by OS>

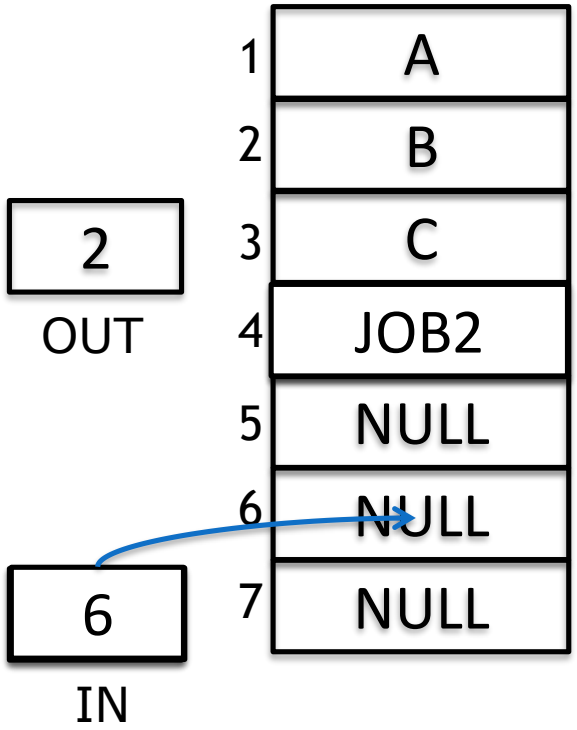
Process 2: Spooler[IN] = JOB2

<Context Switch by OS>

Process 1: IN++

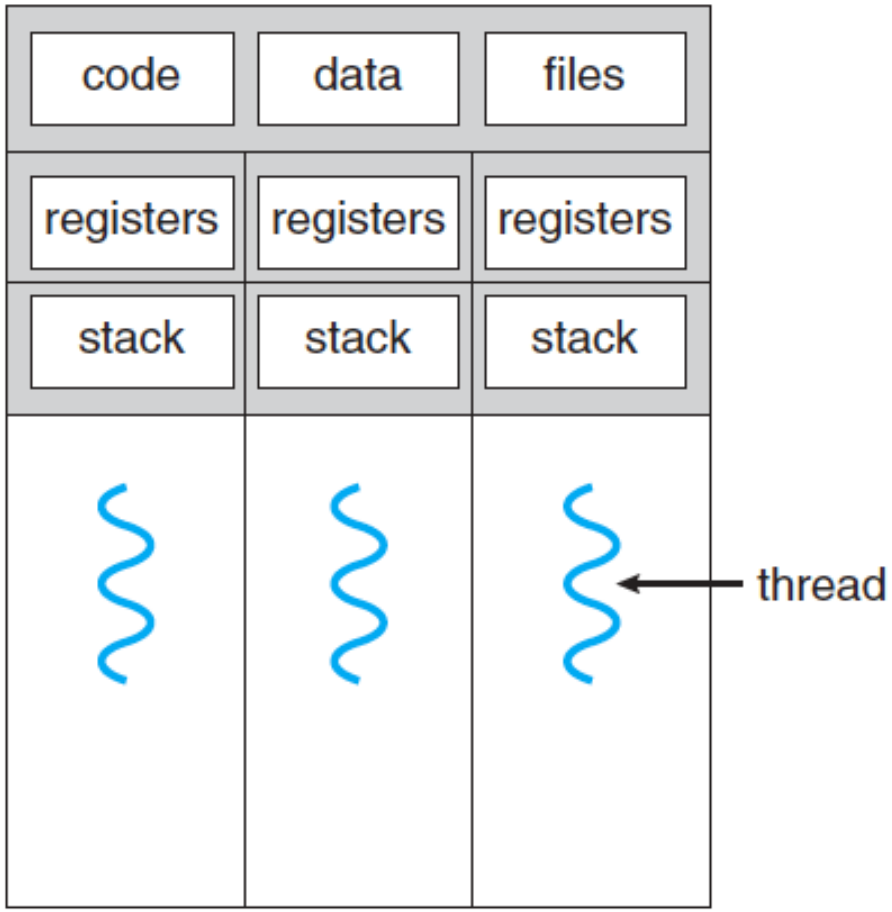
<Context Switch by OS>

Process 2: IN++

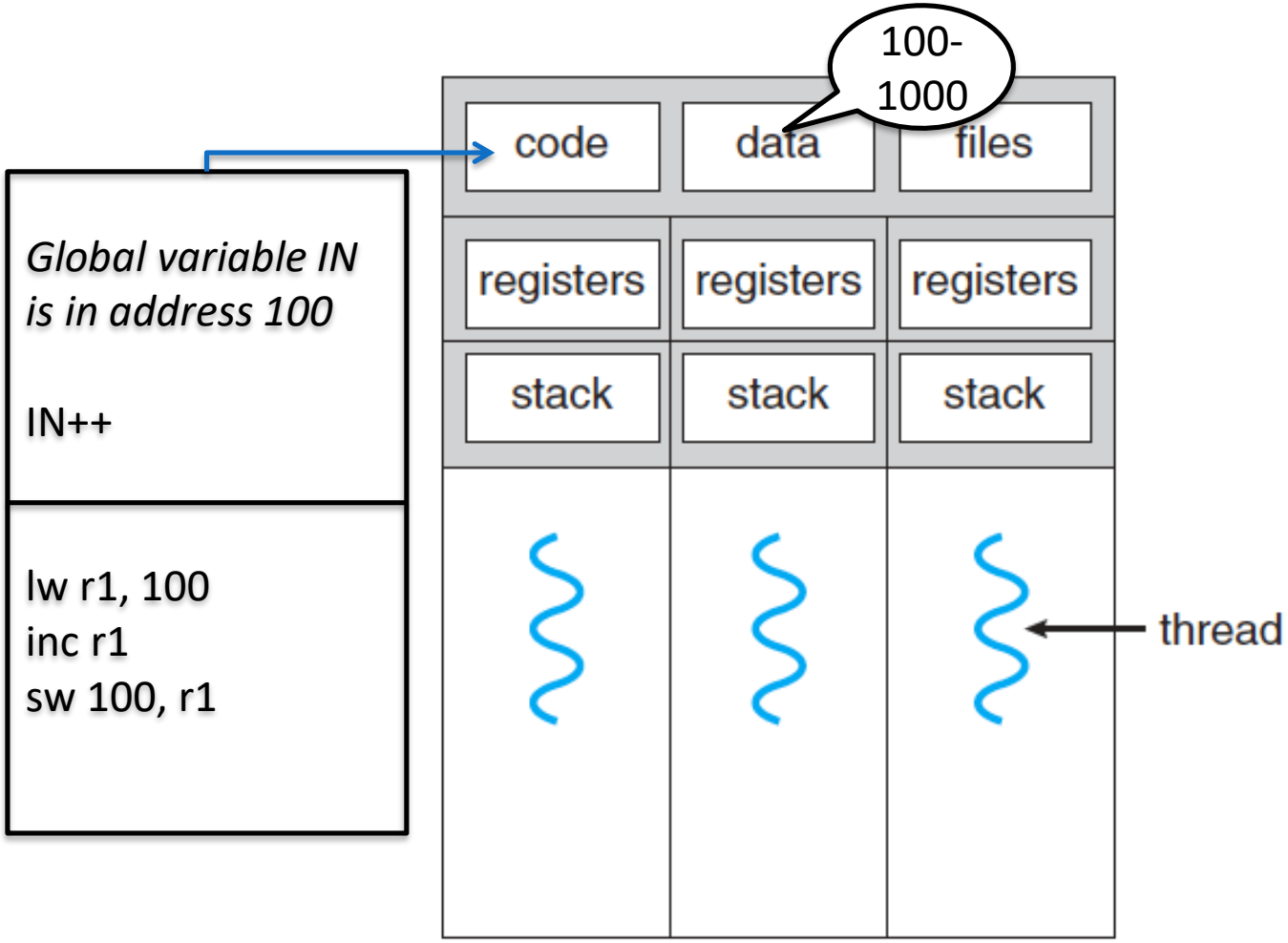


→ JOB1 is lost; all jobs after JOB2 may not be printed as well, because of the gap

Further Motivation

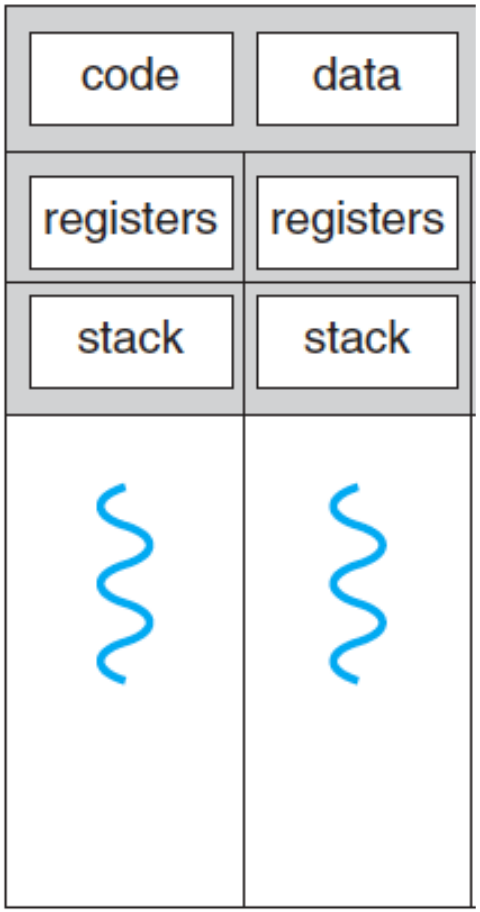


Further Motivation



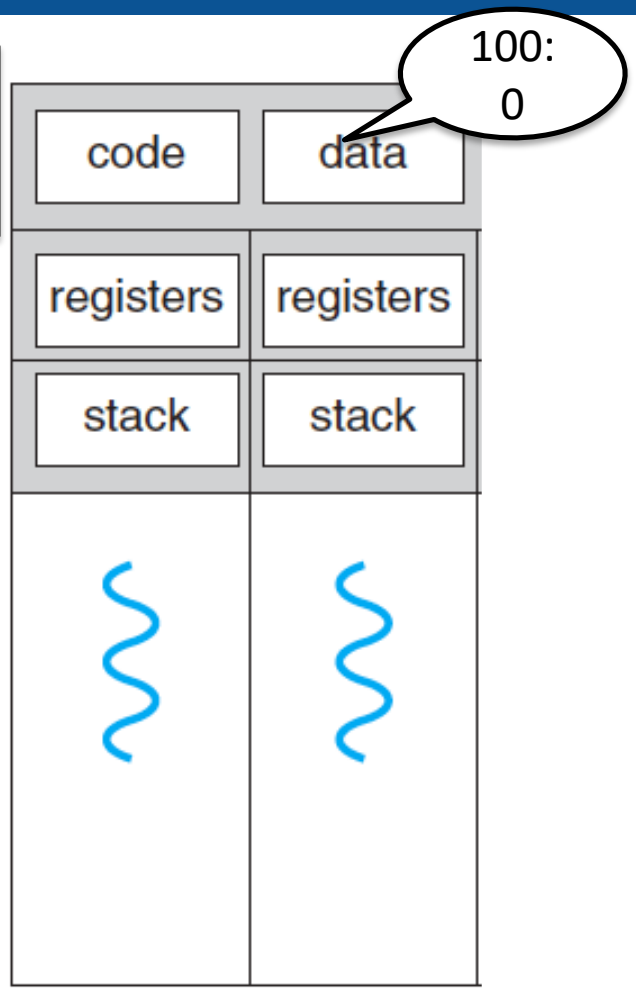
Further Motivation

```
lw r1, 100
inc r1
sw 100, r1
```

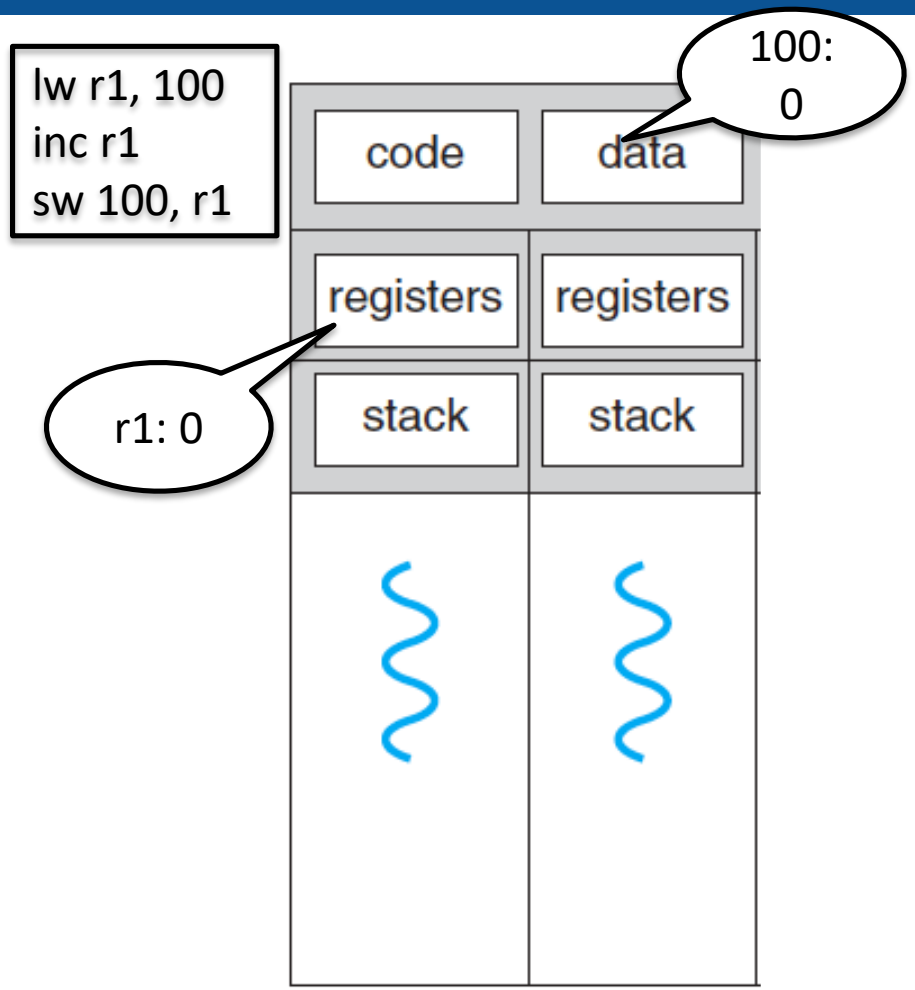


Further Motivation

```
lw r1, 100
inc r1
sw 100, r1
```

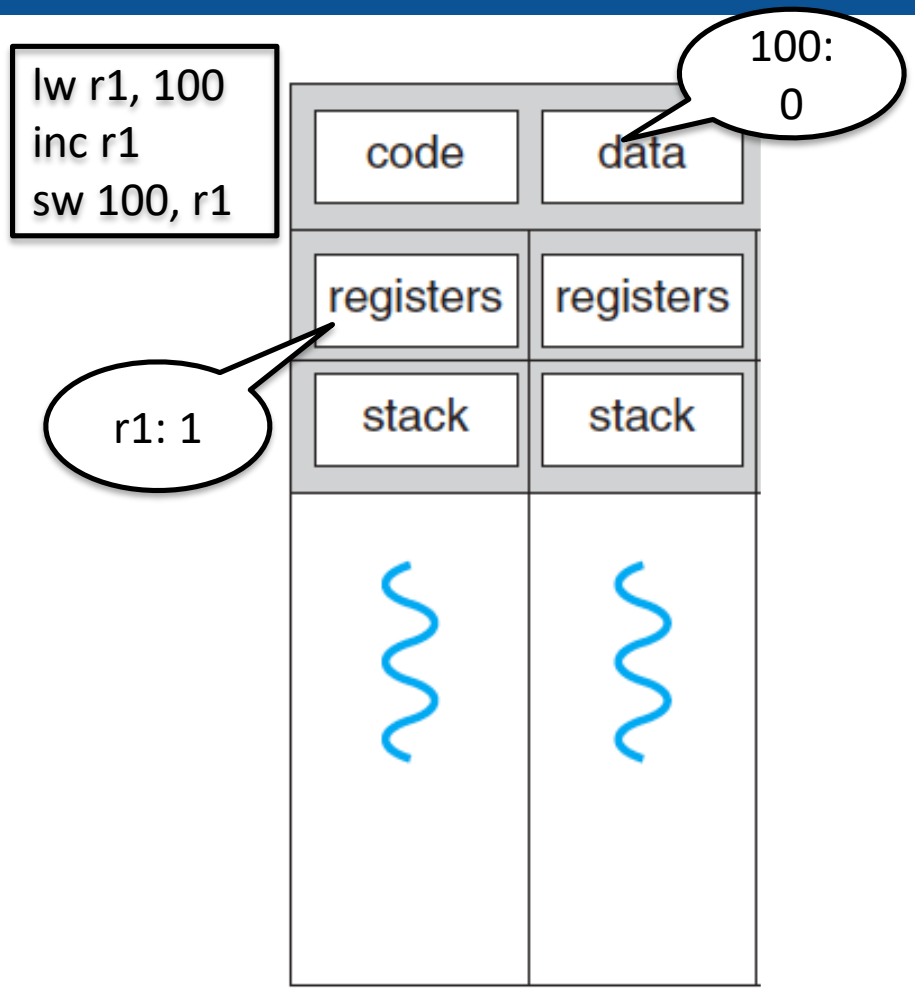


Further Motivation



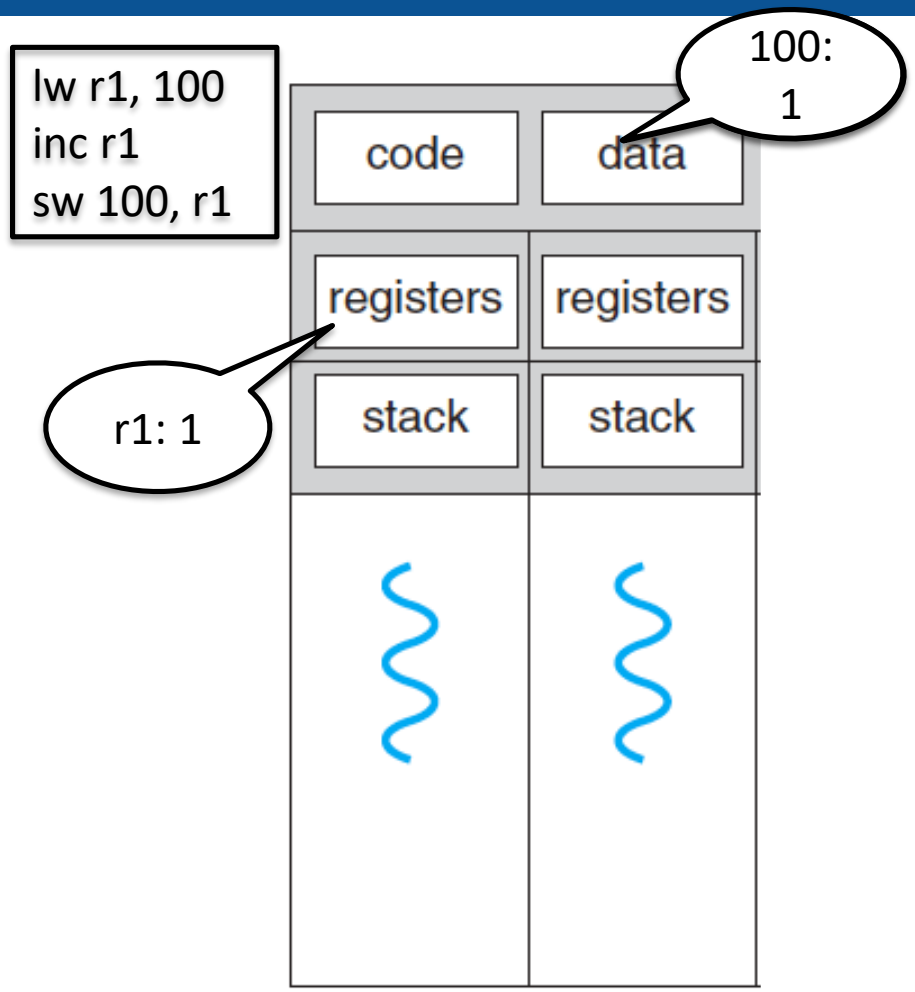
- Thread 1: lw r1,100

Further Motivation



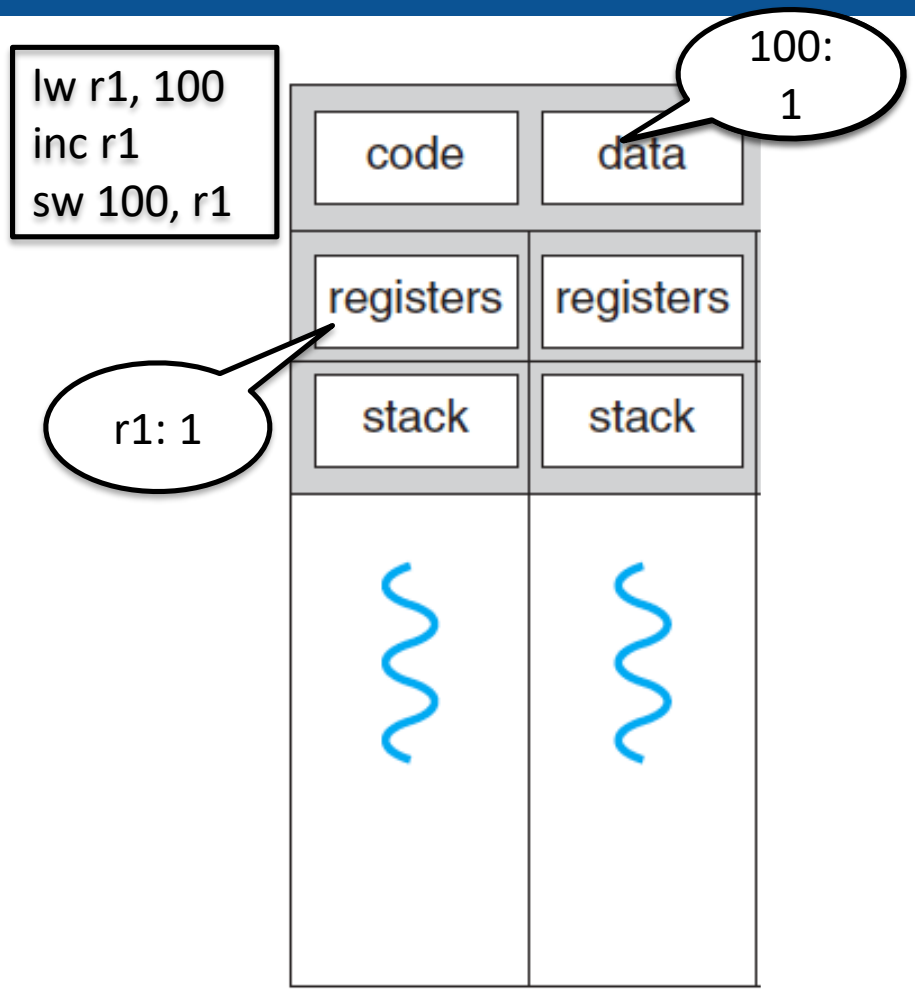
- Thread 1: lw r1,100
- Thread 1: inc r1

Further Motivation



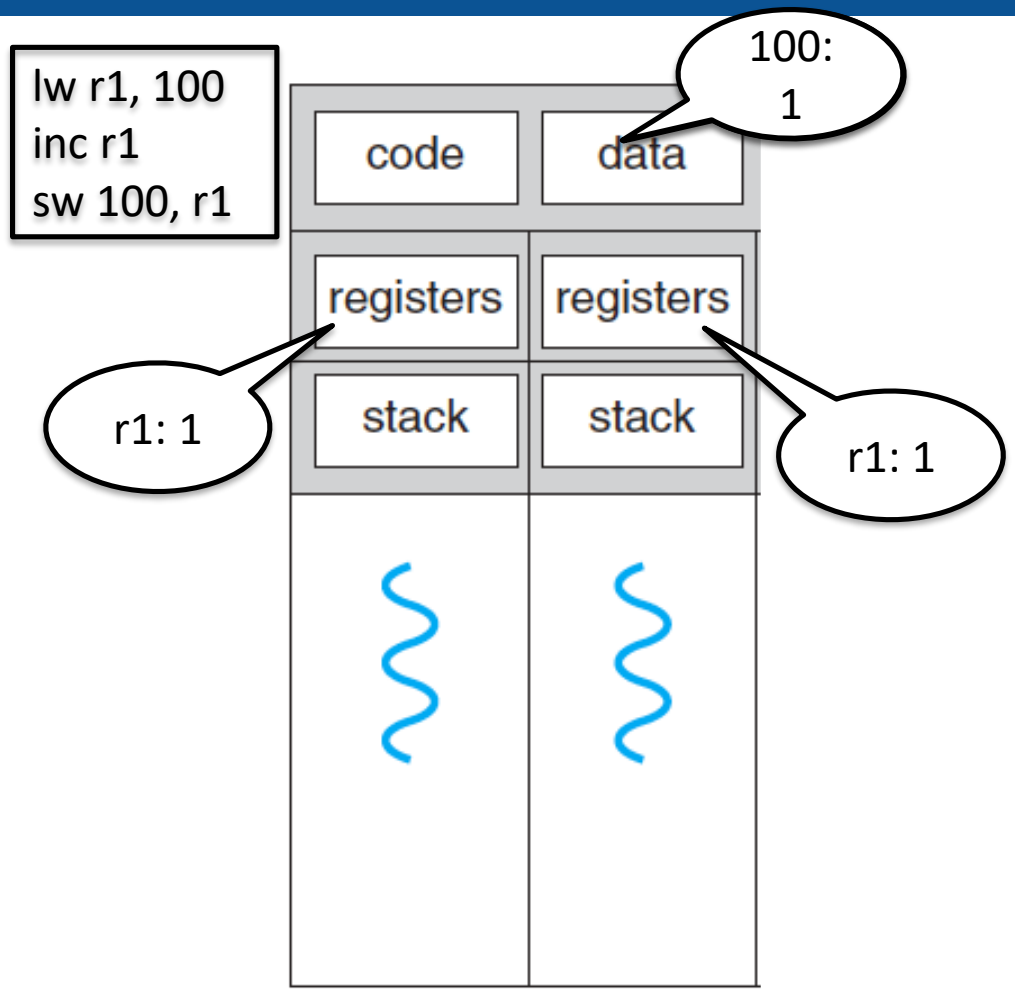
- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1

Further Motivation



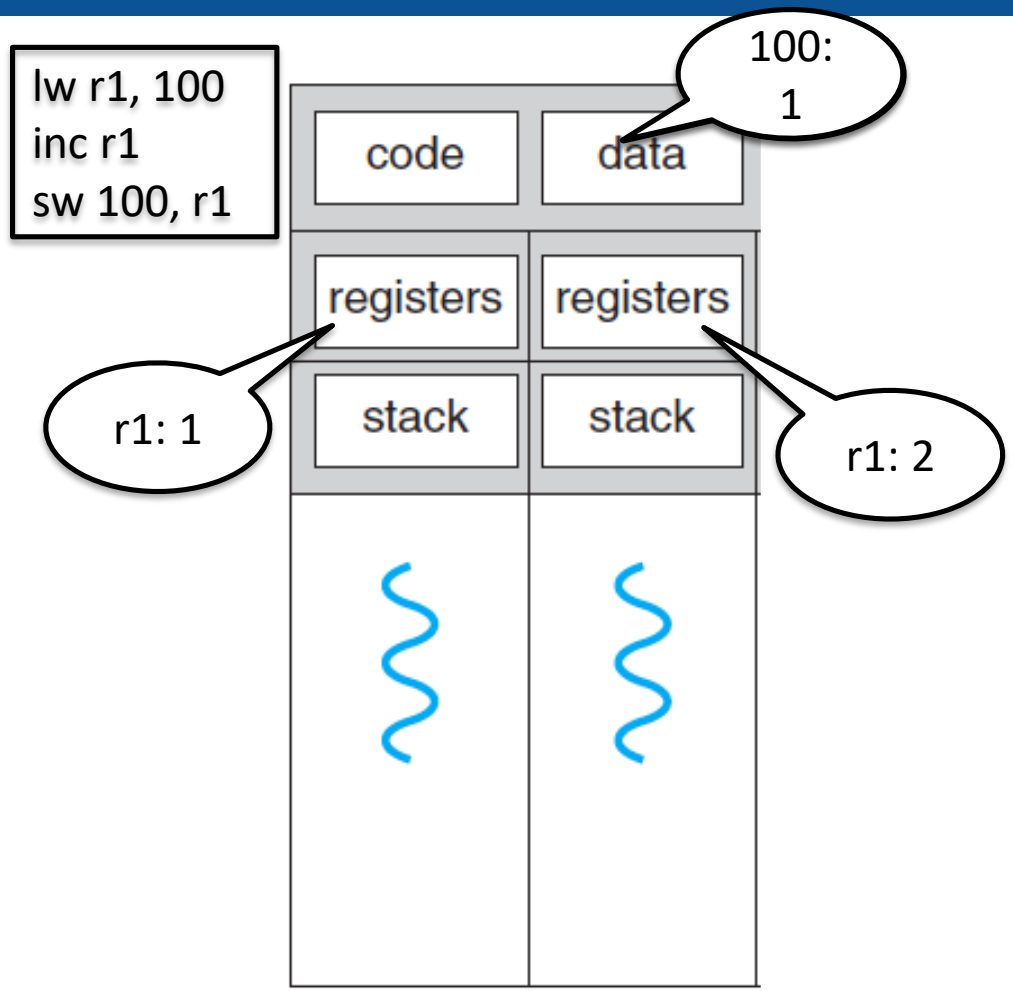
- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>

Further Motivation



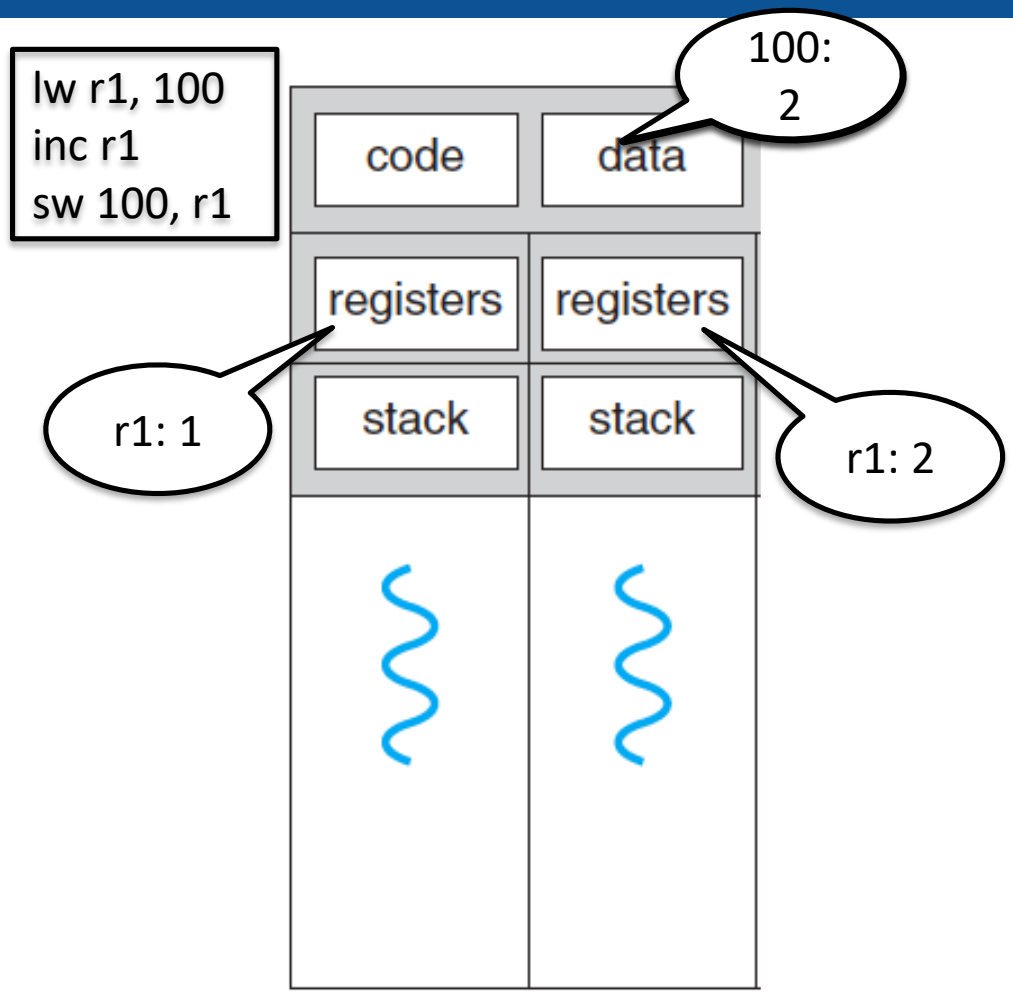
- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100

Further Motivation



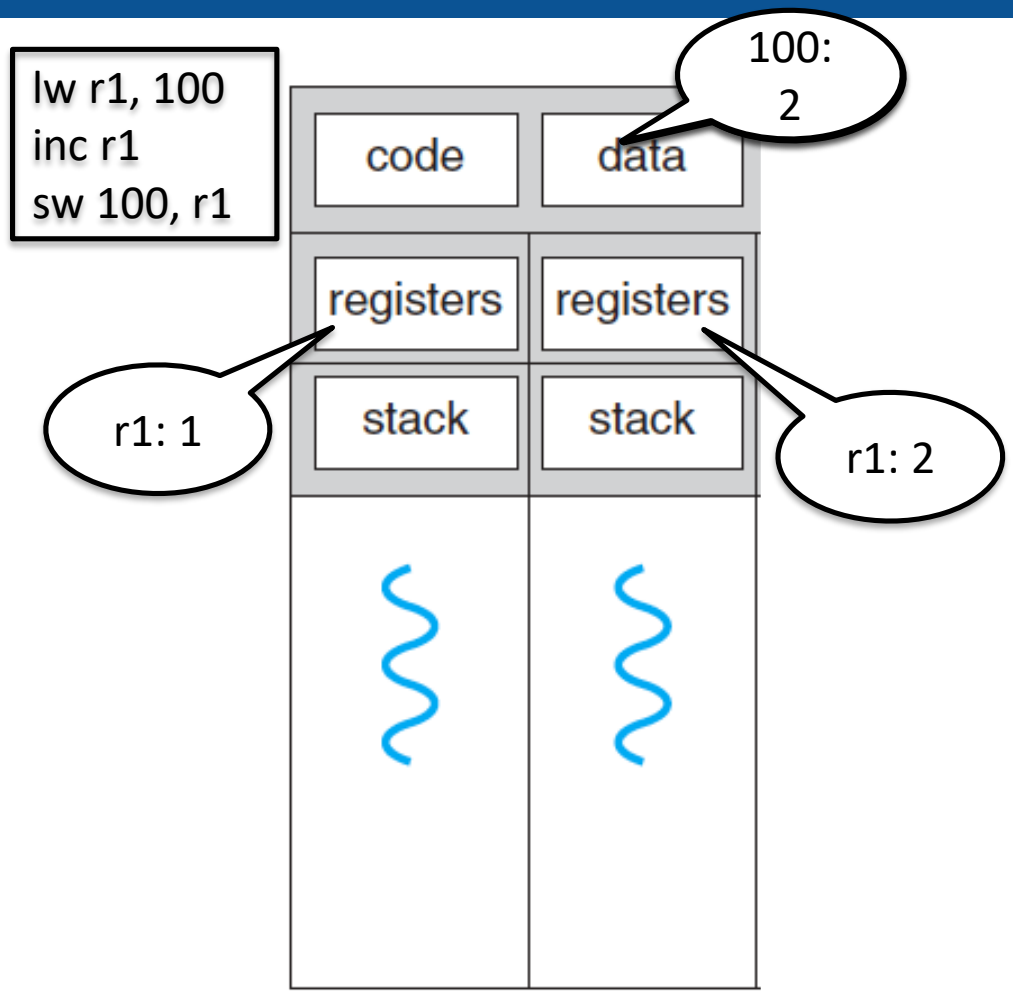
- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1

Further Motivation



- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

Further Motivation

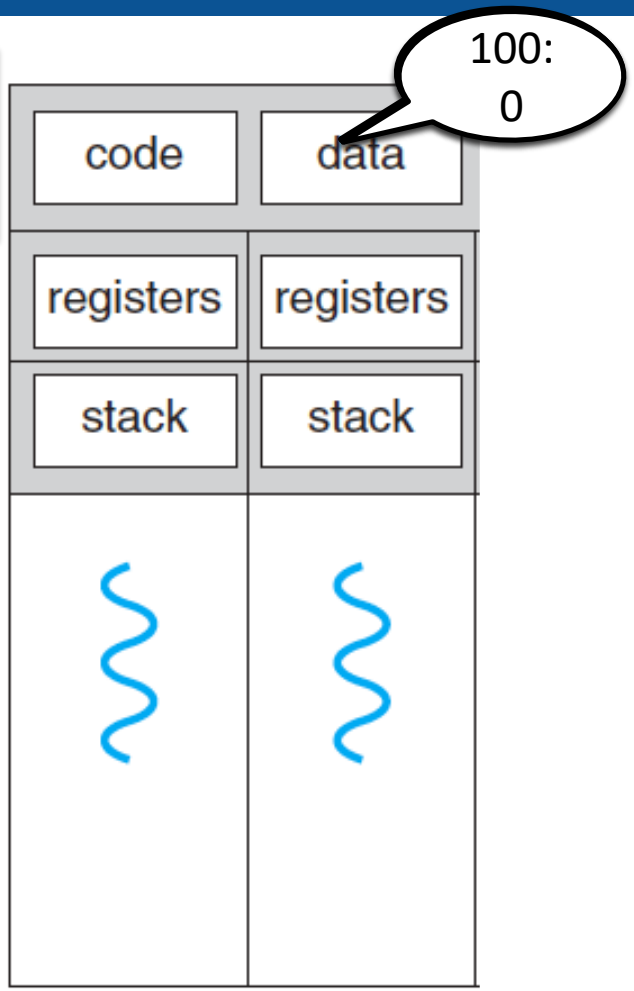


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

Further Motivation

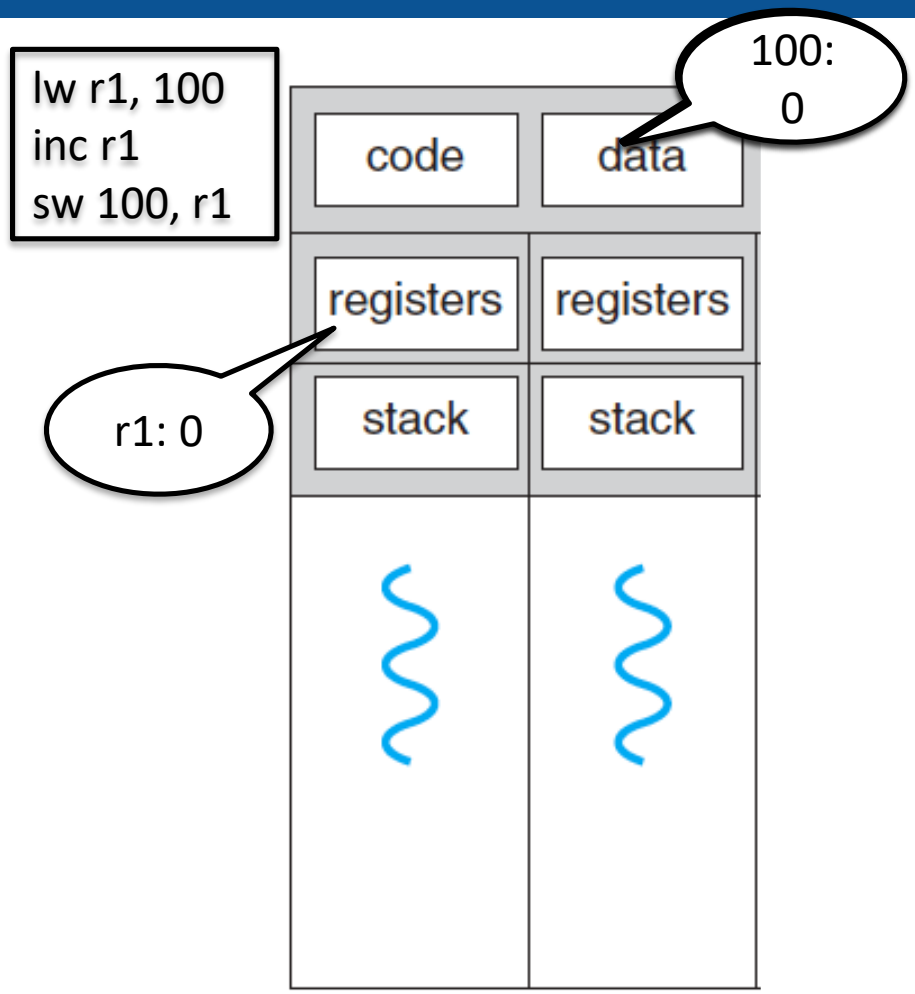
```
lw r1, 100
inc r1
sw 100, r1
```



- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

Further Motivation

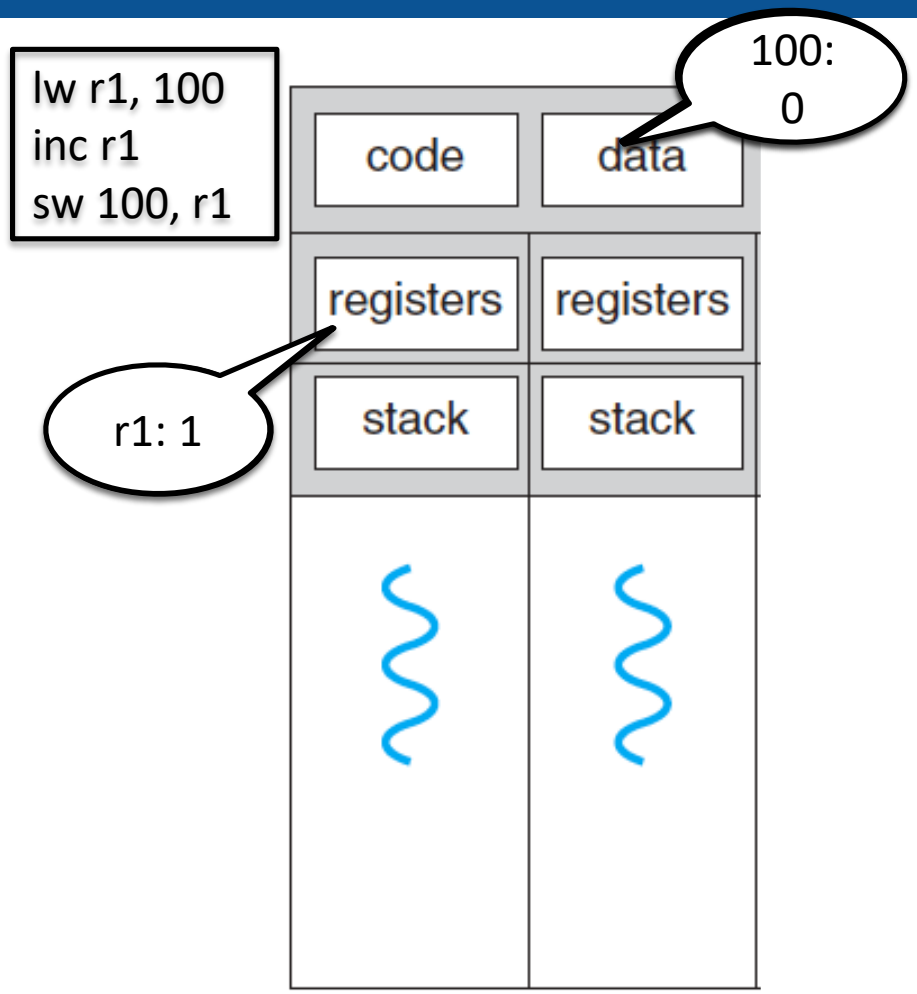


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100

Further Motivation

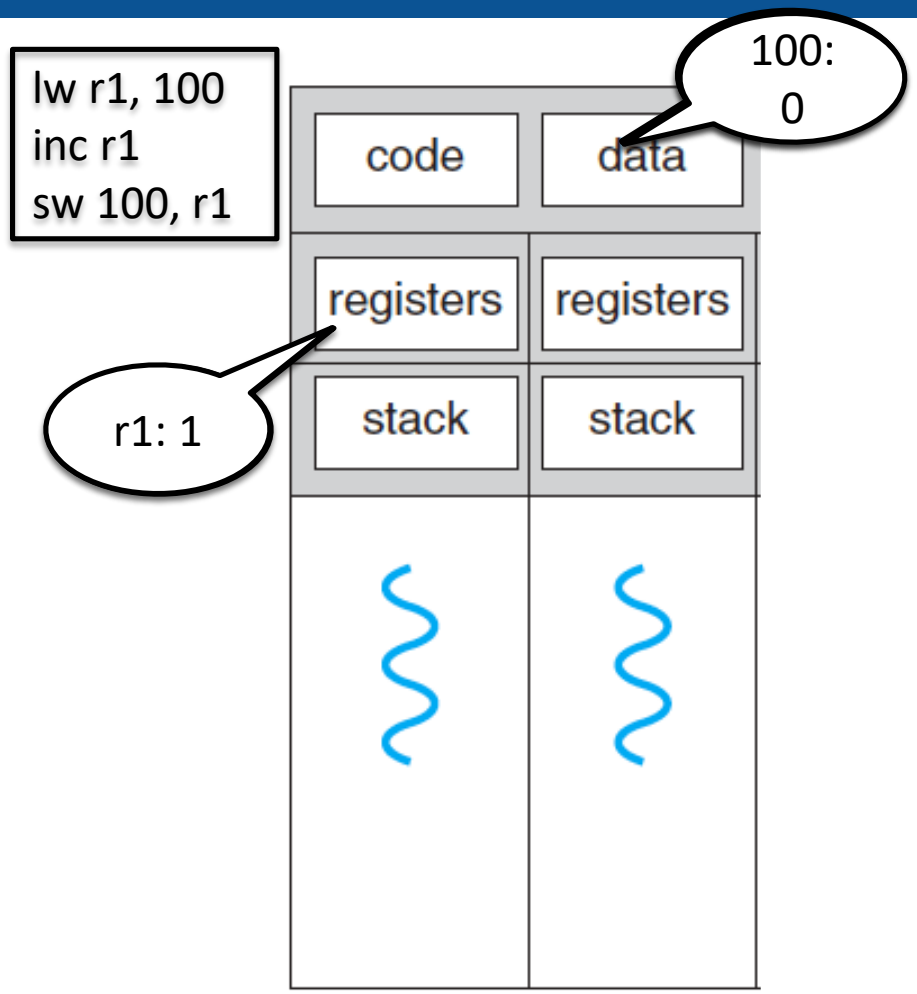


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1

Further Motivation

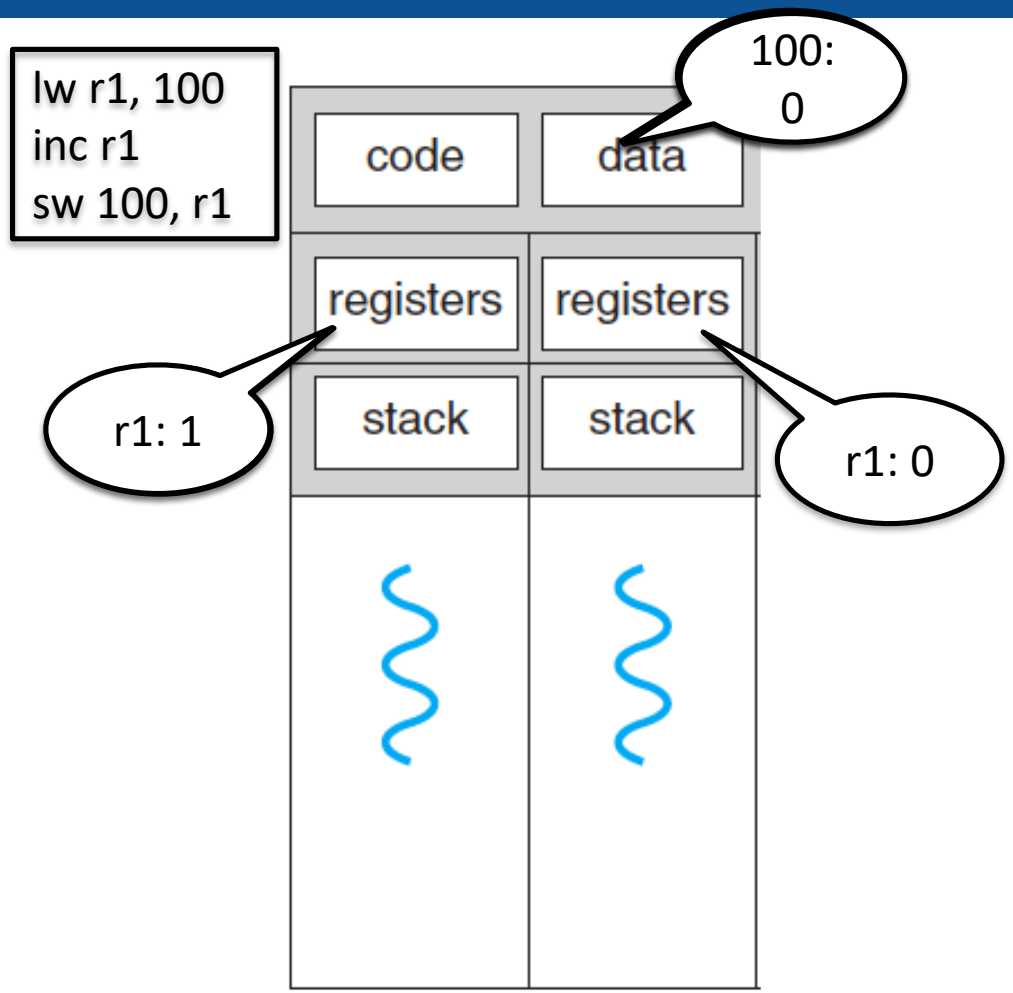


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>

Further Motivation

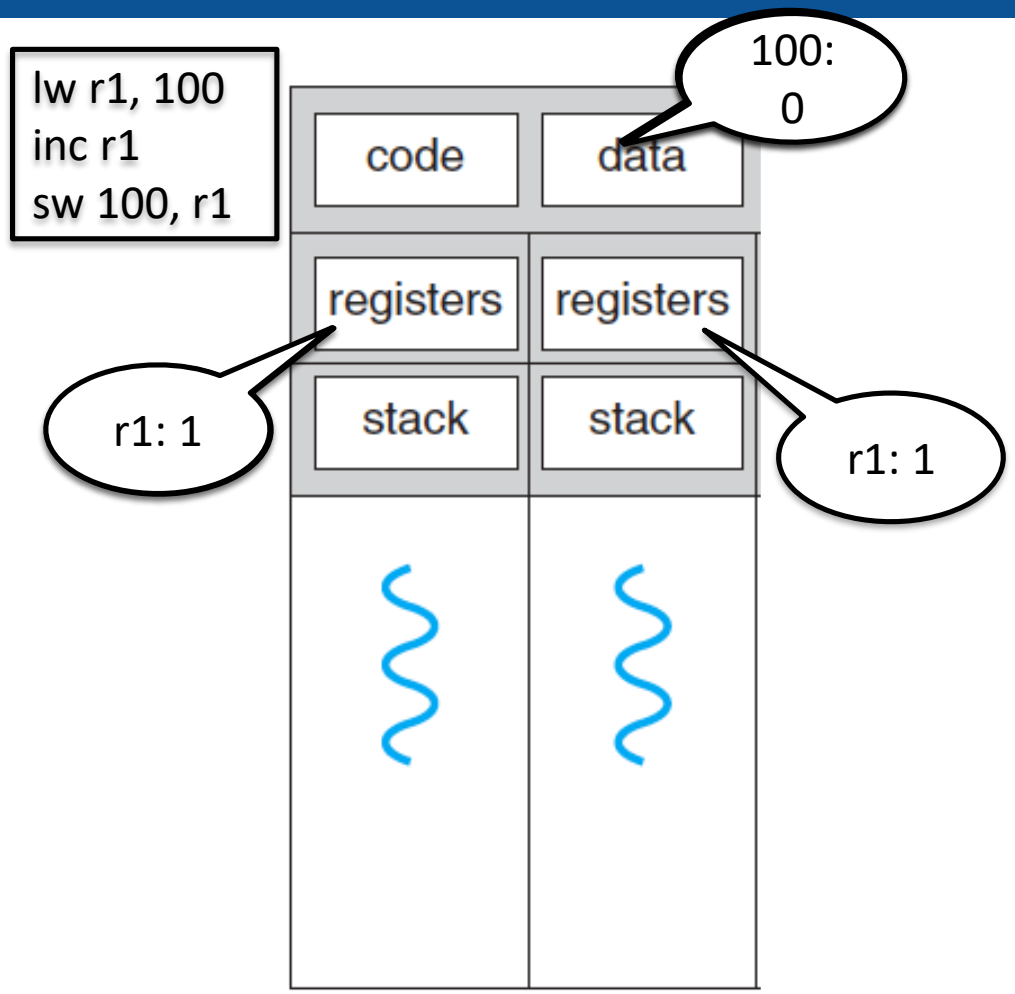


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100

Further Motivation

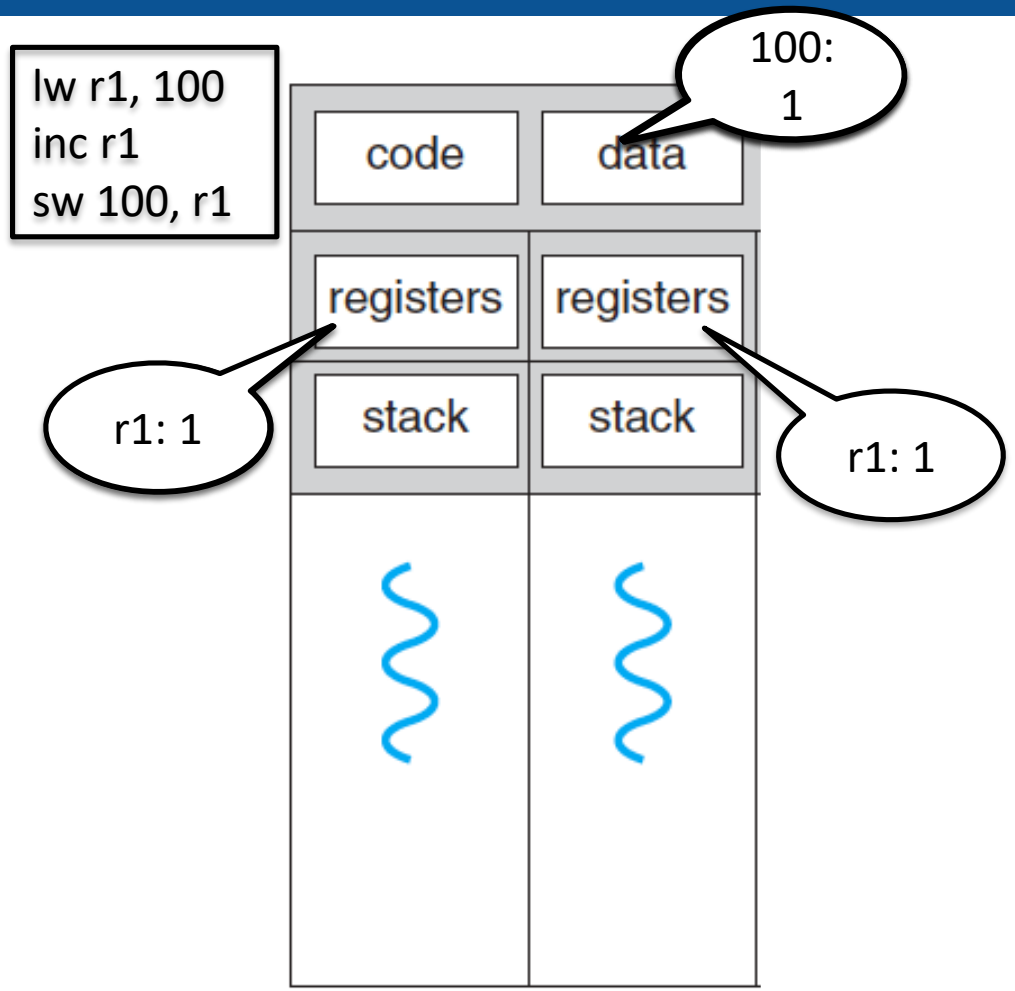


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1

Further Motivation

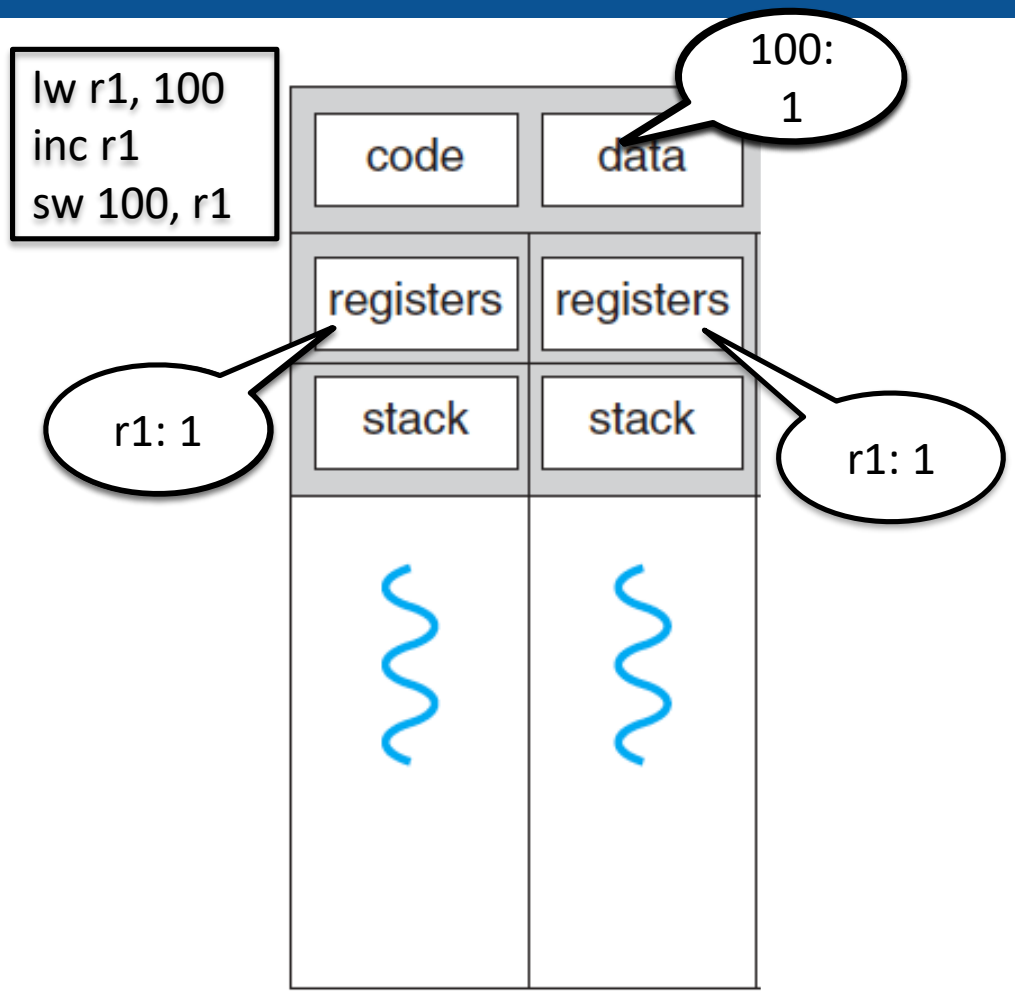


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

Further Motivation

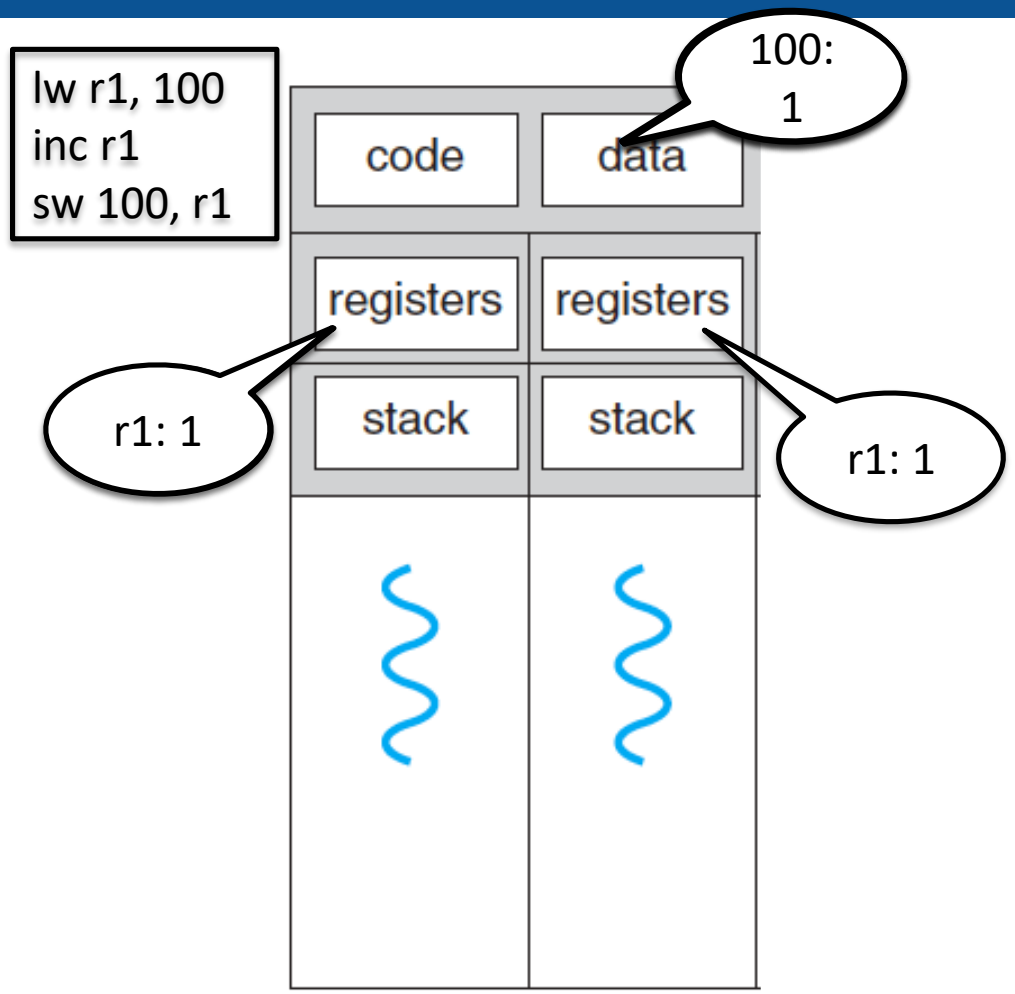


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1
- <Context Switch>

Further Motivation

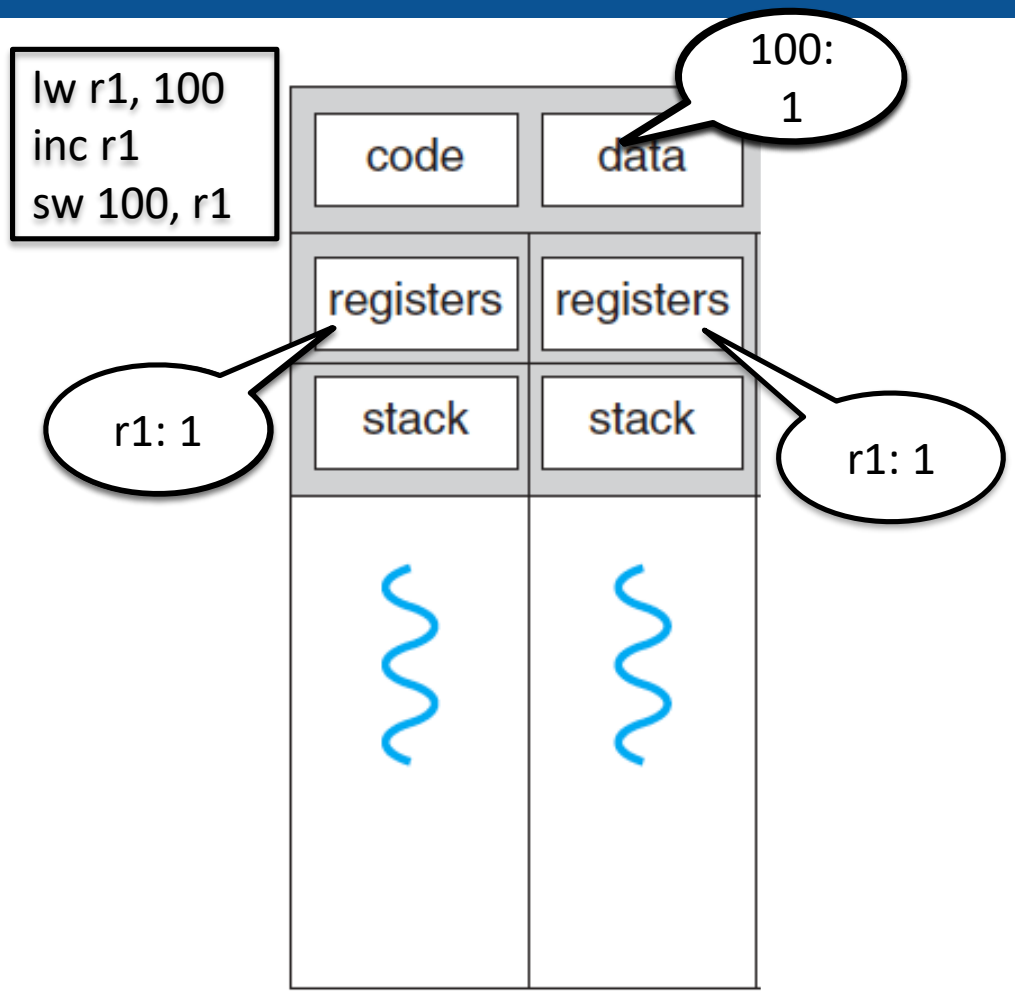


- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1
- <Context Switch>
- Thread 1: sw 100,r1

Further Motivation



- Thread 1: lw r1,100
- Thread 1: inc r1
- Thread 1: sw 100,r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1

→ IN = 2

- Thread 1: lw r1,100
- Thread 1: inc r1
- <Context Switch>
- Thread 2: lw r1, 100
- Thread 2: inc r1
- Thread 2: sw 100, r1
- <Context Switch>
- Thread 1: sw 100,r1

→ IN = 1

“Too Much Milk”…

Time	You	Your Roommate
3:00	Arrive Home	
3:05	Look in fridge; no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge; no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home; put milk in fridge	
3:45		Buy milk
3:50		Arrive home; oh no!

"Too Much Milk"…

Time	You	Your Roommate
3:00	Arrive Home	
3:05	Look in fridge; no milk	
3:10	Leave for grocery	
3:15		Arrive home
3:20	Arrive at grocery	Look in fridge; no milk
3:25	Buy milk	Leave for grocery
3:35	Arrive home; put milk in fridge	
3:45		Buy milk
3:50		Arrive home; oh no!

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

Where: Critical section is a piece of code that accesses the shared resource.

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

Where: Critical section is a piece of code that accesses the shared resource.

- Can be one instruction or very large code

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

Where: Critical section is a piece of code that accesses the shared resource.

- Can be one instruction or very large code
- Correctness: must not be concurrently executed by more than one thread process.

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

Where: Critical section is a piece of code that accesses the shared resource.

- Can be one instruction or very large code
- Correctness: must not be concurrently executed by more than one thread process.

Solution: Mutual exclusion algorithms are used to avoid the simultaneous execution of a critical section

What's happening here?

The Problem: Uncoordinated access to a shared resource (data structure or device)

- By multiple threads, by multiple processes, by multiple cores/processors

Where: Critical section is a piece of code that accesses the shared resource.

- Can be one instruction or very large code
- Correctness: must not be concurrently executed by more than one thread process.

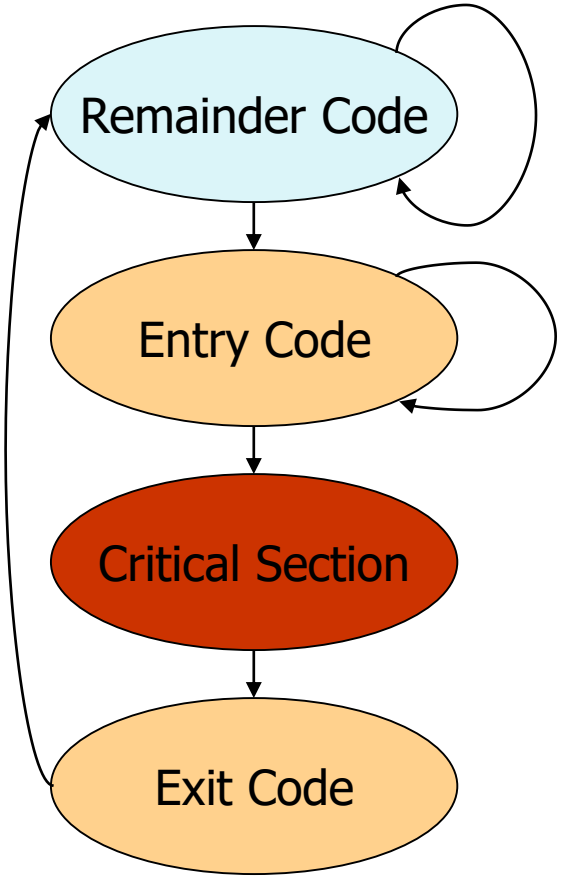
Solution: Mutual exclusion algorithms are used to avoid the simultaneous execution of a critical section

Important: Sometimes it is not enough to ensure mutual exclusion (e.g., when the order of execution matters)

- More on this later

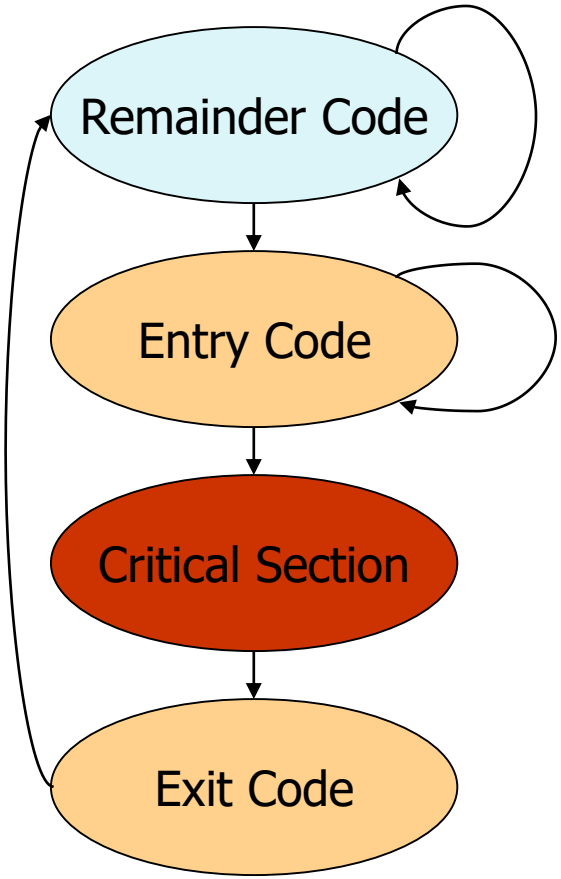
Modeling the Mutual Exclusion Problem

[First modeling by Dijkstra, 1965]



Modeling the Mutual Exclusion Problem

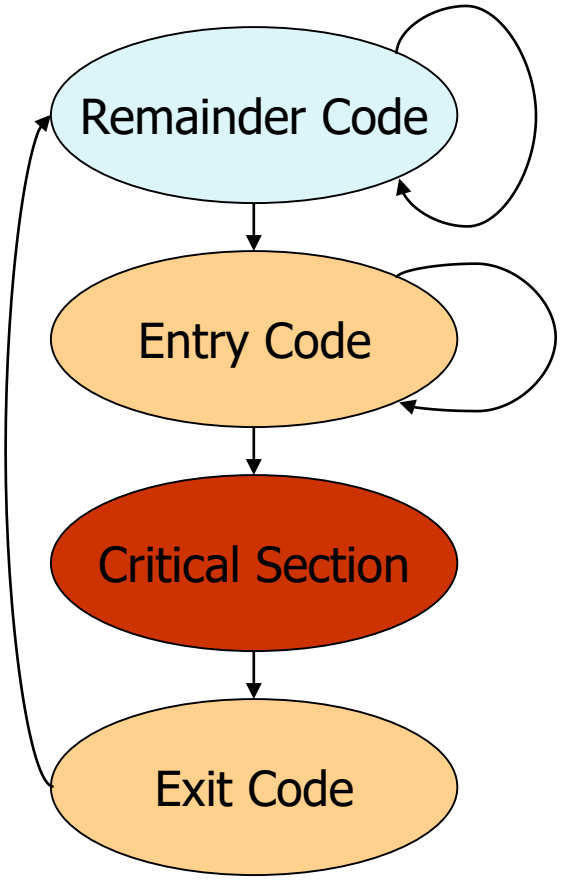
[First modeling by Dijkstra, 1965]



The Problem: Write entry and exit code. We will examine the code against the following properties (the goal is to ensure all of them)

Modeling the Mutual Exclusion Problem

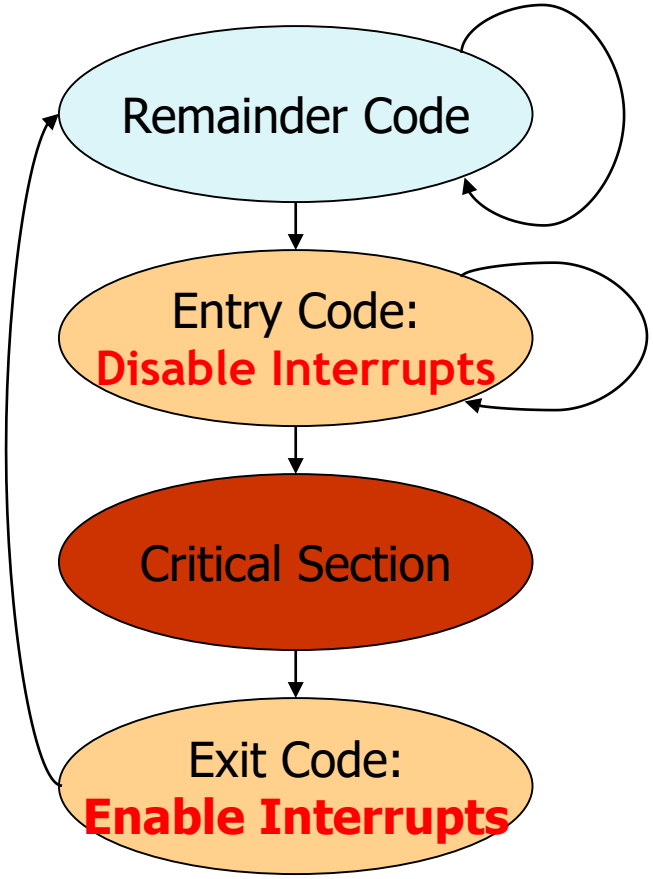
[First modeling by Dijkstra, 1965]



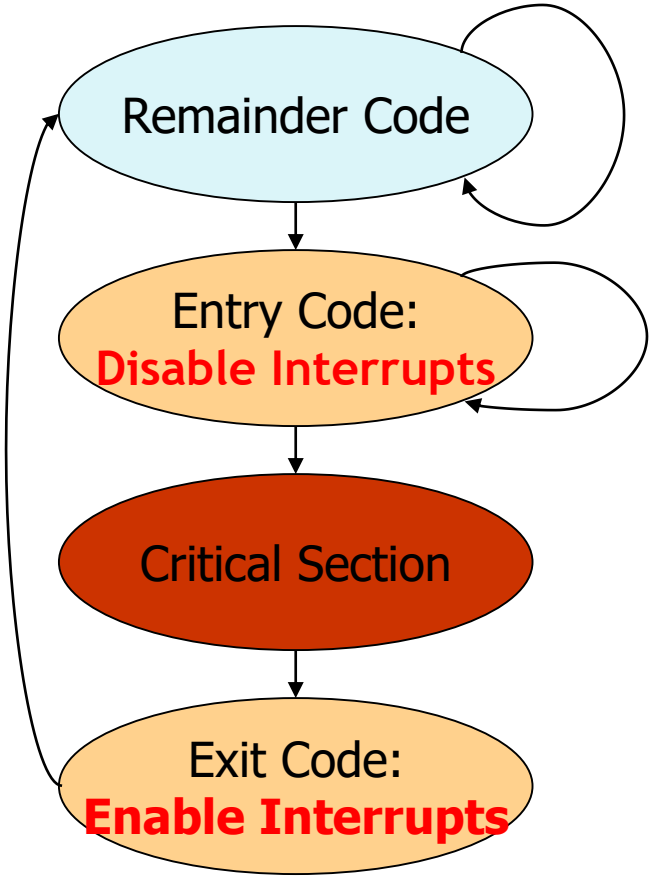
The Problem: Write entry and exit code. We will examine the code against the following properties (the goal is to ensure all of them)

1. **Mutual Exclusion:** No two processes are in their critical section at the same time.
2. **Progress:** If a process in trying to enter the critical section then **some** process must eventually enter the critical section.
3. **Starvation freedom:** If a process in trying to enter the critical section then **this** process must eventually enter the critical section.
4. **Generality:** no assumptions may be made about speeds or the number of participants (threads/processes/cores/CPUs)
5. **No blocking in the remainder** No process running outside its critical section may block other process

First Try: Disabling interrupts

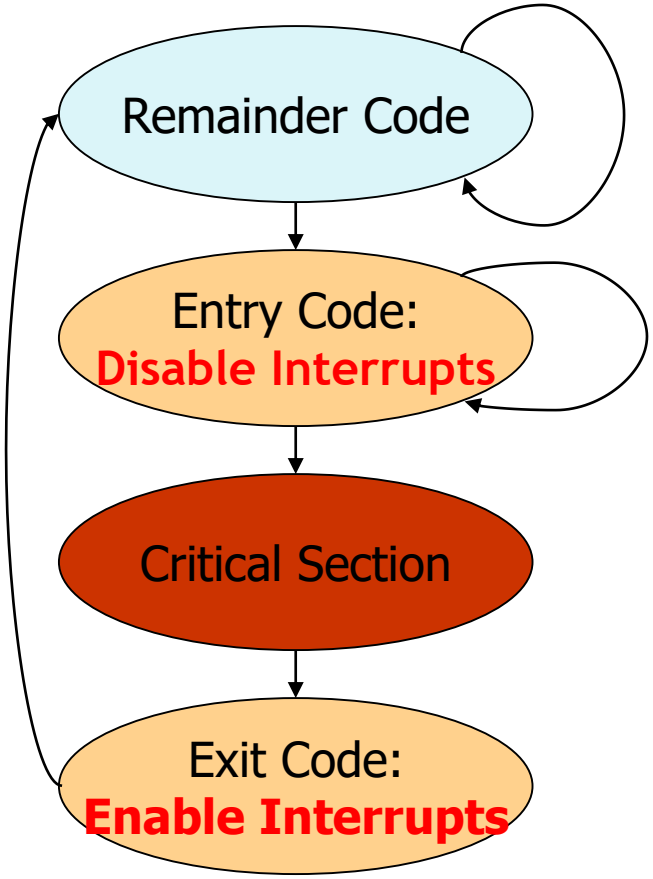


First Try: Disabling interrupts



Satisfies Properties 1,2,3,5 but:

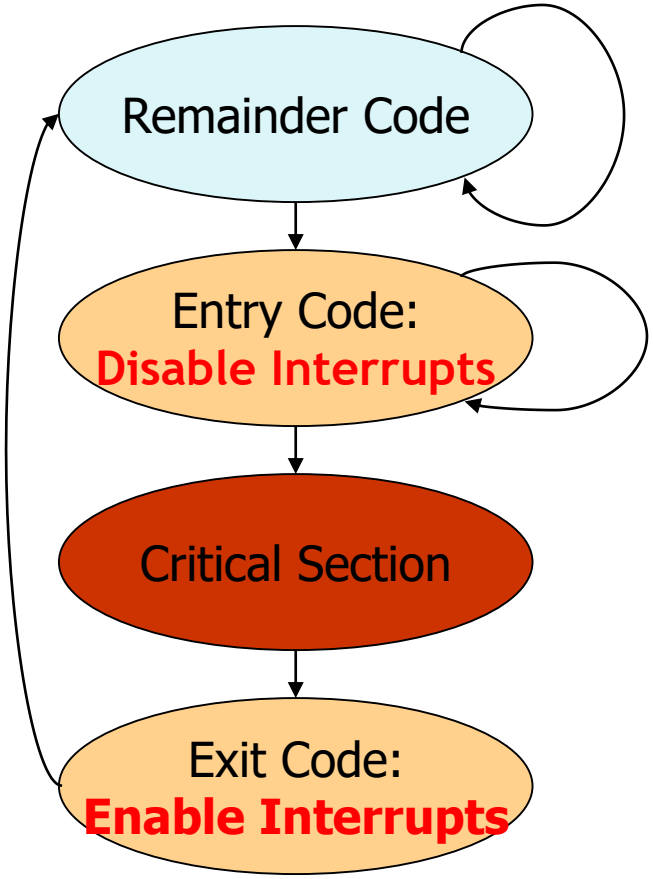
First Try: Disabling interrupts



Satisfies Properties 1,2,3,5 but:

1. Does not solve the problem in a multi-processor system (where real parallelism exists)→ **Property 4 is violated**

First Try: Disabling interrupts



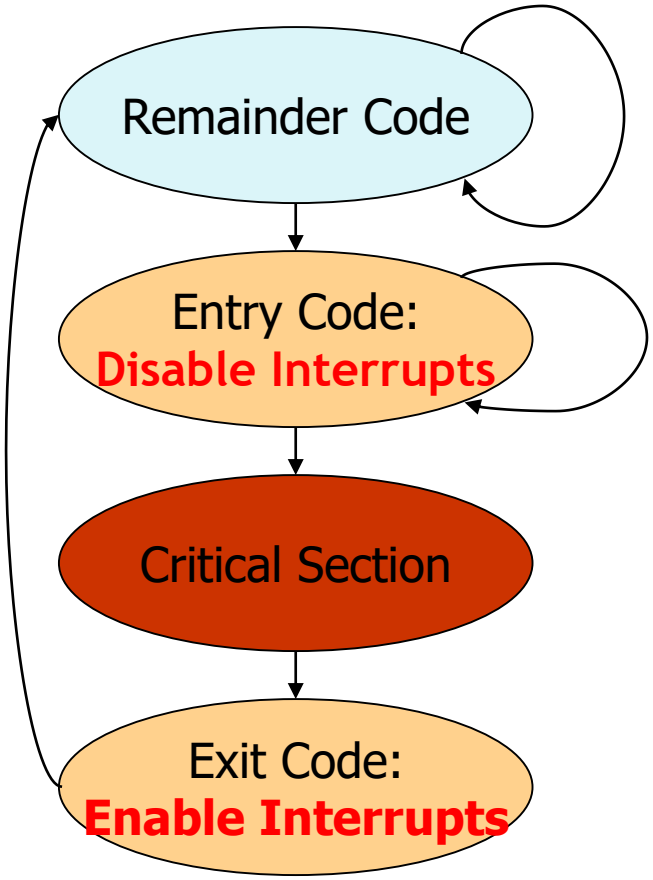
Satisfies Properties 1,2,3,5 but:

1. Does not solve the problem in a multi-processor system (where real parallelism exists)→ **Property 4 is violated**

And:

2. User processes should not be allowed to disable interrupts

First Try: Disabling interrupts



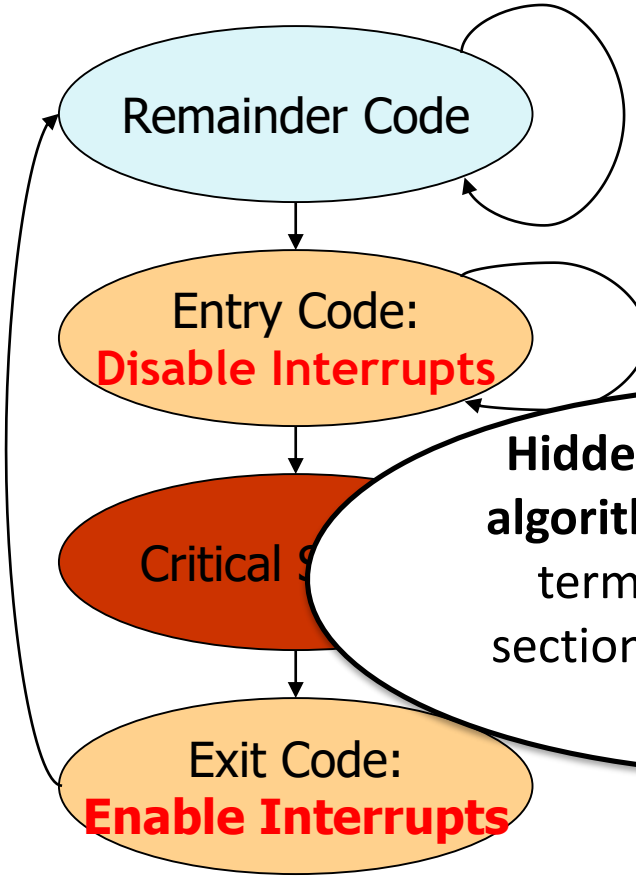
Satisfies Properties 1,2,3,5 but:

1. Does not solve the problem in a multi-processor system (where real parallelism exists)→ **Property 4 is violated**

And:

2. User processes should not be allowed to disable interrupts
3. Disabling interrupts should be done for a very short period of time

First Try: Disabling interrupts



Satisfies Properties 1,2,3,5 but:

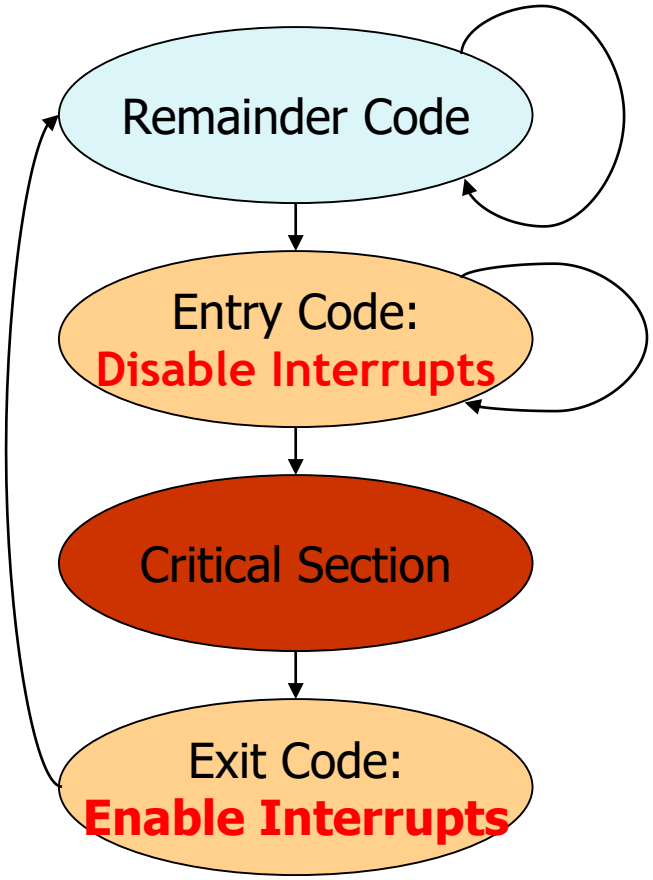
1. Does not s ve the problem in a multi-p cessor system (where rea allelism exists)→ **P ty 4 is violated**

Hidden assumption for all algorithms: Processes never terminate in the critical section and finish it in finite time

s should not be disable interrupts

3. Disabling interrupts should be done for a very short period of time

First Try: Disabling interrupts



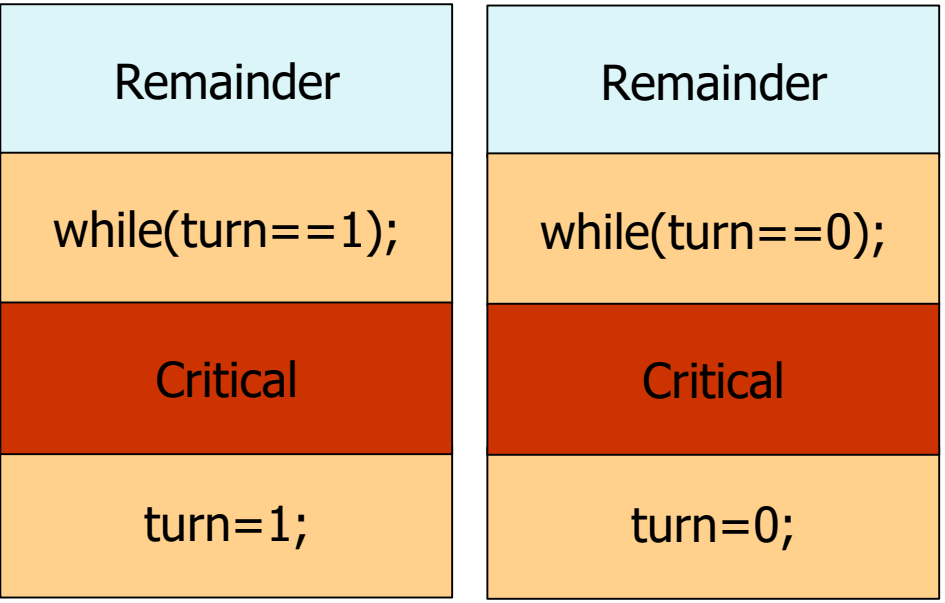
Satisfies Properties 1,2,3,5 but:

1. Does not solve the problem in a multi-processor system (where real parallelism exists)→ **Property 4 is violated**

And:

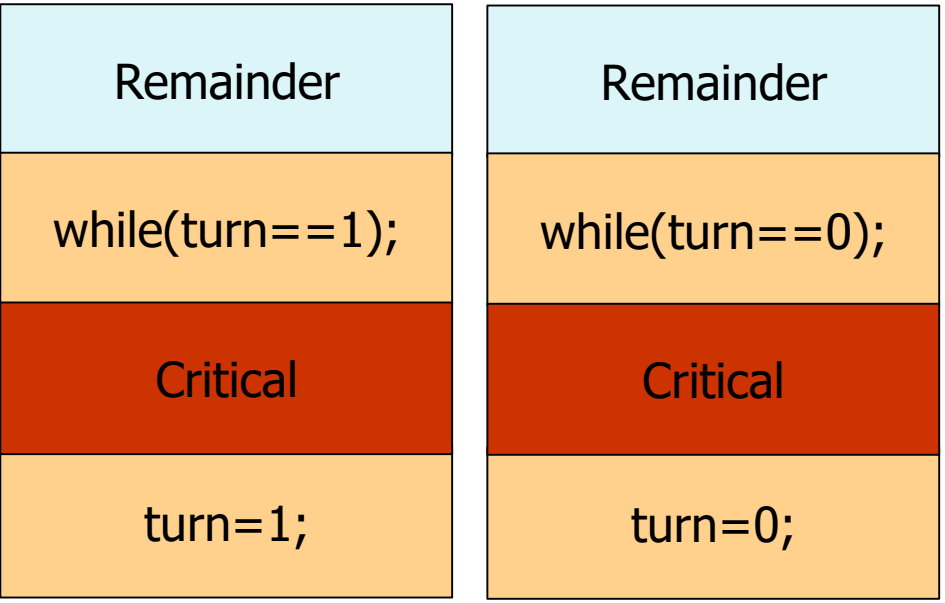
2. User processes should not be allowed to disable interrupts
3. Disabling interrupts should be done for a very short period of time

No “Cheating” – First Try



- Algorithm for 2 threads/processes
- Uses global variable (1 bit) named **turn.** Cannot be used in the remainder/critical
- (Slightly) different code for each thread

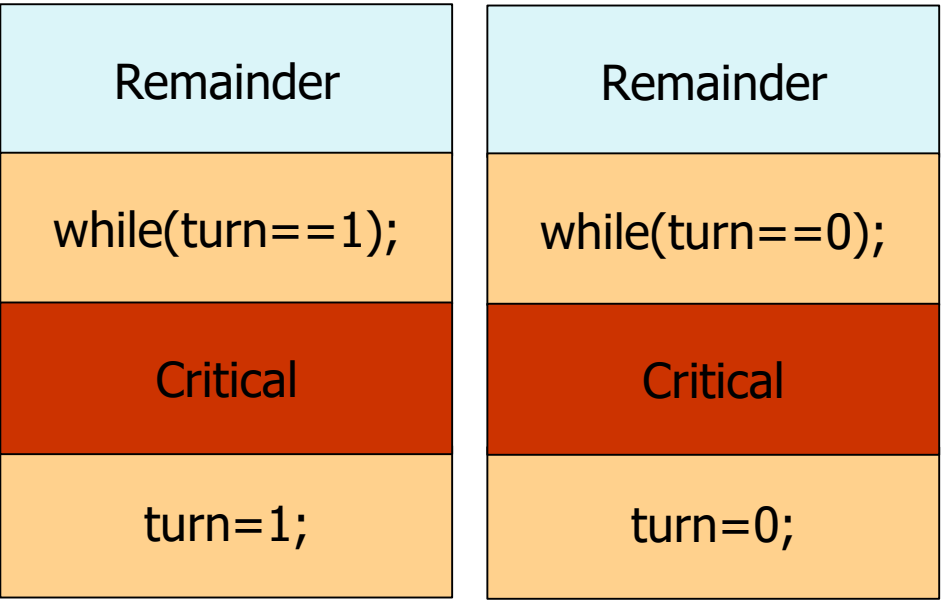
No “Cheating” – First Try



Code for Thread 0 Code for Thread 1

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical condition, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, turn≠1→turn=0. The only place turn can be changed to 1 is when thread 0 exits the critical section, thus turn=0 throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – First Try

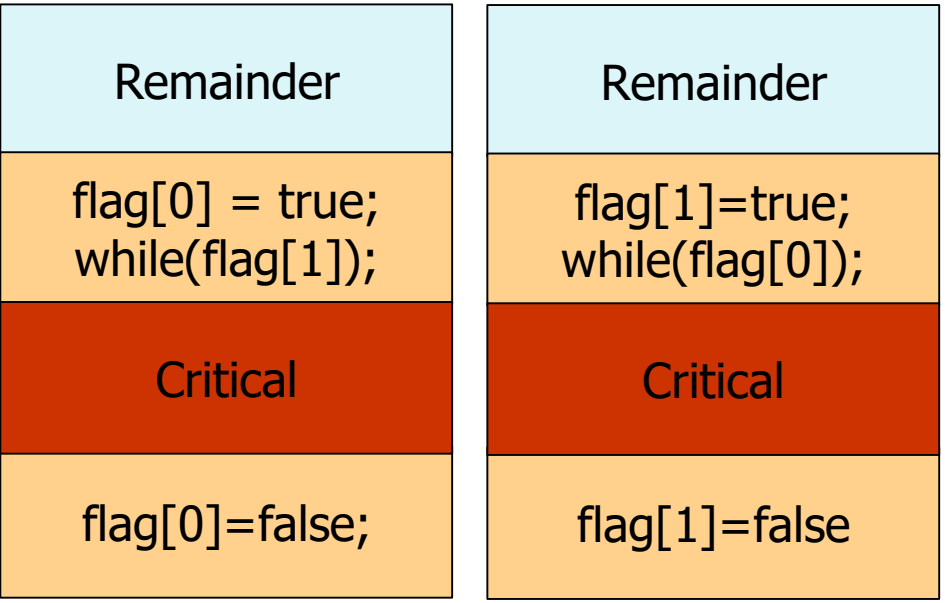


Code for Thread 0 Code for Thread 1

Bad News:

Progress / no blocking in the remainder does not hold. Thread 1 can simply stay in its remainder and never get to its entry (and therefore exit) code → **Starvation Freedom** does not hold

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical condition, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, turn≠1→turn=0. The only place turn can be changed to 1 is when thread 0 exits the critical section, thus turn=0 throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.



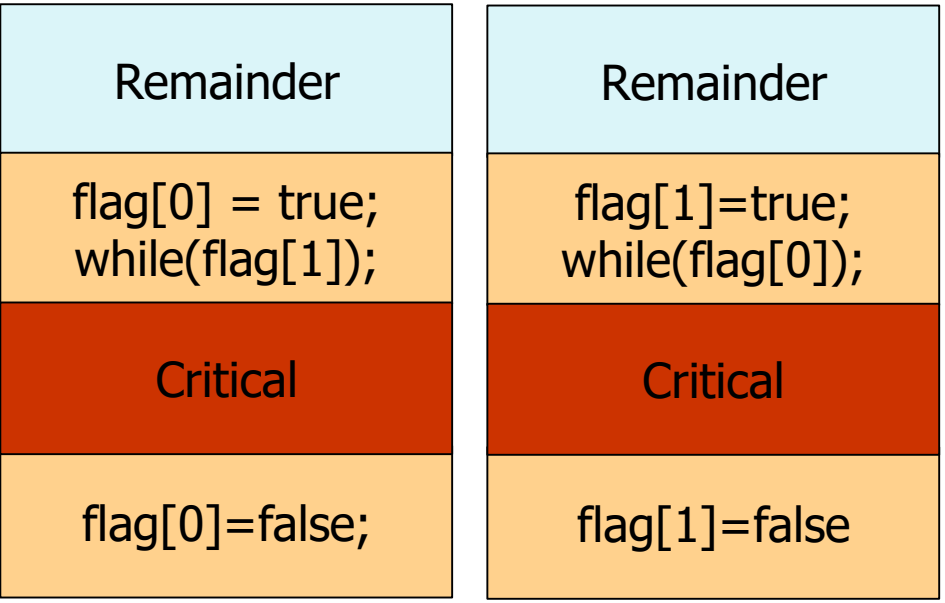
Code for Thread 0 Code for Thread 1

Algorithm for 2 threads/processes

Uses global two global variables (1 bit), named **flag[0], flag[1].** Cannot be used in the remainder/critical section

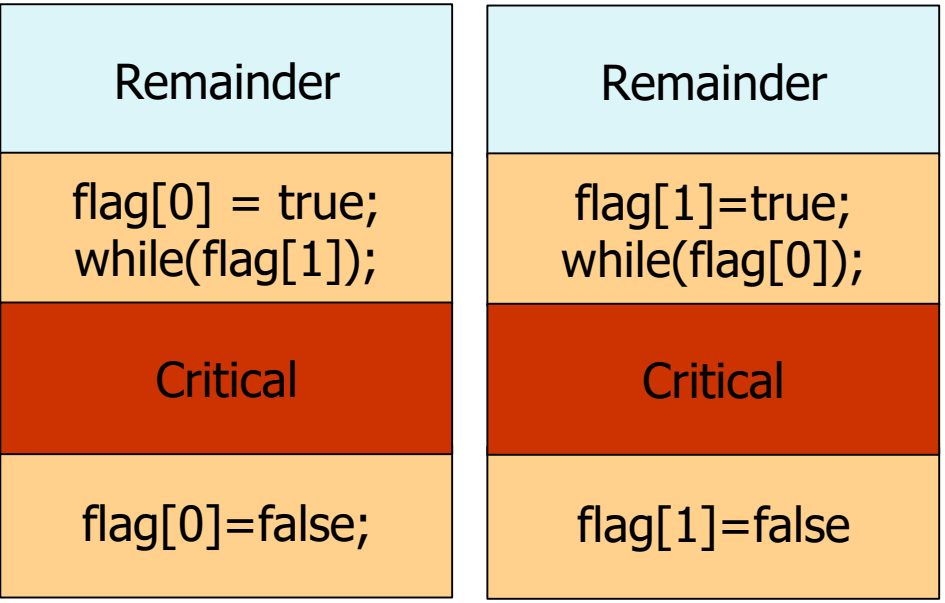
(Slightly) different code for each thread

No “Cheating” – Second Try



Code for Thread 0 Code for Thread 1

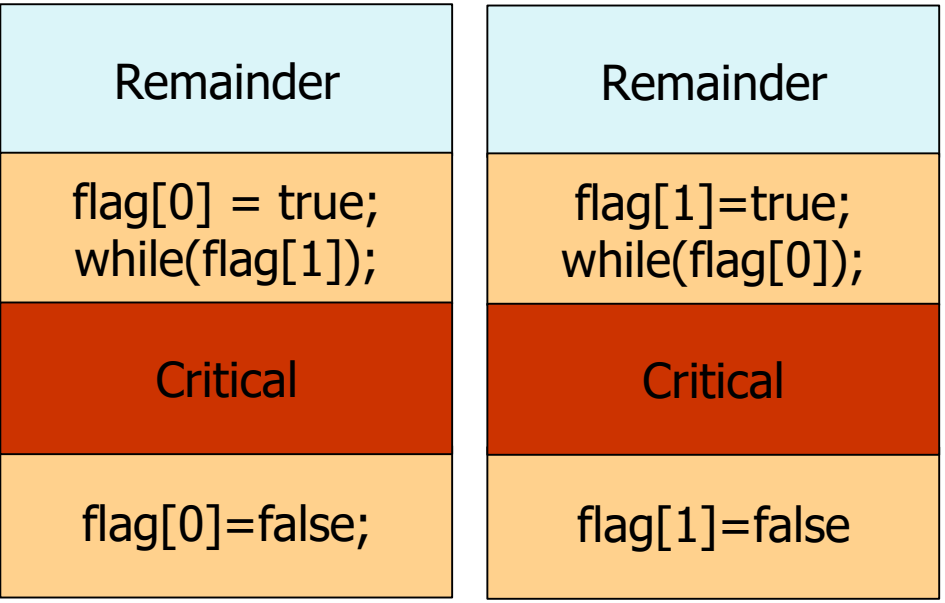
No "Cheating" – Second Try



Code for Thread 0 Code for Thread 1

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try

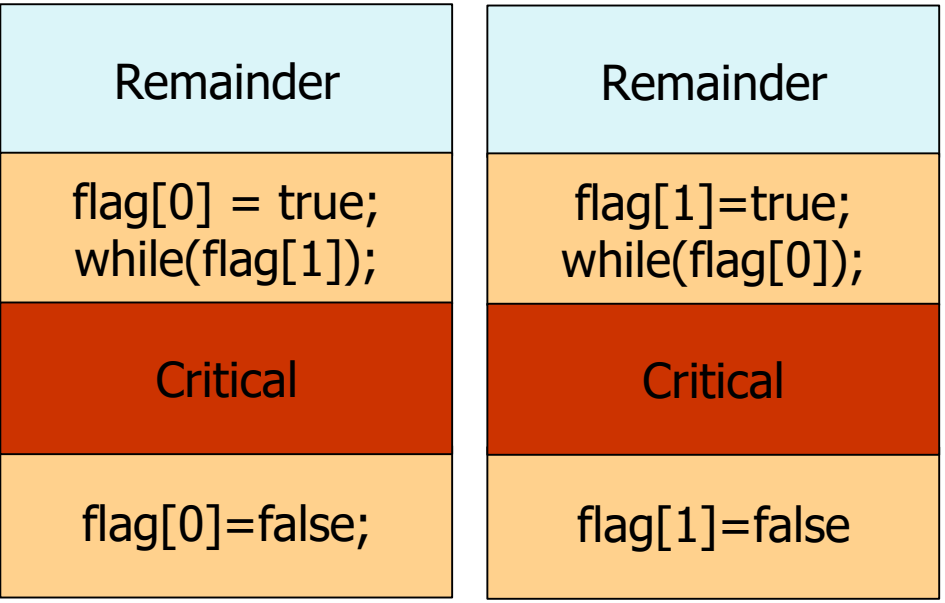


Code for Thread 0 Code for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try



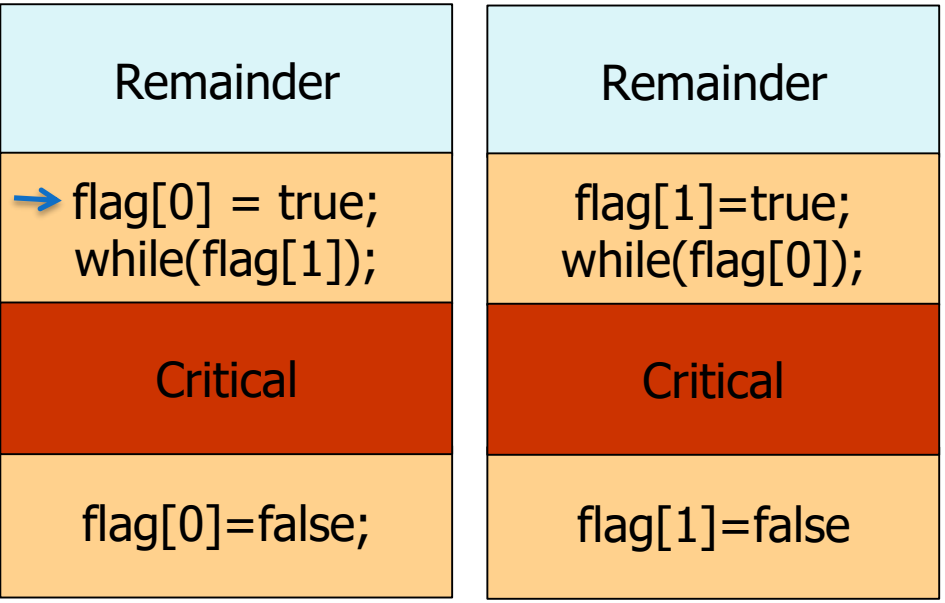
Code for Thread 0 Code for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

But still **Progress** does not hold → **Starvation Freedom** does not hold

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try



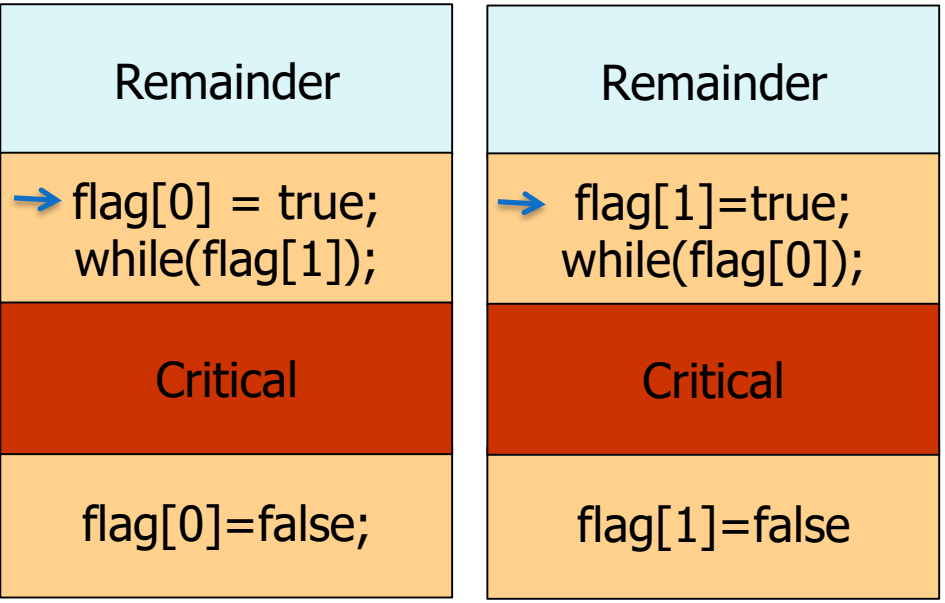
Code for Thread 0 Code for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

But still **Progress** does not hold → **Starvation Freedom** does not hold

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try



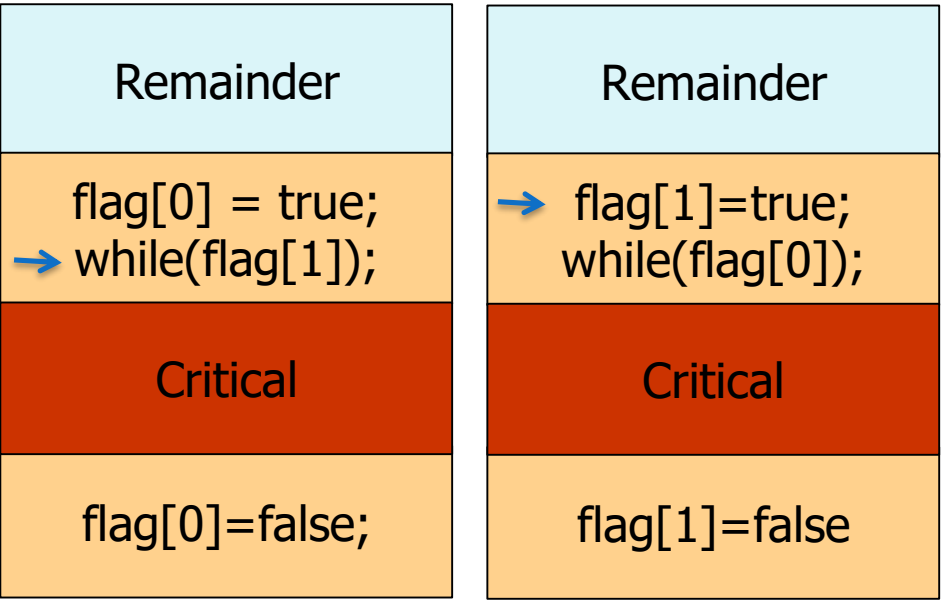
Code for Thread 0 Code for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

But still **Progress** does not hold → **Starvation Freedom** does not hold

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try



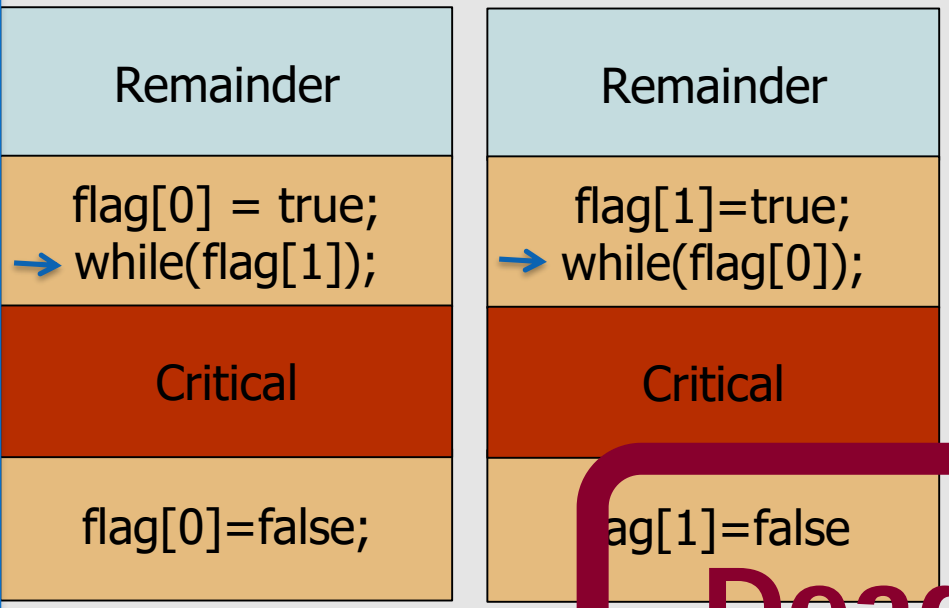
Code for Thread 0 Code for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

But still **Progress** does not hold → **Starvation Freedom** does not hold

√ **Mutual Exclusion:** Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

No "Cheating" – Second Try



Code for Thread 0 Cod for Thread 1

No Blocking in the Remainder: A thread can block another thread only if its flag is true, implying it is either in its entry, critical, or exit section.

But still **Prog ess** does not hold → **Starvation Freedom** does not hold

Deadlock!

√ Mutual Exclusion: Assume that the two of the processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[0]=true** → The only place **flag[0]** can be changed to **false** is when thread 0 exits the critical section, thus **flag[0]=true** throughout the entire execution of the critical section by Thread 0→Thread 1 is in its entry code when Thread 0 is the critical section → Contradiction.

Illustration of Deadlocks

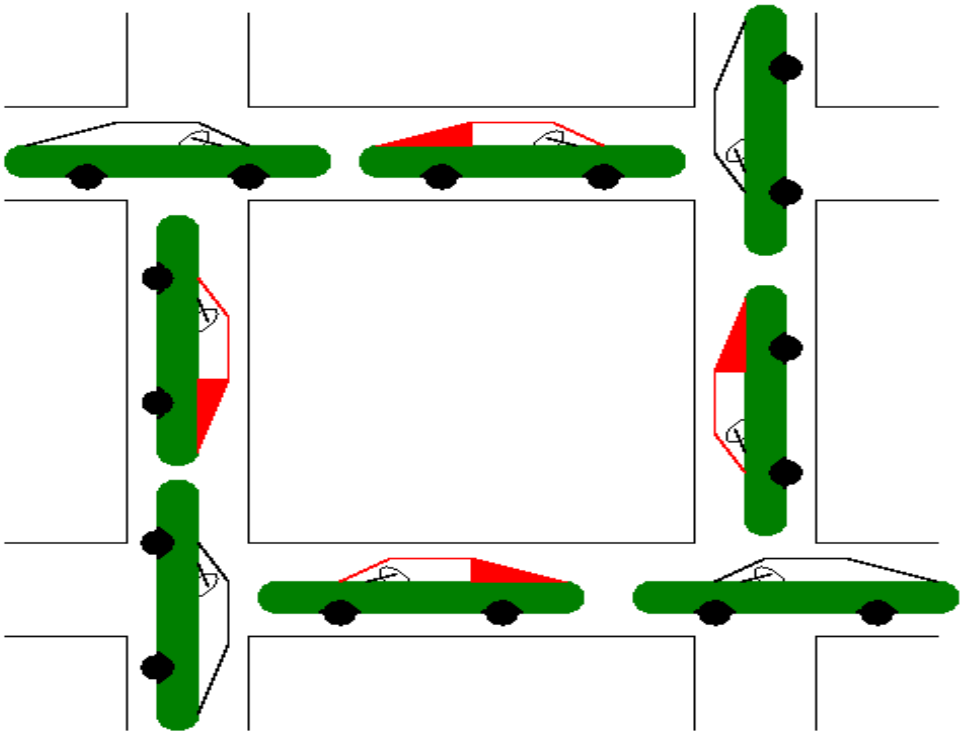


Illustration of Deadlocks

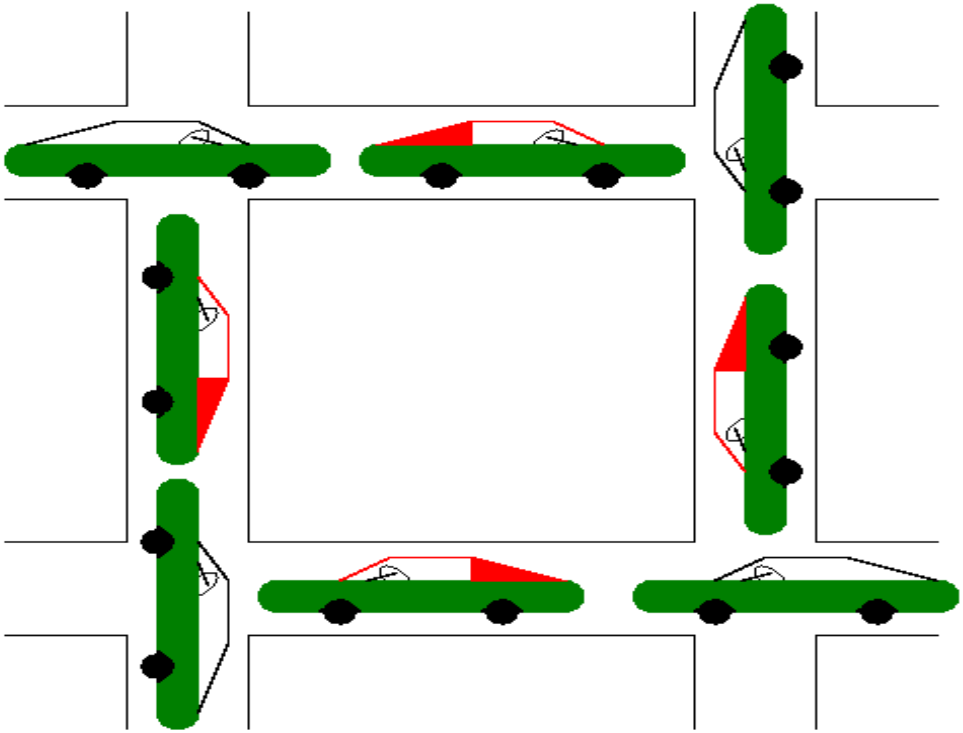
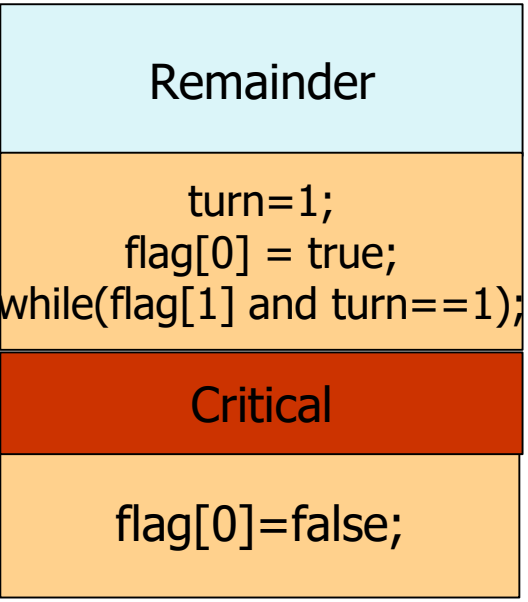


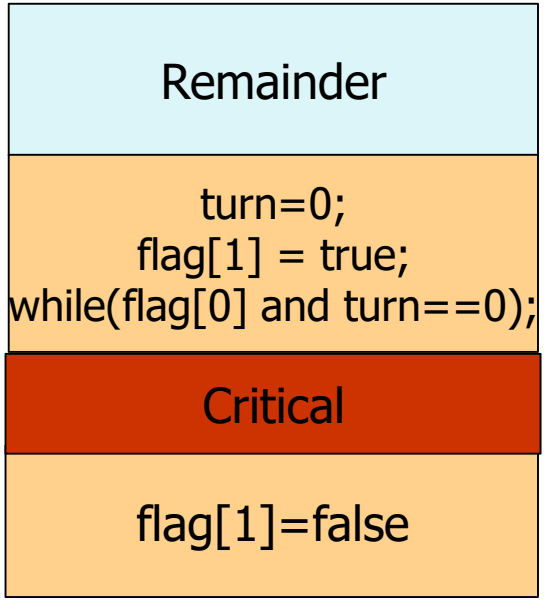
Illustration of Deadlocks

Tel Aviv, Winter 2011

No “Cheating” – Third Try

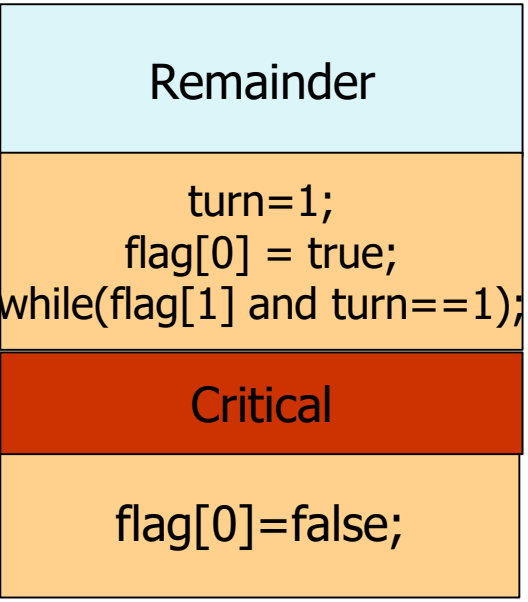


Code for Thread 0

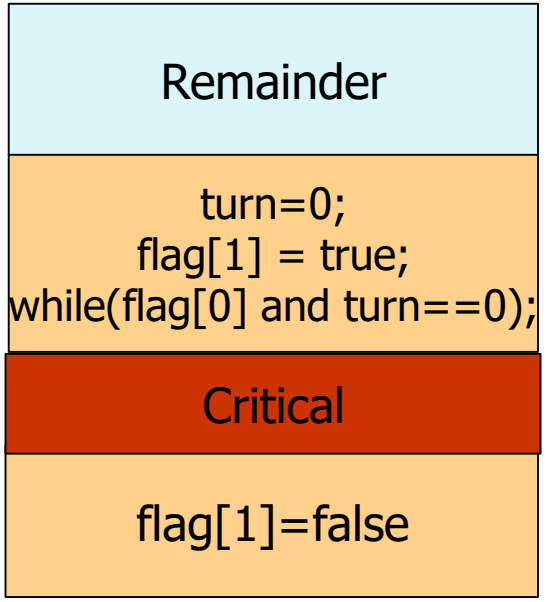


Code for Thread 1

No “Cheating” – Third Try



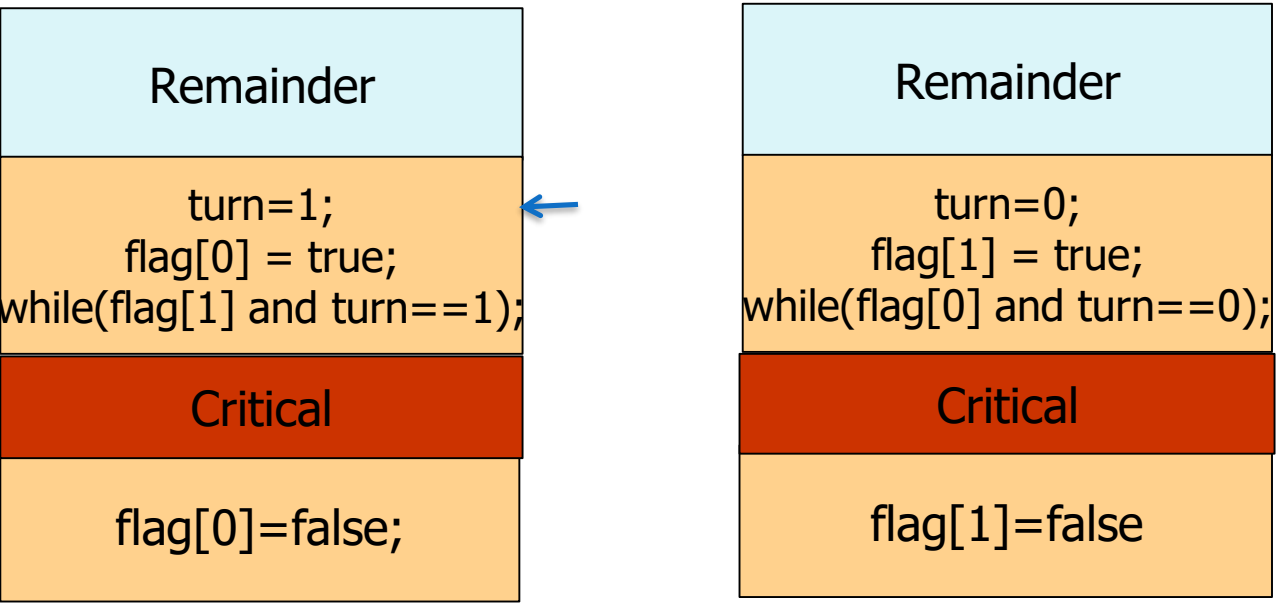
Code for Thread 0



Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

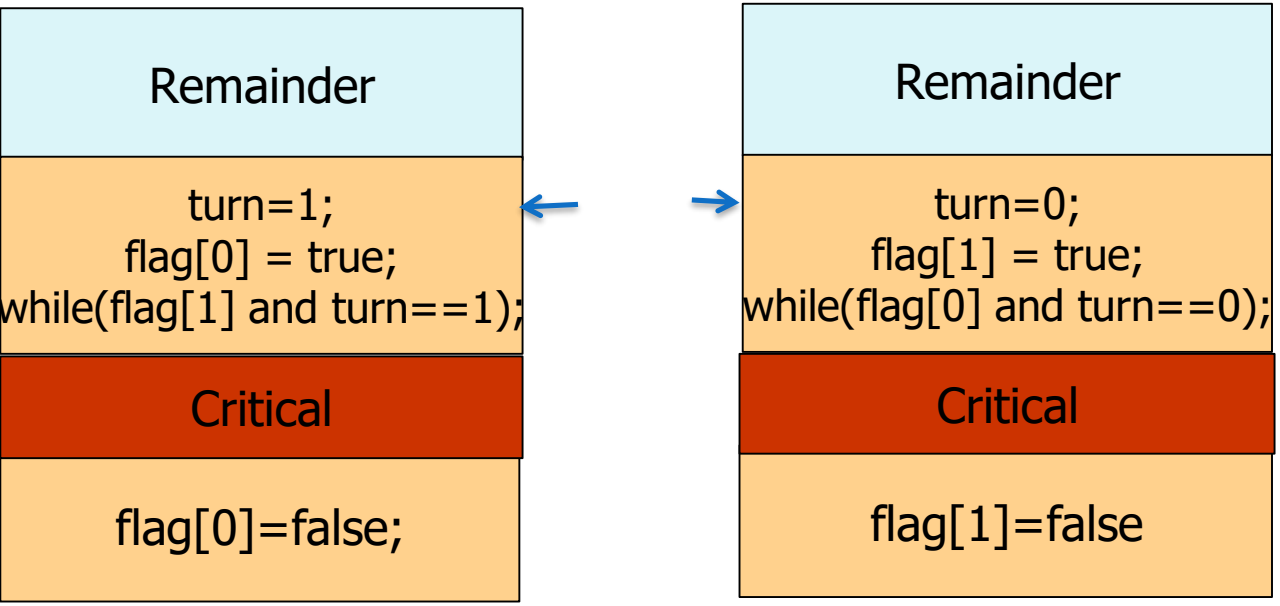


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

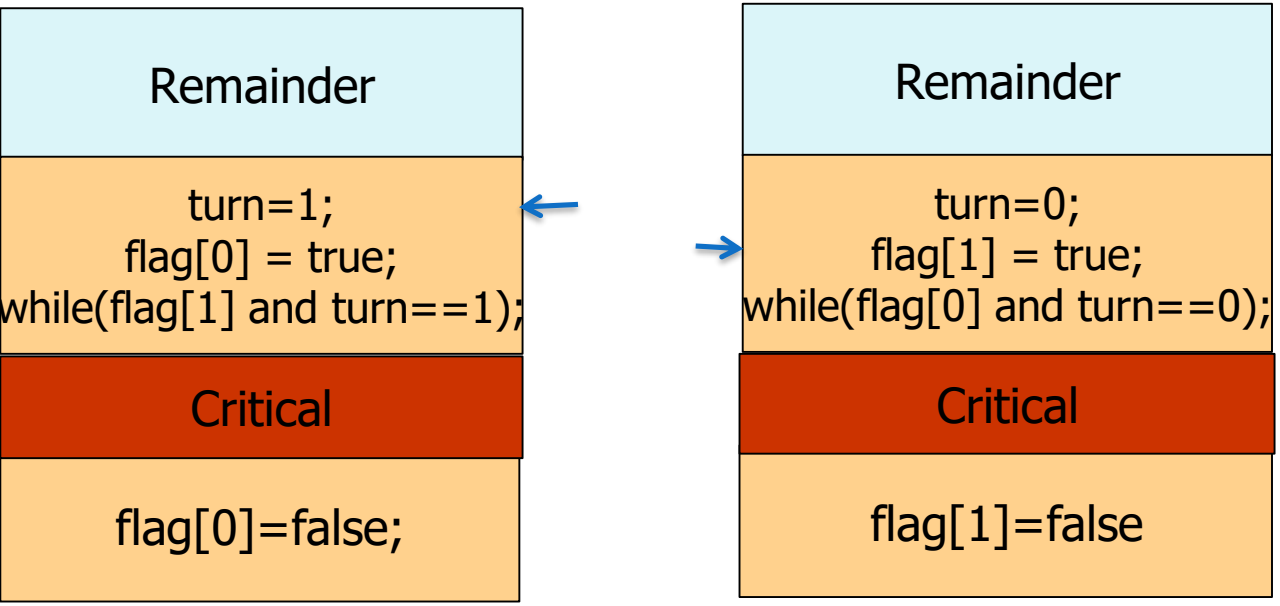


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

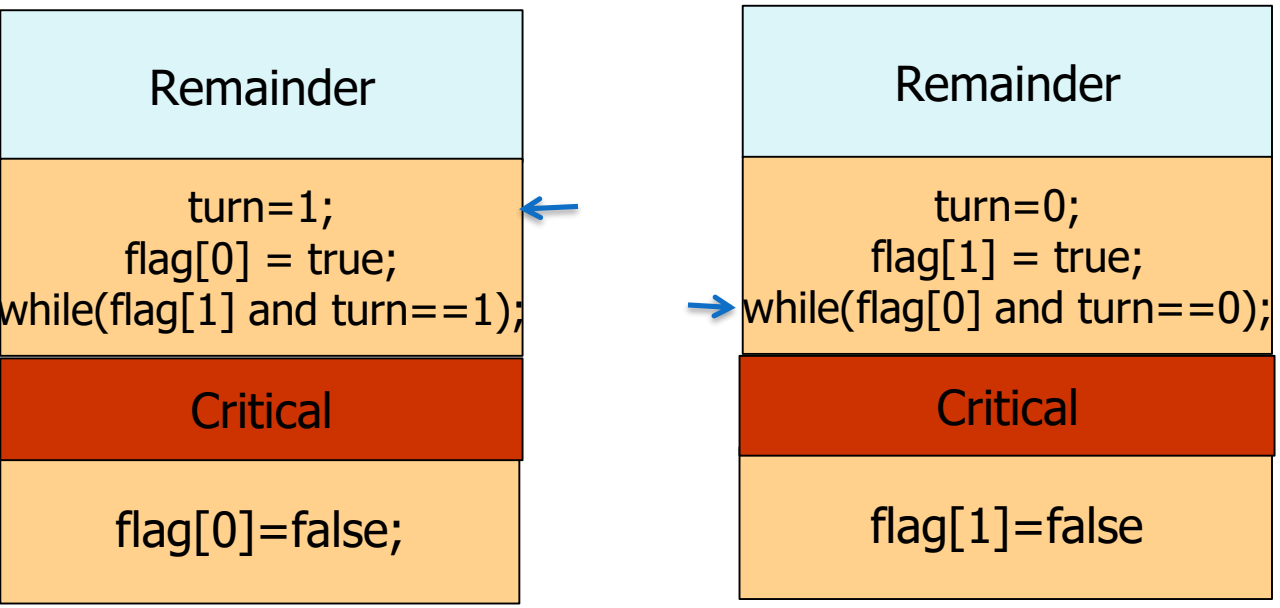


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

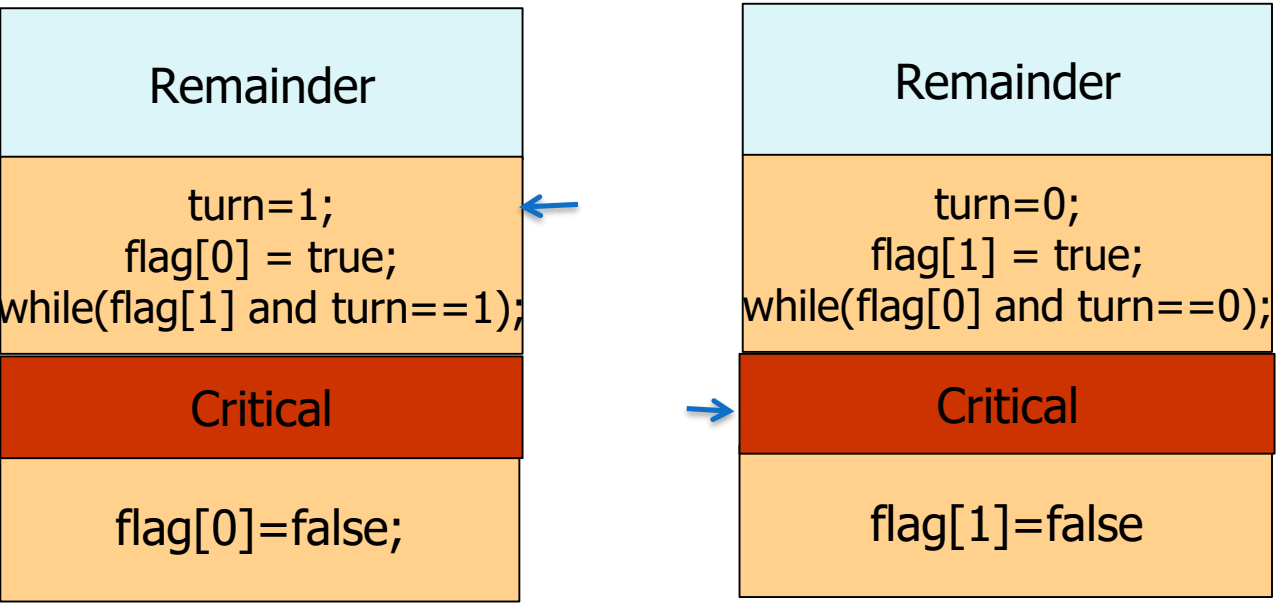


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

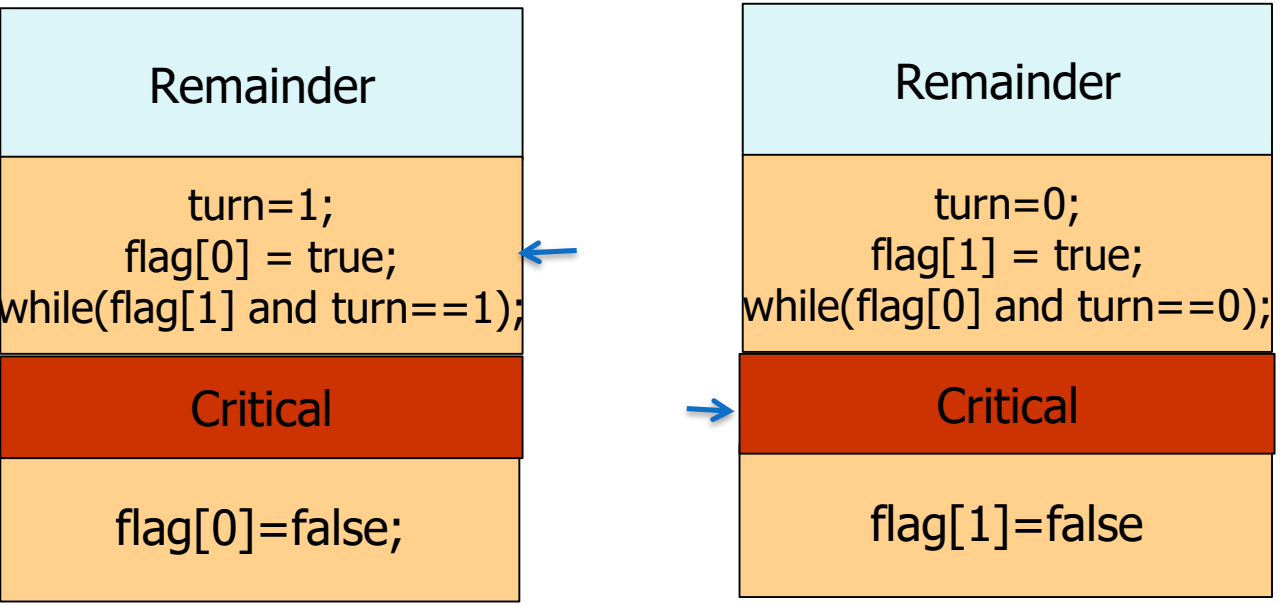


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

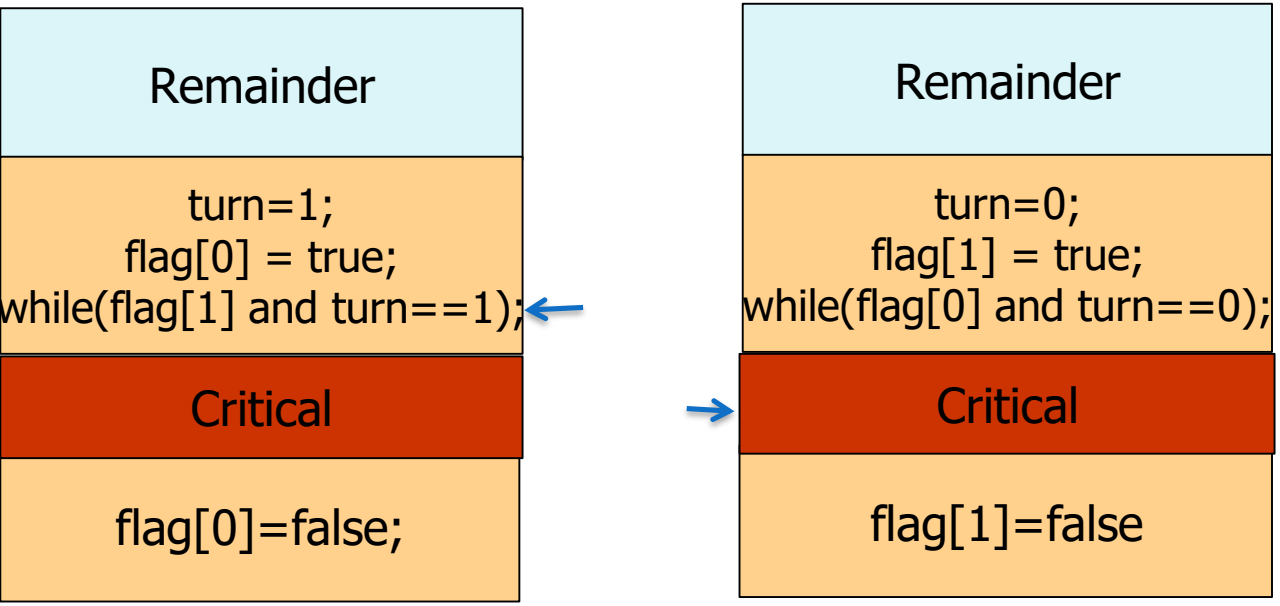


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

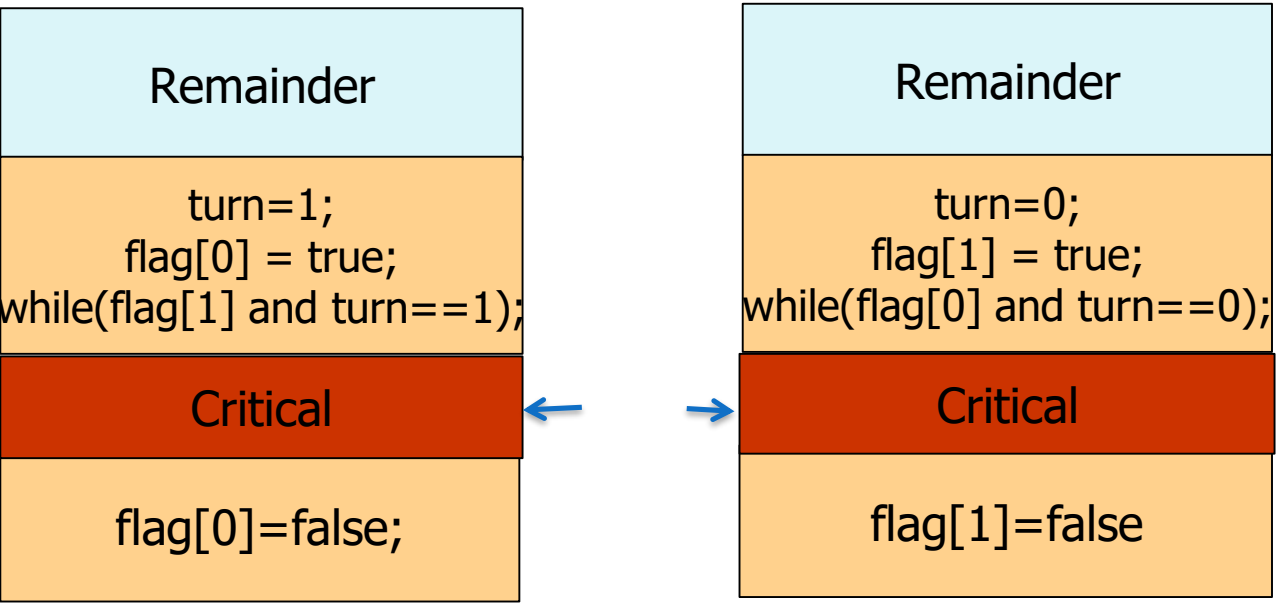


Code for Thread 0

Code for Thread 1

Mutual Exclusion does not hold

No “Cheating” – Third Try

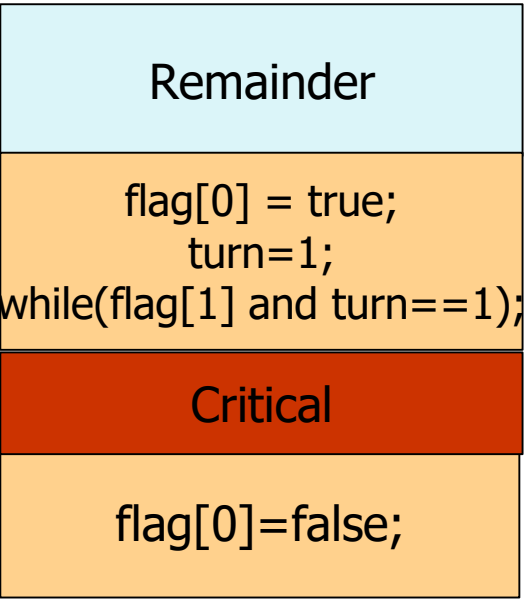


Code for Thread 0

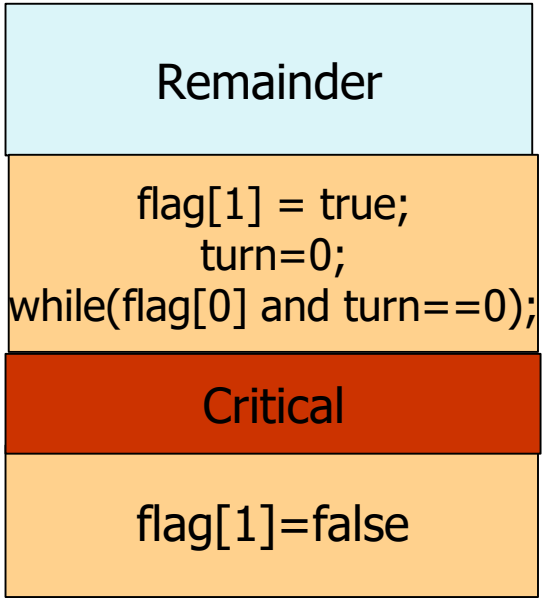
Code for Thread 1

Mutual Exclusion does not hold

Peterson's Algorithm - 1981

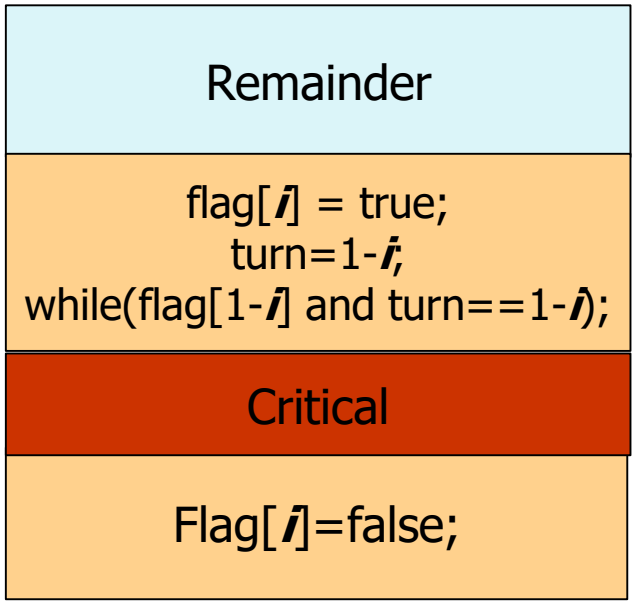


Code for Thread 0



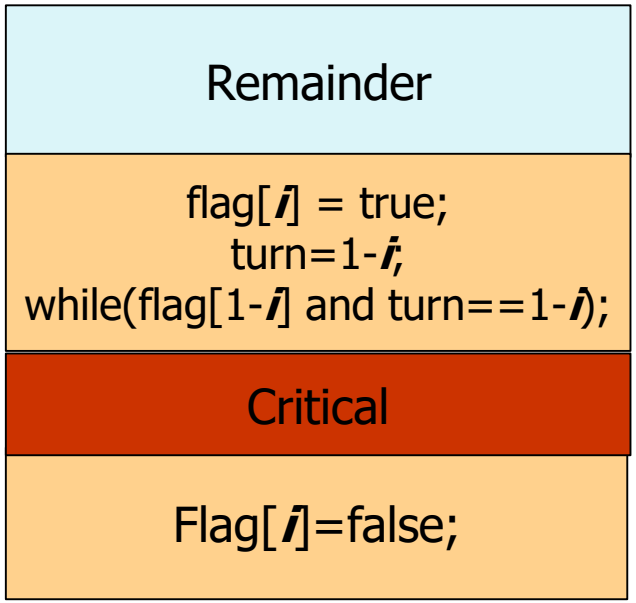
Code for Thread 1

Peterson's Algorithm - 1981



Code for Thread ***i***

Peterson's Algorithm - 1981



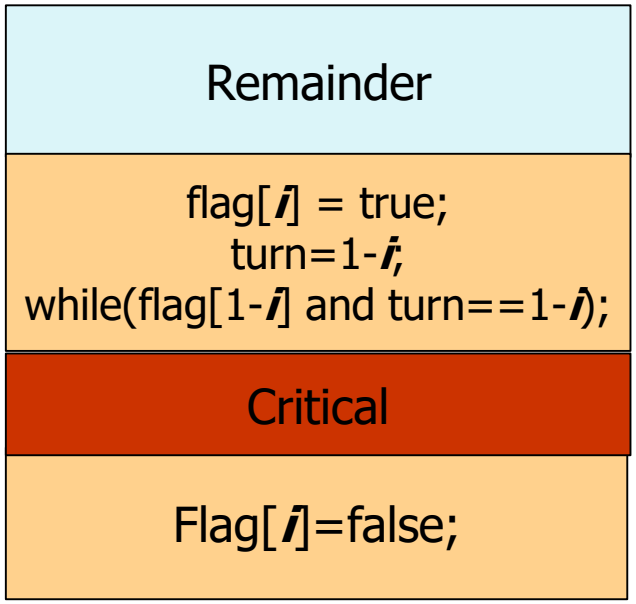
Code for Thread ***i***

√ **Mutual Exclusion:** Assume that the two processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[1]=false or turn=0.**

Case 1: flag[1]=false → Thread 1 has not executed Line 1 of it's entry code → When it will eventually get to Line 3, it will set turn to 0, and wait. The only place **turn** is set to 1, is in Thread's 0 entry code, after it left the critical section → Contradiction.

Case 2: turn=0→Thread 1 executed Line 2 between Thread 0 Line 2 and 3→

Peterson's Algorithm - 1981



Code for Thread ***i***

√ No blocking in the Remainder

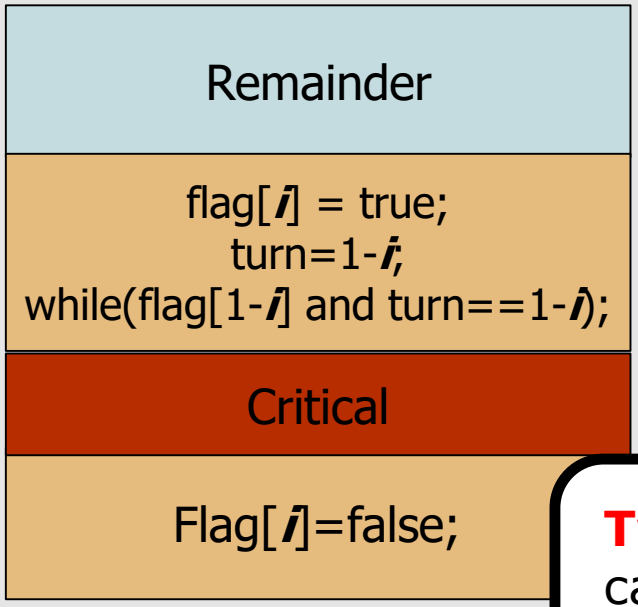
√ Starvation Free: Assume Thread 1 tries to get into the critical section but blocked (on Line 3) for infinite time → Thread 0 enters the critical section infinite number of times → The second time Thread 0 will execute the entry, **flag[0]=true**, **turn=1** and it will block → The next time Thread 1 gets the CPU, the **while** condition does not hold and Thread 1 enters the critical section → Contradiction.

√ Mutual Exclusion: Assume that the two processes are in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[1]=false or turn=0.**

Case 1: flag[1]=false → Thread 1 has not executed Line 1 of it's entry code → When it will eventually get to Line 3, it will set turn to 0, and wait. The only place **turn** is set to 1, is in Thread's 0 entry code, after it left the critical section → Contradiction.

Case 2: turn=0→Thread 1 executed Line 2 between Thread 0 Line 2 and 3→

Peterson's Algorithm - 1981



Code for Thread ***i***

√ No blocking in the Remainder

√ Starvation Free: Assume Thread 1 tries to get into the critical section but blocked (on Line 3) for infinite time → Thread 0 enters the critical section infinite number of times → The second time Thread 0 will execute the entry, **flag[0]=true**, **turn=1** and it will block → The next time Thread 1 gets the CPU, the **while** Thread 1 enters the on.

Two threads only
cannot be easily extended to more
Busy wait (spinning)

√ Mutual Exclusion: A re in the critical section, and then w.l.o.g. thread 0 left its entry section (and entered the critical section) first → At that point, **flag[1]=false or turn=0.**

Case 1: flag[1]=false → Thread 1 has not executed Line 1 of it's entry code → When it will eventually get to Line 3, it will set turn to 0, and wait. The only place **turn** is set to 1, is in Thread's 0 entry code, after it left the critical section → Contradiction.

Case 2: turn=0→Thread 1 executed Line 2 between Thread 0 Line 2 and 3→

Three Concepts We've Covered

- Race Condition: A situation in which the scheduling of processes/threads (the order of who is getting the CPU) changes the final result
  - Different order → different results
  - Typical scenario: Most of the time everything is OK, but sometime not → very hard to debug
- Atomic Instruction: instruction that completes in a single step relative to other threads
  - x=x+1 was not atomic
  - Read (lw) and write (sw) were atomic
  - Sometimes this doesn't hold (wide words that spans more than one memory address)

Three Concepts We've Covered

- Busy-waiting (spinning, busy-looping): technique in which a process repeatedly checks to see if a condition is true
  - "do nothing" loops: while(flag[1-***i***] and turn==1-***i***);

Three Concepts We've Covered

- Busy-waiting (spinning, busy-looping): technique in which a process repeatedly checks to see if a condition is true
  - "do nothing" loops: while(flag[1-***i***] and turn==1-***i***);
  - Usually, should be avoided as it wastes CPU cycles and yields large overhead

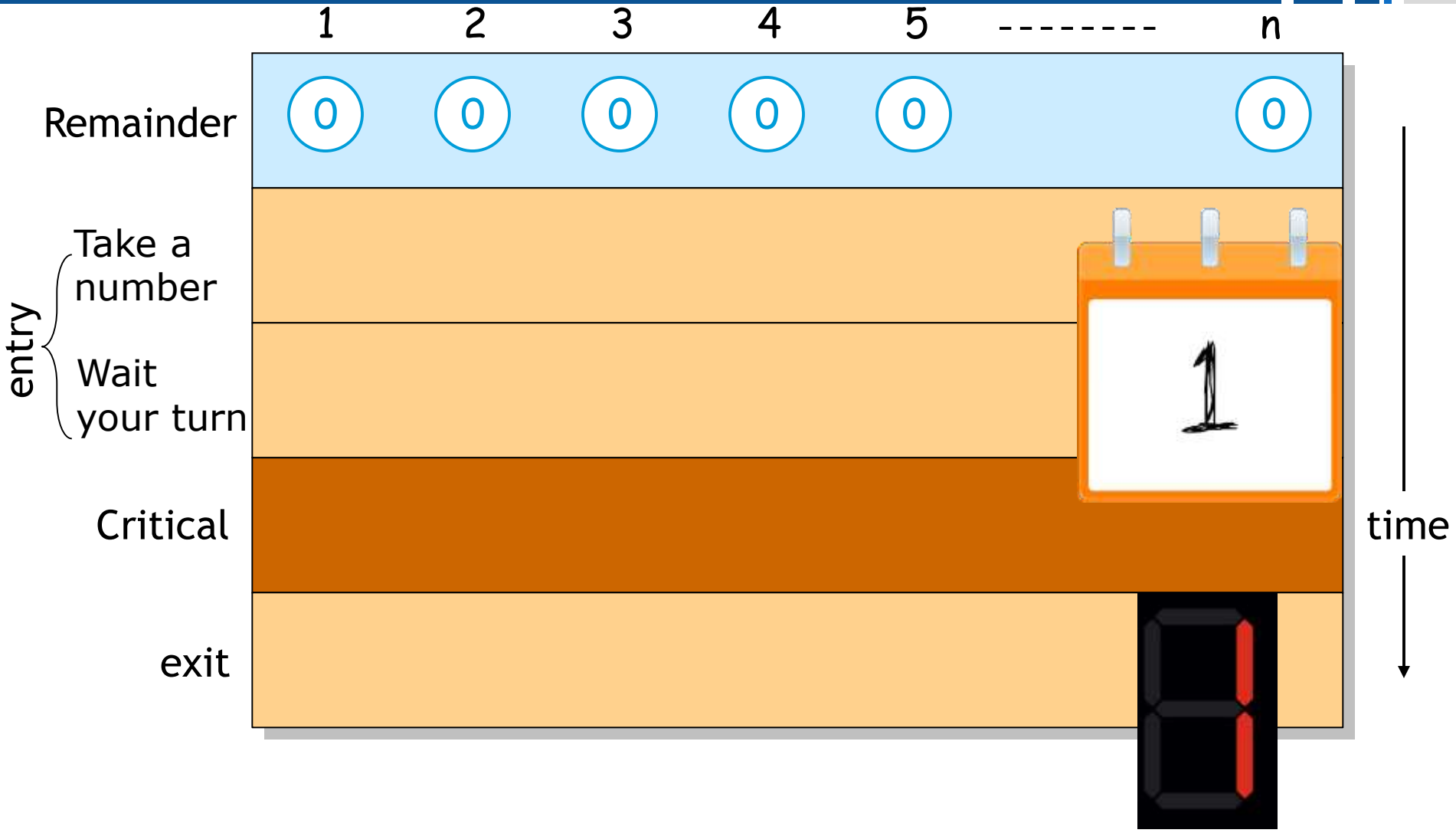
Three Concepts We've Covered

- Busy-waiting (spinning, busy-looping): technique in which a process repeatedly checks to see if a condition is true
  - "do nothing" loops: while(flag[1-***i***] and turn==1-***i***);
  - Usually, should be avoided as it wastes CPU cycles and yields large overhead
  - Polling vs. interrupts

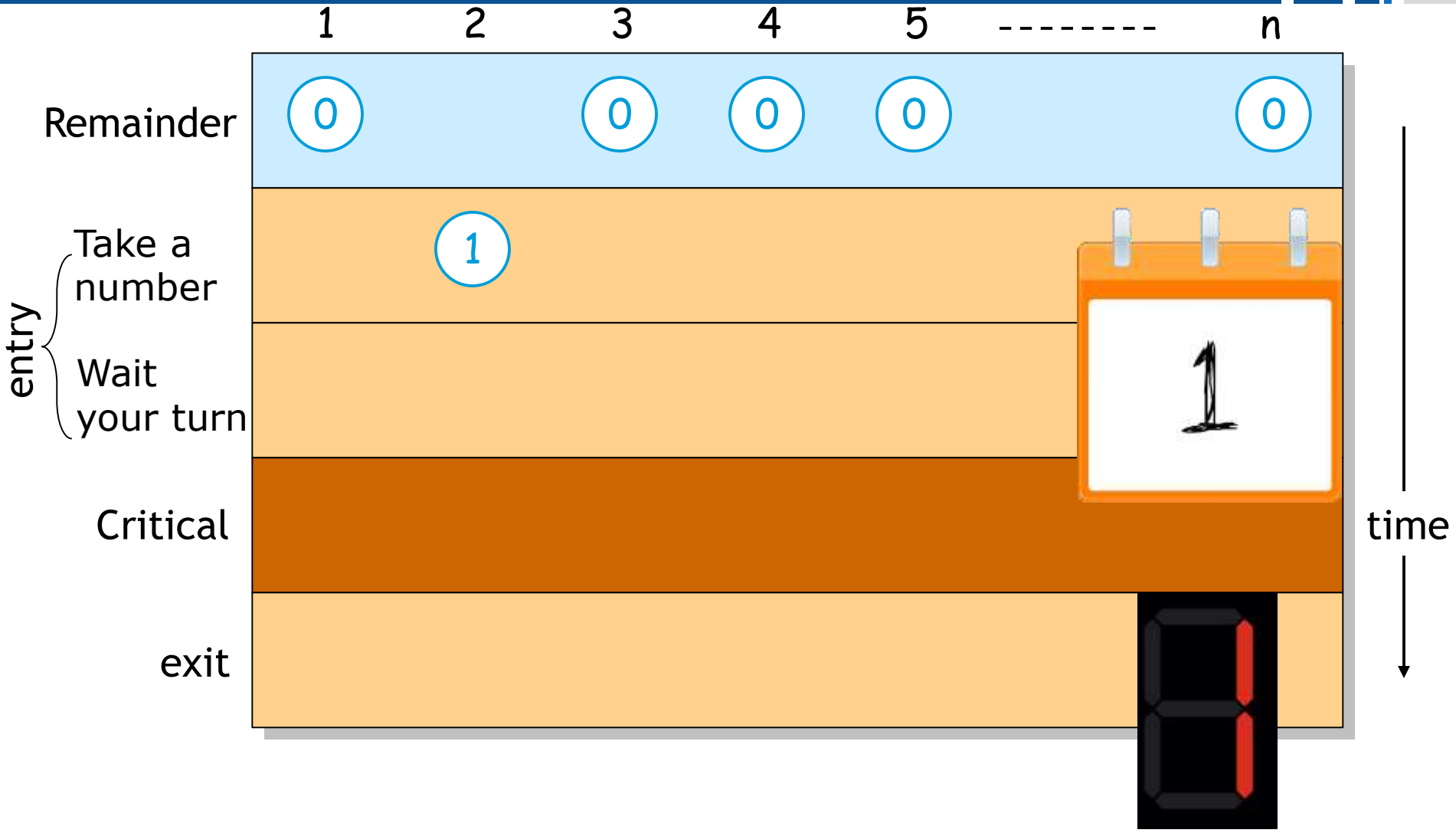
Three Concepts We've Covered

- Busy-waiting (spinning, busy-looping): technique in which a process repeatedly checks to see if a condition is true
  - "do nothing" loops: while(flag[1-***i***] and turn==1-***i***);
  - Usually, should be avoided as it wastes CPU cycles and yields large overhead
  - Polling vs. interrupts
  - We will discuss alternative later

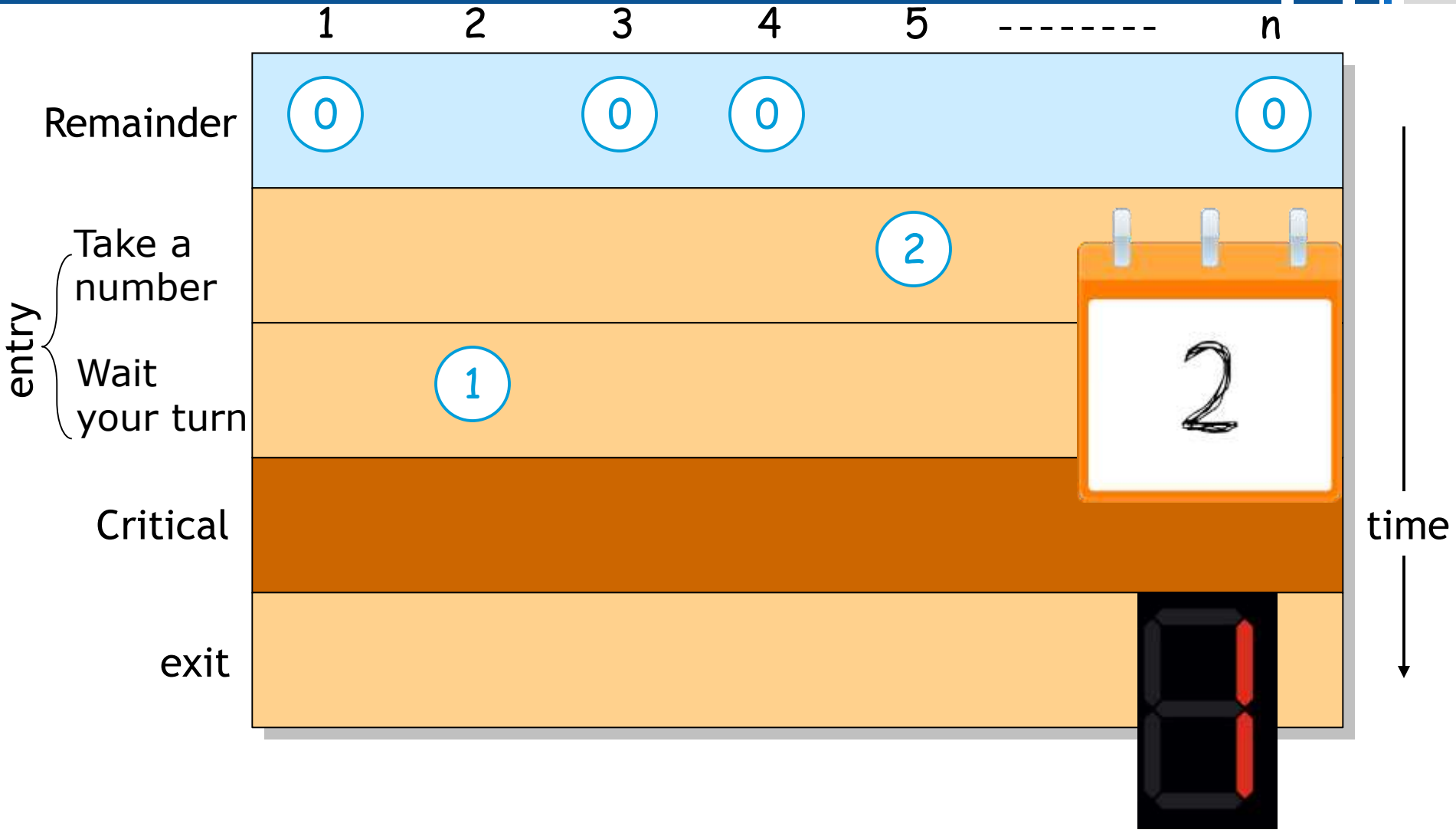
Lamport's Bakery Algorithm - 1974



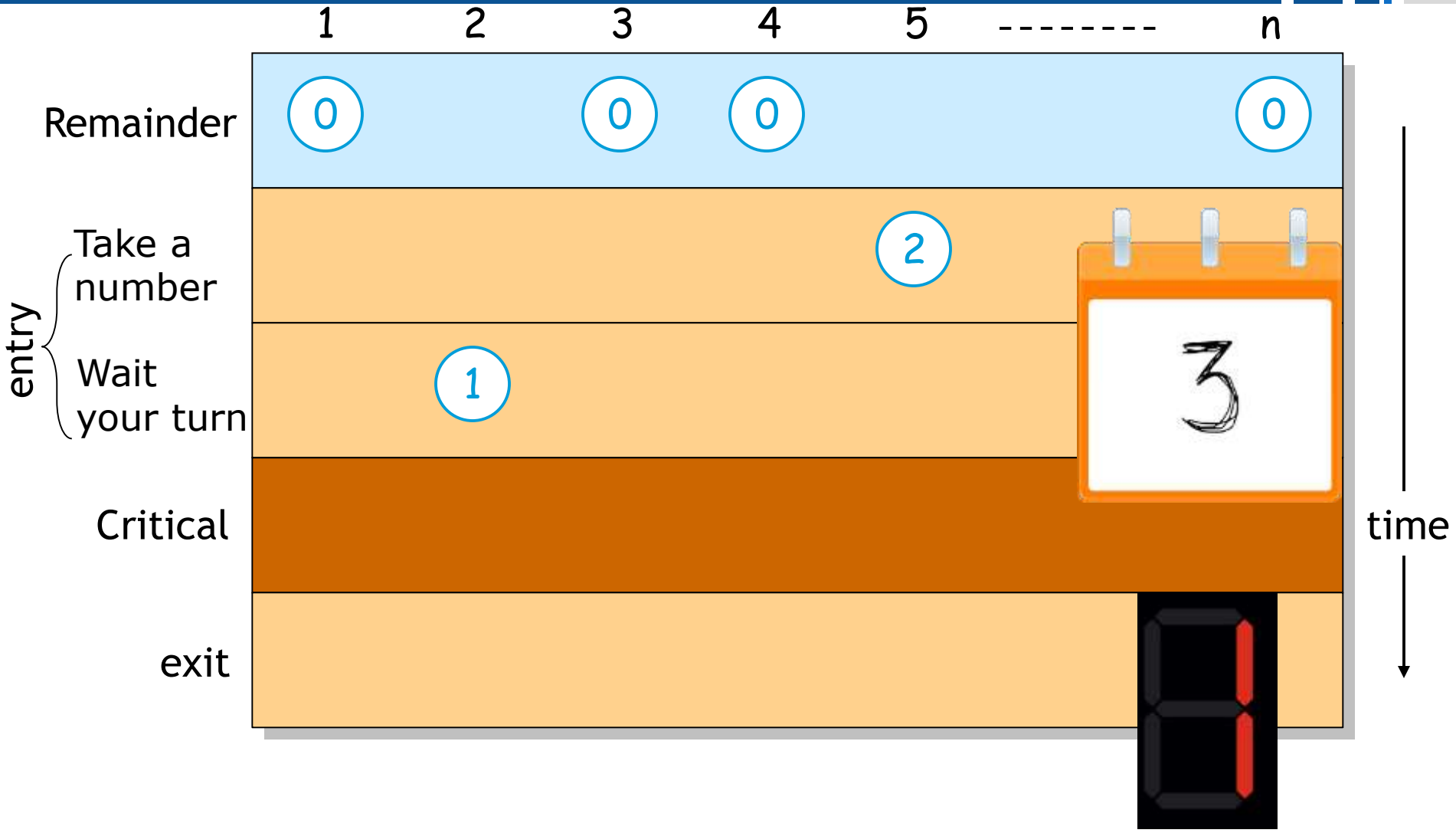
Lamport's Bakery Algorithm - 1974



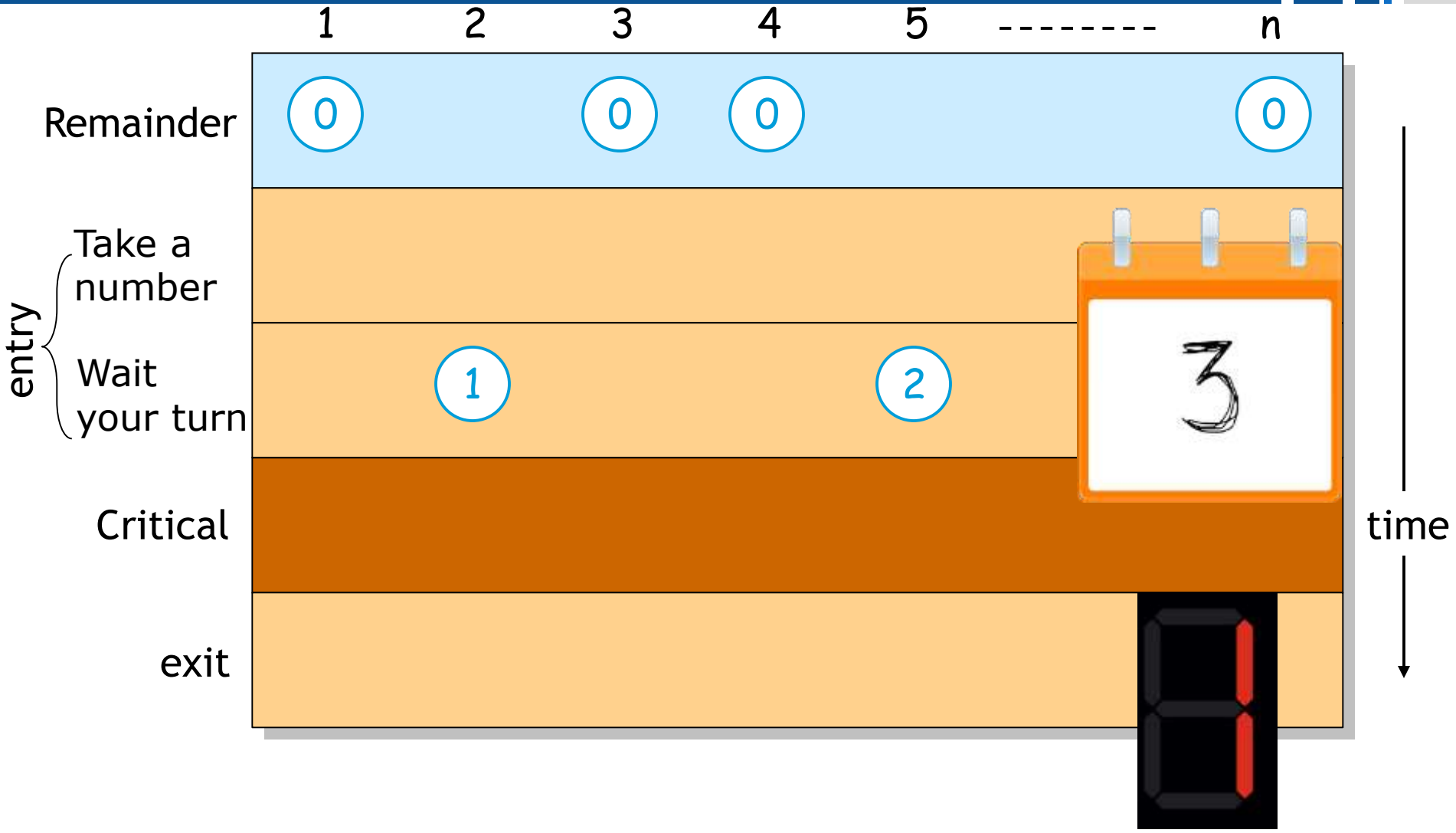
Lamport's Bakery Algorithm - 1974



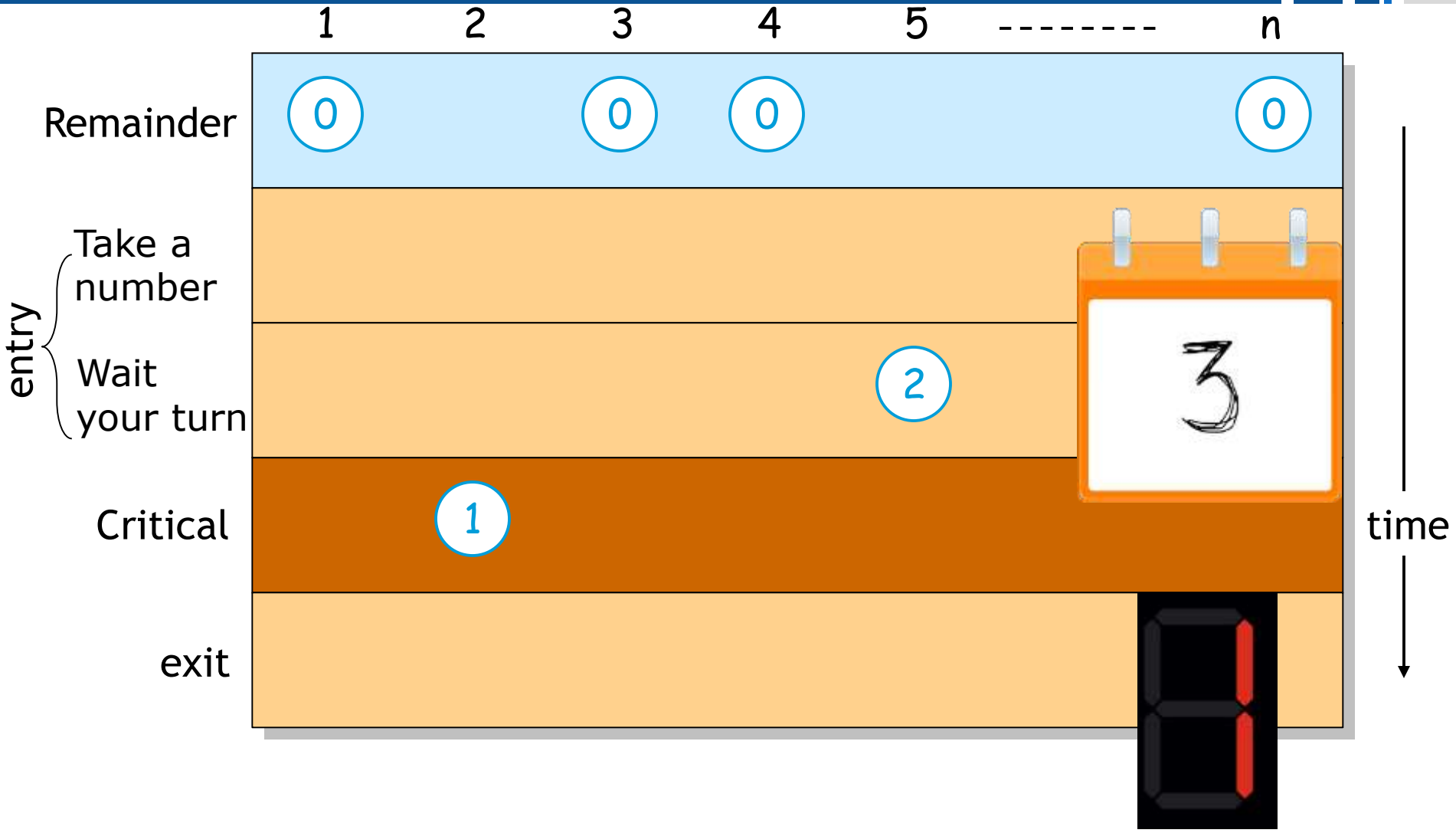
Lamport's Bakery Algorithm - 1974



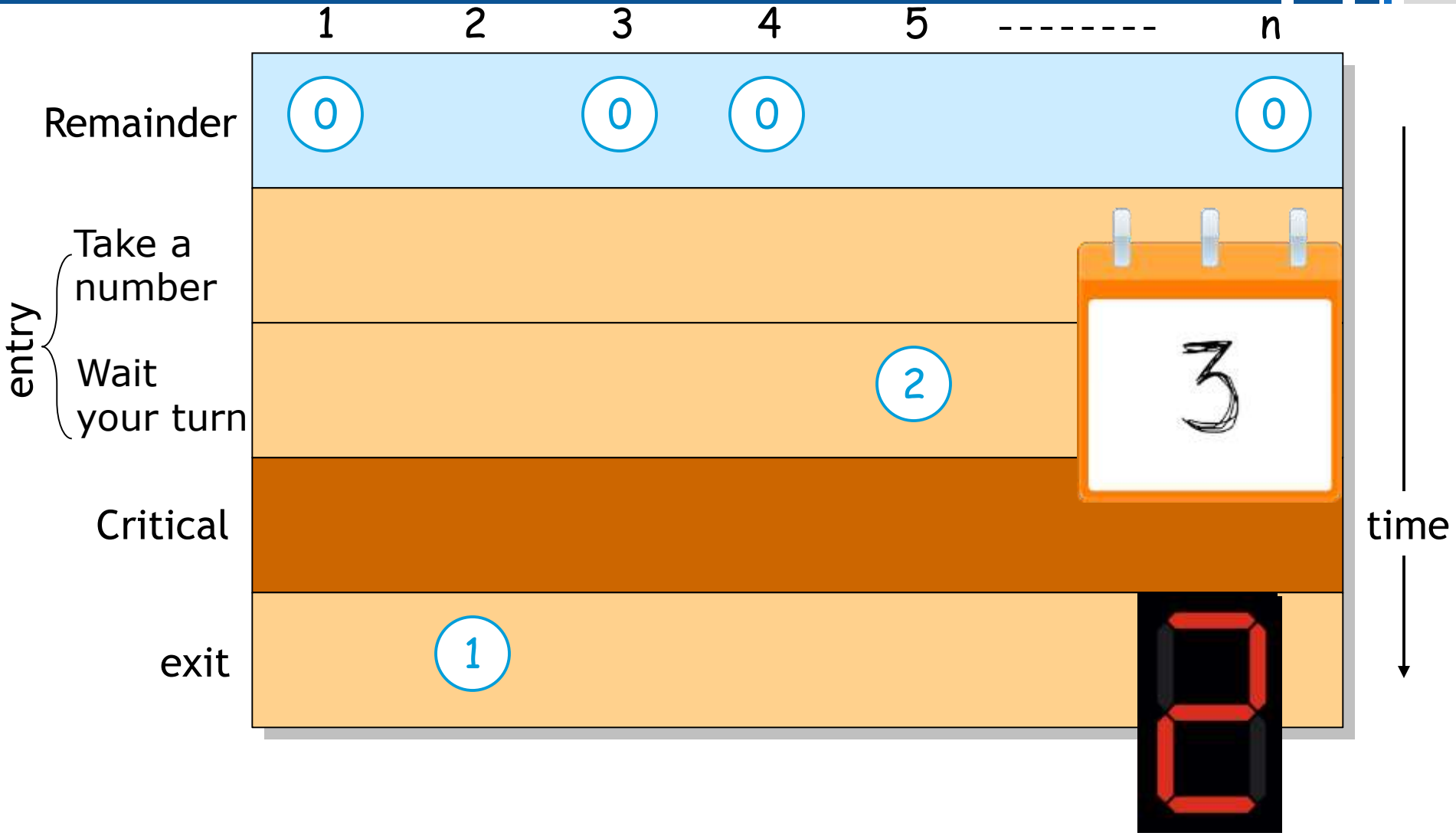
Lamport's Bakery Algorithm - 1974



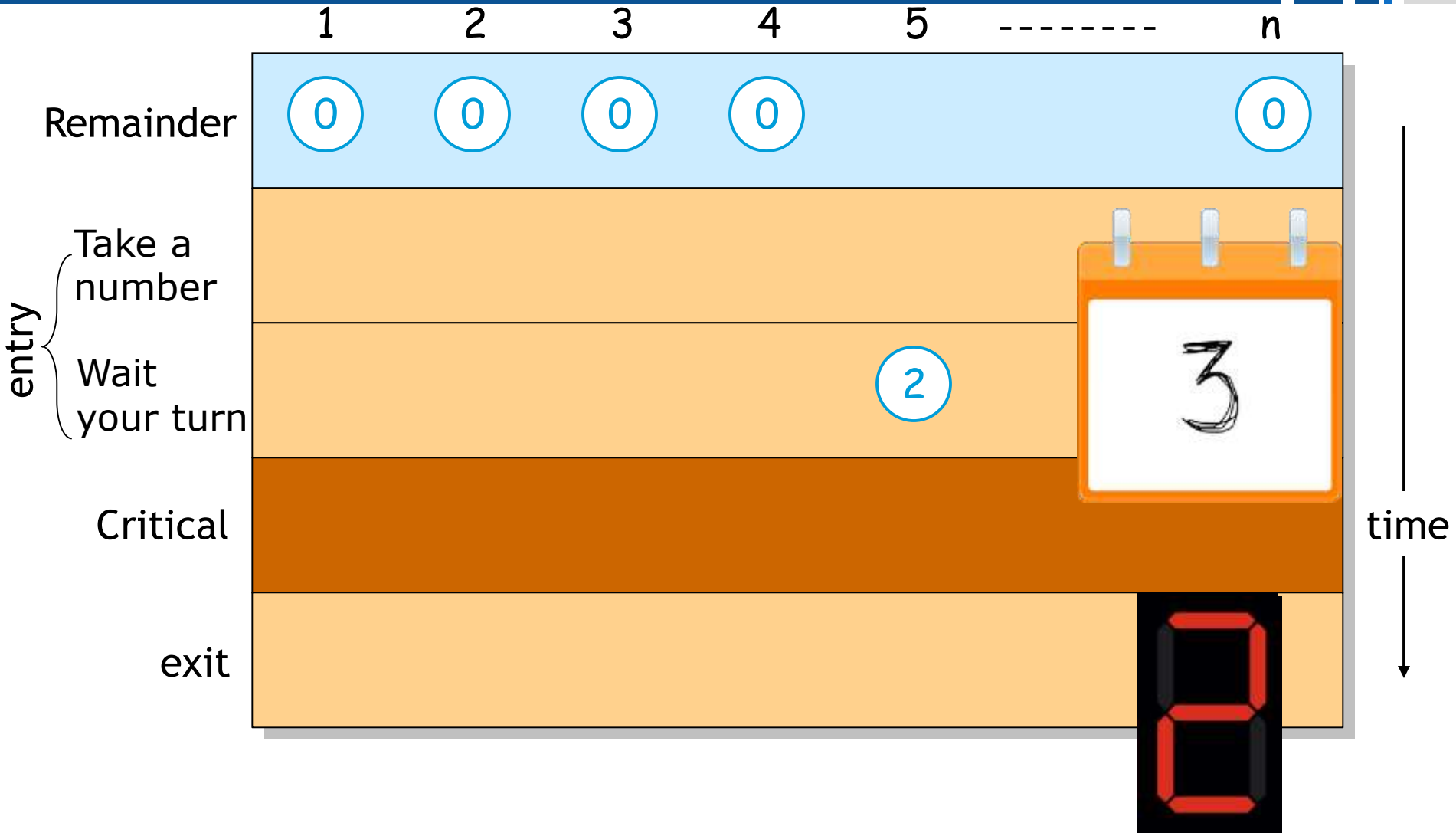
Lamport's Bakery Algorithm - 1974



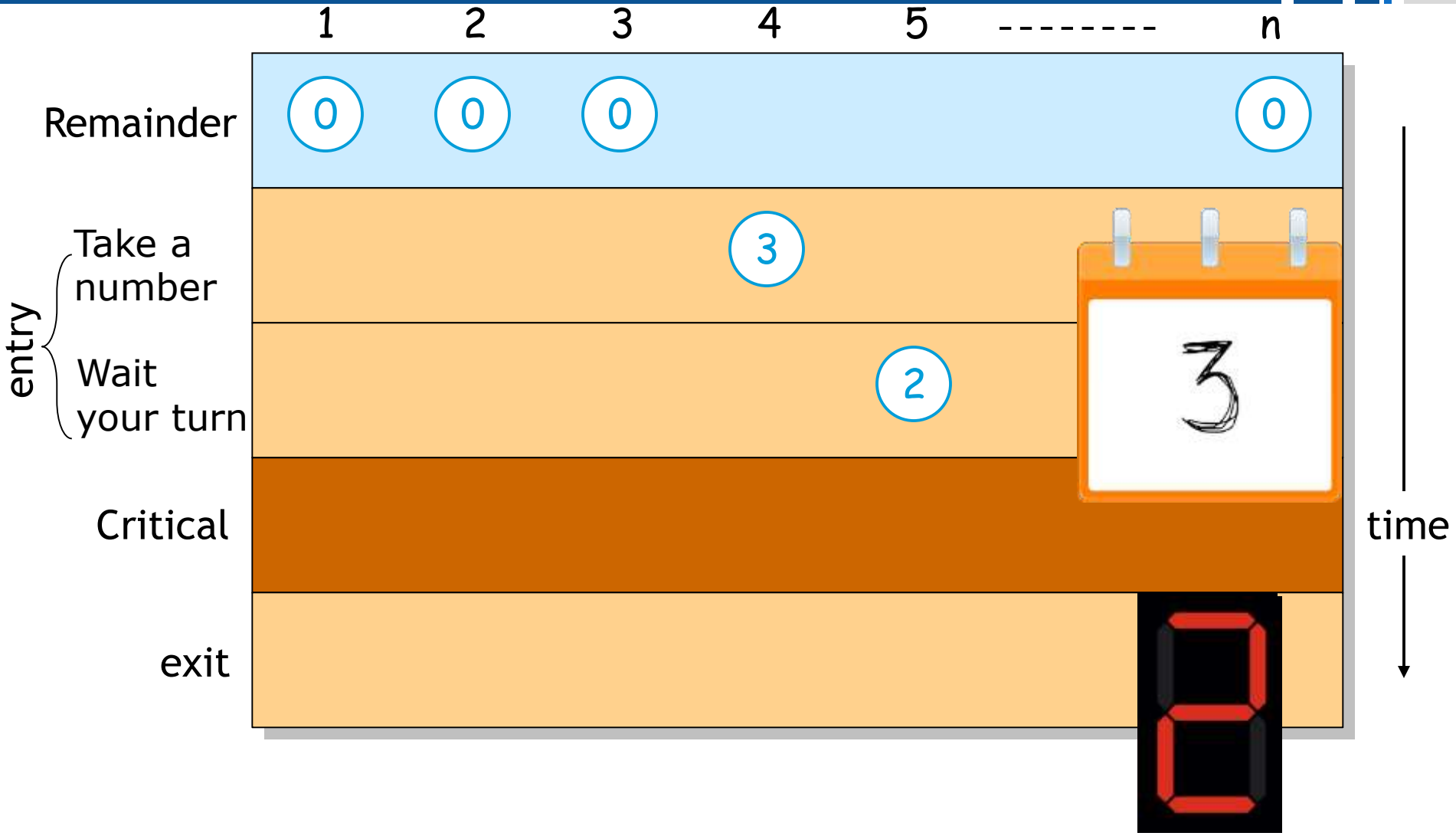
Lamport's Bakery Algorithm - 1974



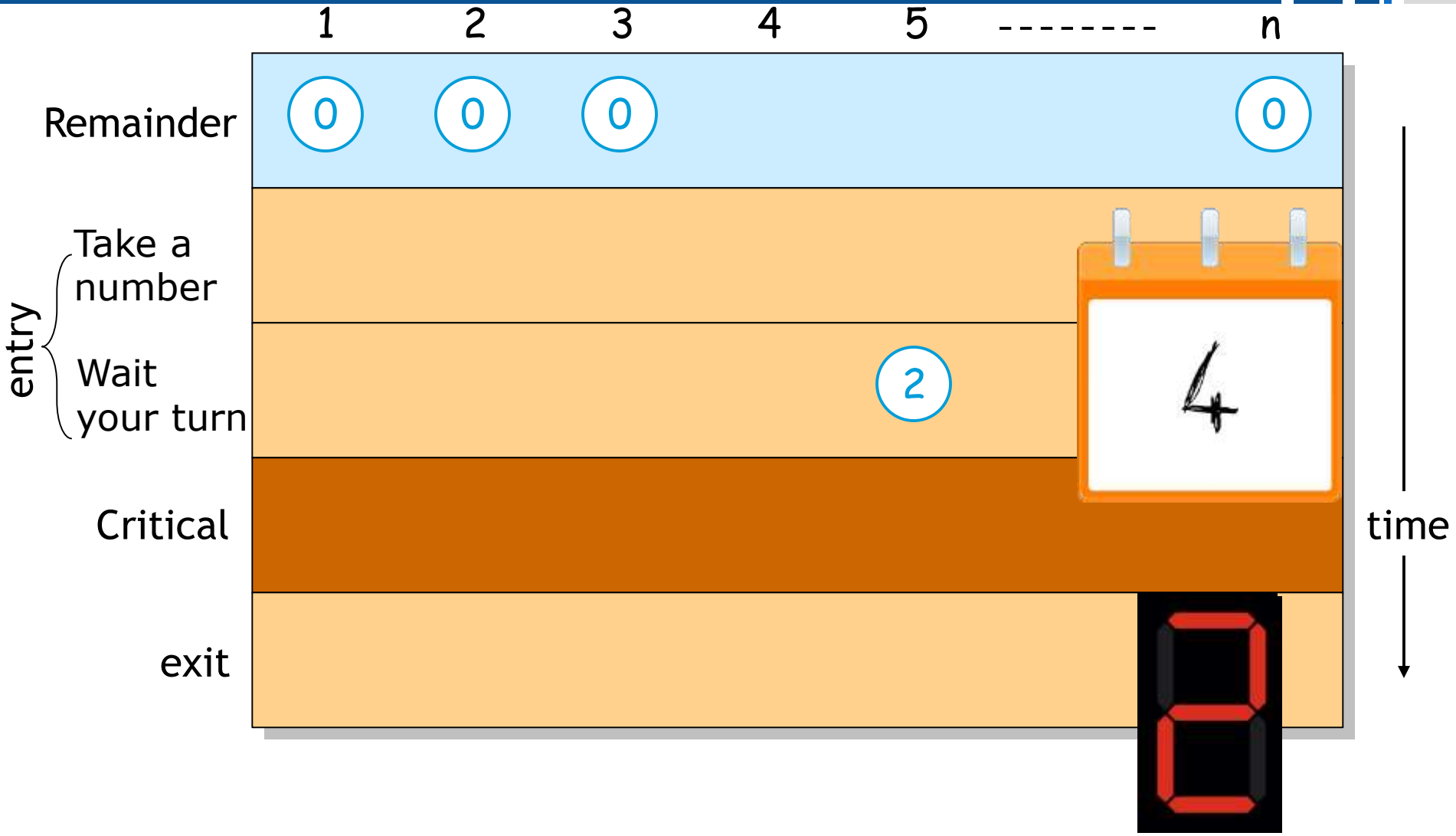
Lamport's Bakery Algorithm - 1974



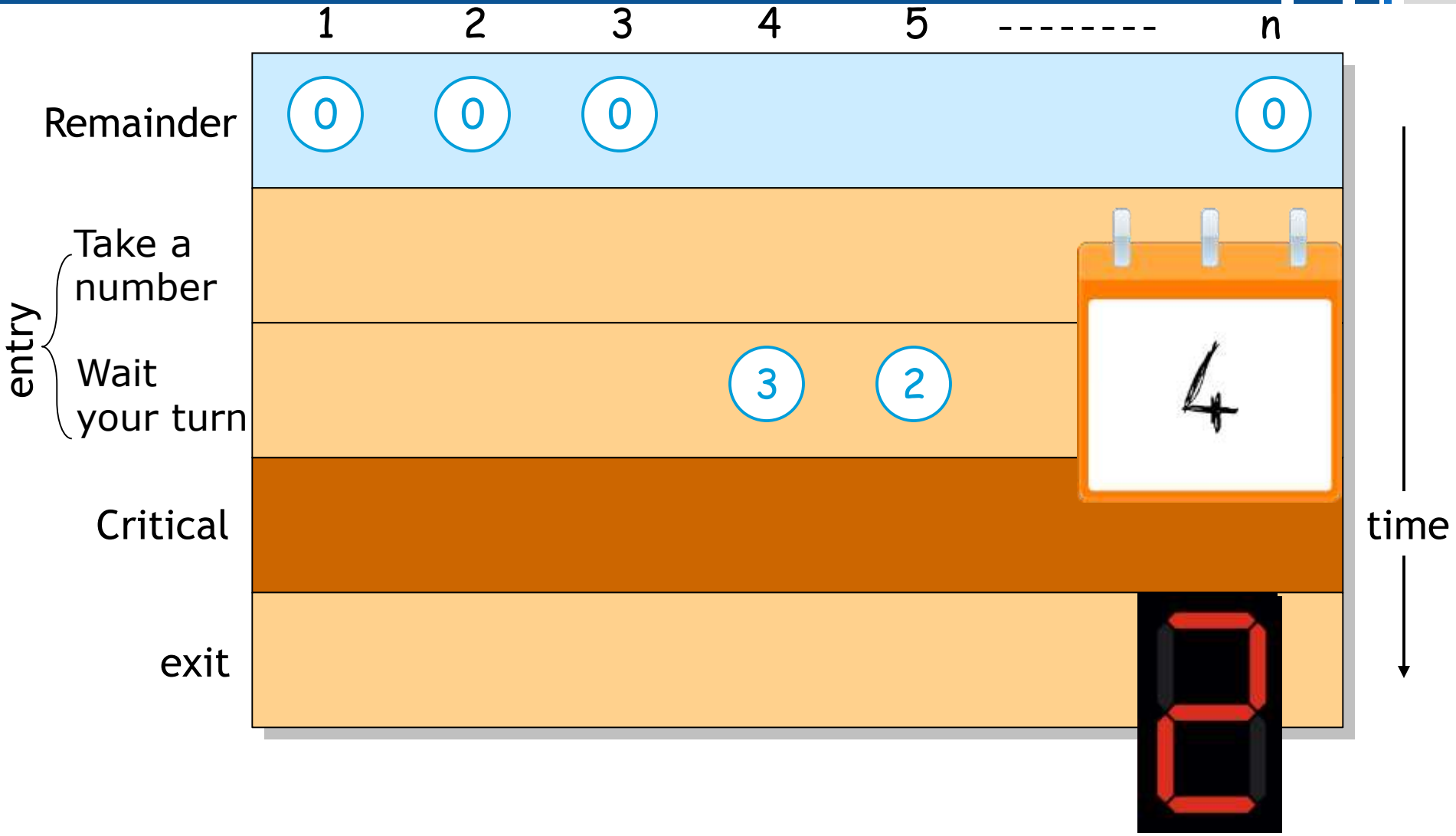
Lamport's Bakery Algorithm - 1974



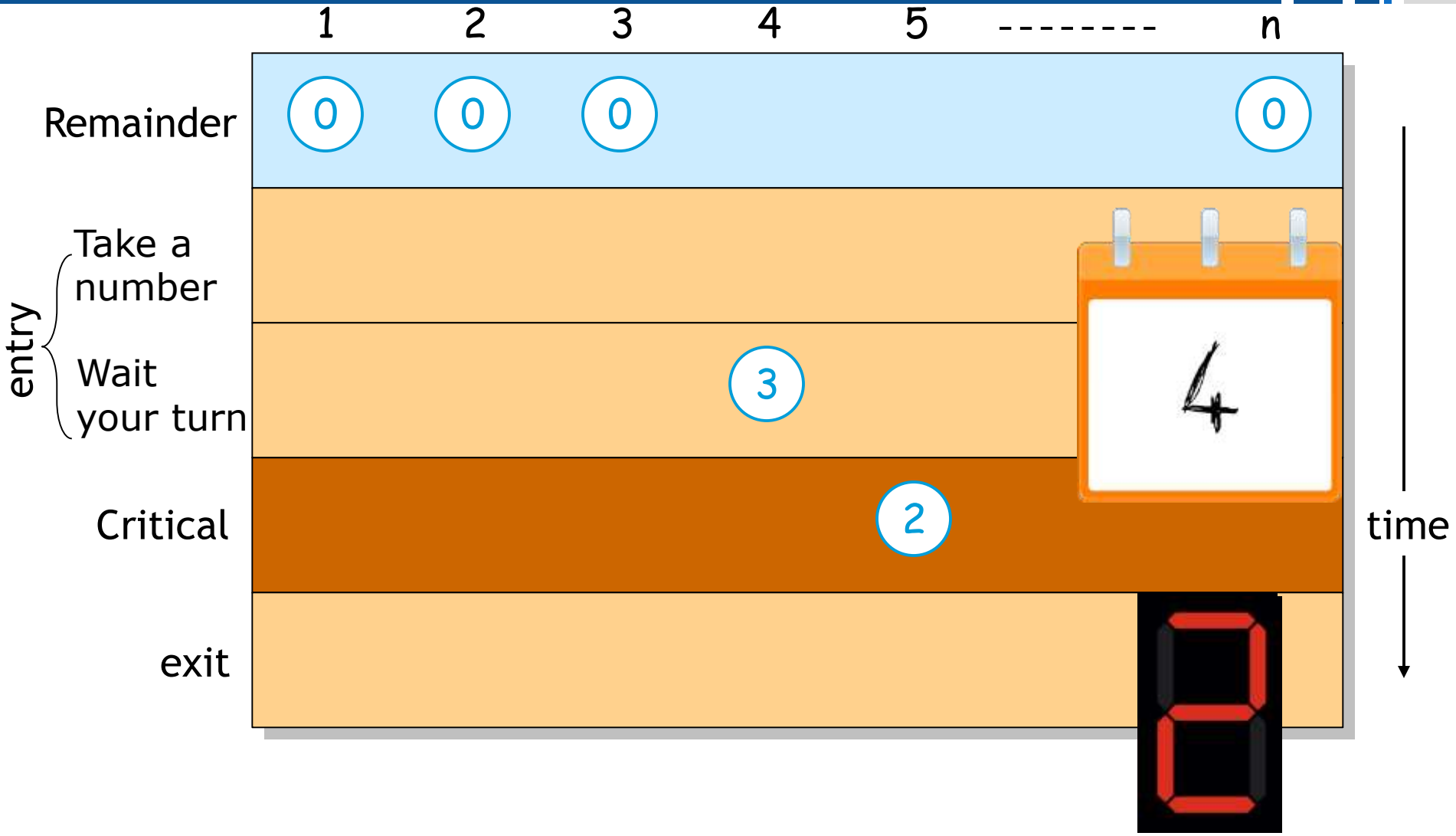
Lamport's Bakery Algorithm - 1974



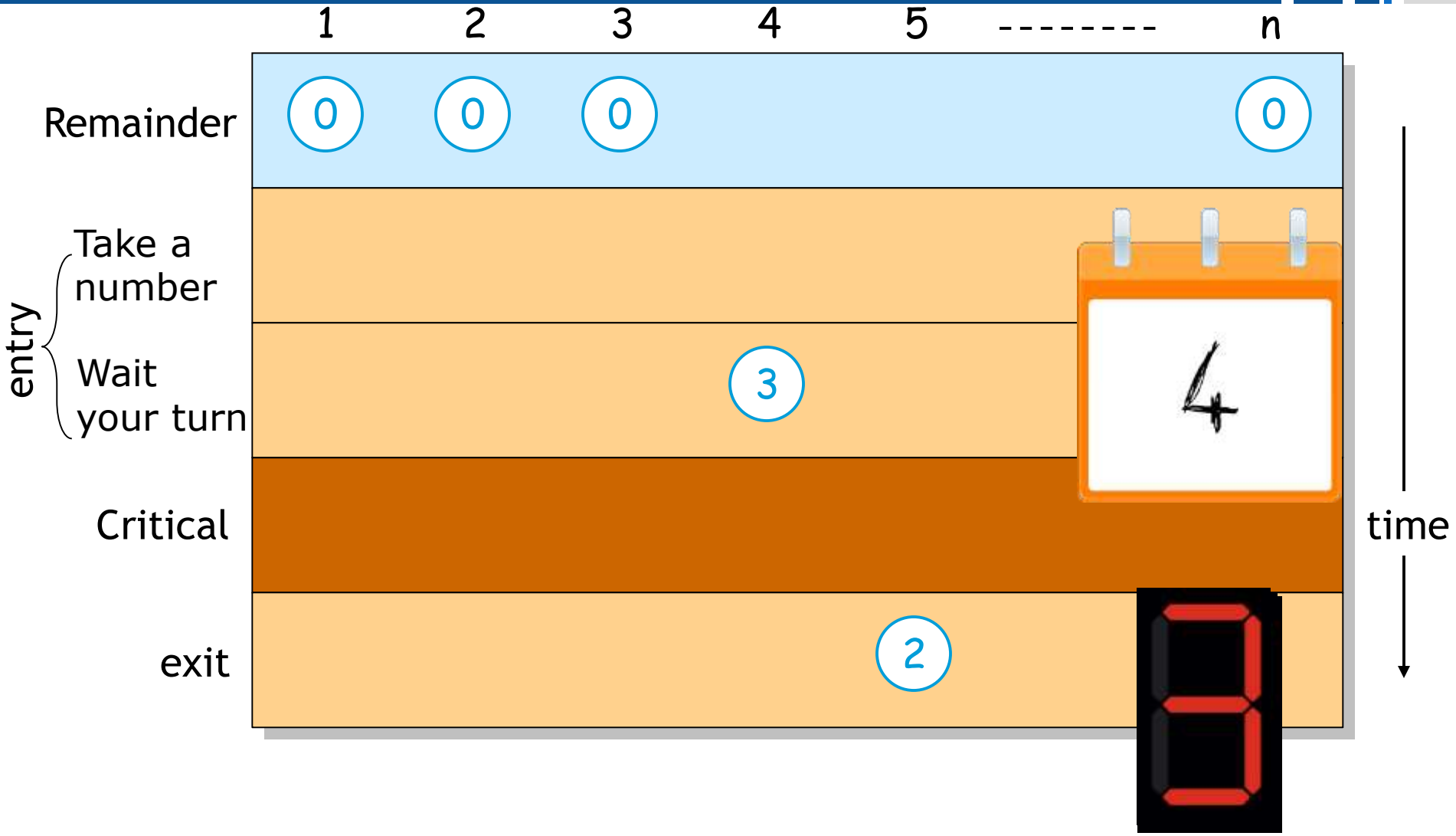
Lamport's Bakery Algorithm - 1974



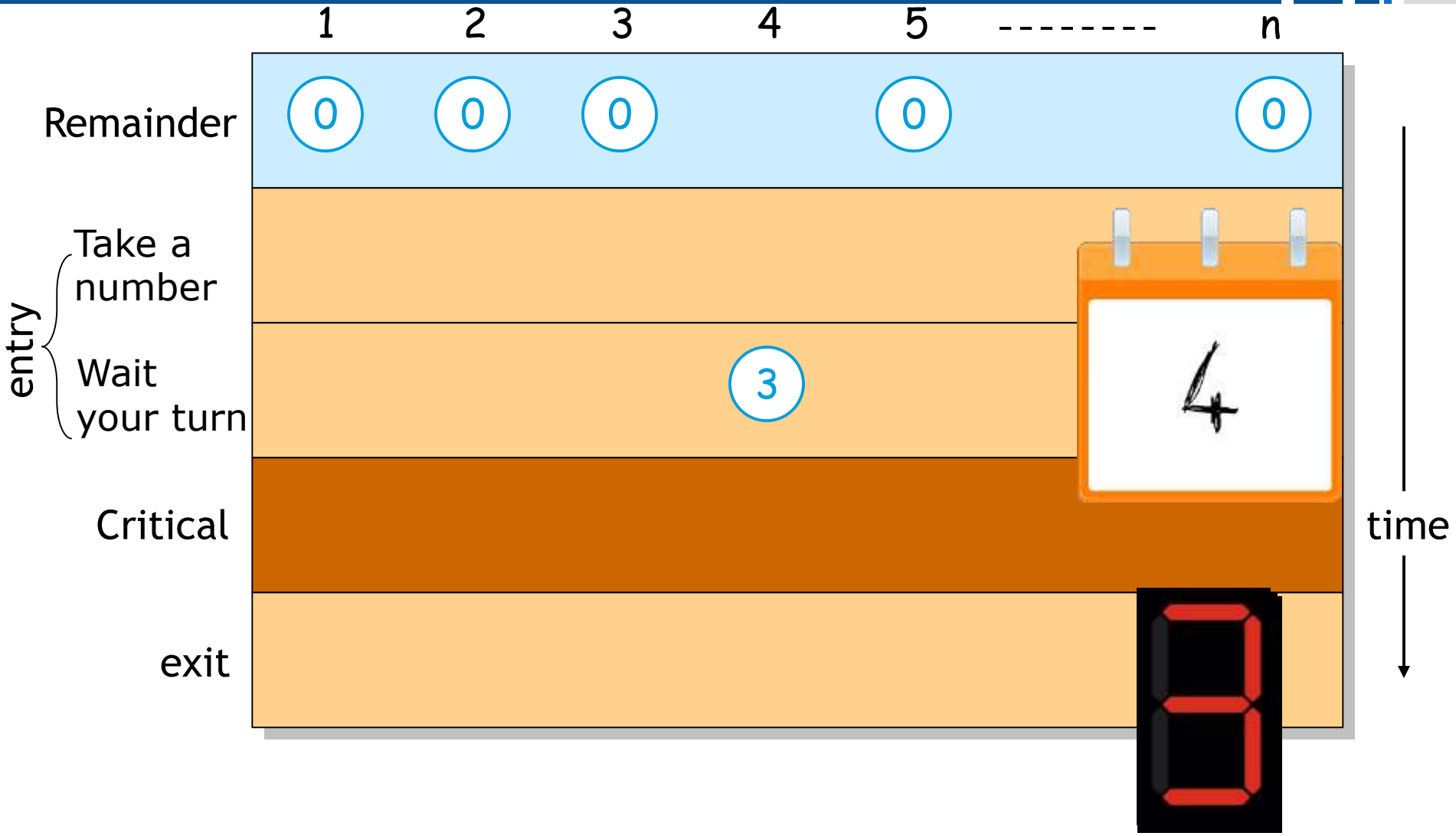
Lamport's Bakery Algorithm - 1974



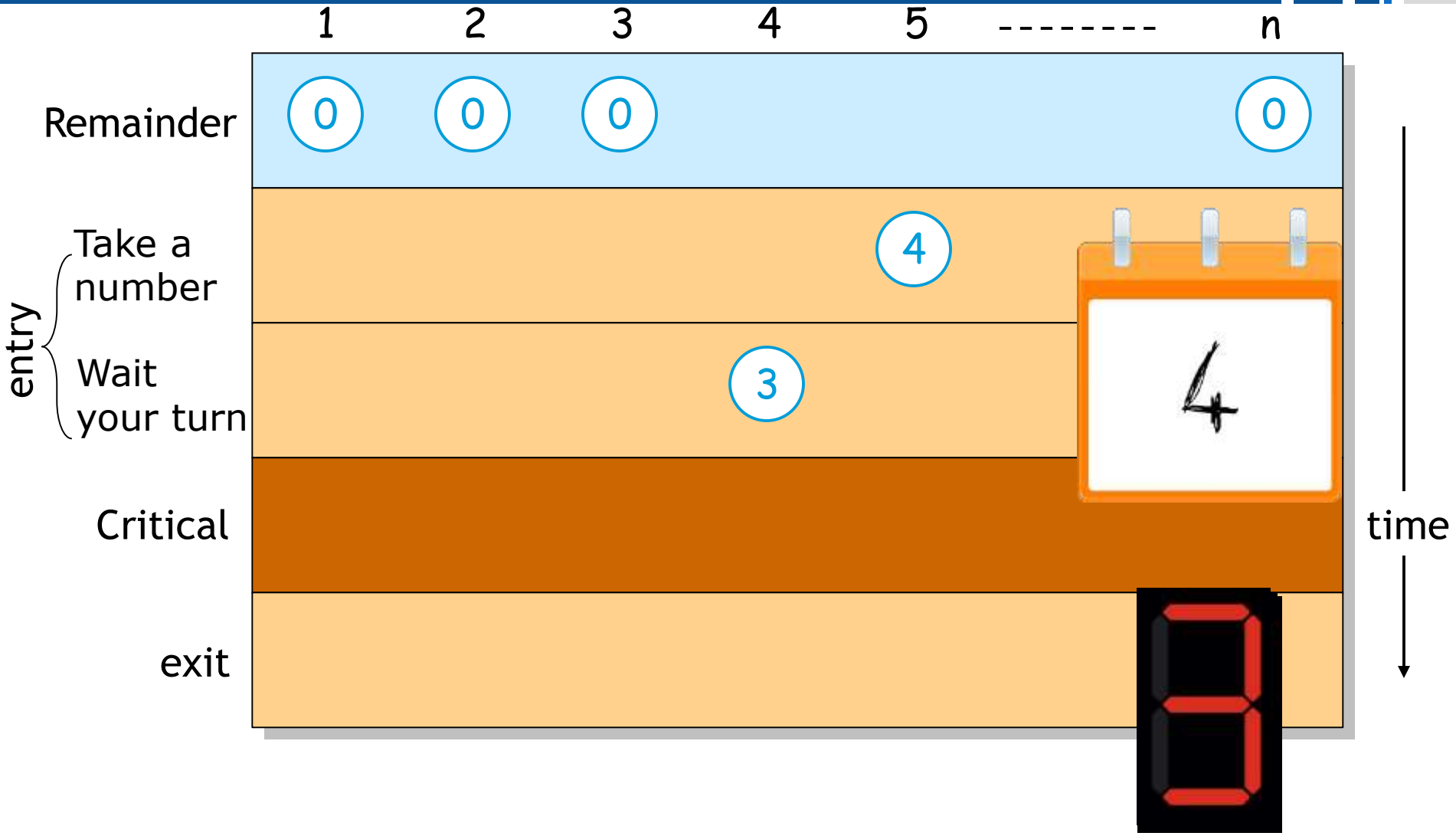
Lamport's Bakery Algorithm - 1974



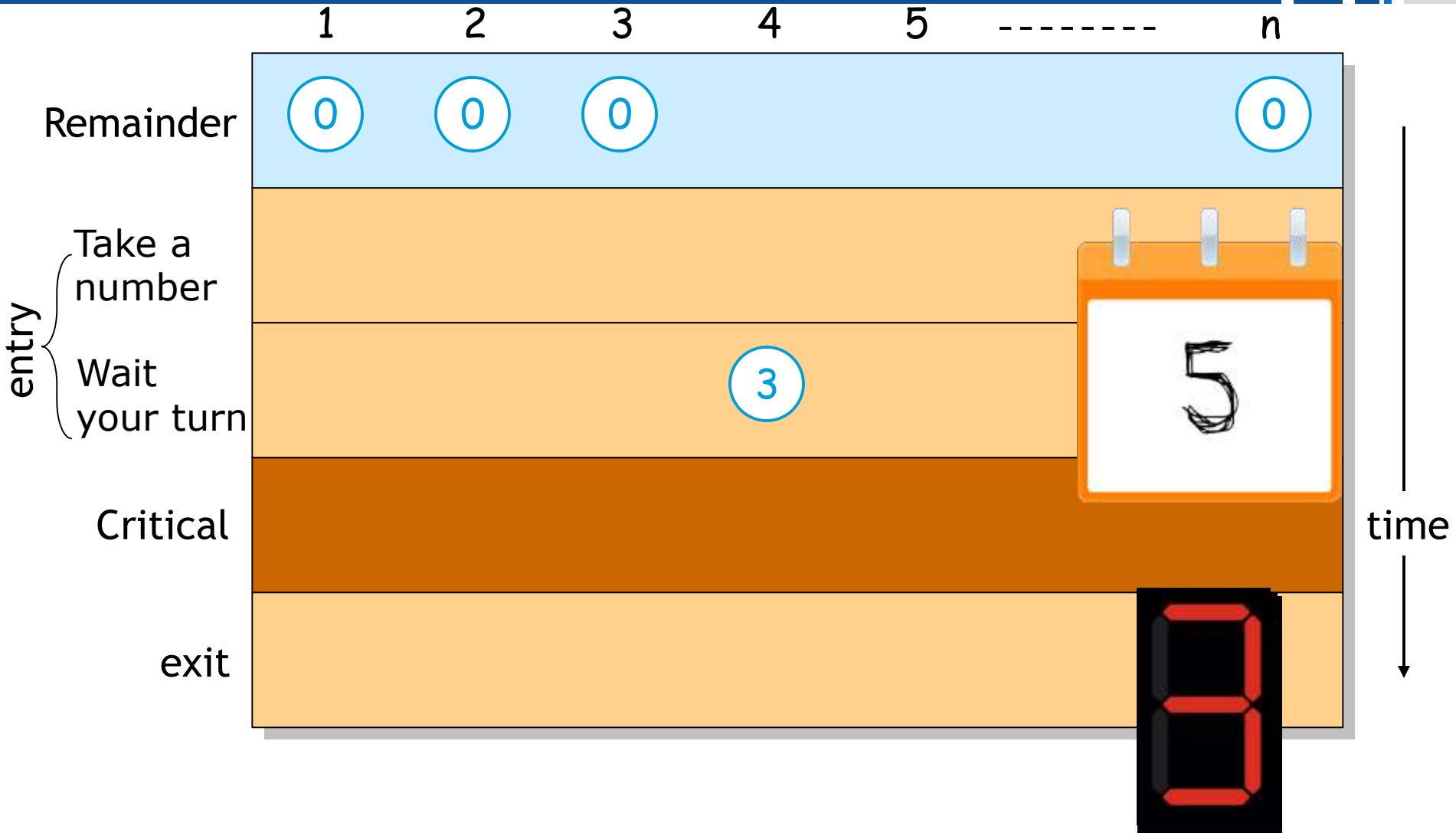
Lamport's Bakery Algorithm - 1974



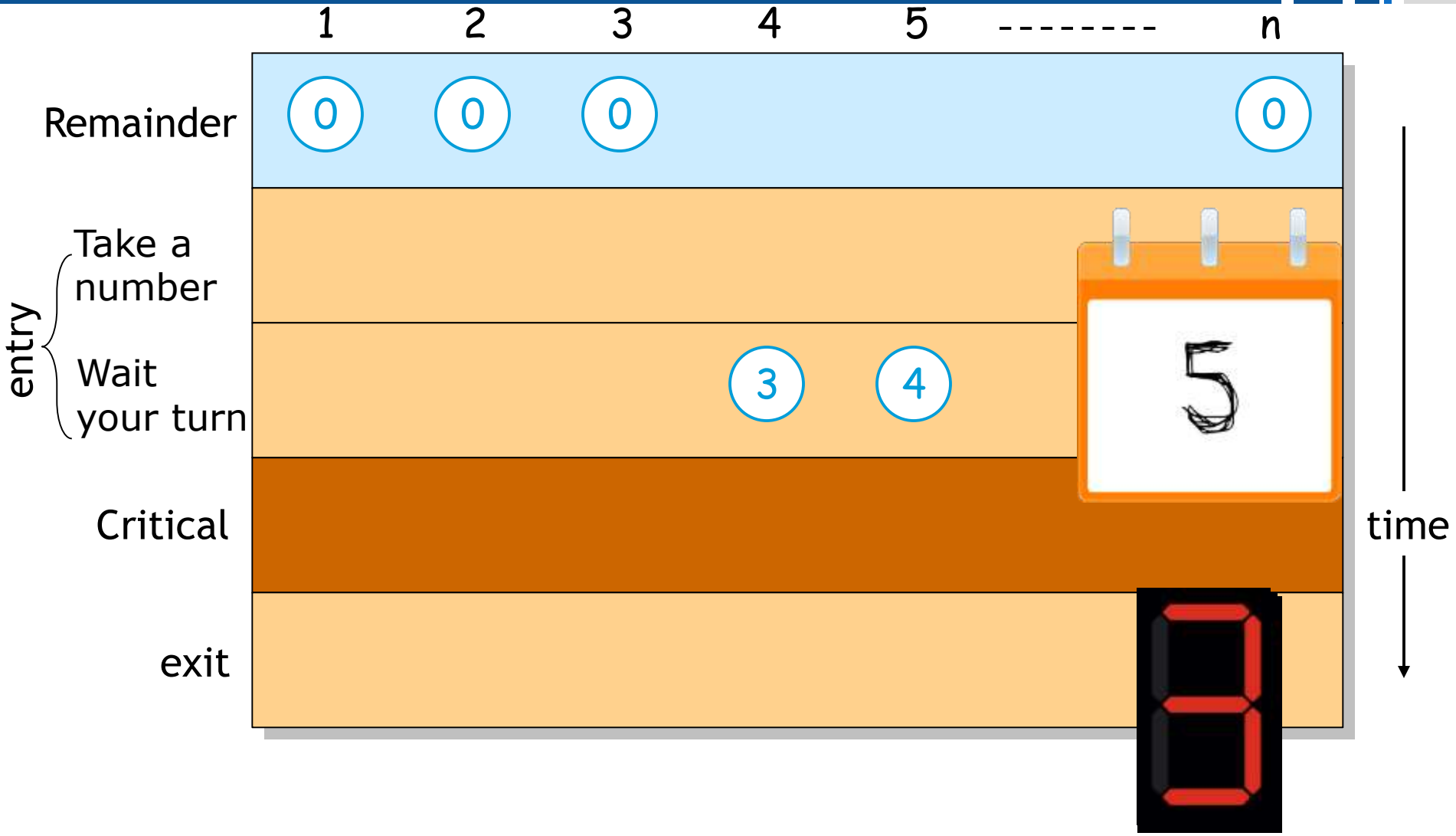
Lamport's Bakery Algorithm - 1974



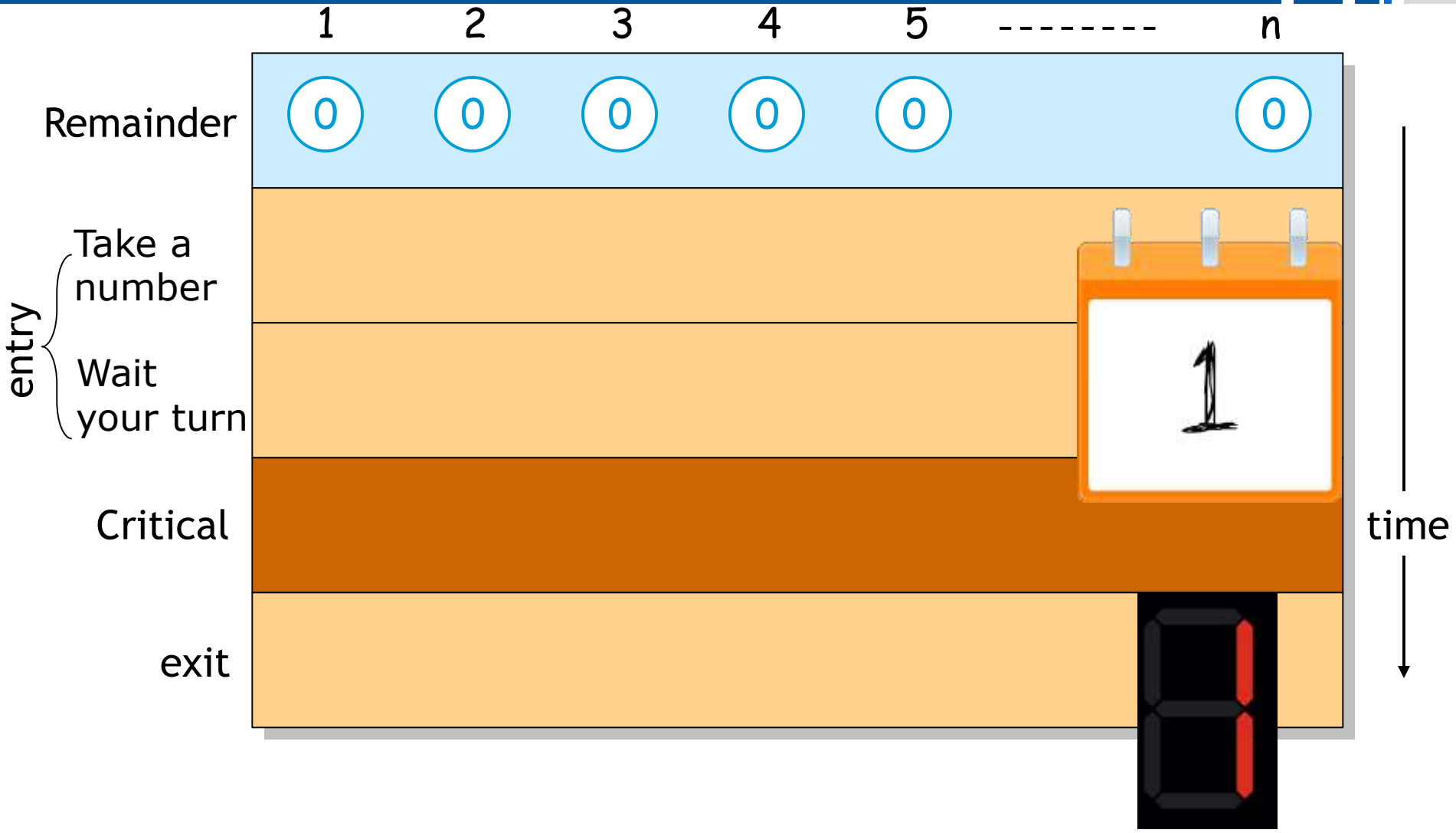
Lamport's Bakery Algorithm - 1974



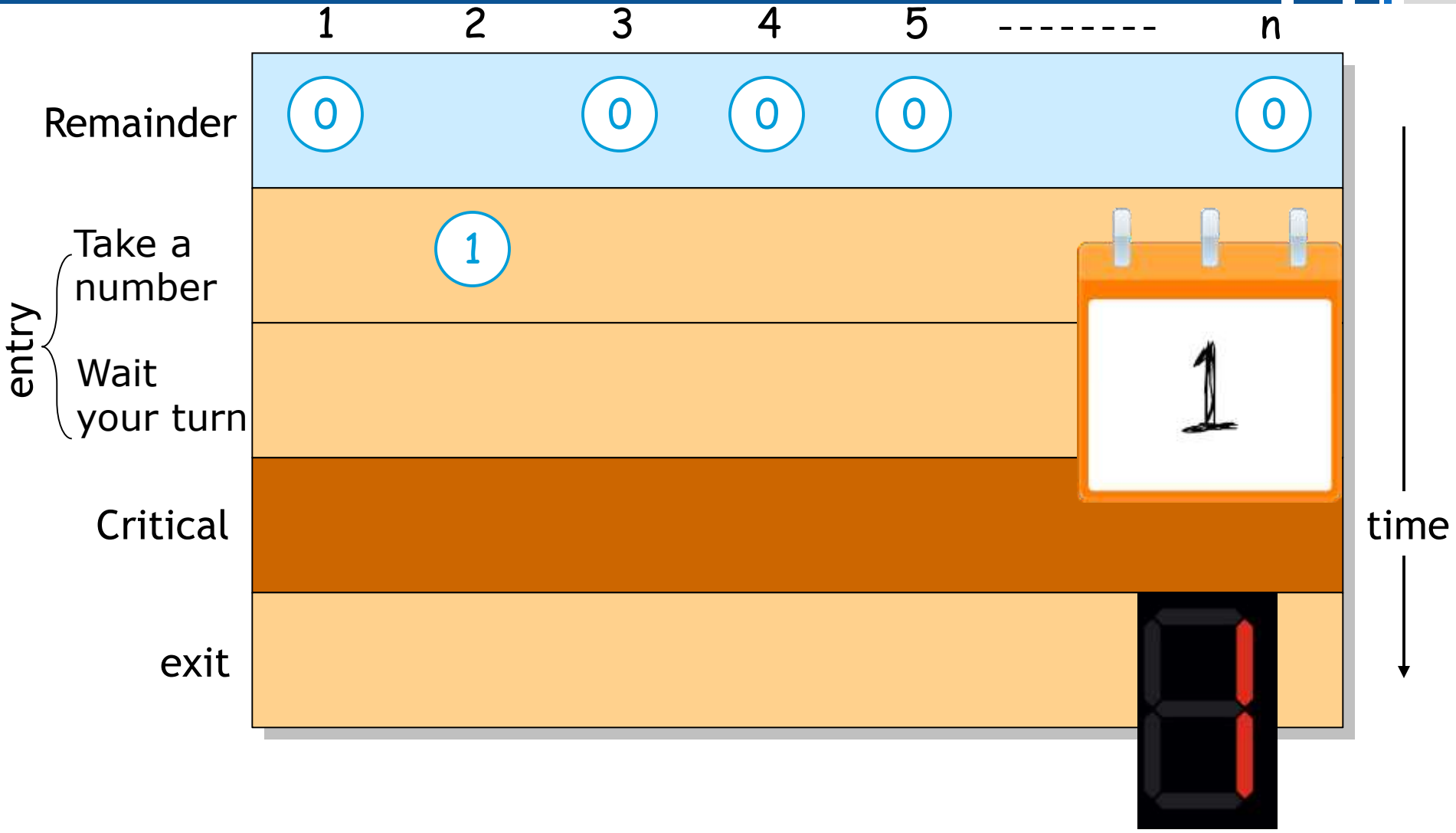
Lamport's Bakery Algorithm - 1974



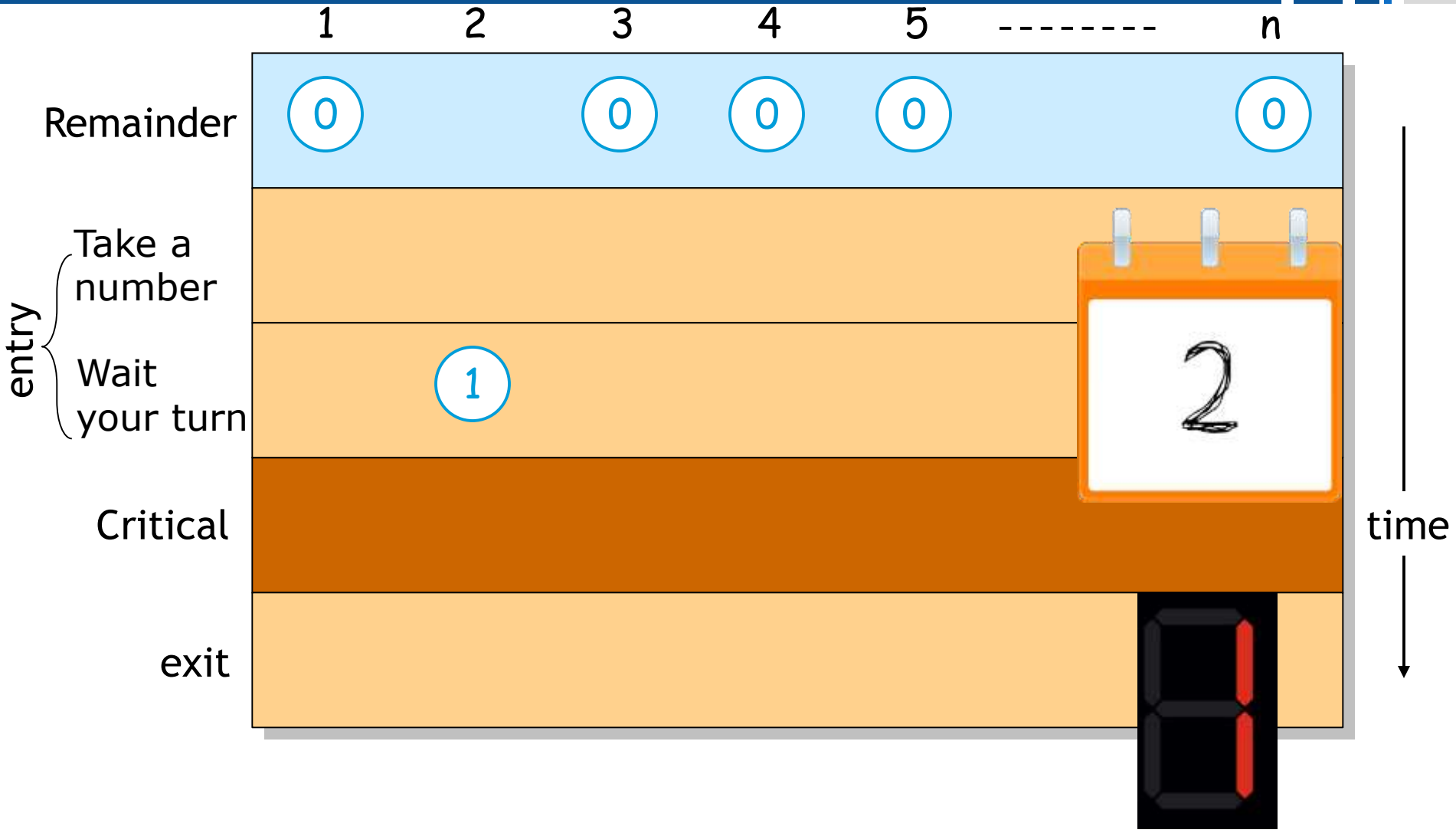
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
0
0
0
Take a number
entry
Wait your turn
1
Critical
exit
time

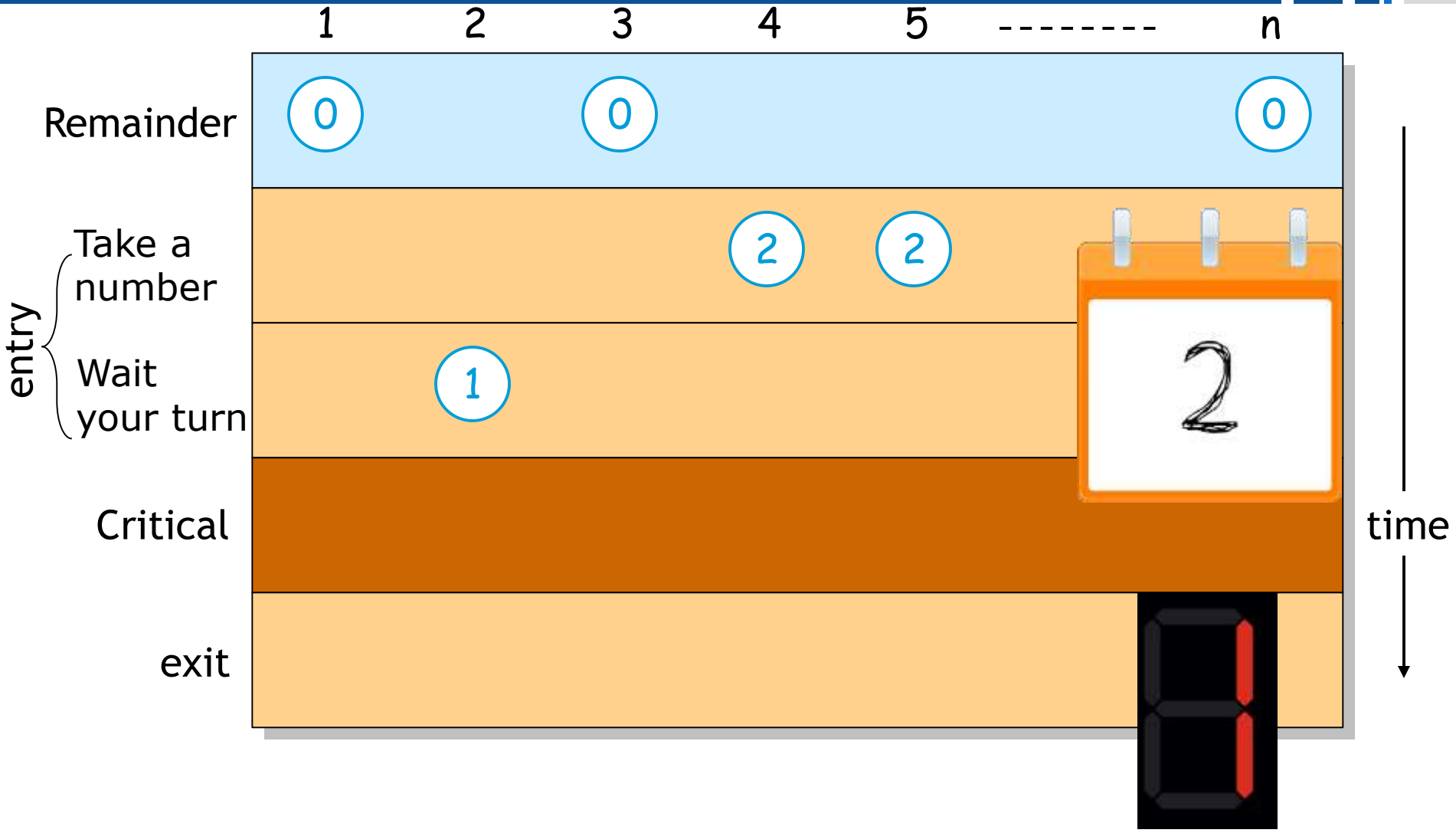
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
0
0
Take a number
1
entry
Wait your turn
1
Critical
exit
time

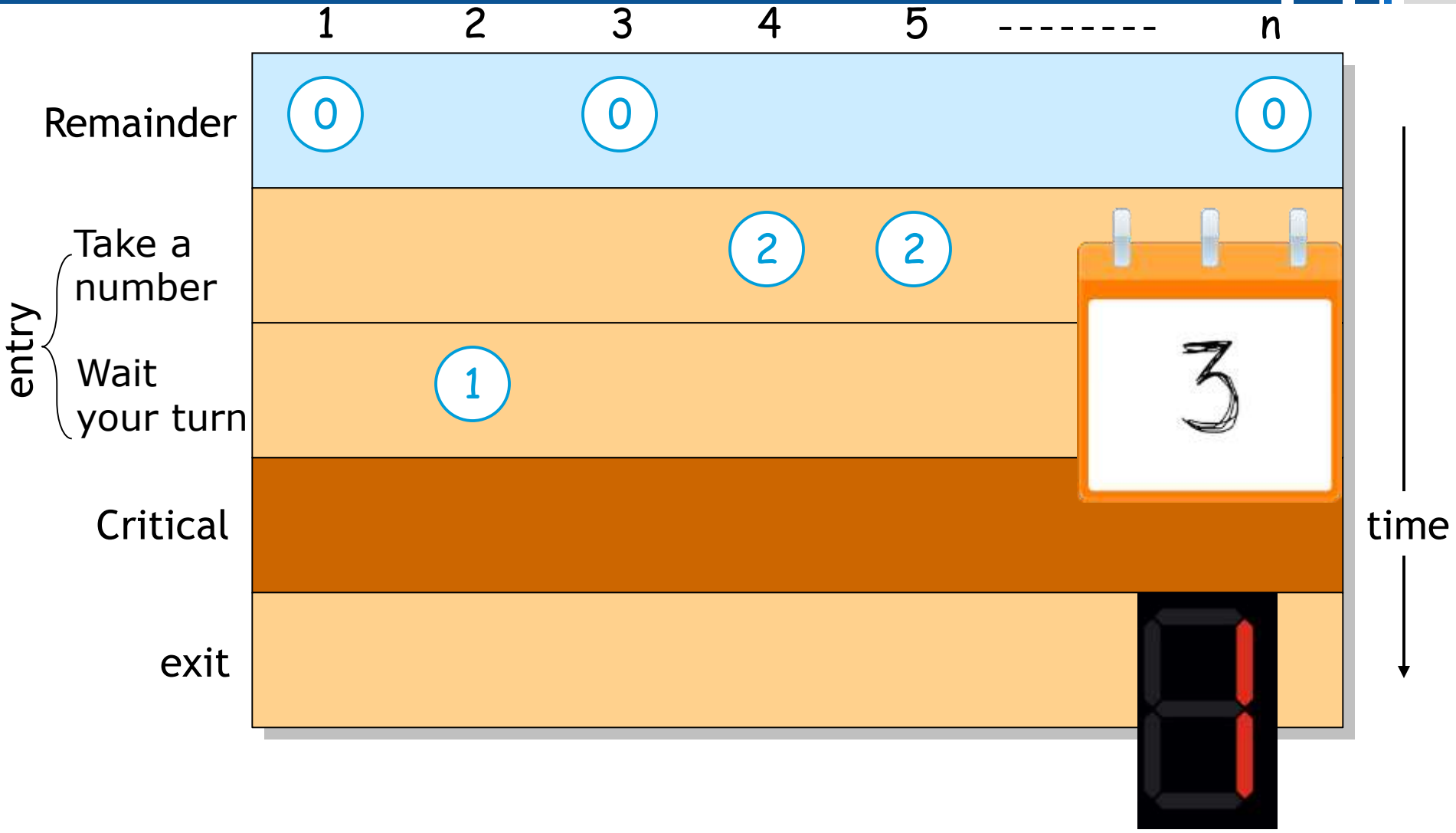
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
0
0
entry
Take a number
Wait your turn
1
2
Critical
exit
time

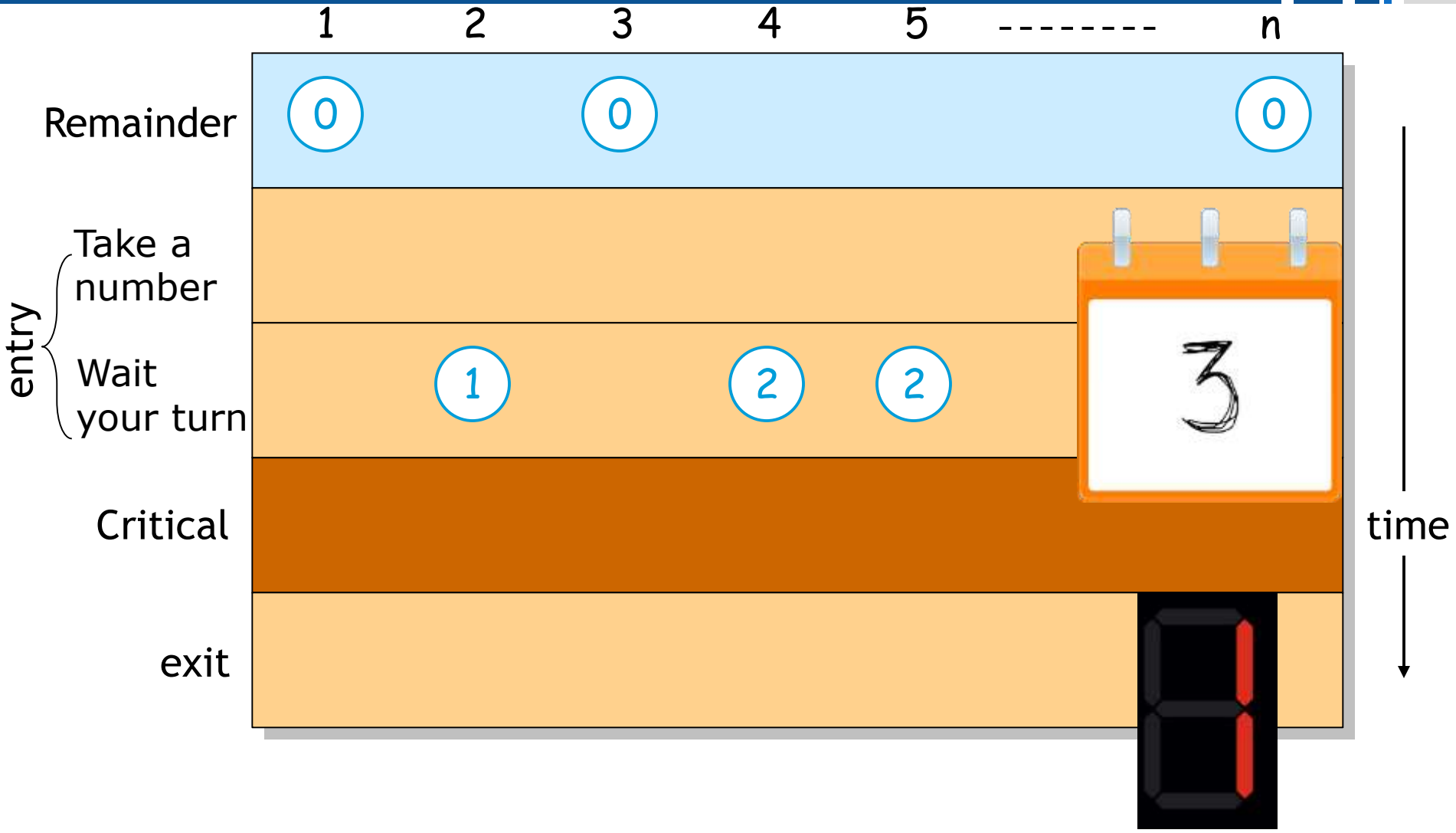
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
entry
Take a number
2
2
Wait your turn
1
2
Critical
exit
time

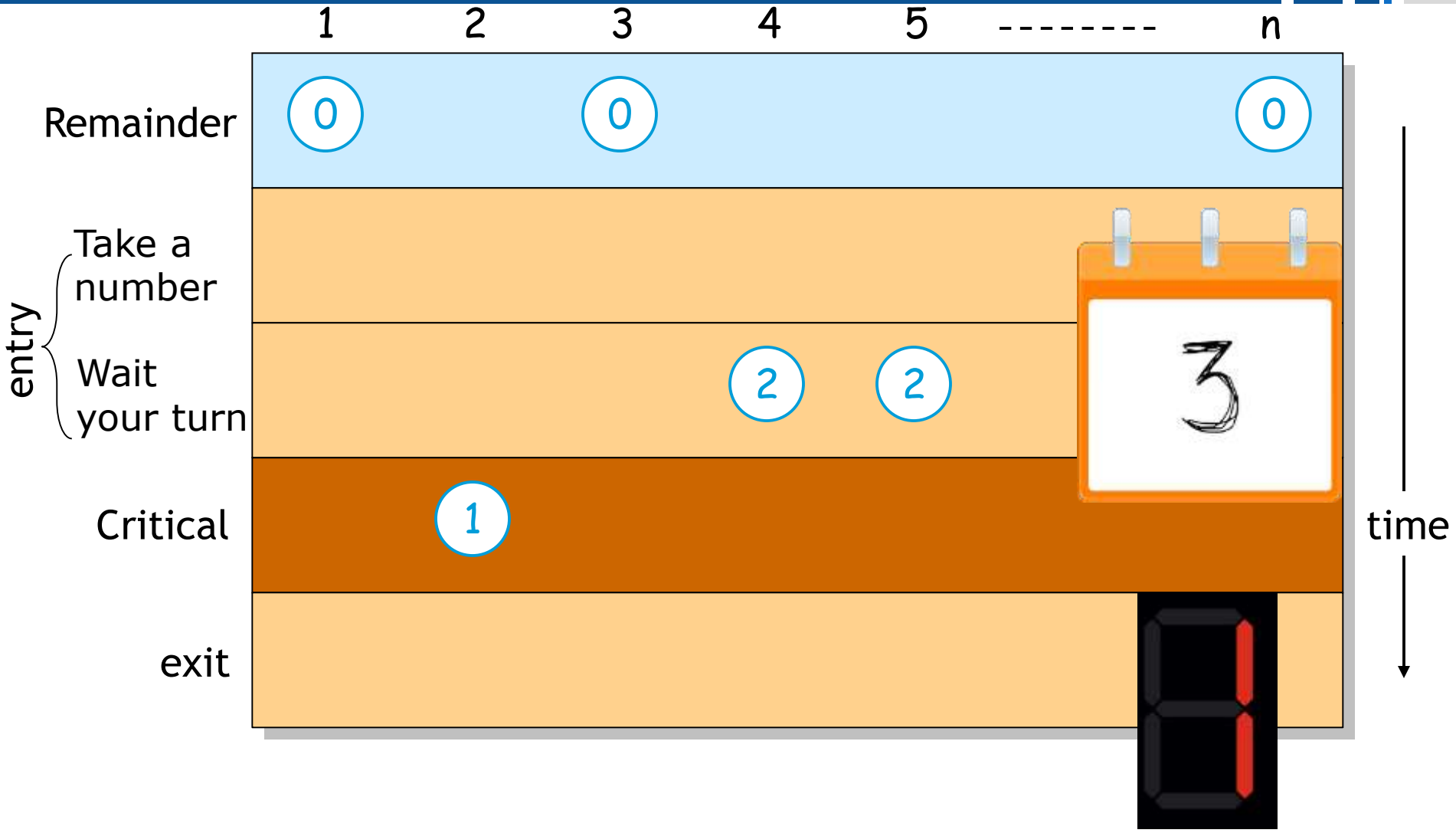
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
entry
Take a number
2
2
Wait your turn
1
3
Critical
exit
time

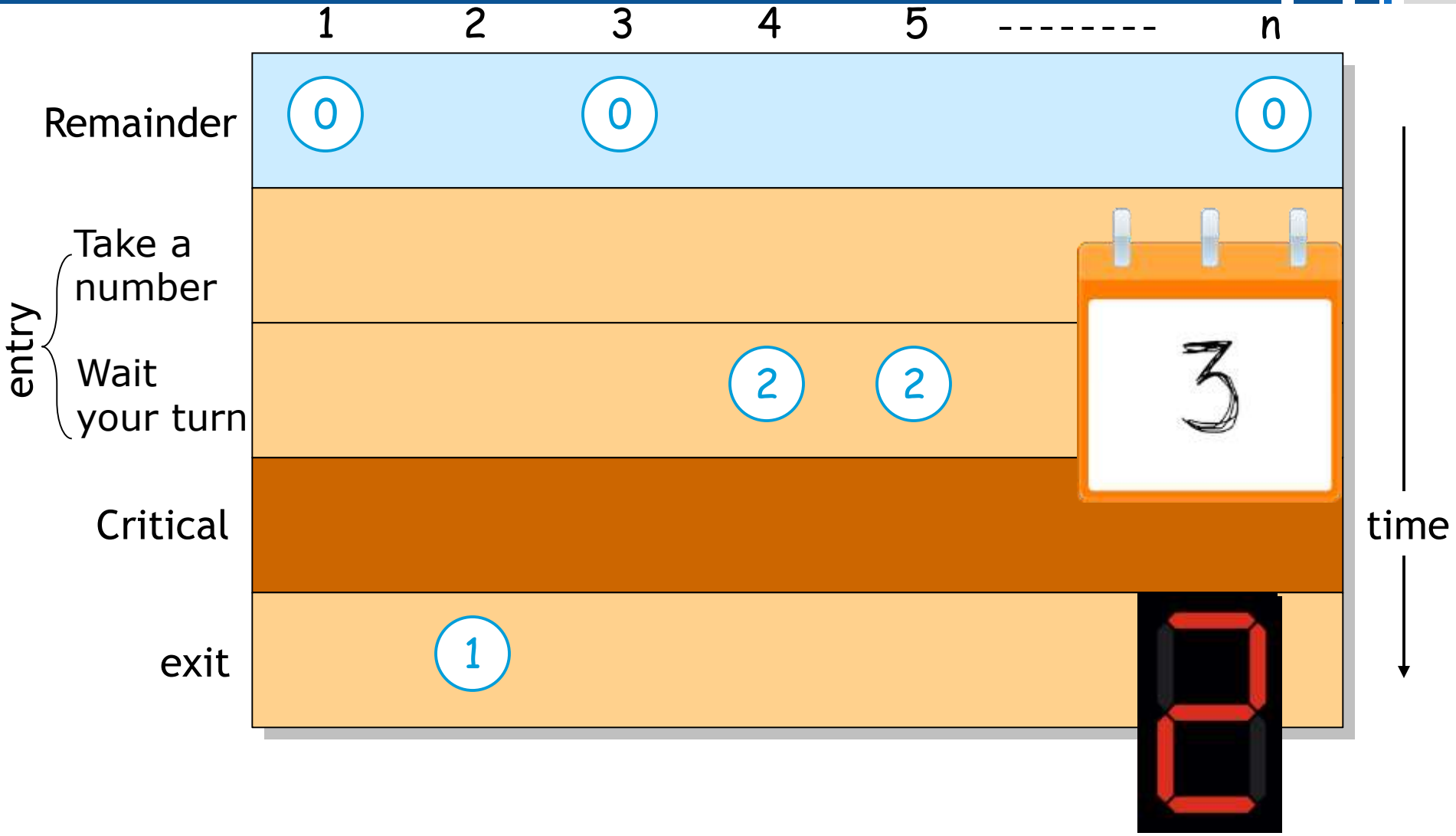
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
entry
Take a number
Wait your turn
1
2
2
3
Critical
exit
time

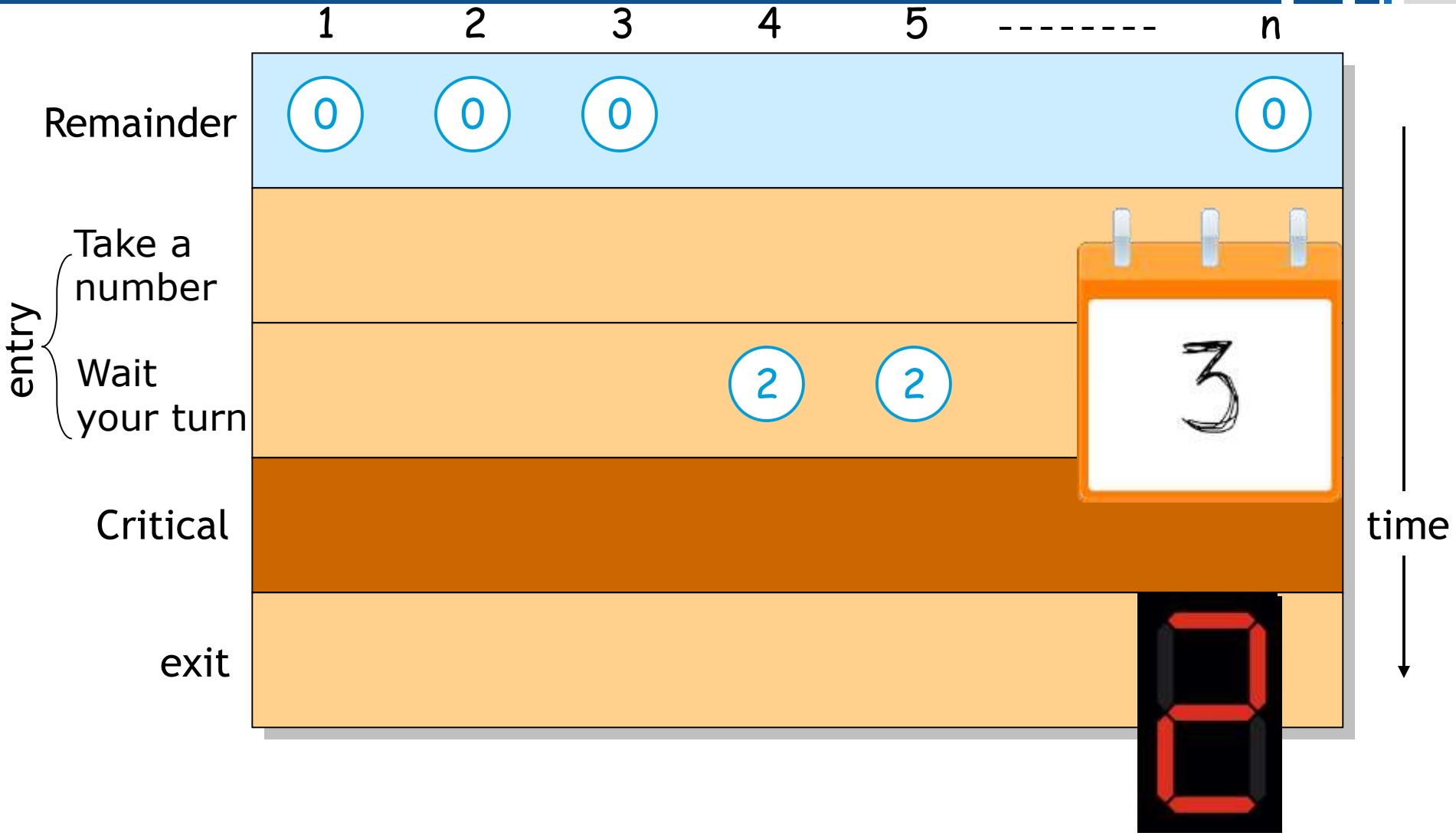
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
entry
Take a number
Wait your turn
2
2
3
Critical
1
exit
time

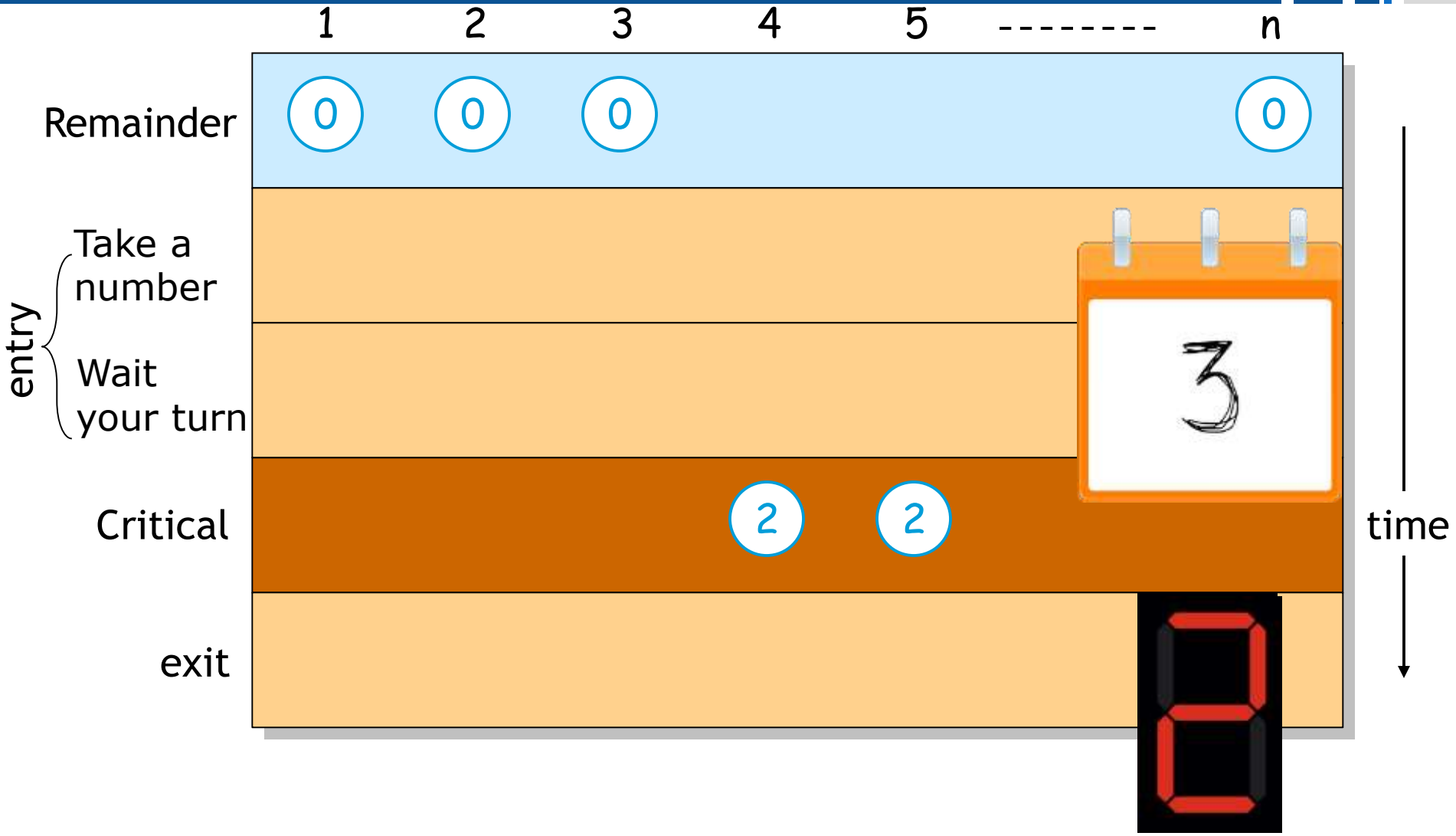
Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
entry
Take a number
Wait your turn
2
2
3
Critical
exit
1
time

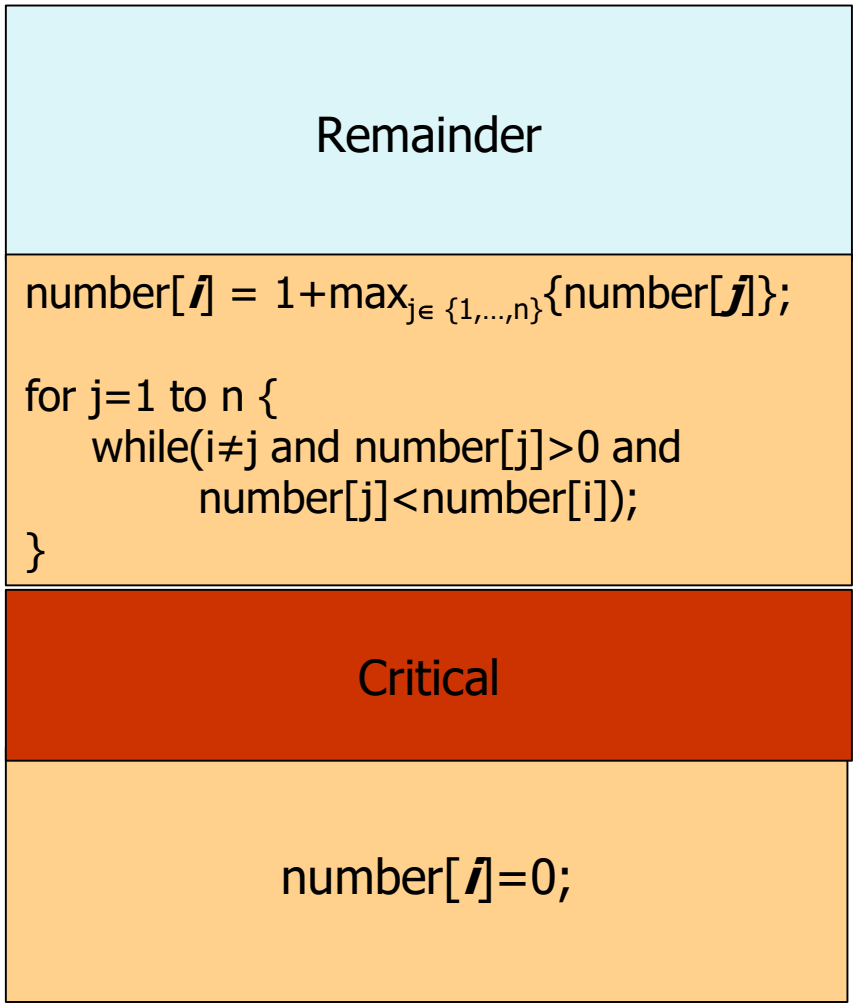
Not quite…


1
2
3
4
5
----
n
Remainder
0
0
0
0
entry
Take a number
Wait your turn
2
2
3
Critical
exit
time

Not quite...


1
2
3
4
5
----
n
Remainder
0
0
0
0
entry
Take a number
Wait your turn
3
Critical
2
2
exit
time

Implementation 1



Code for Thread ***i***
(out of ***n*** threads)

Implementation 1

Remainder

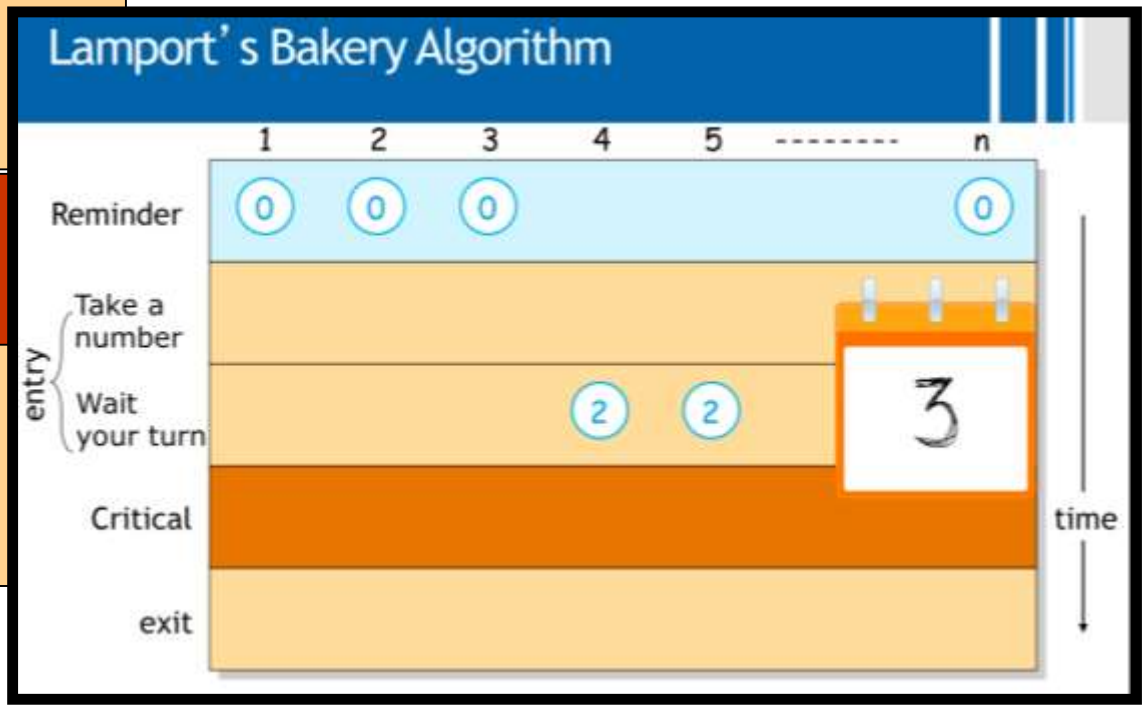
number[***i***] = 1+max$_{j \in \{1,\ldots,n\}}${number[***j***]};

```
for j=1 to n {
    while(i≠j and number[j]>0 and
          number[j]<number[i]);
}
```

Critical

number[***i***]=0;

Code for Thread ***i***
(out of ***n*** threads)



Implementation 1

Remainder

number[***i***] = 1+max$_{j \in \{1,\ldots,n\}}${number[***j***]};

for j=1 to n {
    while(i≠j and num[i]> 0 and
        number[j] number[i]);
}

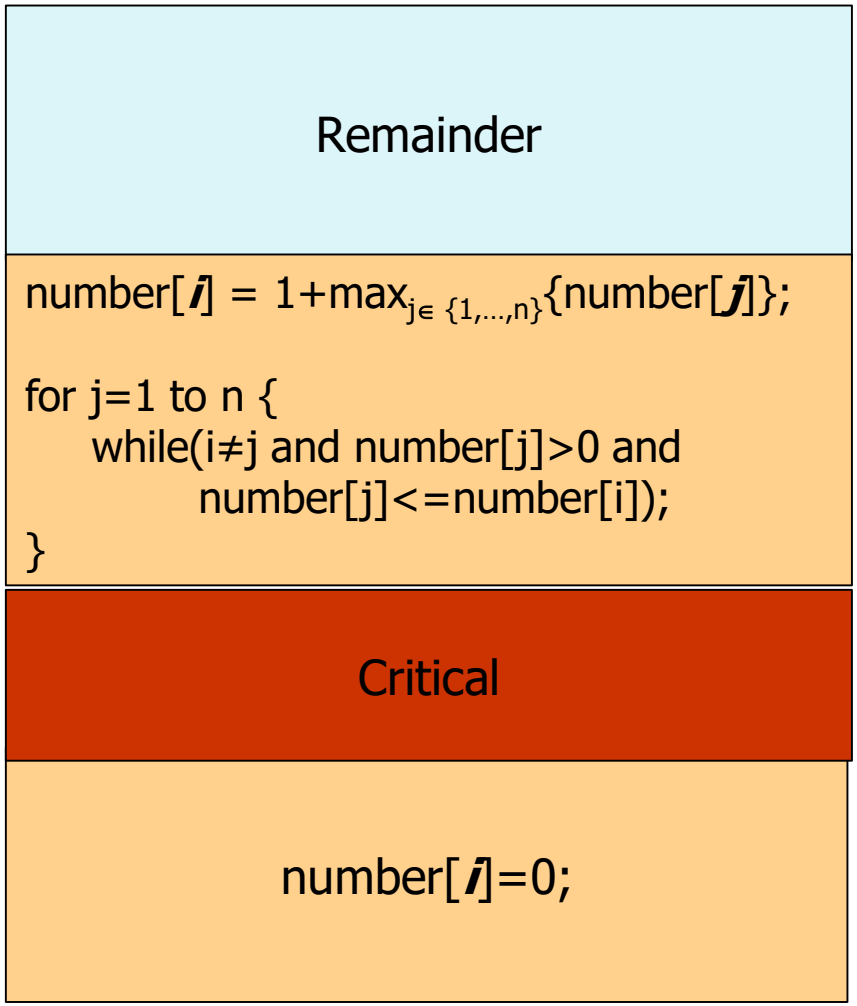
Critical

number[***i***]=0;

Code for Thread ***i***
(out of ***n*** threads)

No mutual exclusion

Implementation 1.1



Code for Thread ***i***
(out of ***n*** threads)

Implementation 1.1

Remainder

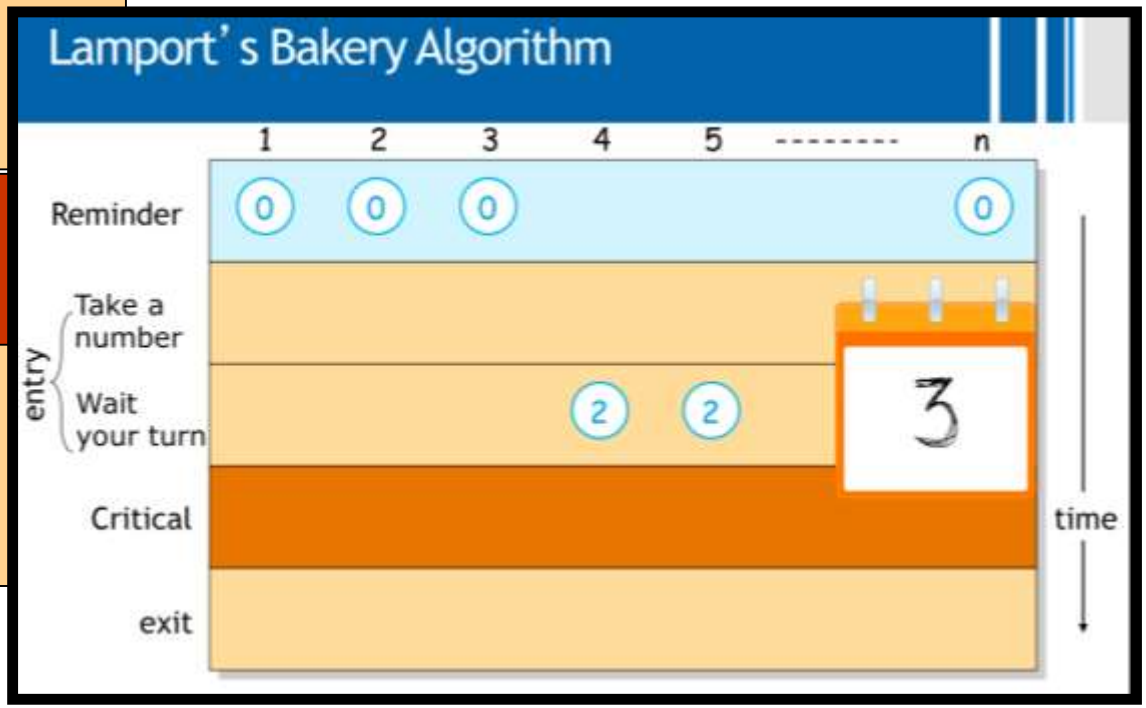
number[***i***] = 1+max$_{j \in \{1,\ldots,n\}}${number[***j***]};

```
for j=1 to n {
    while(i≠j and number[j]>0 and
          number[j]<=number[i]);
}
```

Critical

number[***i***]=0;

Code for Thread ***i***
(out of ***n*** threads)



Implementation 1.1

Remainder

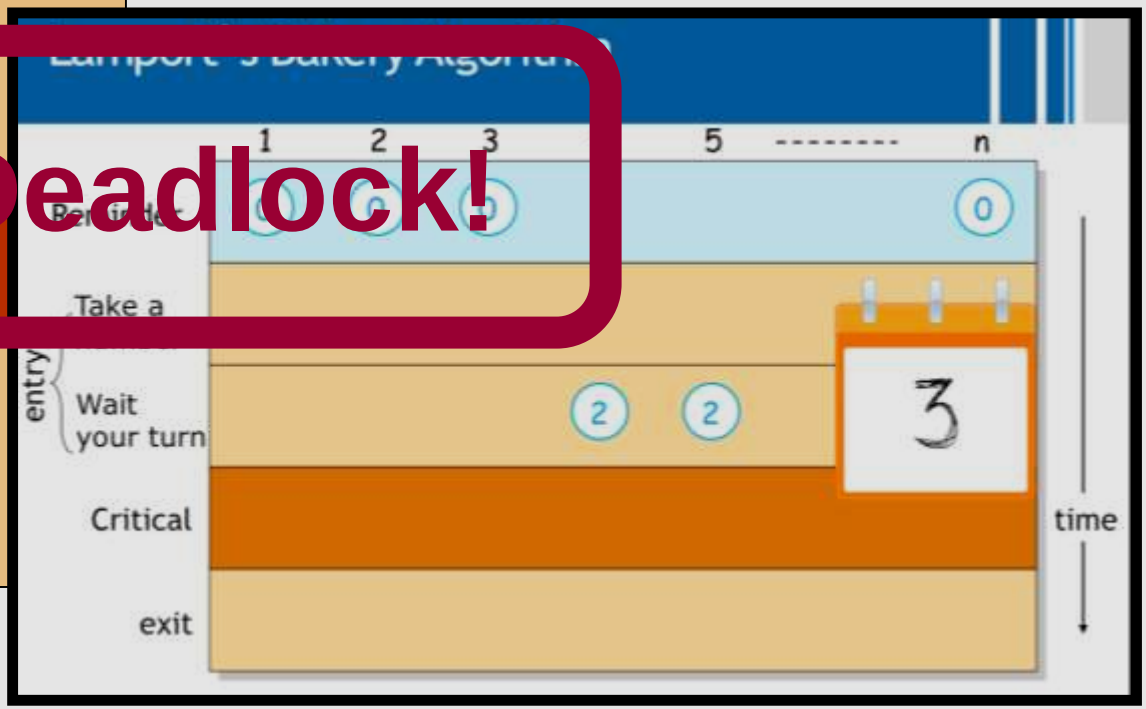
number[*i*] = 1+max$_{j \in \{1,\ldots,n\}}${number[*j*]};

for j=1 to n {
    while(i≠j and number[j]>0 and
        number[j]<=number[i]);
}

Critical

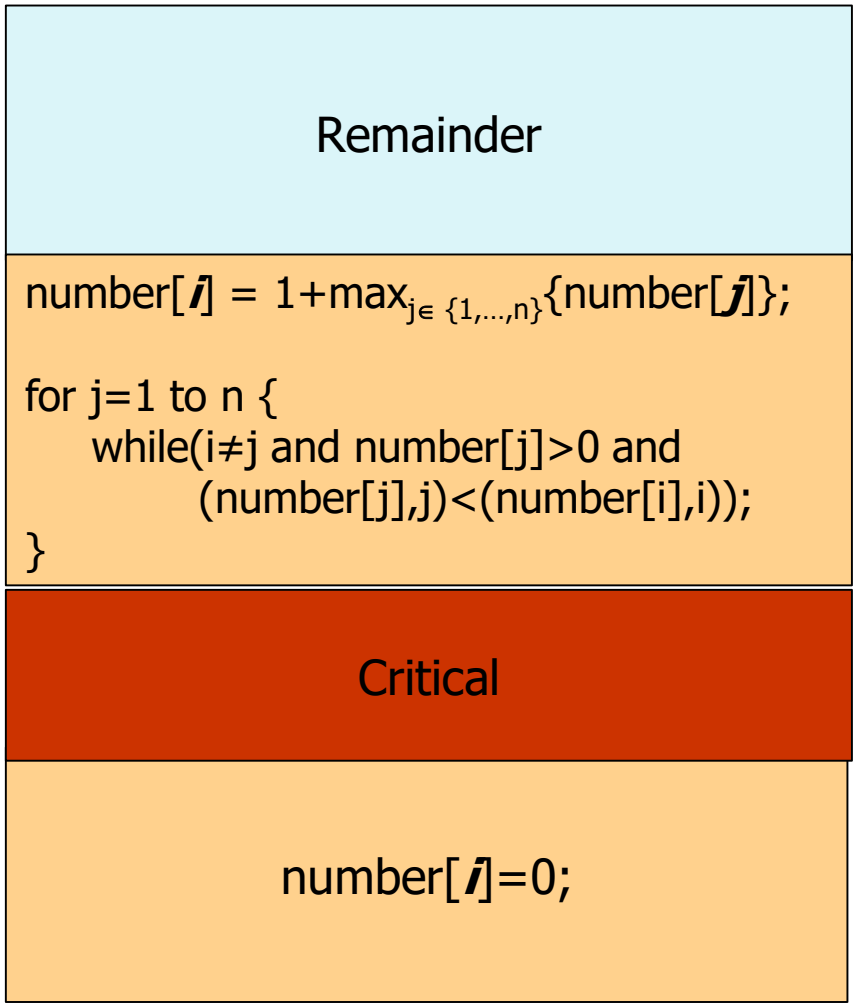
number[*i*]=0;

Code for Thread ***i***
(out of ***n*** threads)



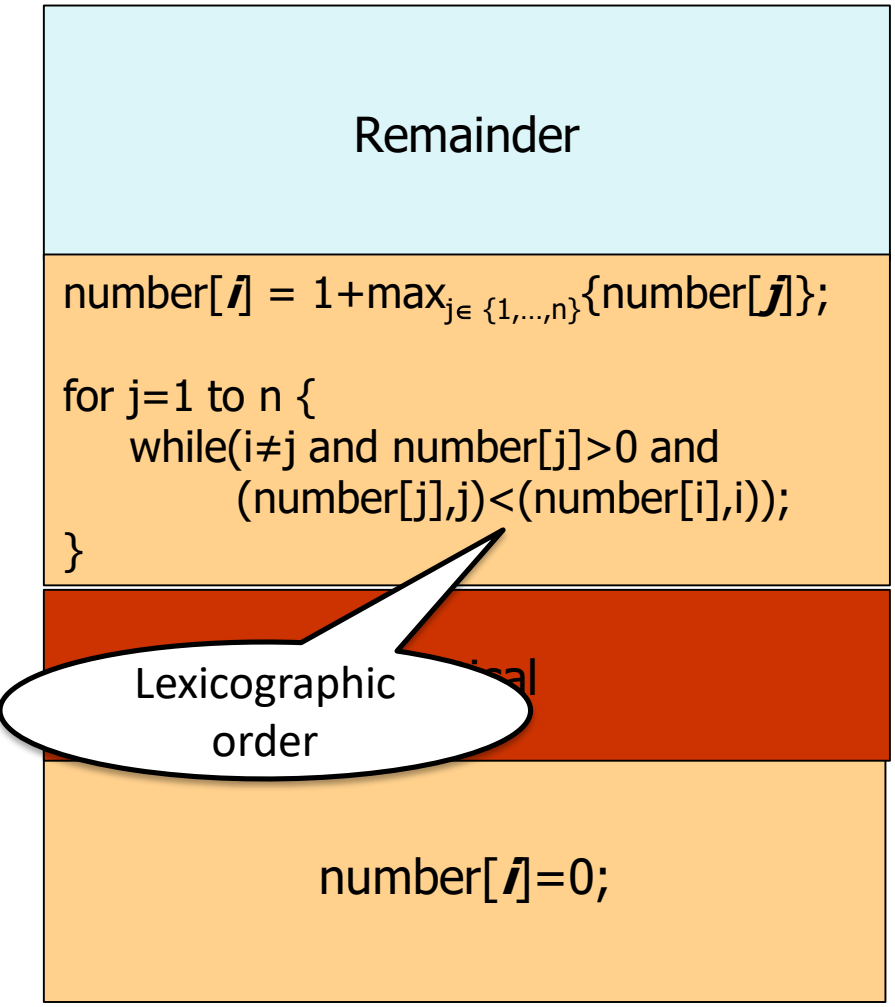
Deadlock!

Implementation 2



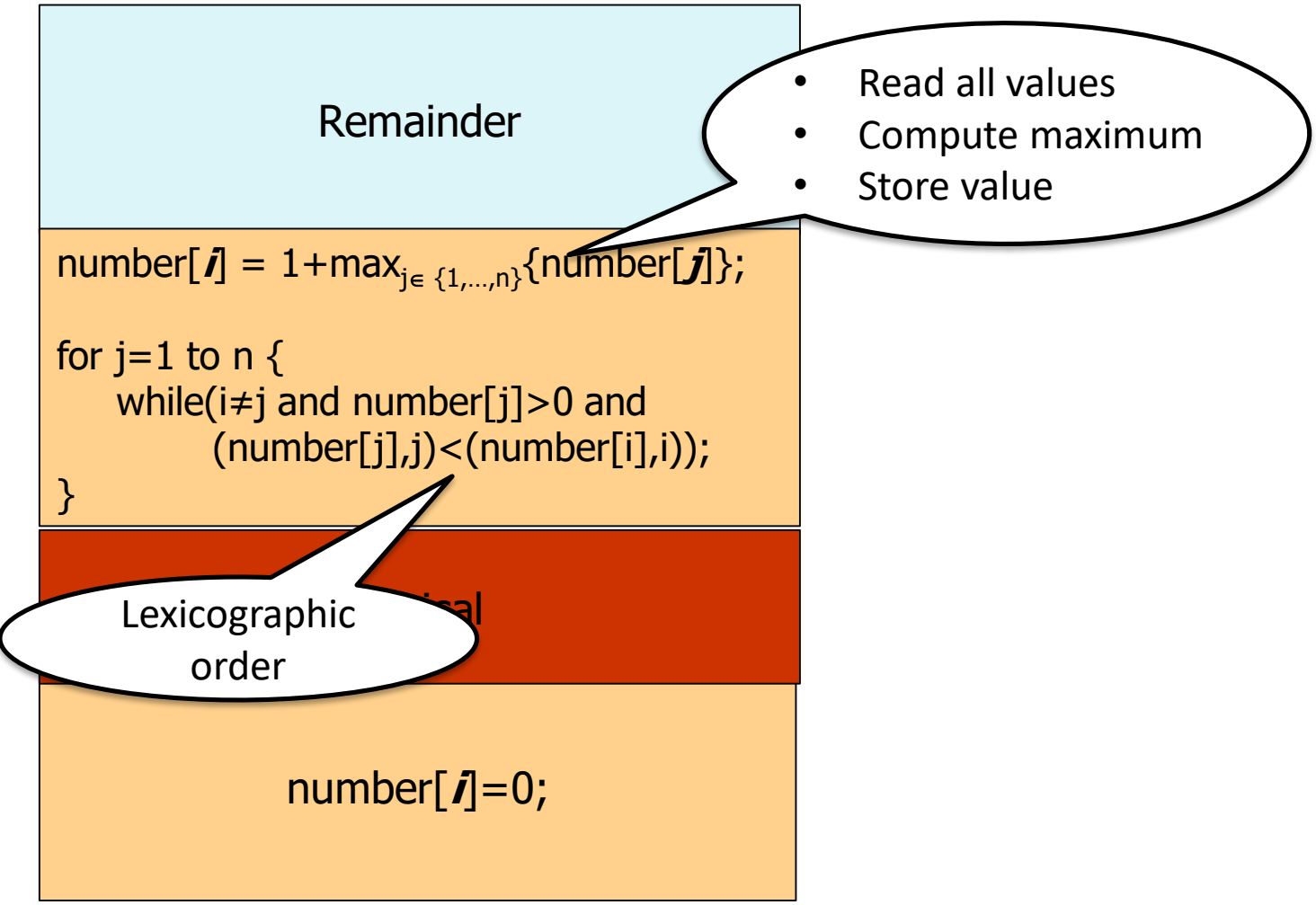
Code for Thread ***i***
(out of ***n*** threads)

Implementation 2



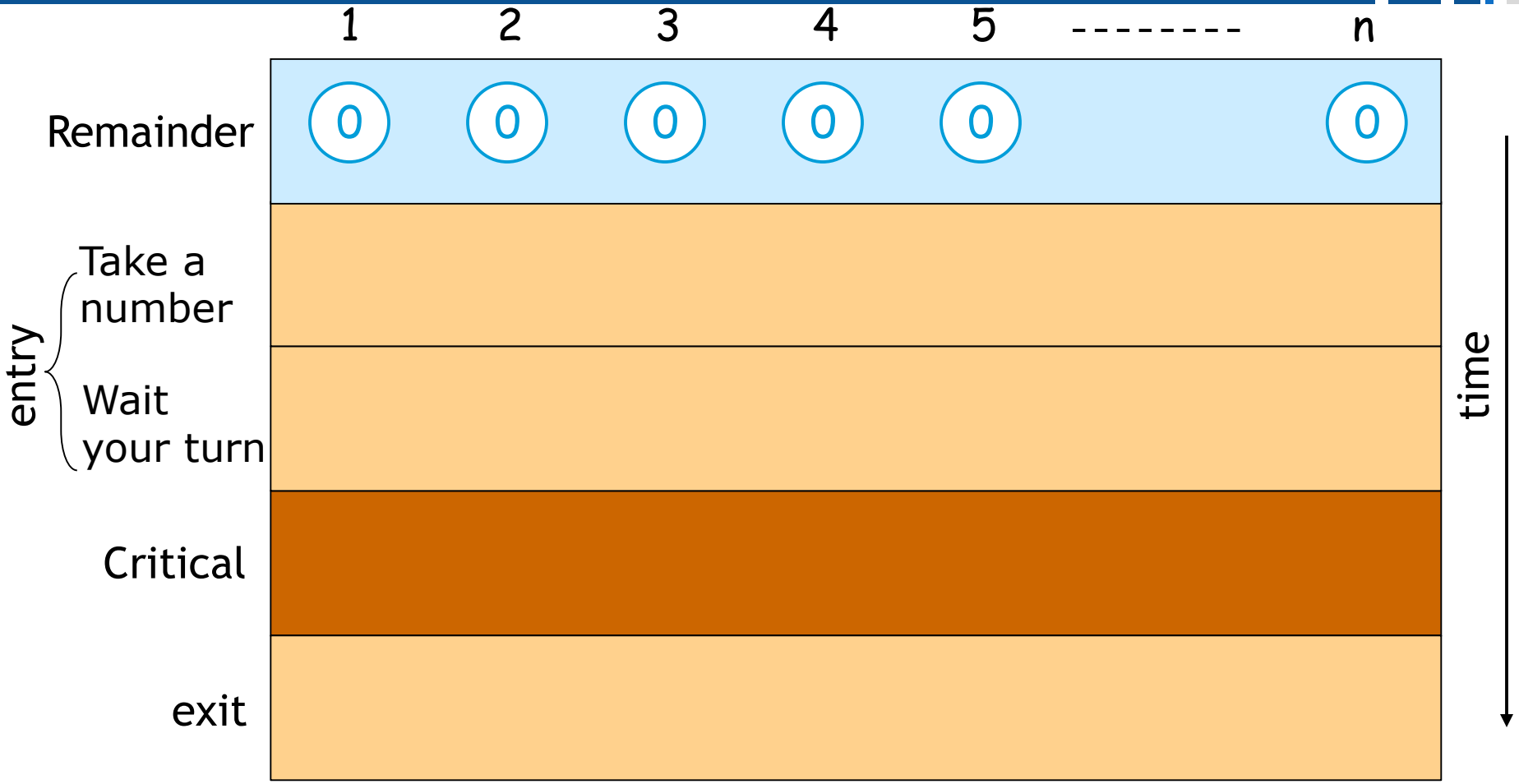
Code for Thread ***i***
(out of ***n*** threads)

Implementation 2

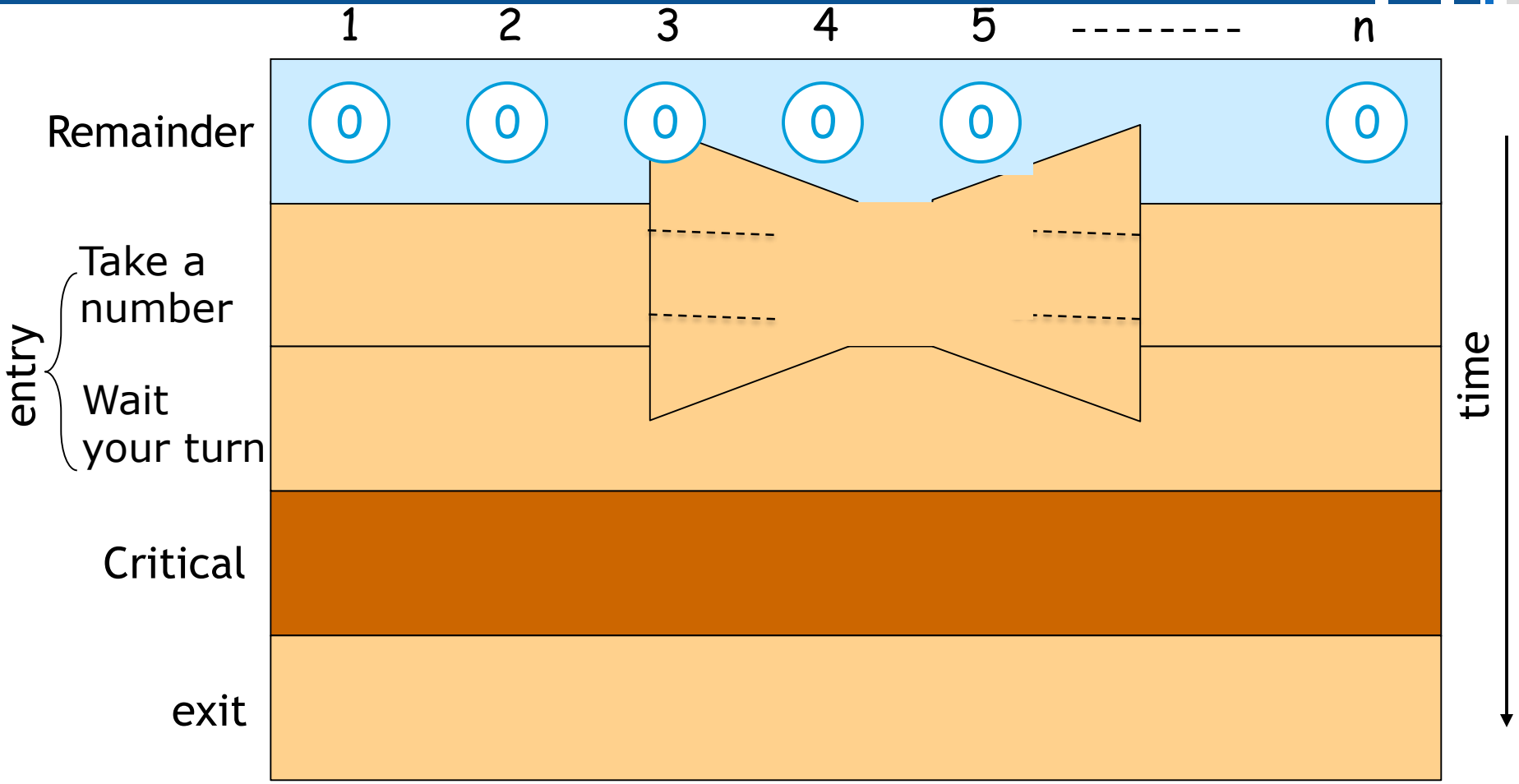


Code for Thread ***i***
(out of ***n*** threads)

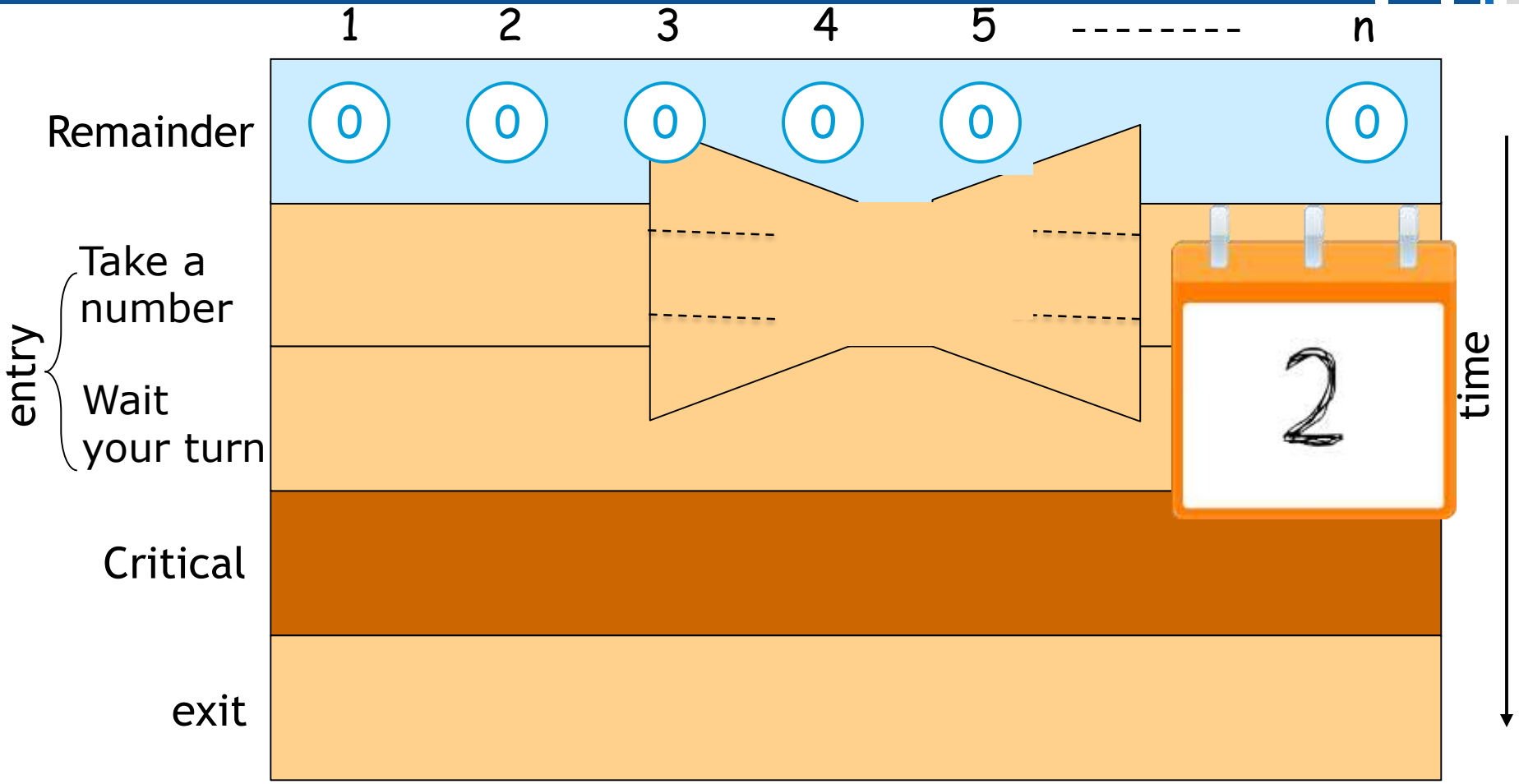
Bakery Algorithm - Implementation 2



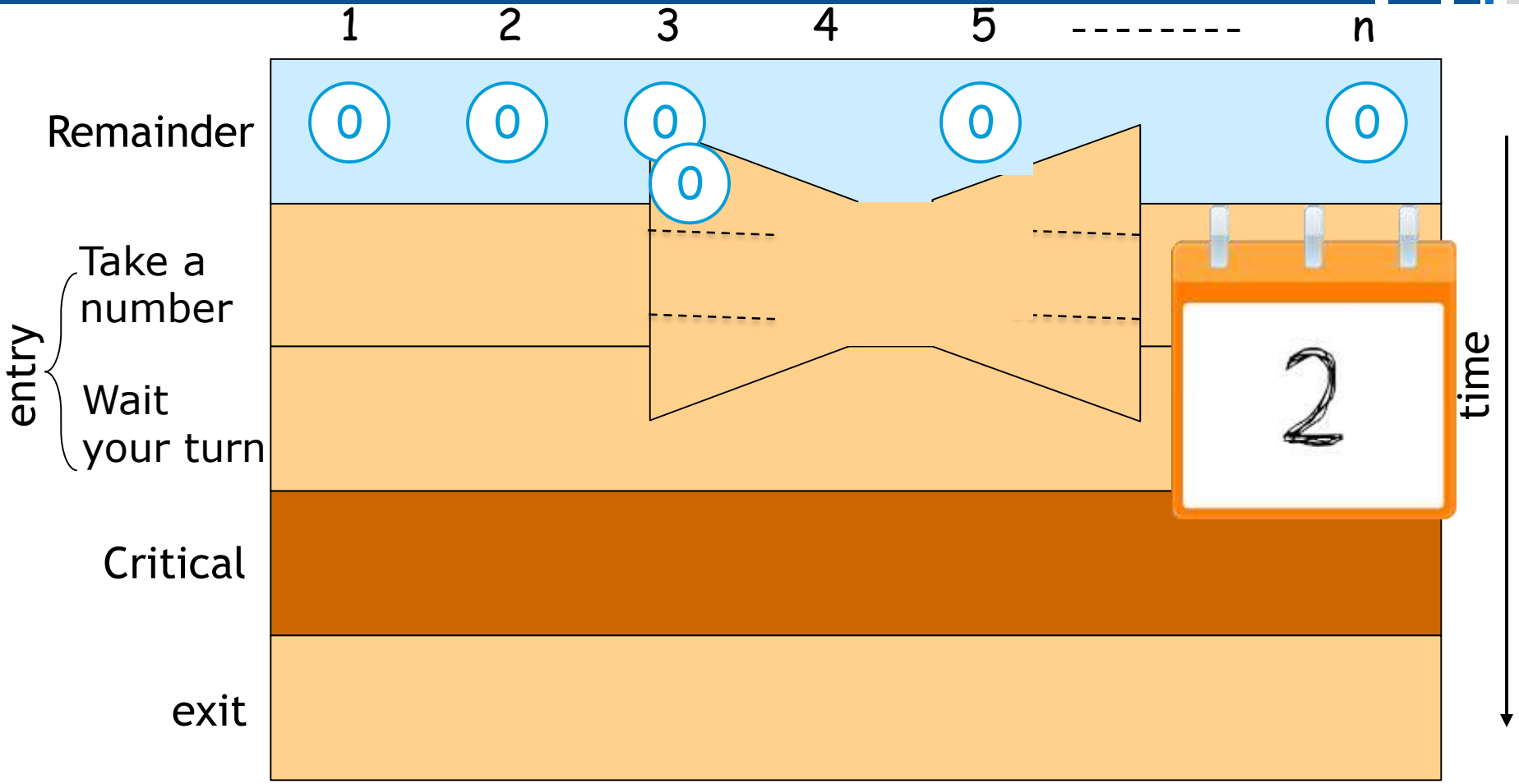
Bakery Algorithm - Implementation 2



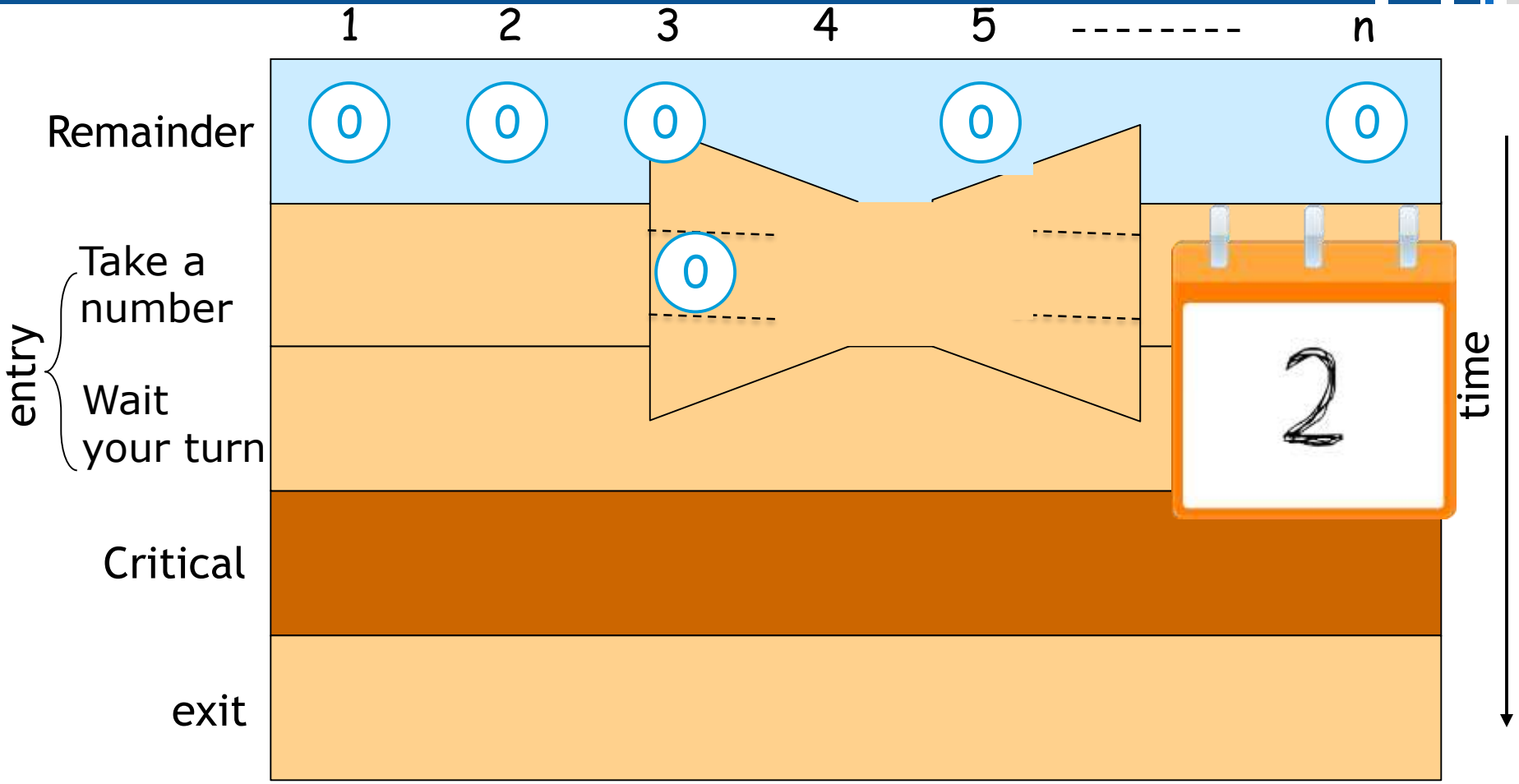
Bakery Algorithm - Implementation 2



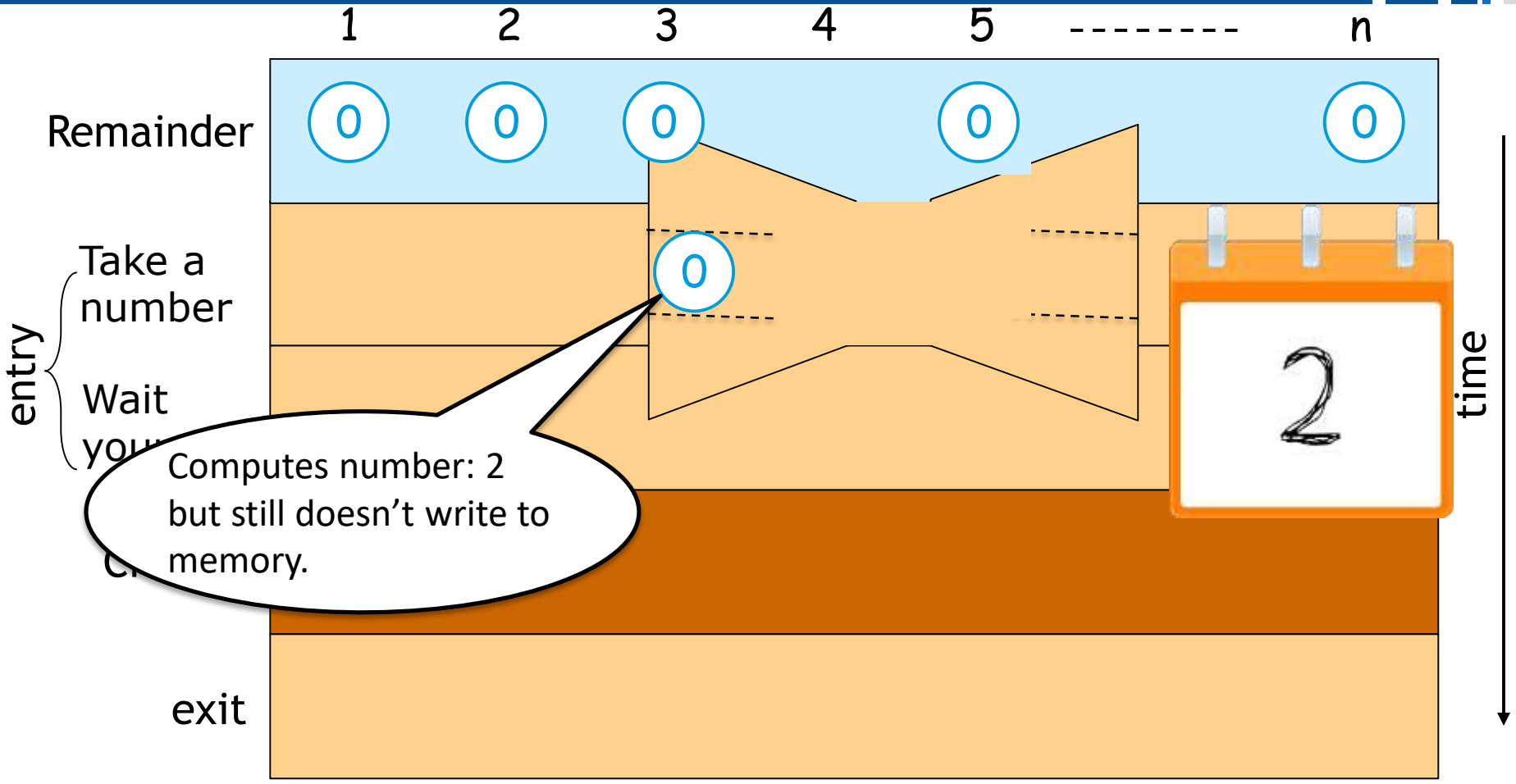
Bakery Algorithm - Implementation 2



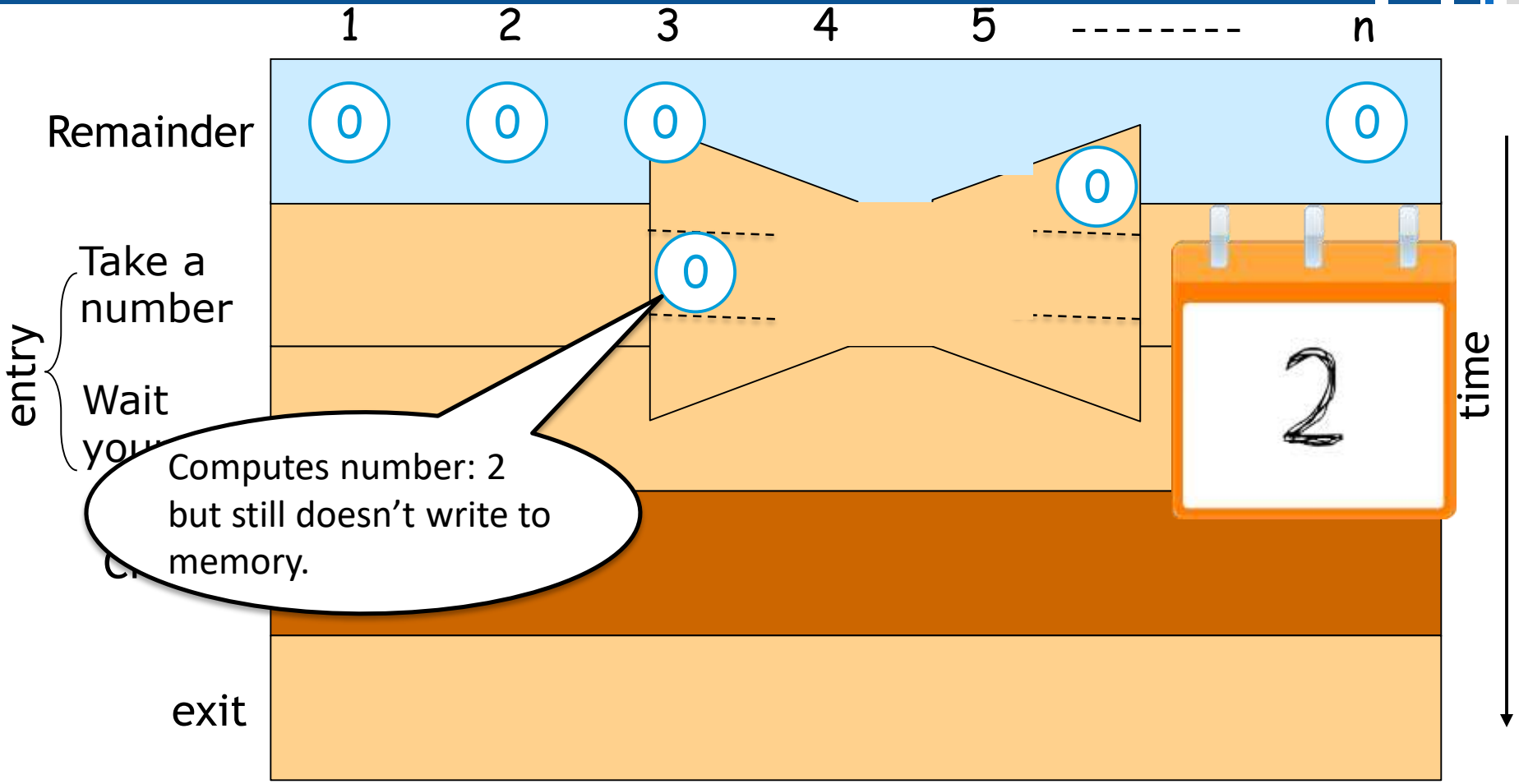
Bakery Algorithm - Implementation 2



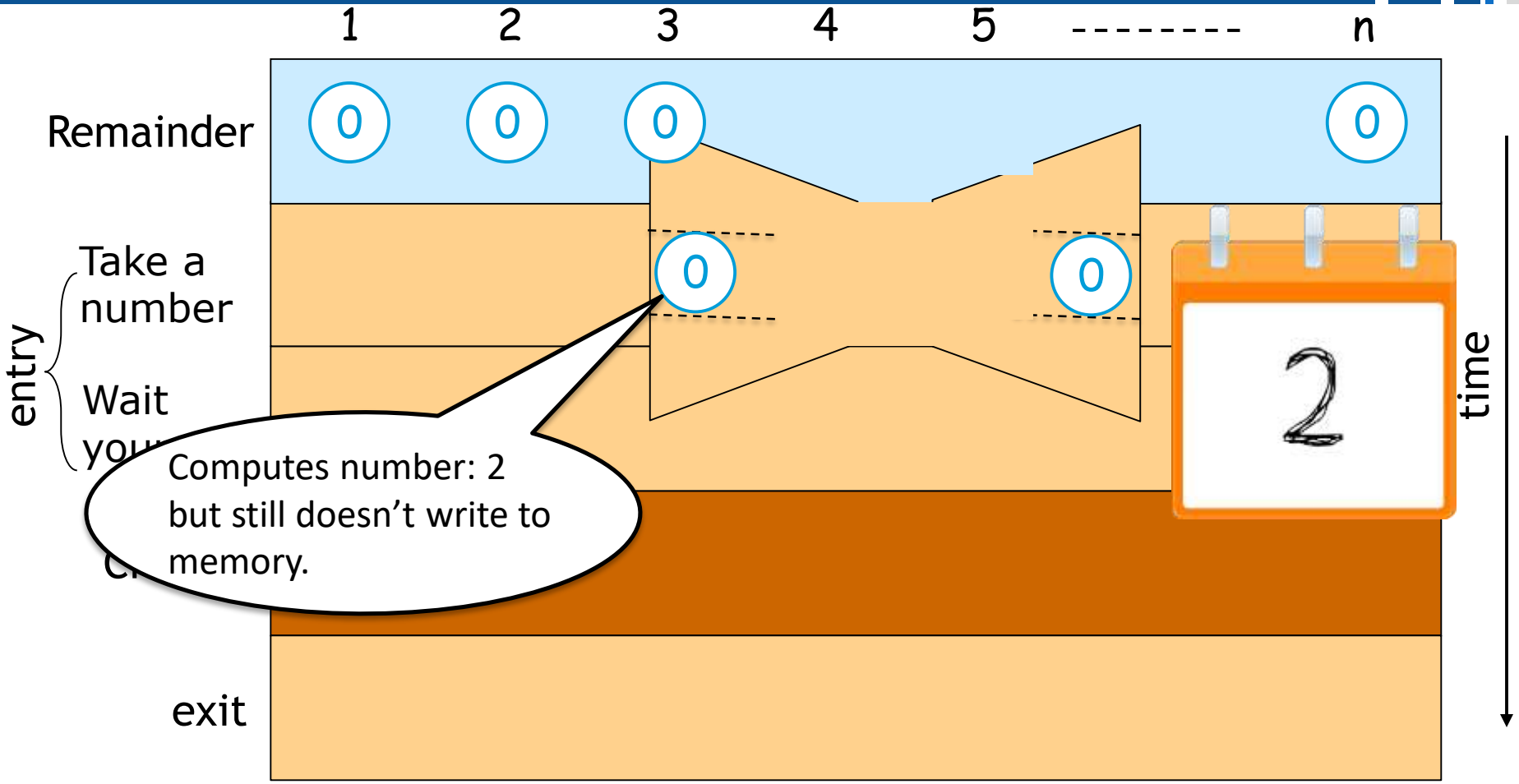
Bakery Algorithm - Implementation 2



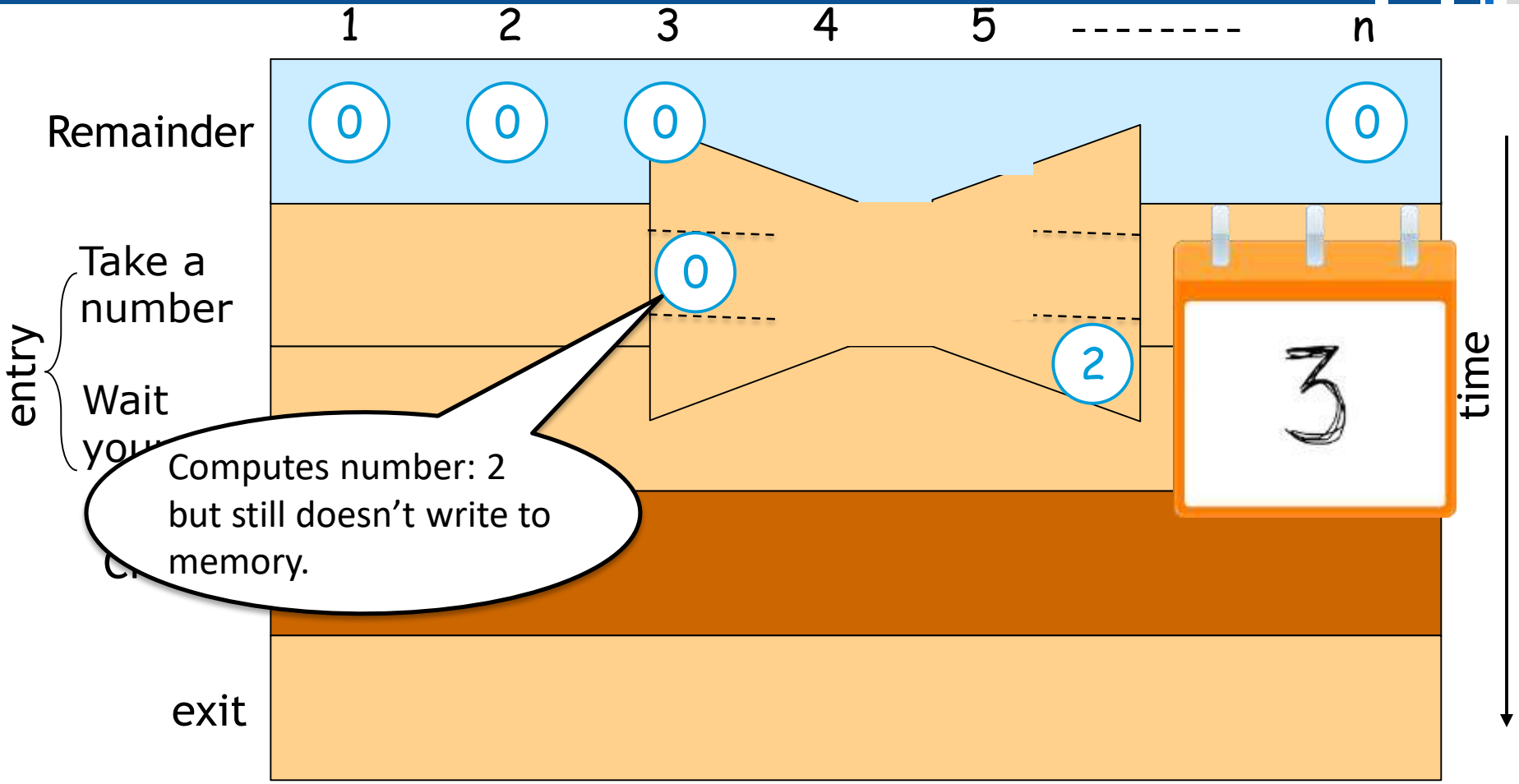
Bakery Algorithm - Implementation 2



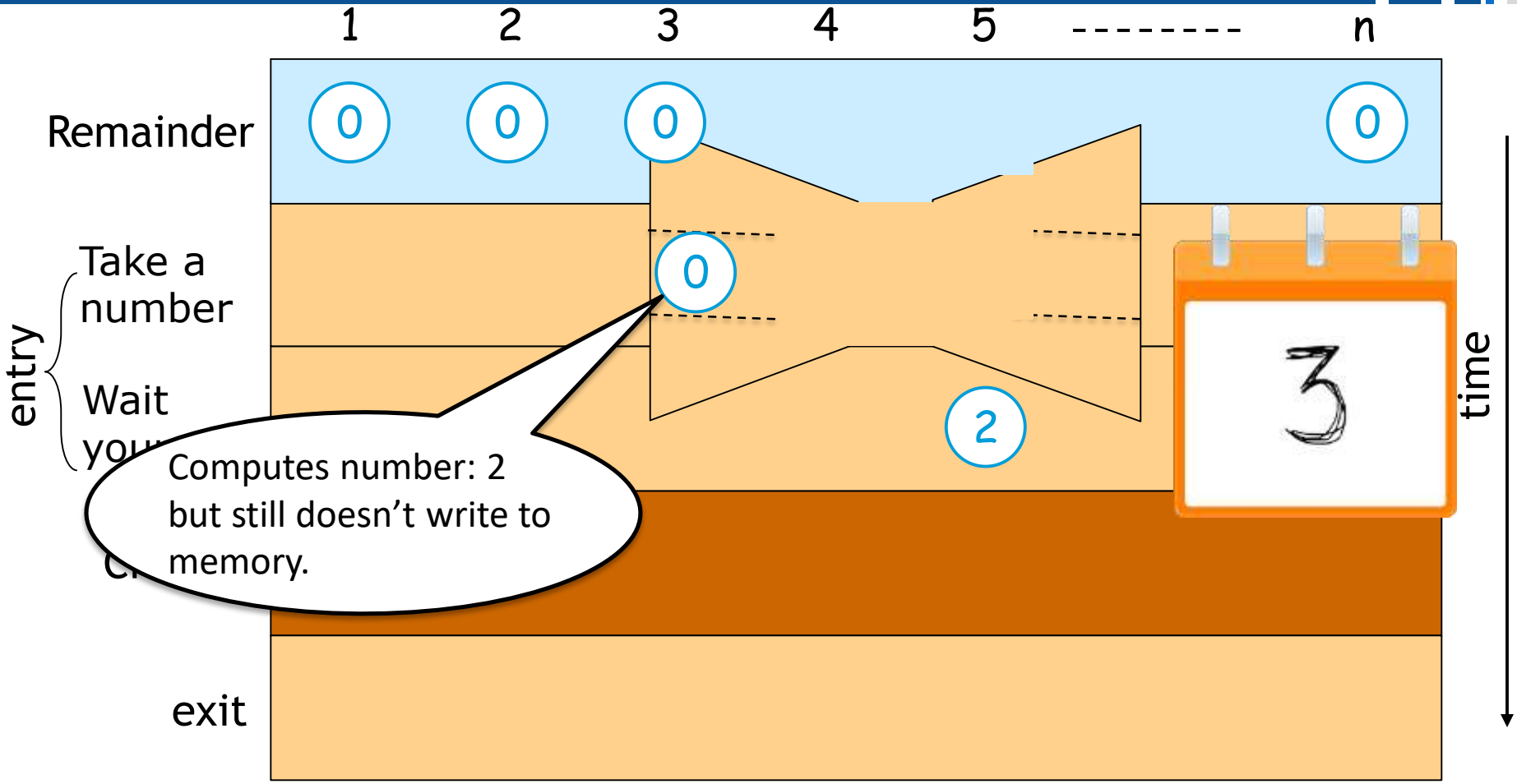
Bakery Algorithm - Implementation 2



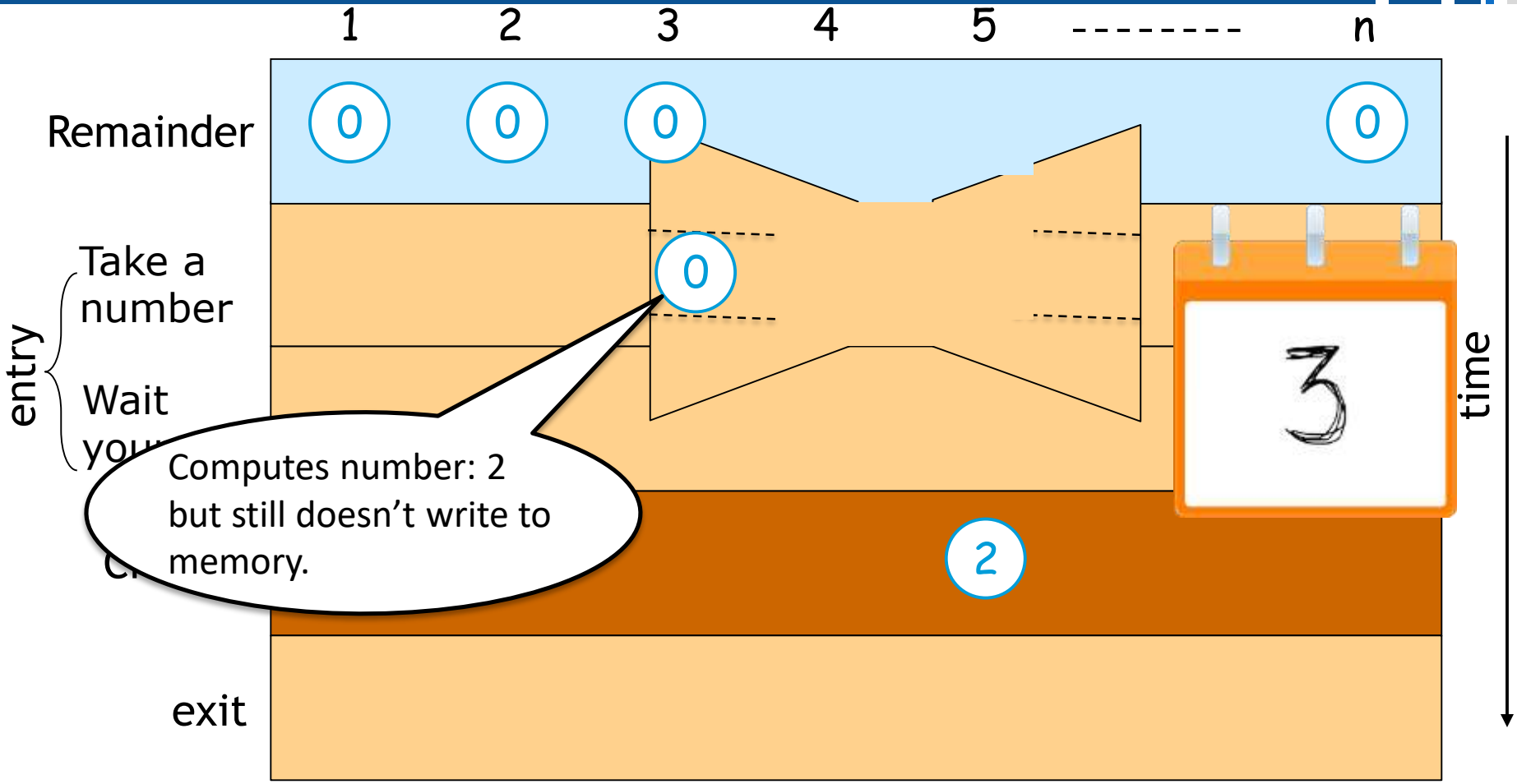
Bakery Algorithm - Implementation 2



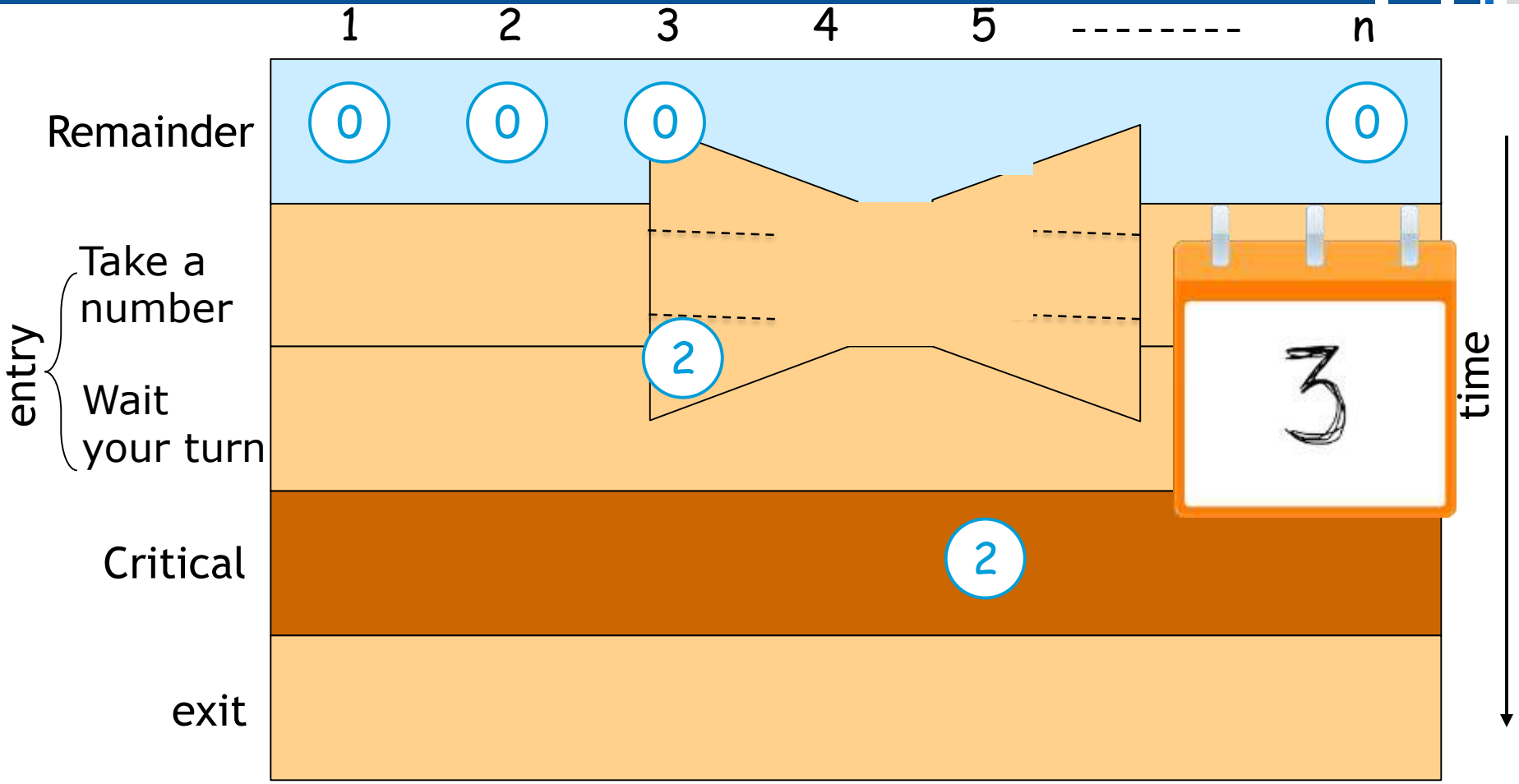
Bakery Algorithm - Implementation 2



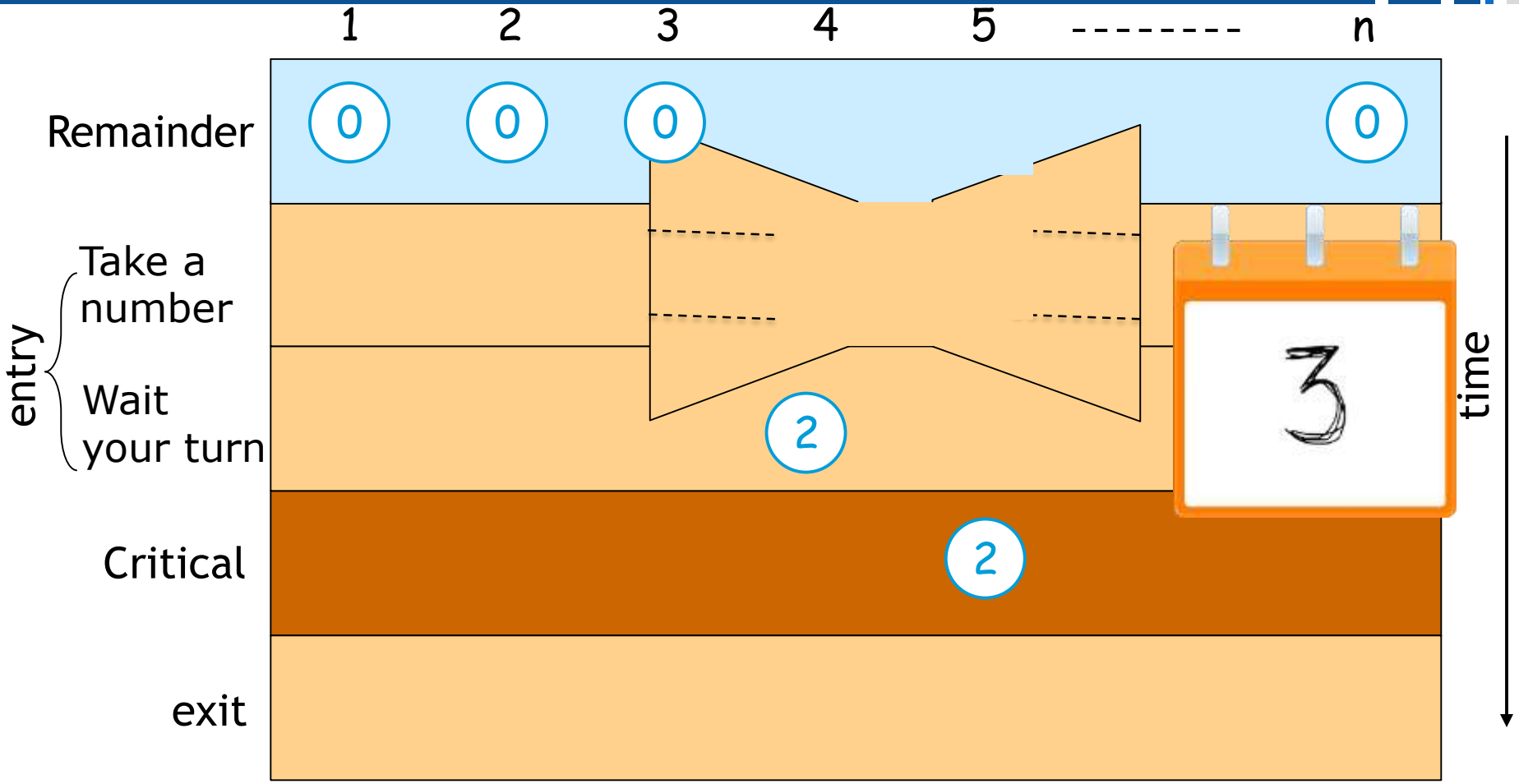
Bakery Algorithm - Implementation 2



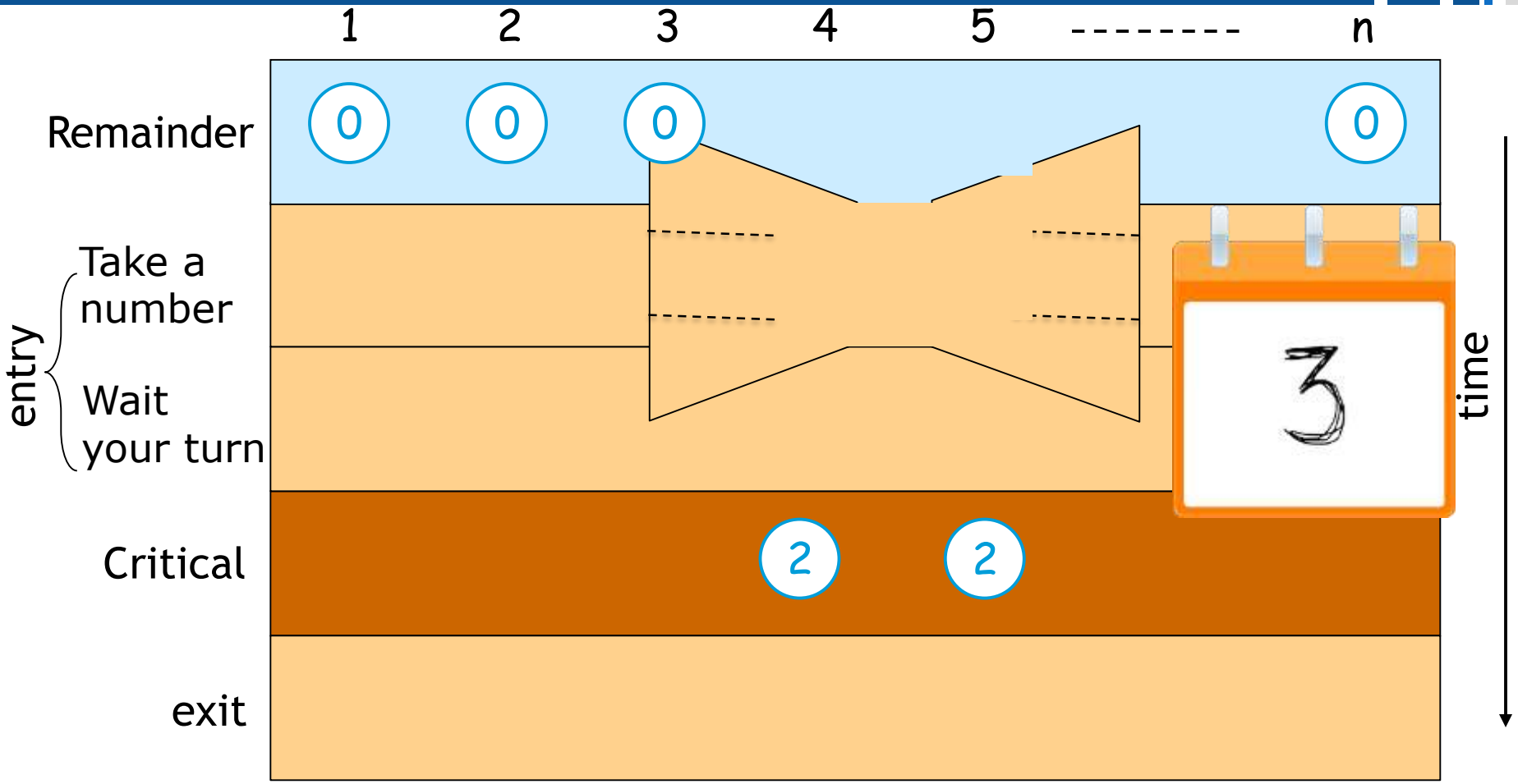
Bakery Algorithm - Implementation 2



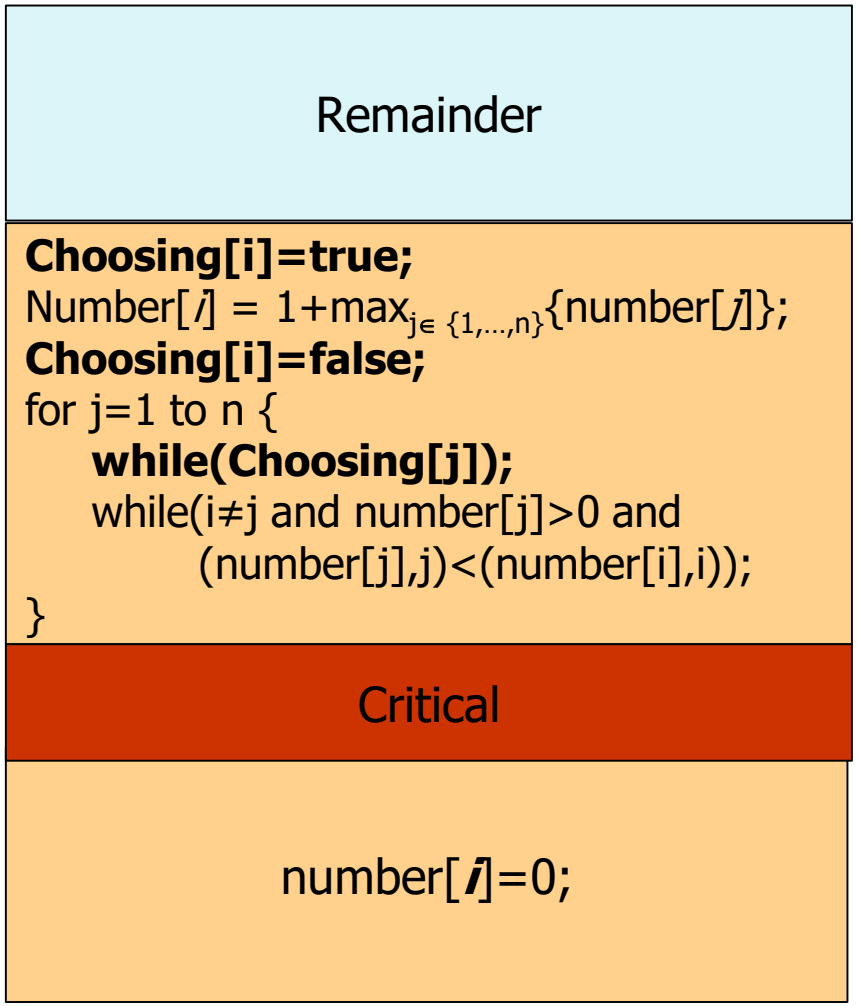
Bakery Algorithm - Implementation 2



Bakery Algorithm - Implementation 2



Correct Implementation (1974)



Code for Thread ***i***
(out of ***n*** threads)

Correct Implementation (1974)

Remainder

Choosing[i]=true;
Number[*i*] = 1+max$_{j \in \{1,\ldots,n\}}${number[*j*]};
Choosing[i]=false;
for j=1 to n {
    while(Choosing[j]);
    while(i≠j and number[j]>0 and
        (number[j],j)<(number[i],i));
}

Critical

number[***i***]=0;

Code for Thread ***i***
(out of ***n*** threads)

Also ensures a notion of **First-In-First-Out:**
If a process *i* is waiting (somewhere in the for loop) and thread *j* has not yet started the entry, then *j* will not enter the CS before *i*.

Read-Modify-Write Instructions

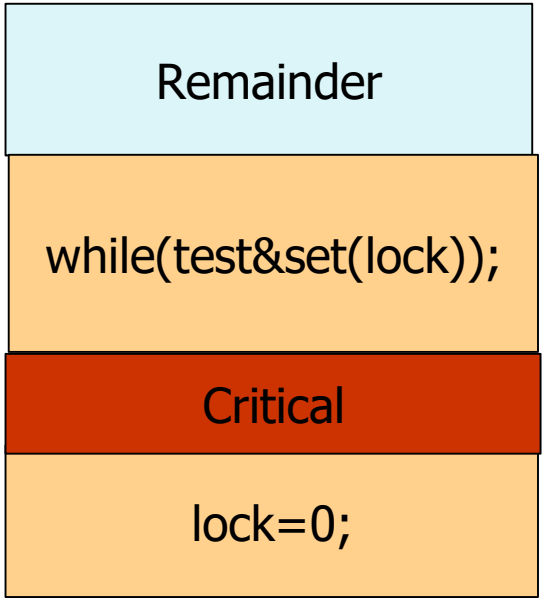
- Up until now we assume that read (load) and write (store) are atomic instructions
  - There might be a context switch between a read and its corresponding write
- Modern CPUs support in **hardware** some RMW instructions:
  - **Test&Set(&lock):**

    {i=*lock; *lock=1; return i;}
  - **Fetch&Add(&p,inc):**

    {val=*p; *p=val+inc; return val}
  - **Compare&Swap(&p,old,new):**

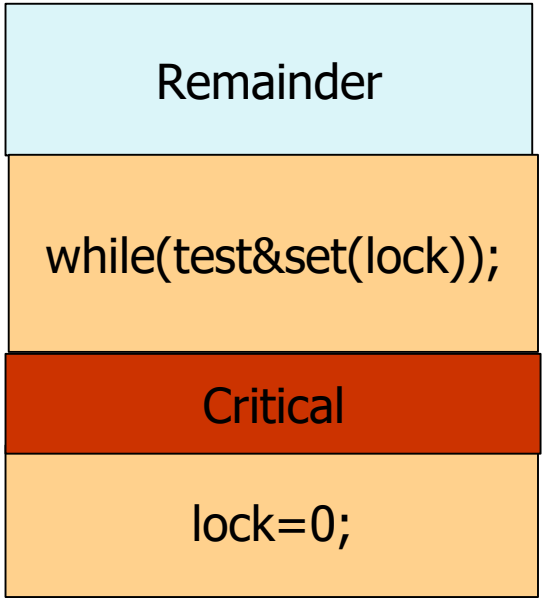
    {if(*p≠old) return false; *p=new; return true;}

Mutual Exclusion with Test&Set



Code for Thread i

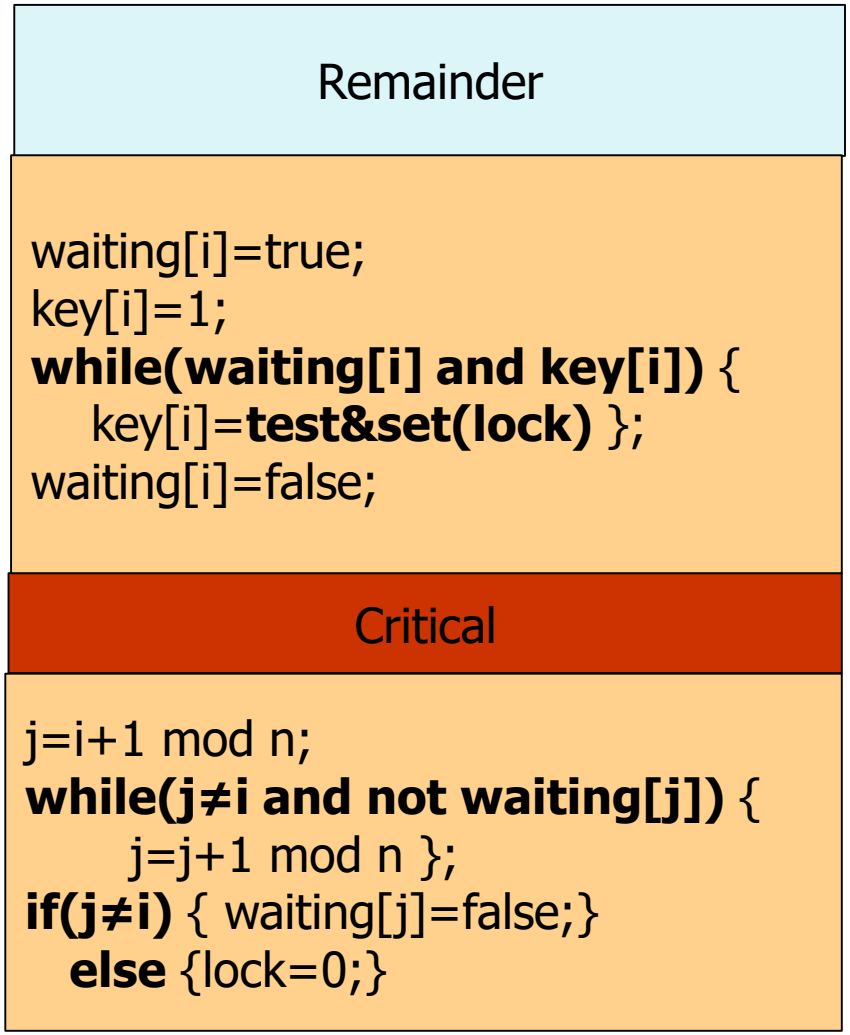
Mutual Exclusion with Test&Set



Code for Thread i

Not so fast: Starvation Freedom condition does not hold!

Burn's Algorithm



Code for Thread i out of n

Up until now…

- We saw several solutions for the **mutual exclusion** problem in different settings
- All the solutions perform busy-waiting:
  - Waste of CPU resources
- Mutual exclusion is only one synchronization problem. There are many more
  - Contention → coordination
  - Some were and will be covered in the tutorials

Next: Synchronization Primitives

- We will define **abstract data types** used to provide synchronization

Next: Synchronization Primitives

- We will define **abstract data types** used to provide synchronization
- Clear **interface** to the data type

Next: Synchronization Primitives

- We will define **abstract data types** used to provide synchronization
- Clear **interface** to the data type
- The data type (and its inner variables) cannot be accessed in any other way

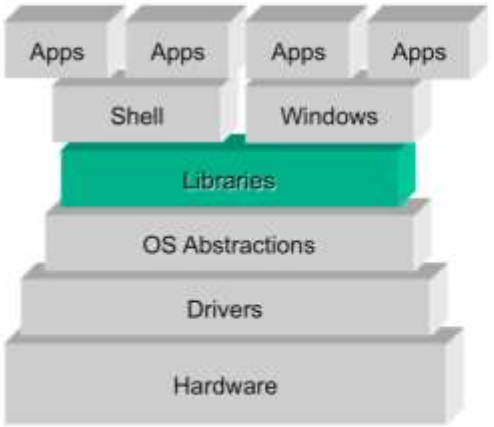
Next: Synchronization Primitives

- We will define **abstract data types** used to provide synchronization
- Clear **interface** to the data type
- The data type (and its inner variables) cannot be accessed in any other way
- Usually provide by the programming language with some support of the OS

Next: Synchronization Primitives

- We will define **abstra**
  provide synchronizati
- Clear **interface** to th
- The data type (and it
  cannot be accessed i
- Usually provide by the programming language with some support of the OS

Layers of System



Libraries

- Usually language specific
  - java.io.*, java.net.*
  - stdio.h; stdlib.h
- Often higher level abstractions

Next: Synchronization Primitives

- We will define **abstract data types** used to provide synchronization
- Clear **interface** to the data type
- The data type (and its inner variables) cannot be accessed in any other way
- Usually provide by the programming language with some support of the OS
- The abstract data type may have many implementations

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Down(S)

S.*value*= S.*value* - 1
if S.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Up(S)

S.*value*= S.*value* + 1
if S.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Init(S,v)

S.*value*= v

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Down(S)

S.*value*= S.*value* - 1
if S.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Up(S)

S.*value*= S.*value* + 1
if S.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Init(S,v)

S.*value*= v

- The operations are executed atomically
  - How? Implementation is orthogonal to the definition

First Data Type: Semaphore

- Record with two fields:

In literature, the operations are often called P(s) and V(s).
In the book, wait(s), signal(s)

Down(S)

S.*value*= S.*value* - 1
if S.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Up(S)

S.*value*= S.*value* + 1
if S.*value*≤0 then
{remove a thread T from S.L; Wakeup(T);}

Init(S,v)

S.*value*= v

- The operations are executed atomically
  - How? Implementation is orthogonal to the definition

First Data Type: Semaphore

- Record with two fields:
  - value
  - List (L)
- Two operations:

Some variations bound the maximum and minimum value (e.g., binary semaphores)

Down(S)

S.*value*= S.*value* - 1
if S.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

Up(S)

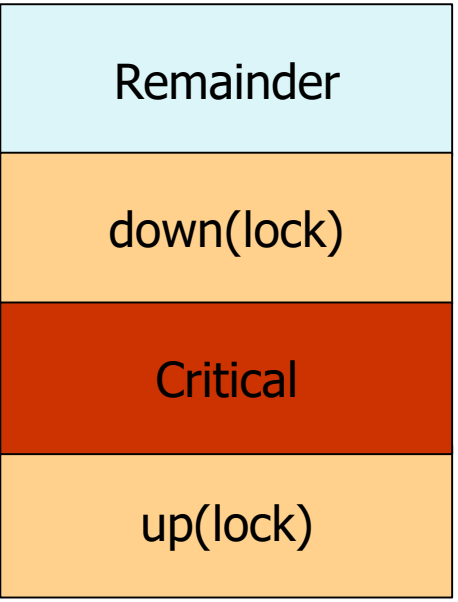
S.*value*= S.*value* + 1
if S.*value*≤0 then
{remove a thread T from S.L; Wakeup(T);}

Init(S,v)

S.*value*= v

- The operations are executed atomically
  - How? Implementation is orthogonal to the definition

Mutual Exclusion w/ Semaphore

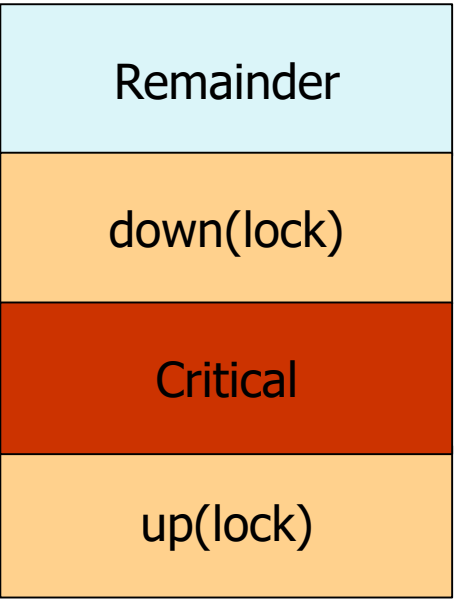


Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

Mutual Exclusion w/ Semaphore



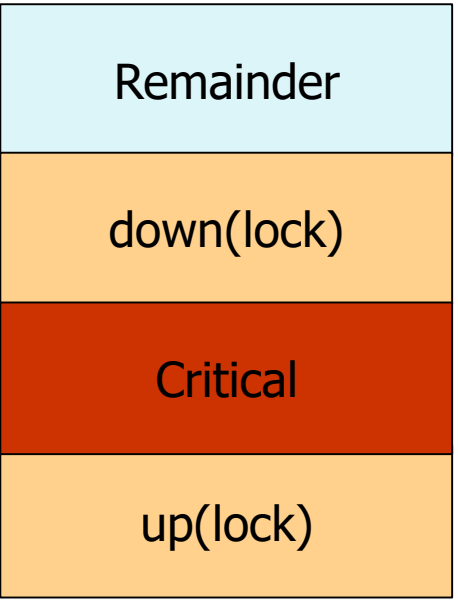
Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1

Mutual Exclusion w/ Semaphore



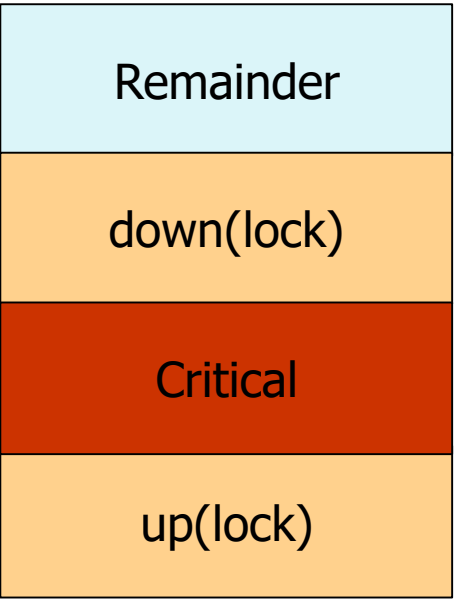
Code for Thread i

```
lock.value= lock.value - 1
if  lock.value < 0 then
 { add this thread to S.L;
    sleep();}
```

```
lock.value= lock.value + 1
If lock.value≤0 then
{remove a thread T from
     S.L;   Wakeup(T);}
```

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting

Mutual Exclusion w/ Semaphore



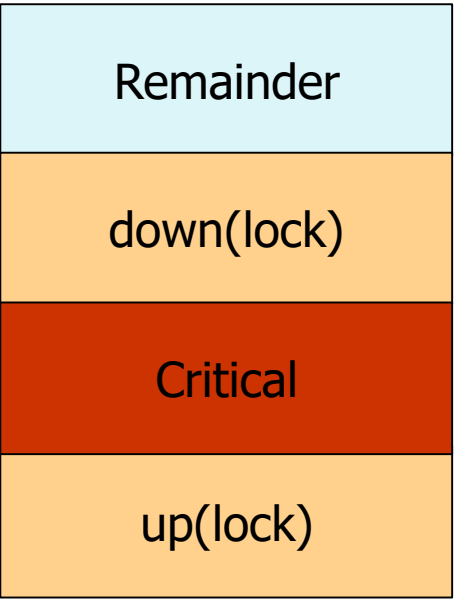
Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting
- All properties hold
  - If L is a FIFO queue

Mutual Exclusion w/ Semaphore



Code for Thread i

lock.*value*= lock.*value* - 1
if lock.*value* < 0 then
{ add this thread to S.*L*;
sleep();}

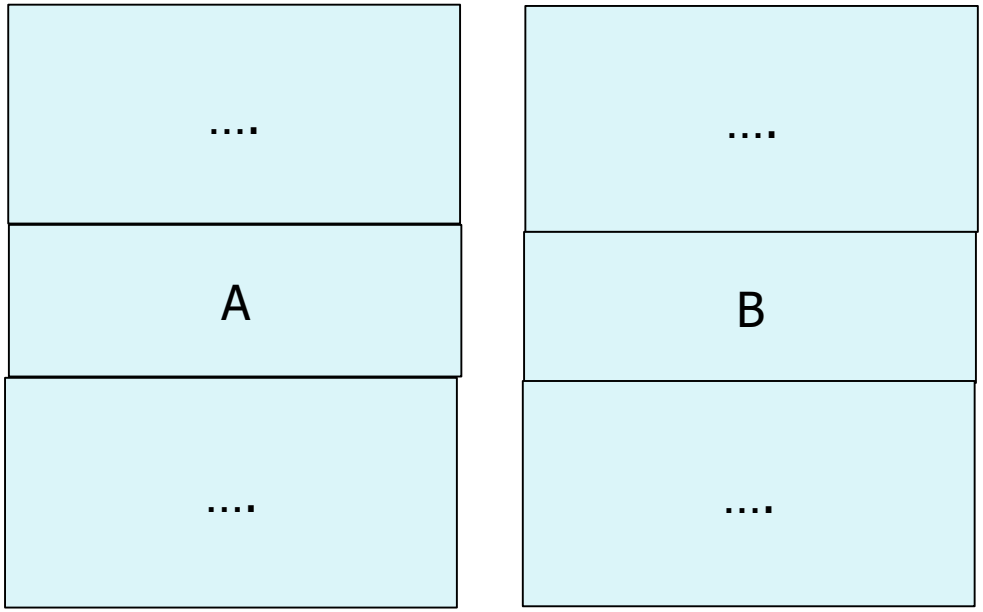
lock.*value*= lock.*value* + 1
If lock.*value*≤0 then
{remove a thread T from
S.L; Wakeup(T);}

- Assume **lock** is a shared semaphore
  - Initially **lock** is 1
- Value:
  - Cannot be more than 1
  - 0: one process in Critical Section
  - (-x): x processes are waiting
- All properties hold
  - If L is a FIFO queue

0/1 (binary) semaphore suffices → Binary semaphores data type are often called Mutex objects

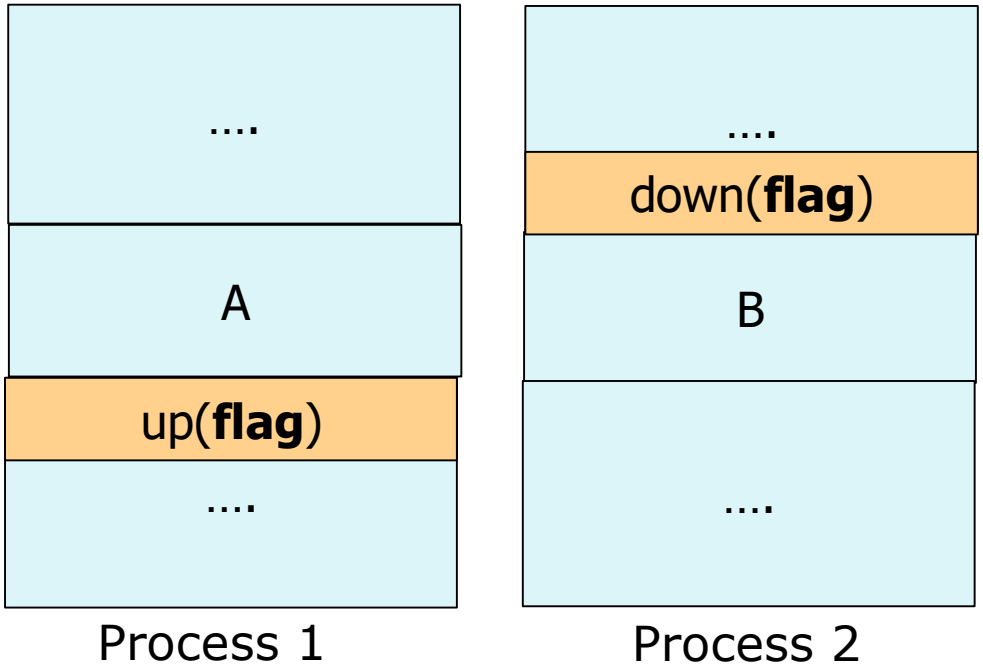
“Execute B after A”

- A different synchronization problem
  - Coordination rather than contention
- One process needs to execute code A, before another process executes code B



“Execute B after A”

- A different synchronization problem
  - Coordination rather than contention
- One process needs to execute code A, before another process executes code B



Semaphore **flag**, initialized to 0

Another Problem: Moving Money Between Accounts

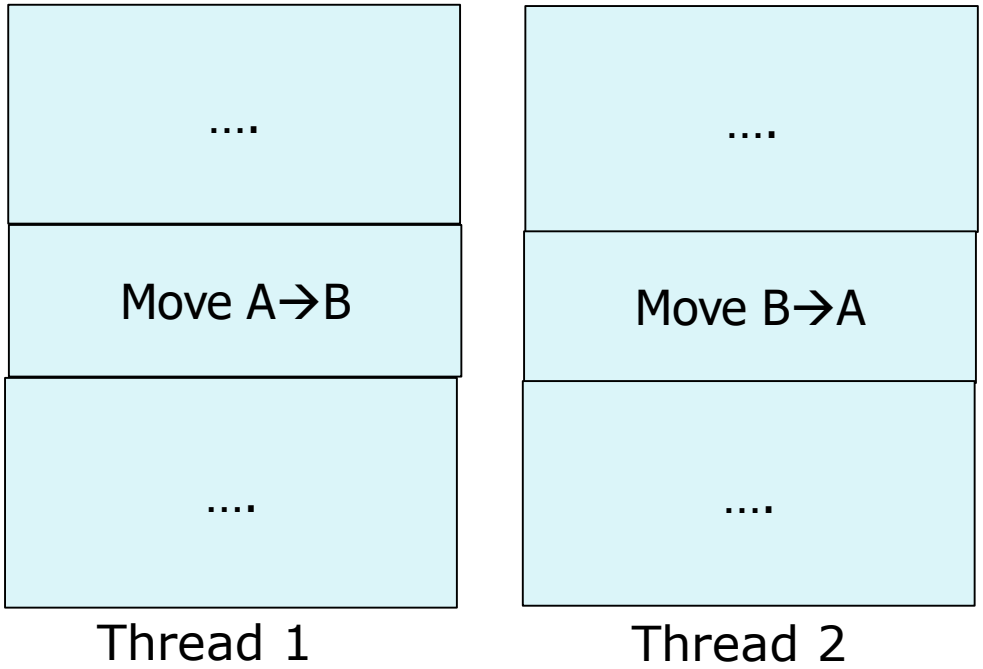
Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)

Another Problem: Moving Money Between Accounts

Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

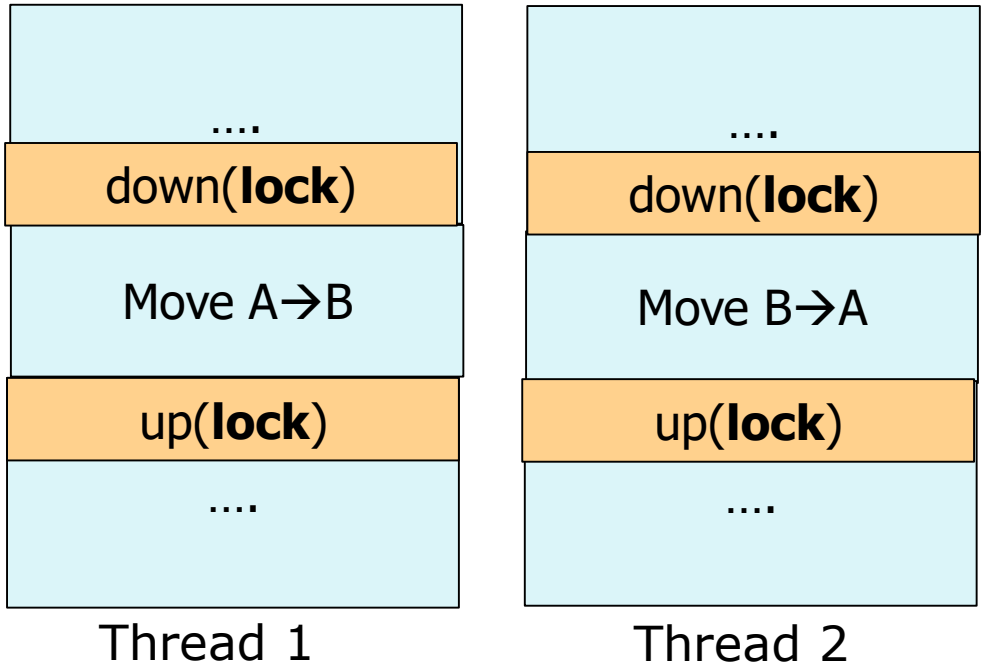
If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)



Another Problem: Moving Money Between Accounts

Thread 1 transfers money from account A to B, Thread 2 transfers money from B to A.

If executed simultaneously, some errors might occur (e.g., Thread 1 executes A--, while Thread 2 executes A++)

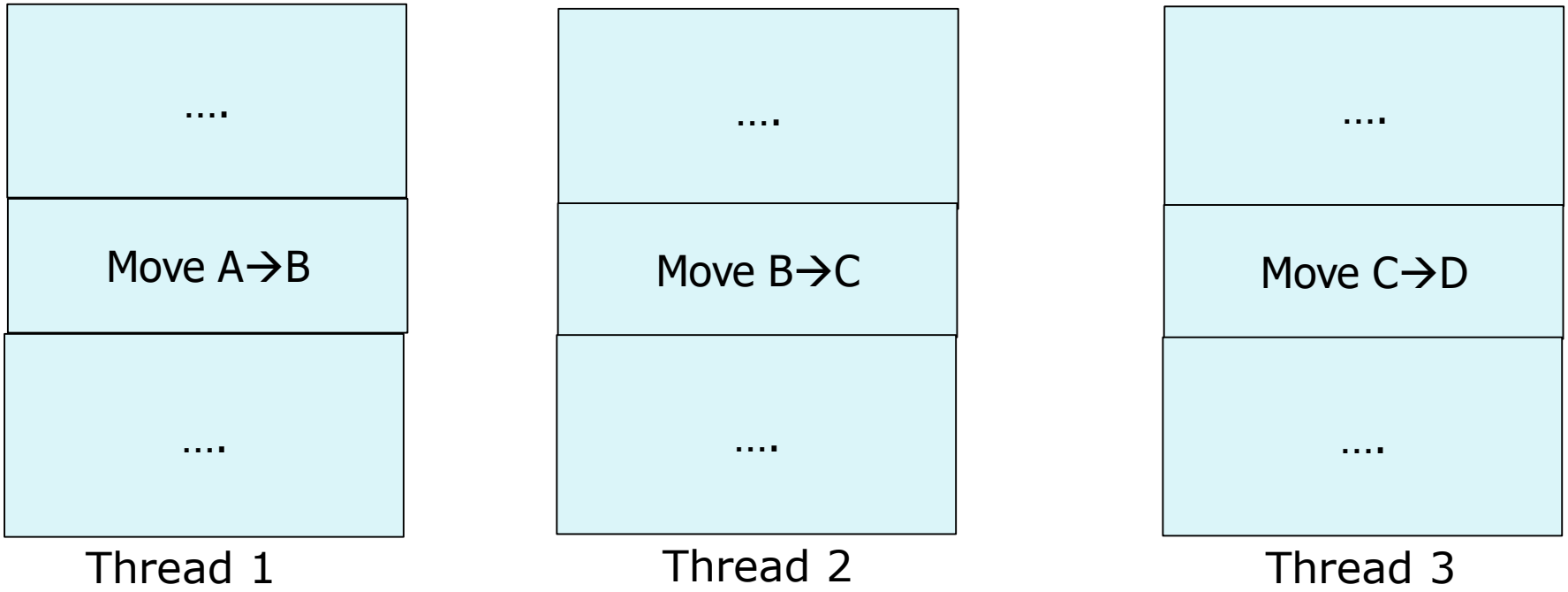


Semaphore **lock**, initialized to 1

Mutual exclusion, where the Move is the critical section

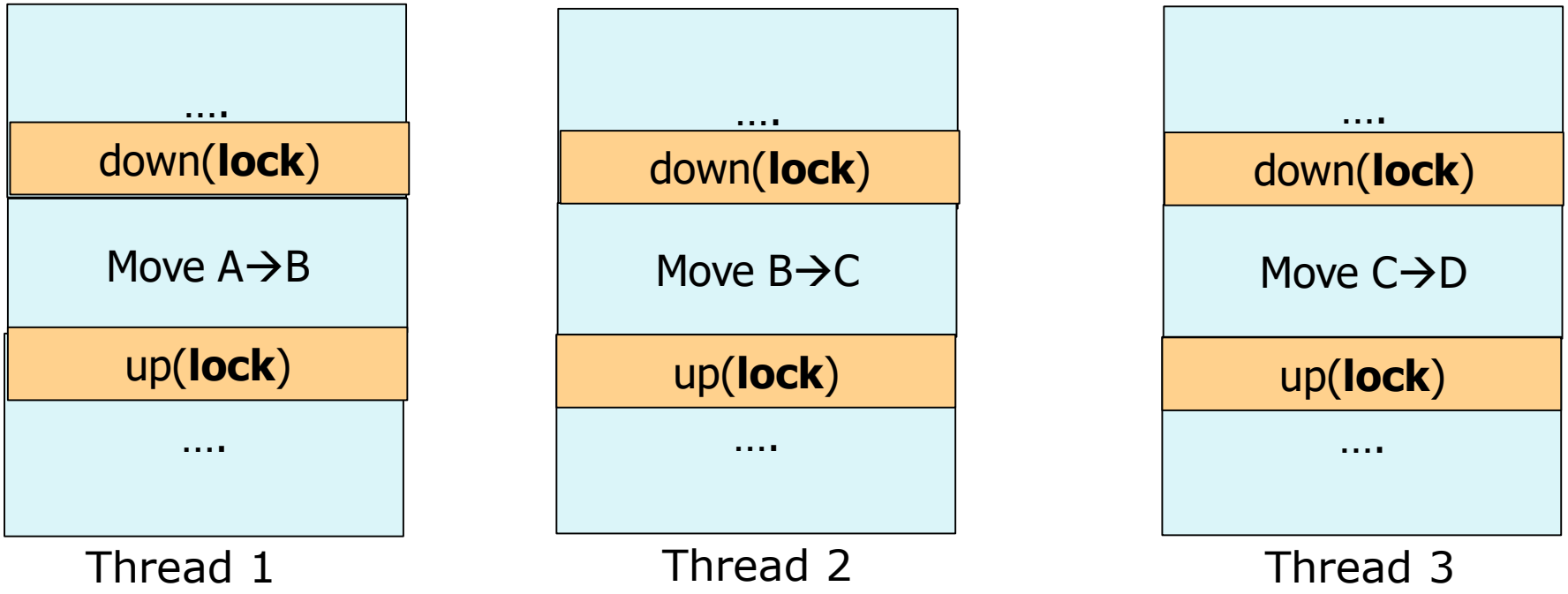
Another Problem: Moving Money Between Accounts

- Three Threads



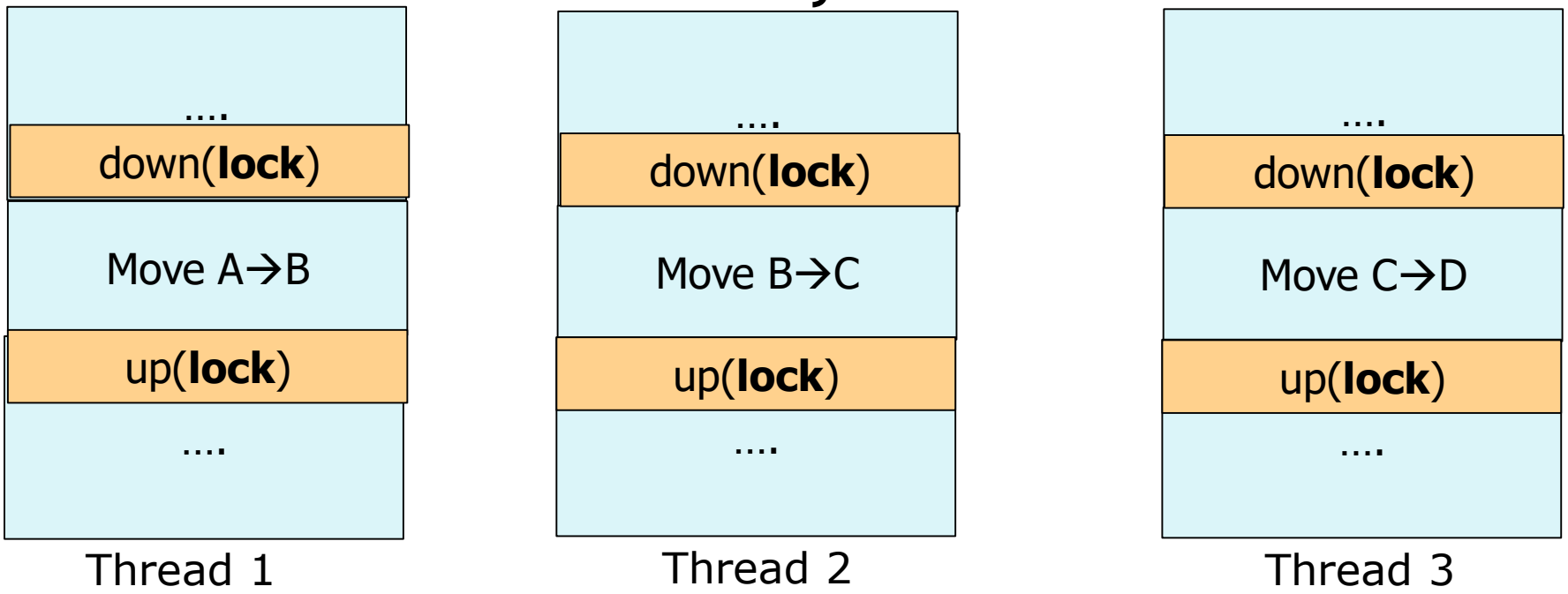
Another Problem: Moving Money Between Accounts

- Three Threads
- Solution 1: mutual exclusion



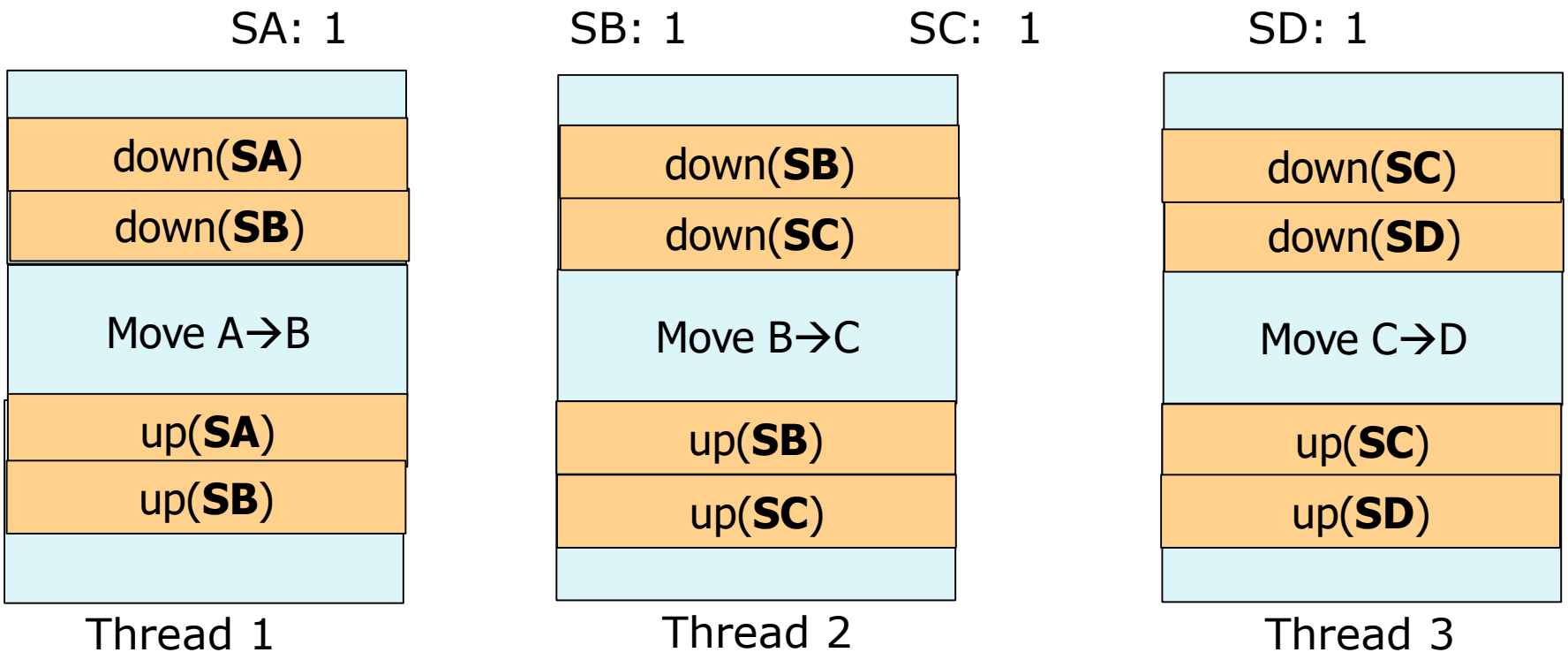
Another Problem: Moving Money Between Accounts

- Three Threads
- Solution 1: mutual exclusion
  - But why Thread 1 and Thread 3 cannot be executed concurrently?



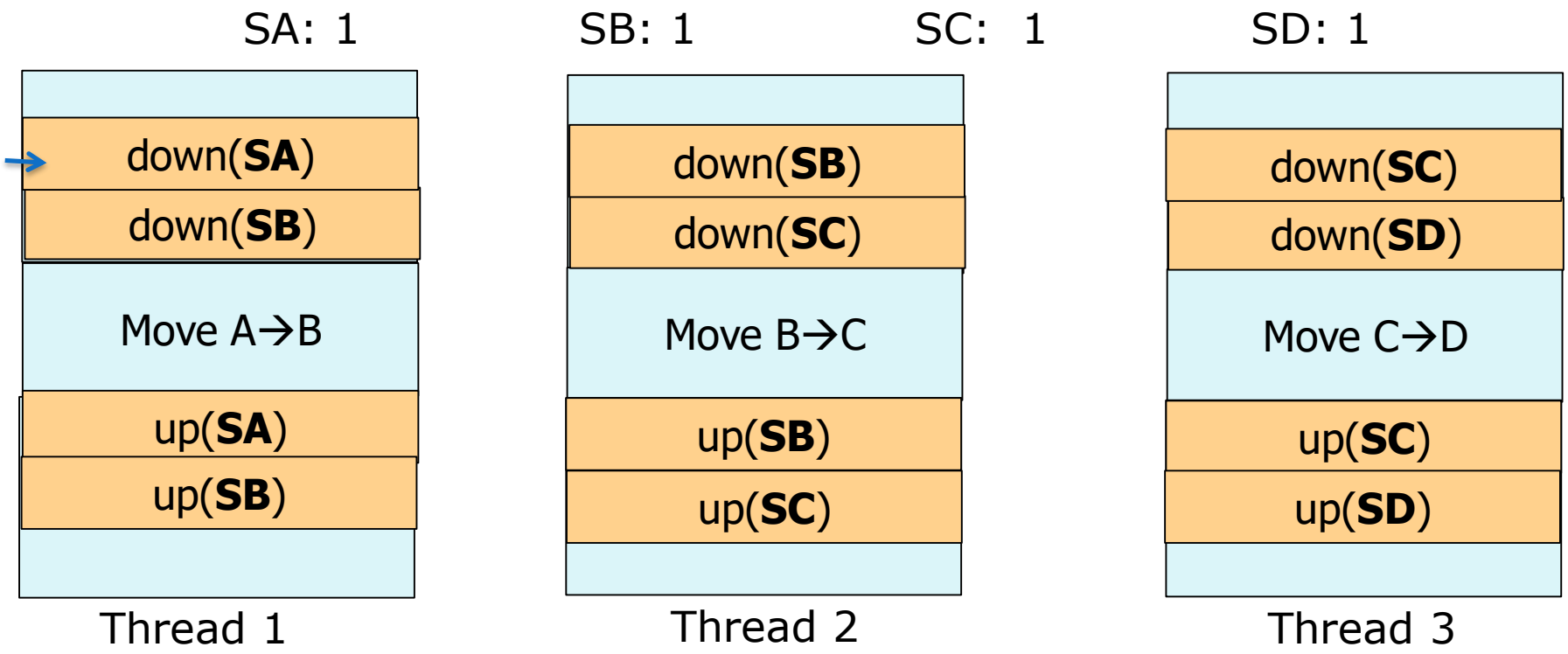
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



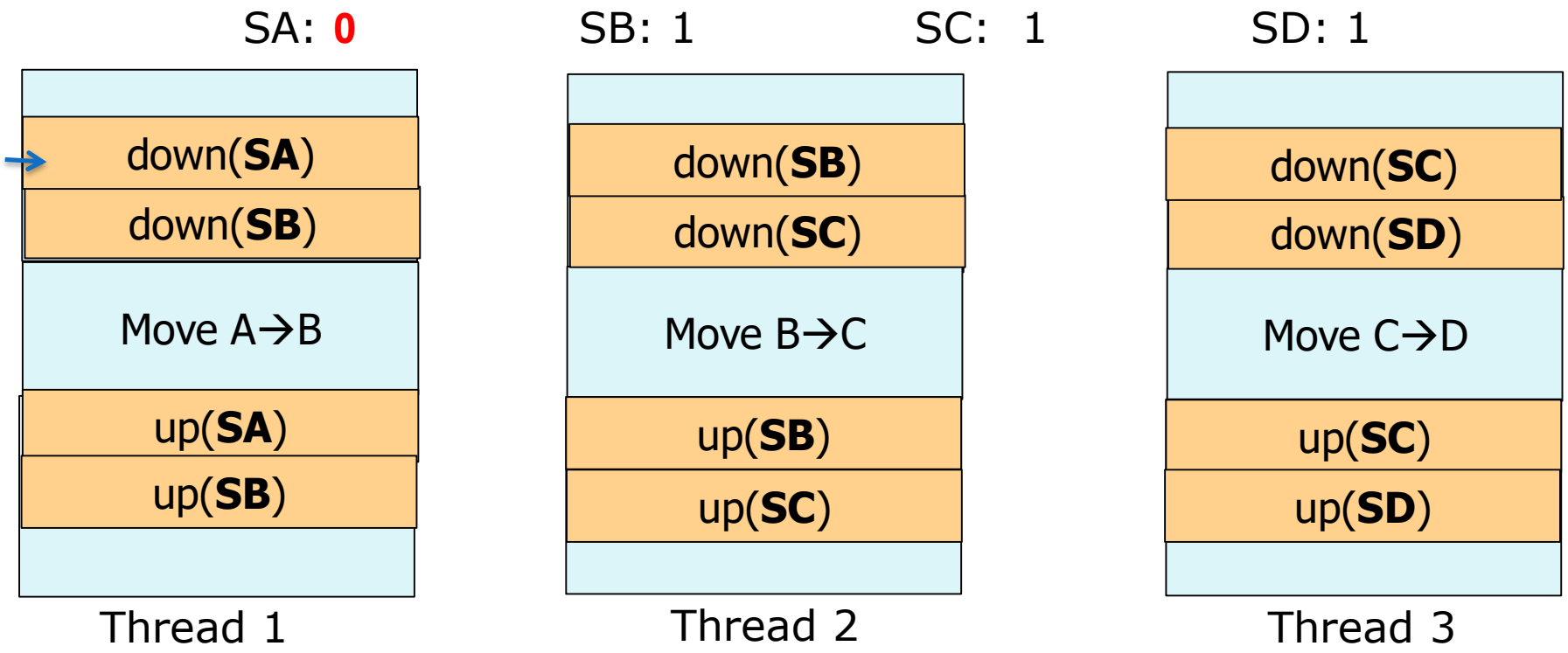
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



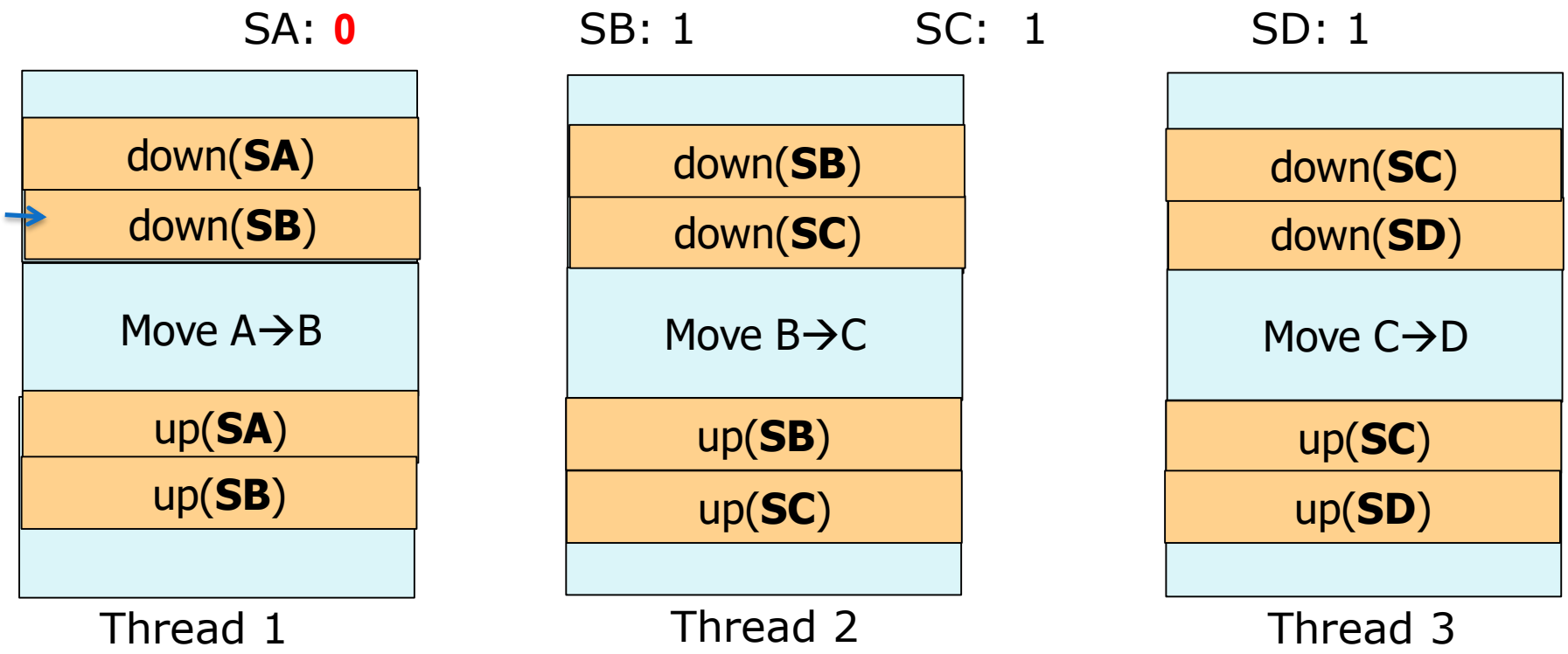
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



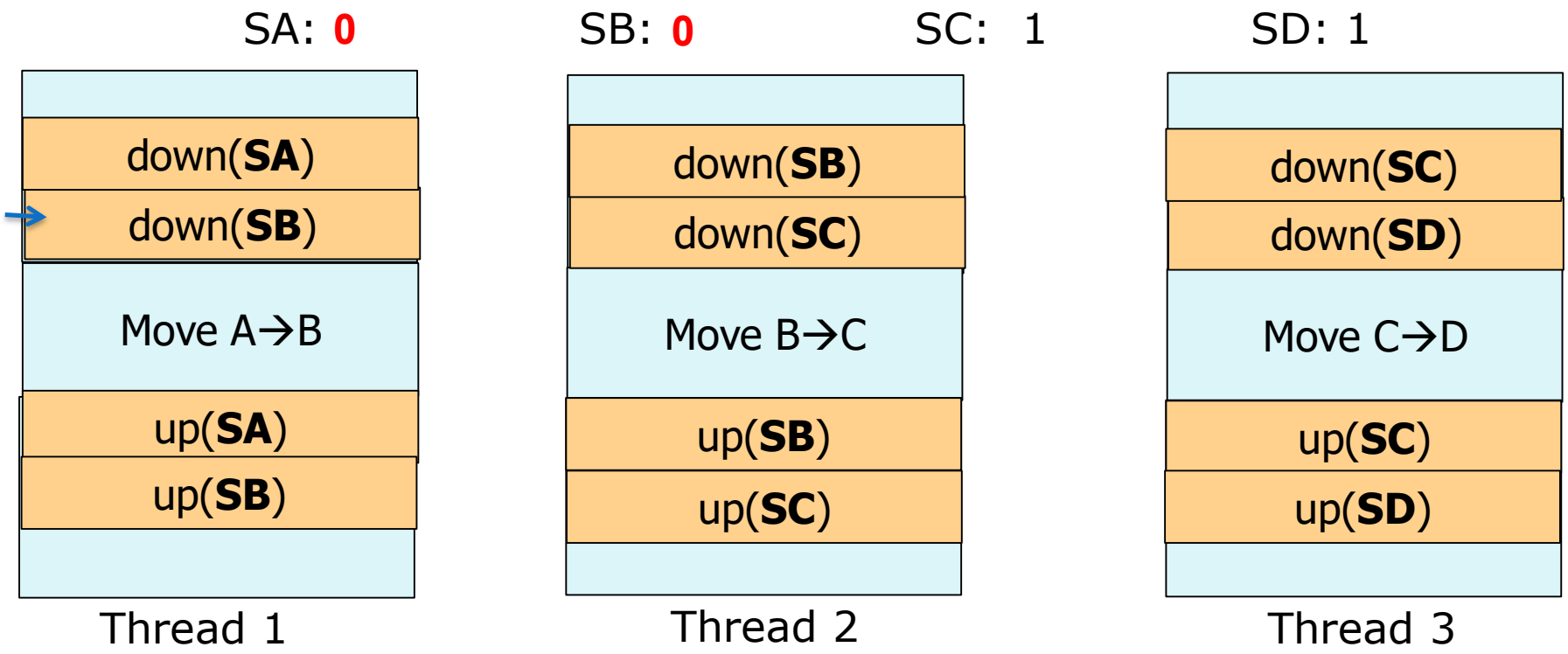
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



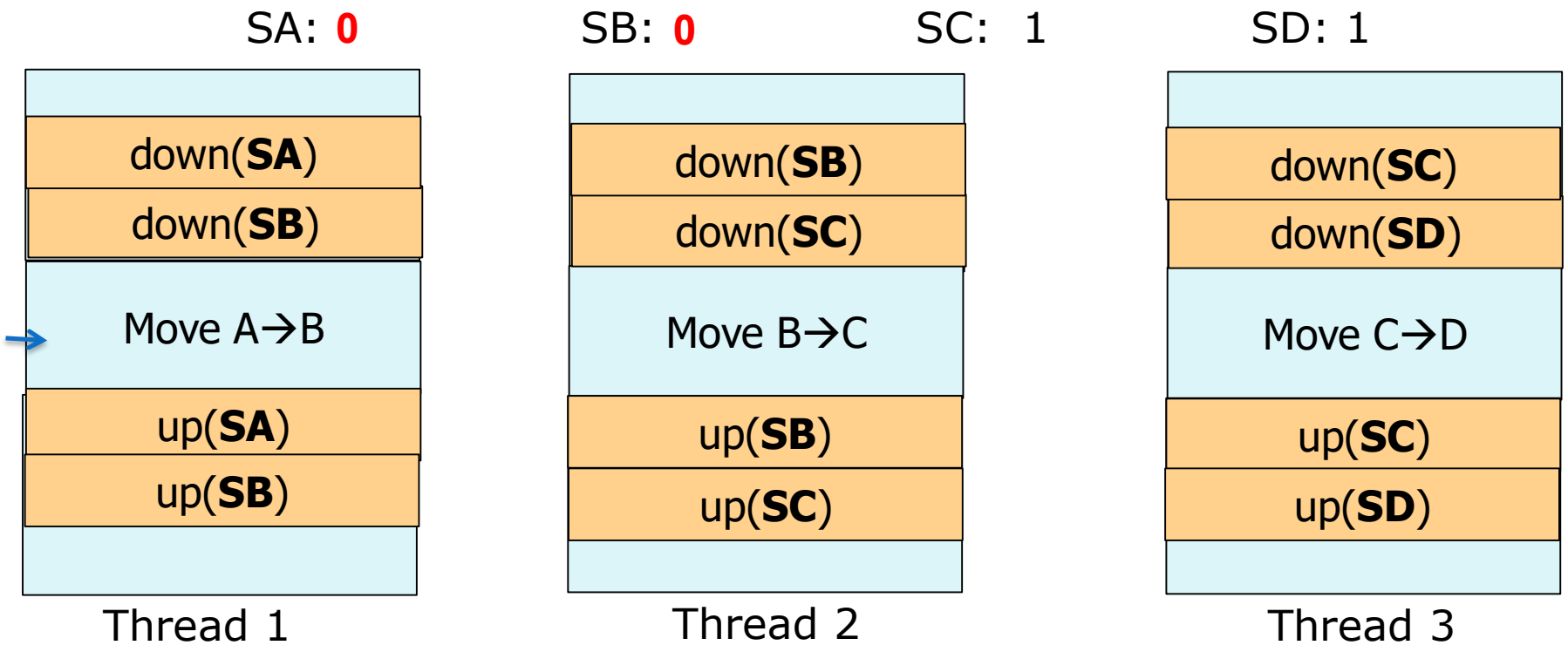
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



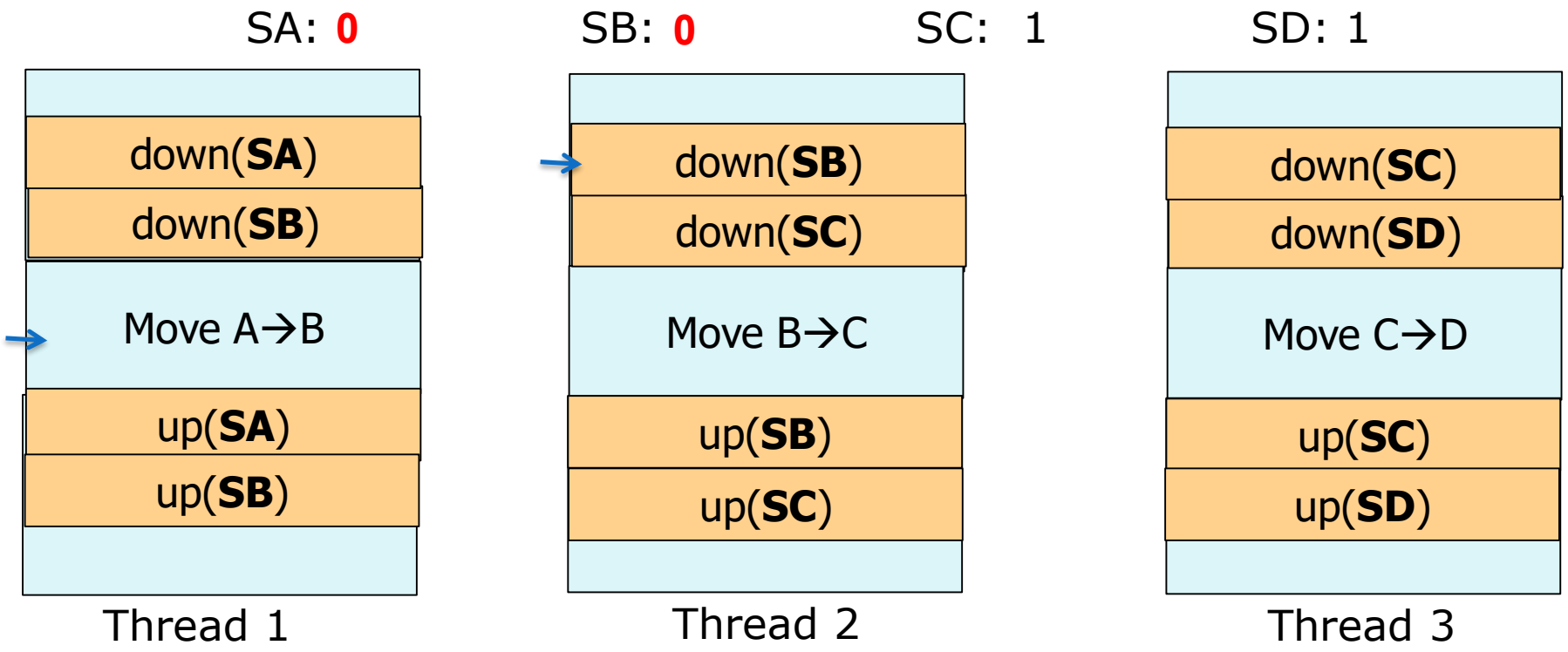
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



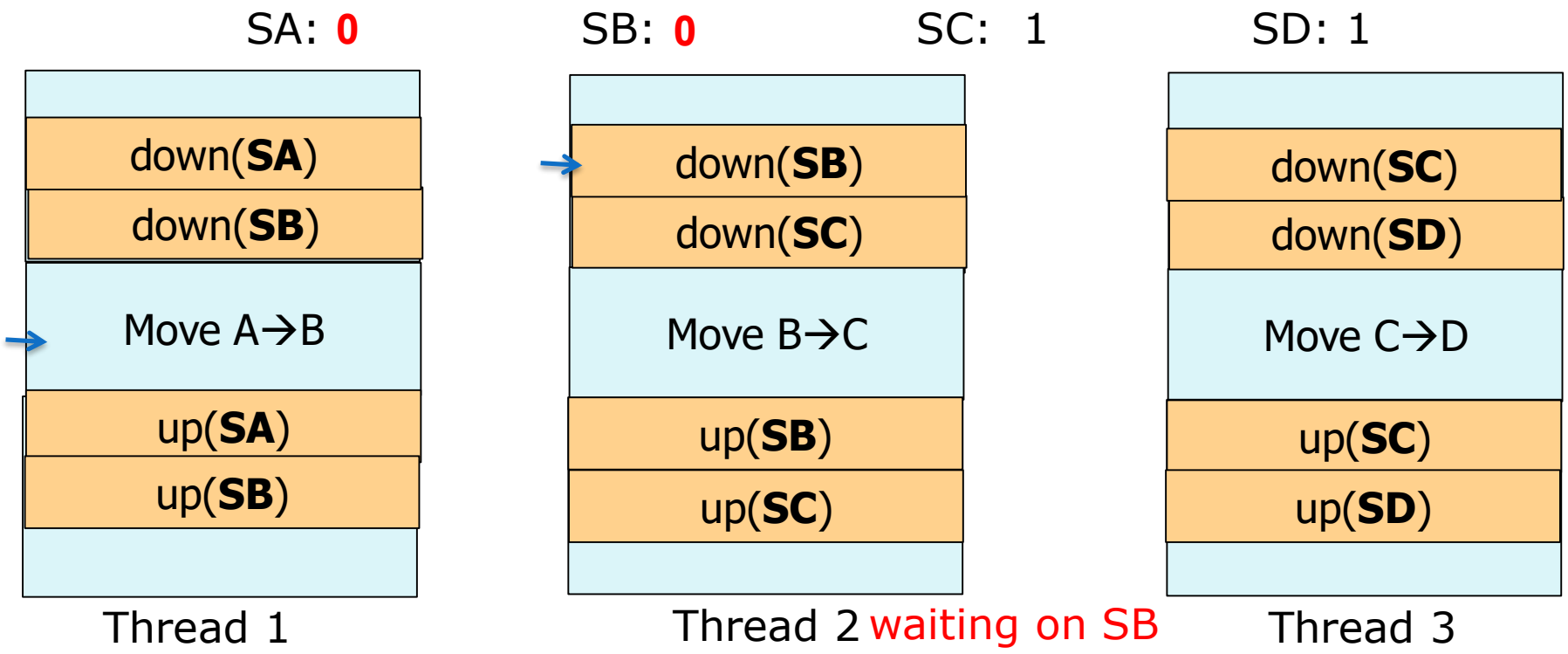
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



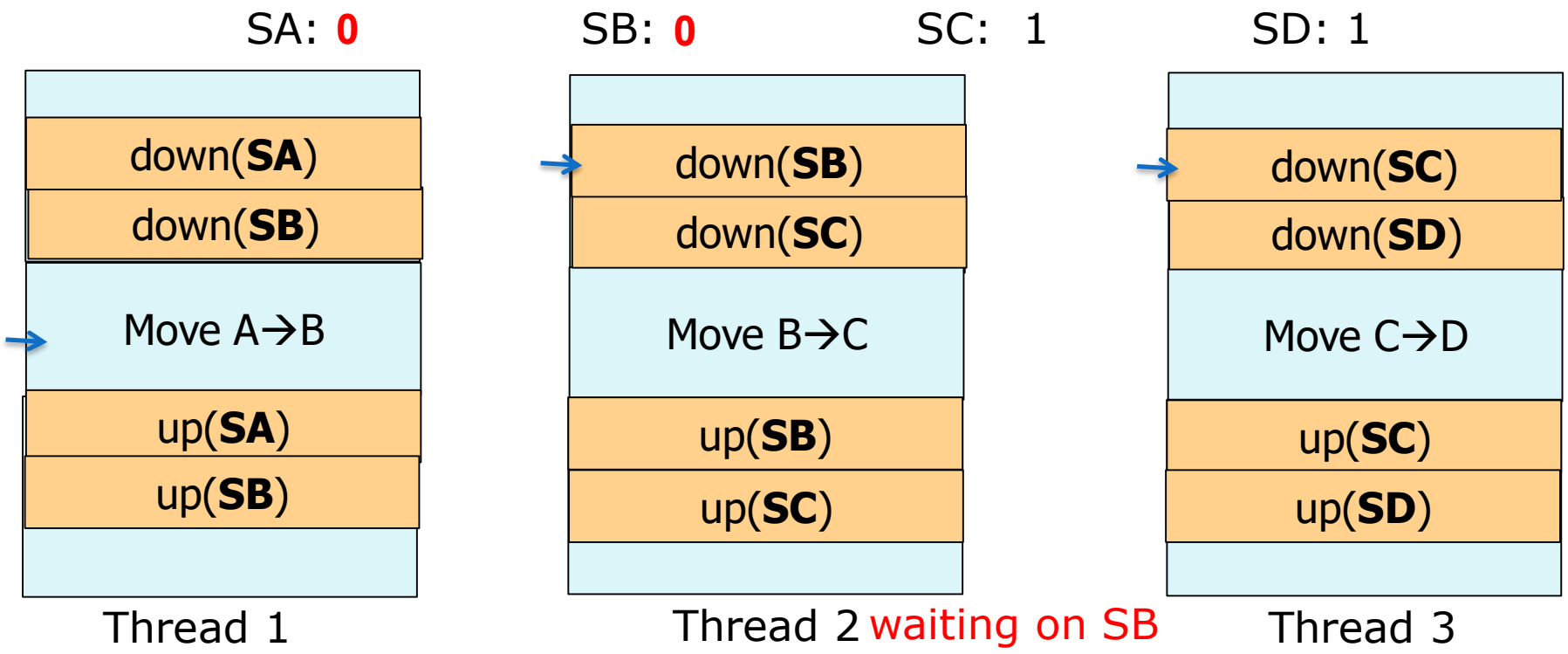
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



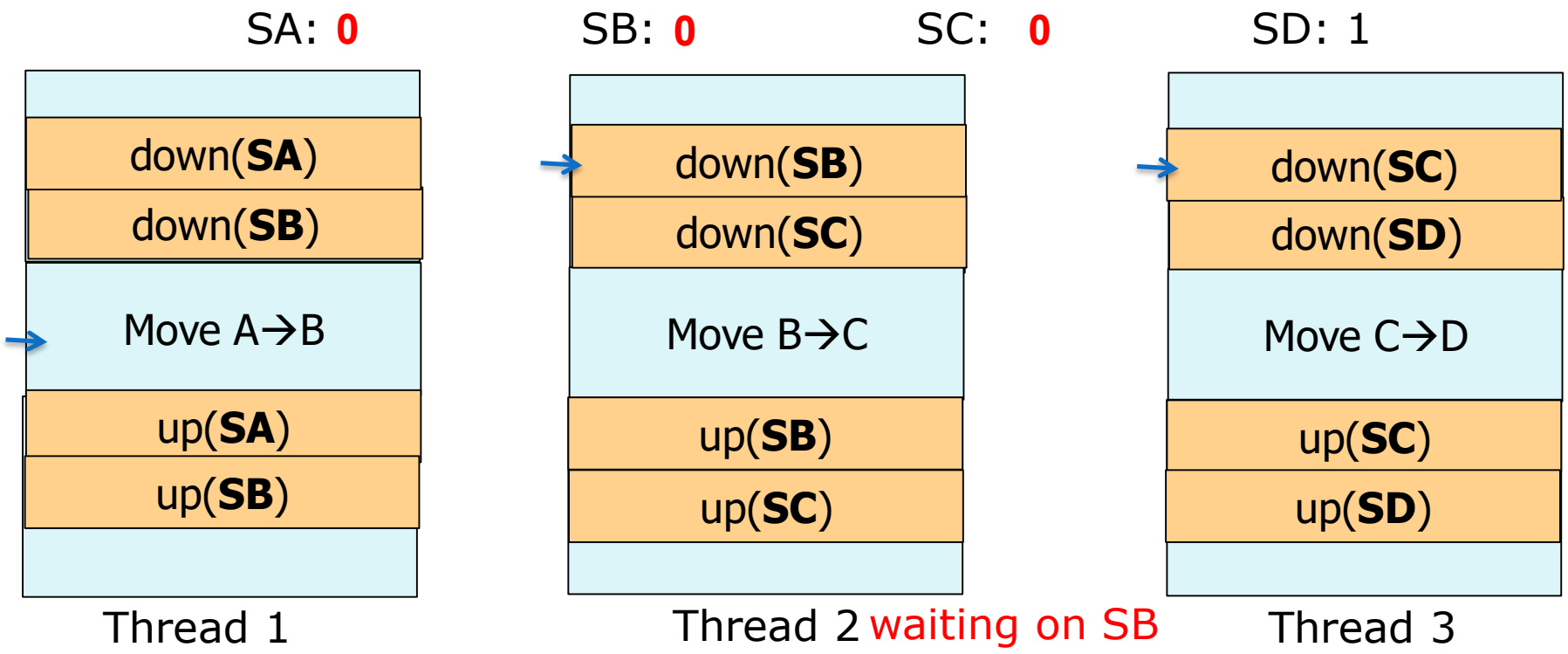
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



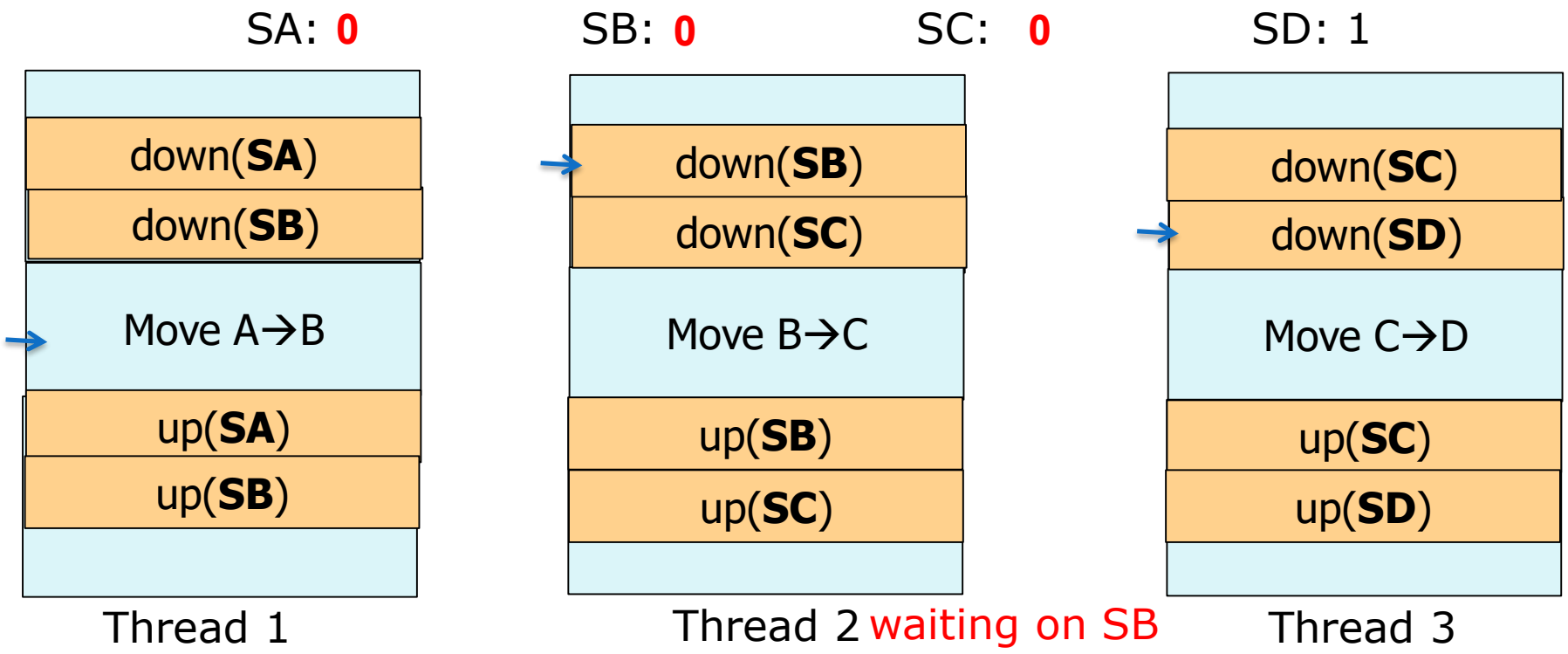
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



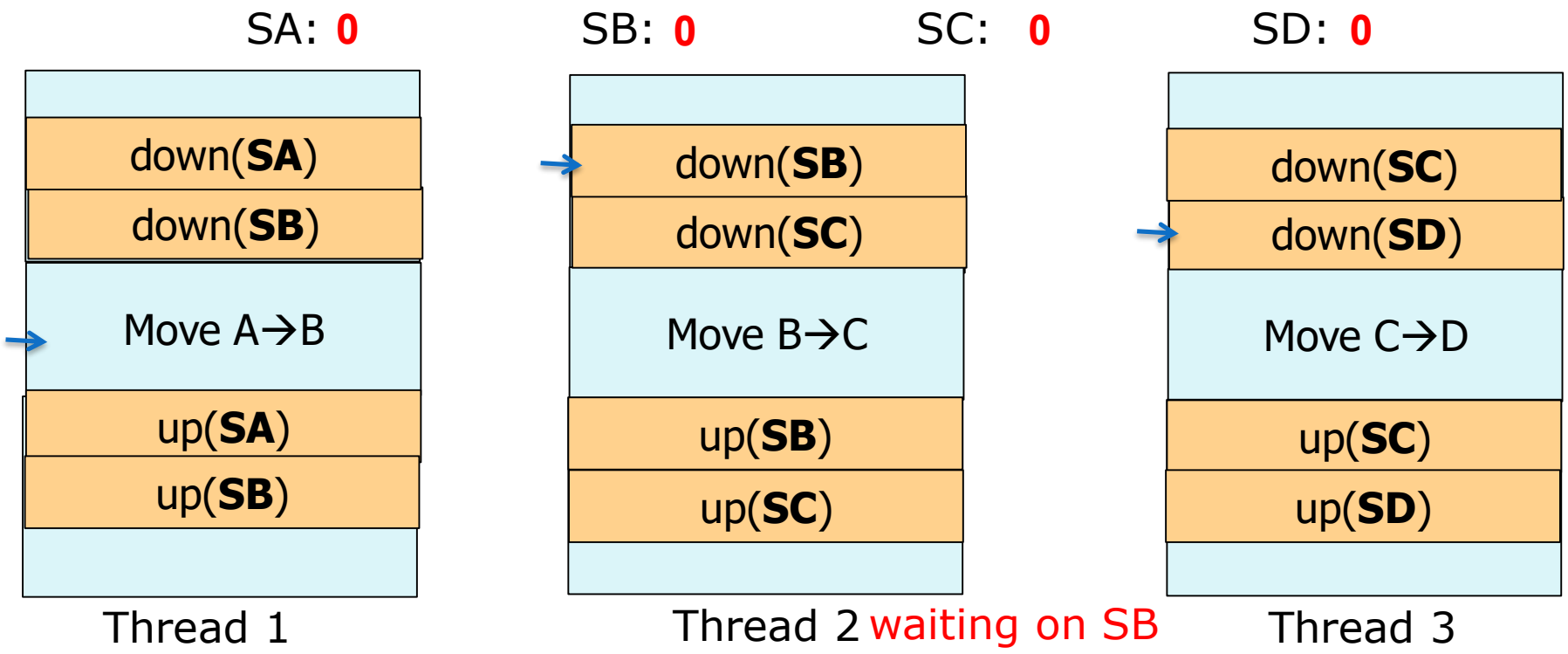
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



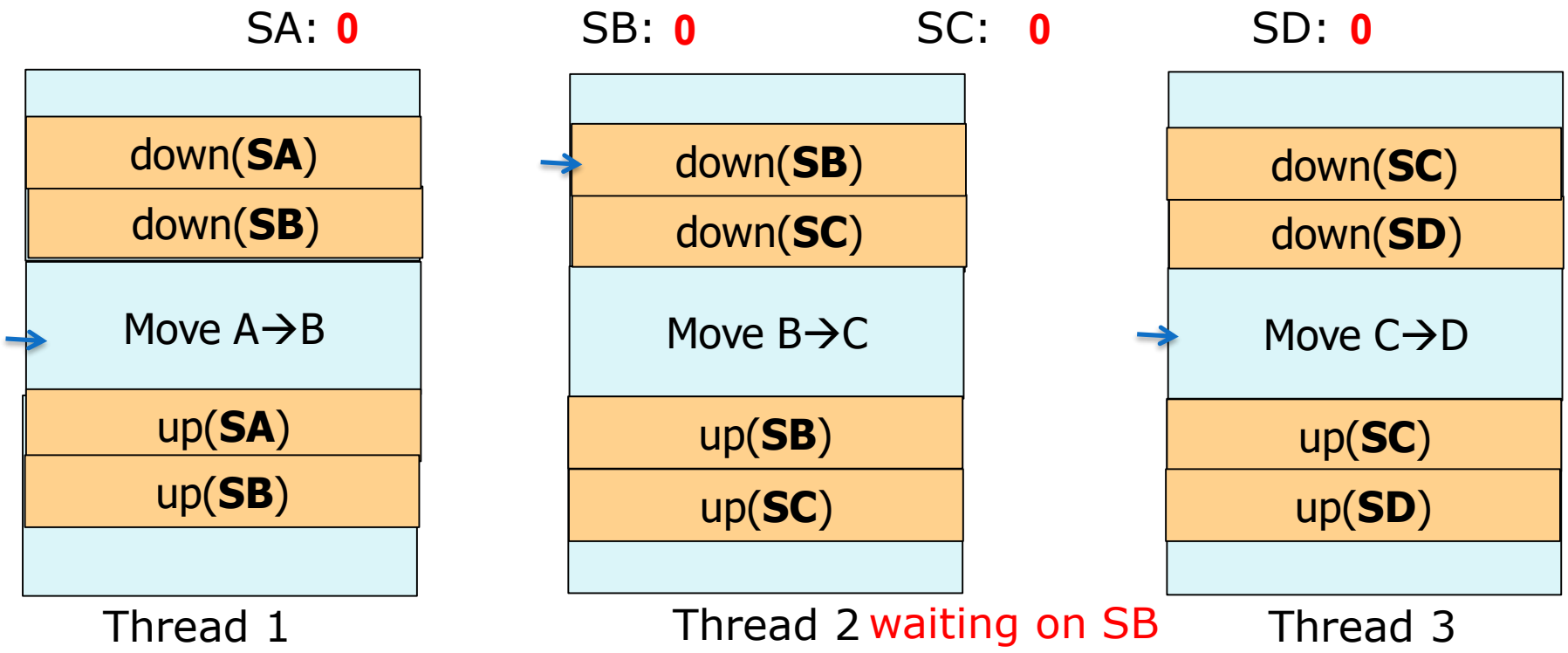
Another Problem: Moving Money Between Accounts

- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.

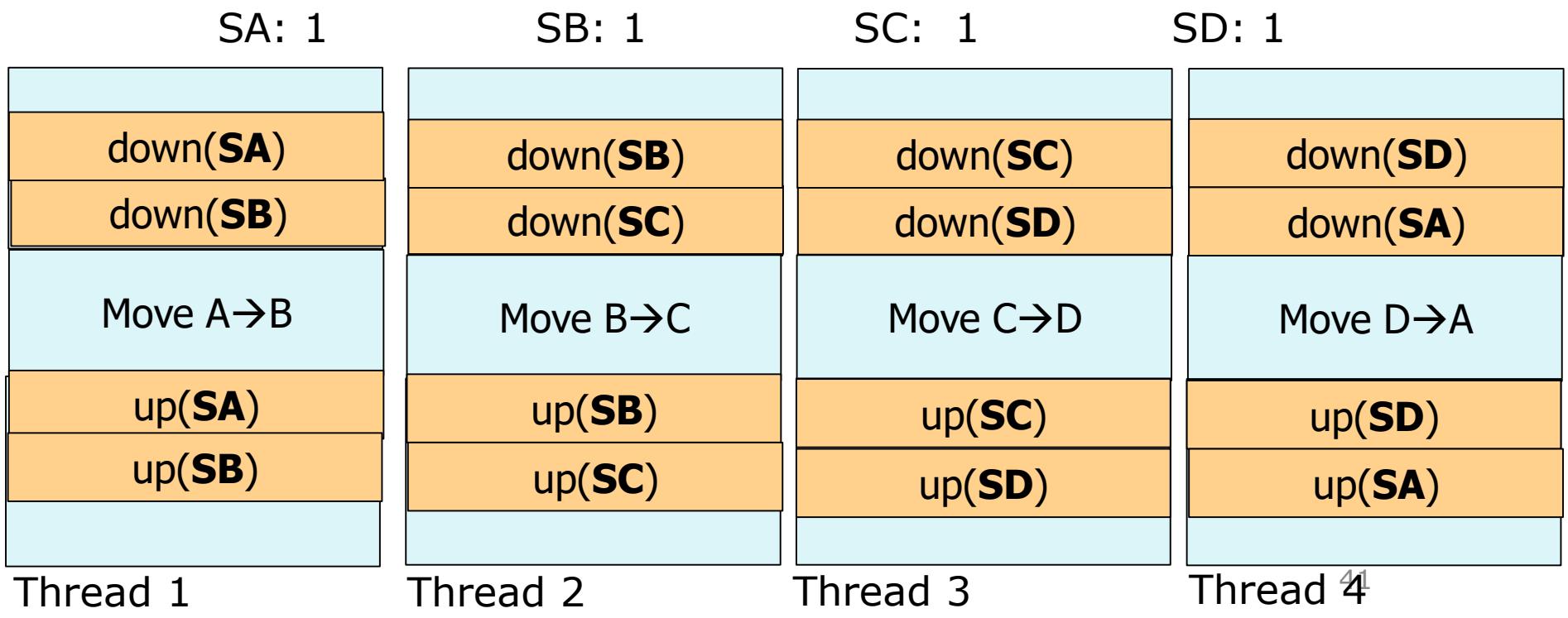


Another Problem: Moving Money Between Accounts

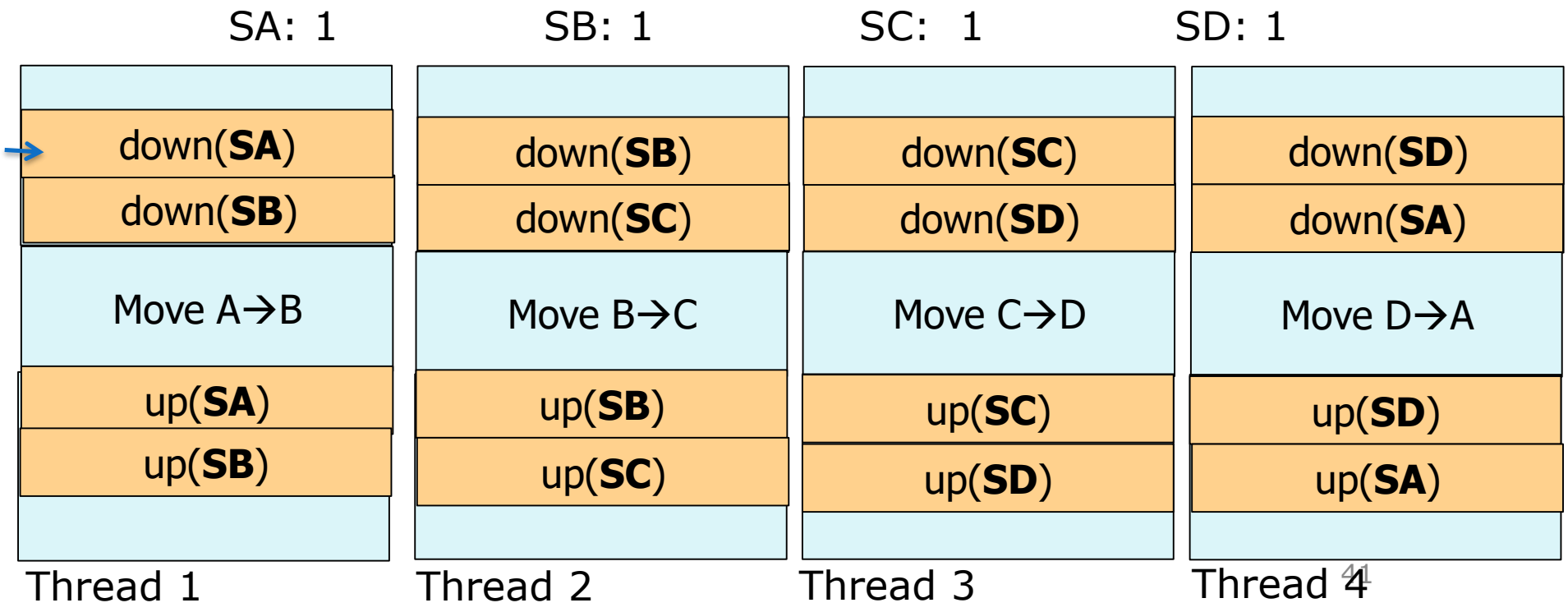
- Solution 2: Lock each account separately
  - Semaphore for each account: SA, SB, SC, SD. All initialized to 1.



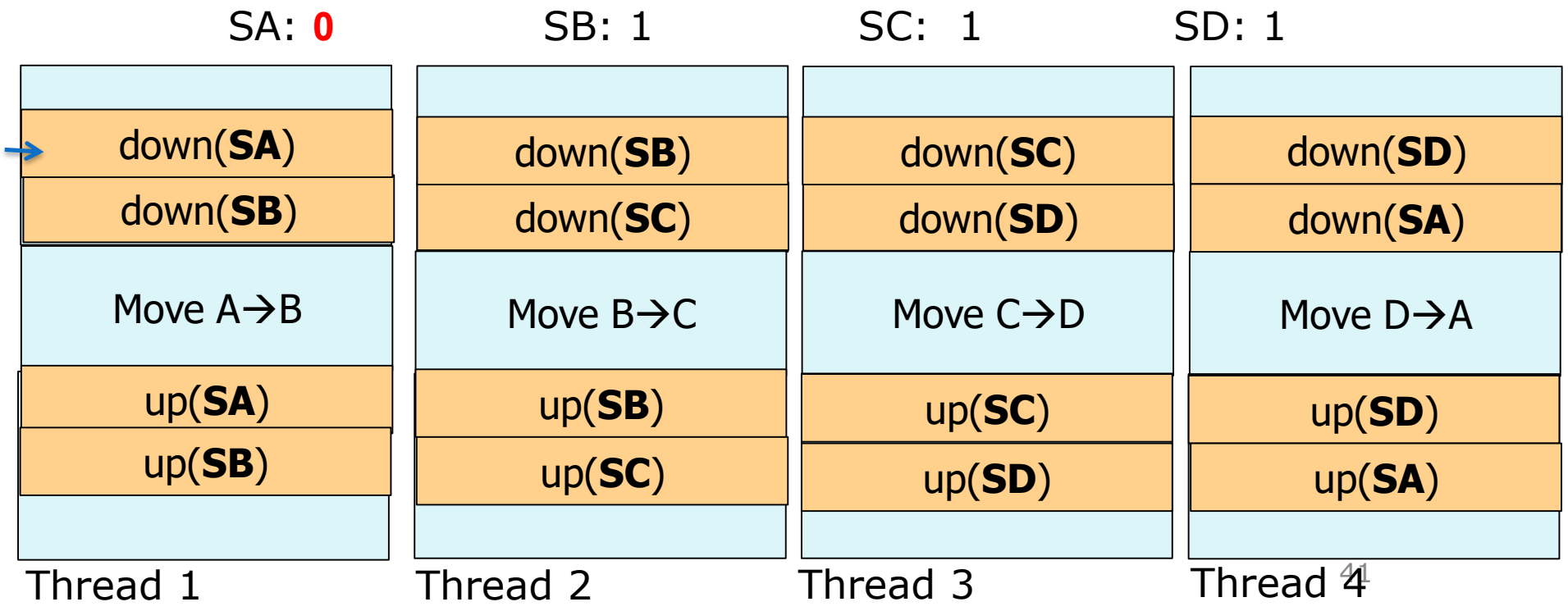
Semaphores are not Silver Bullets



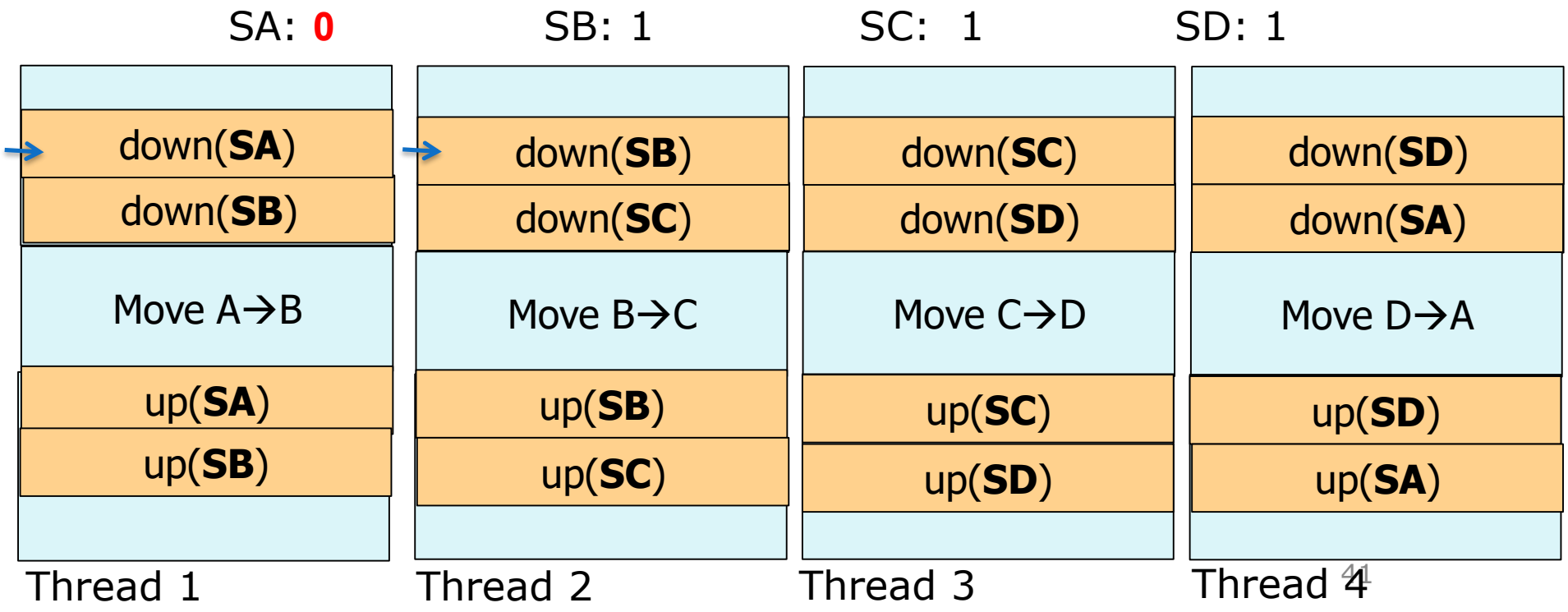
Semaphores are not Silver Bullets



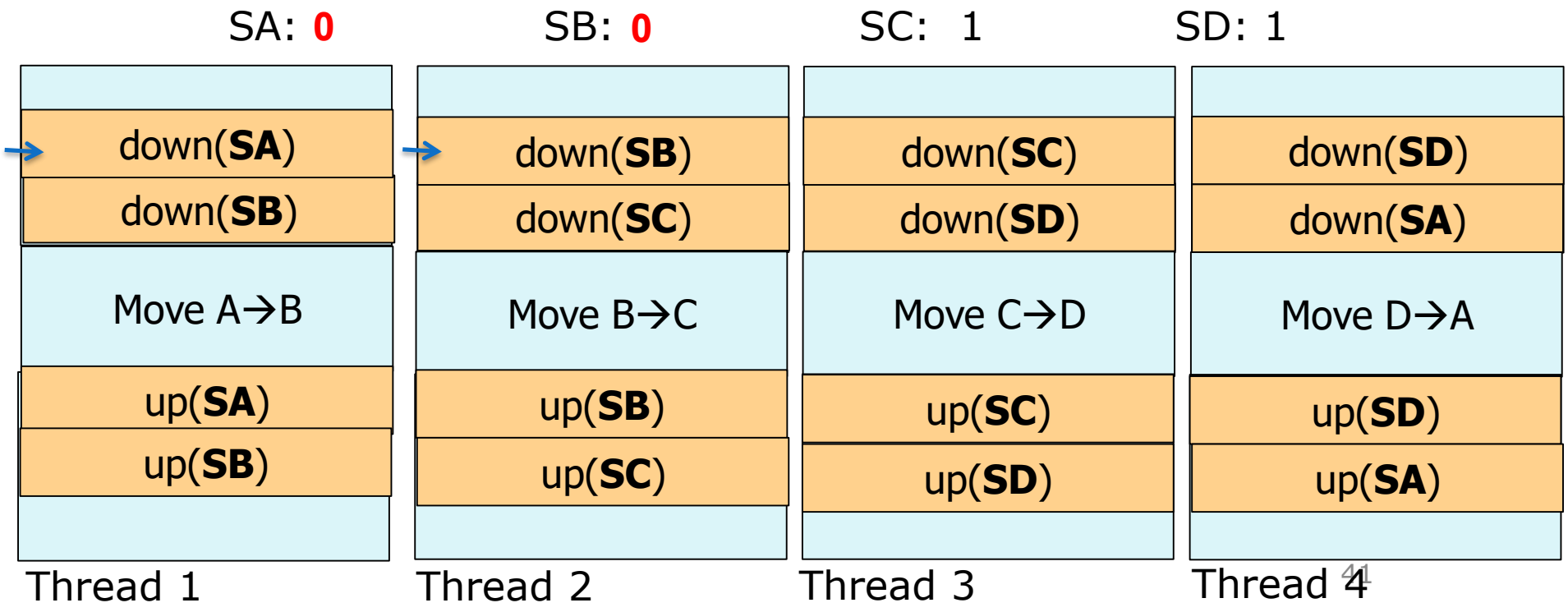
Semaphores are not Silver Bullets



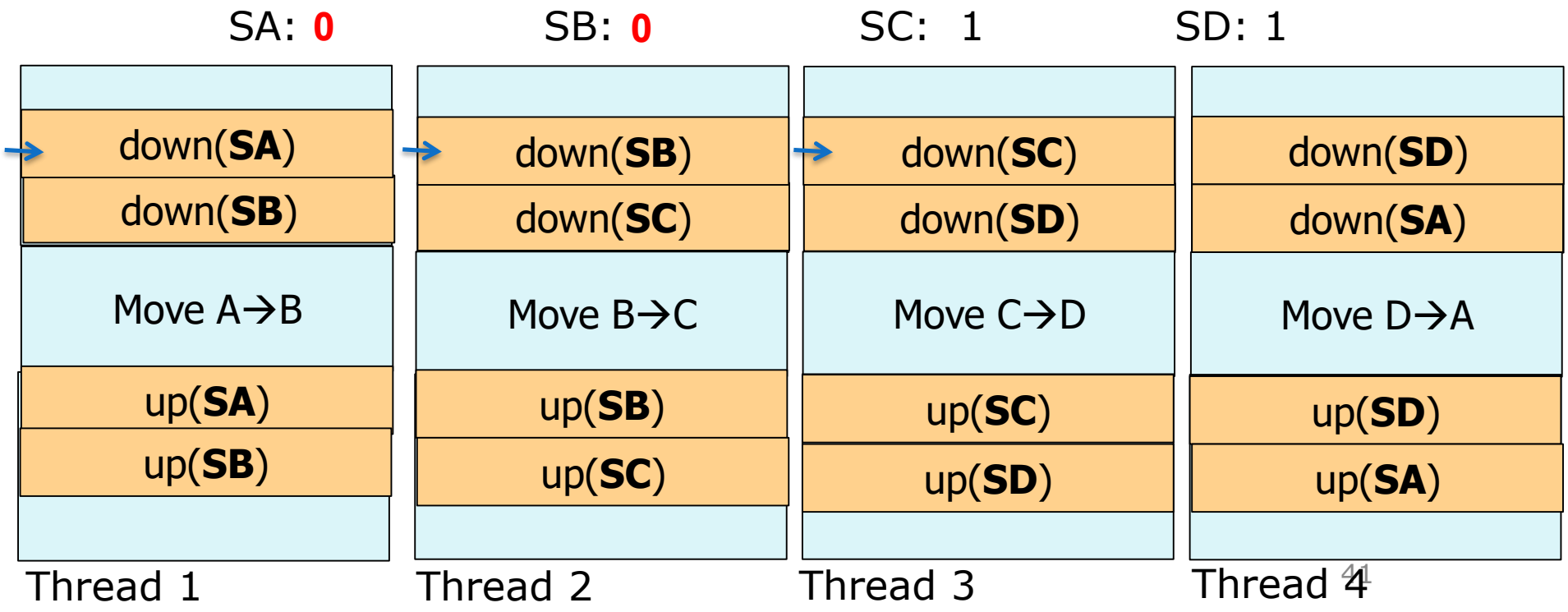
Semaphores are not Silver Bullets



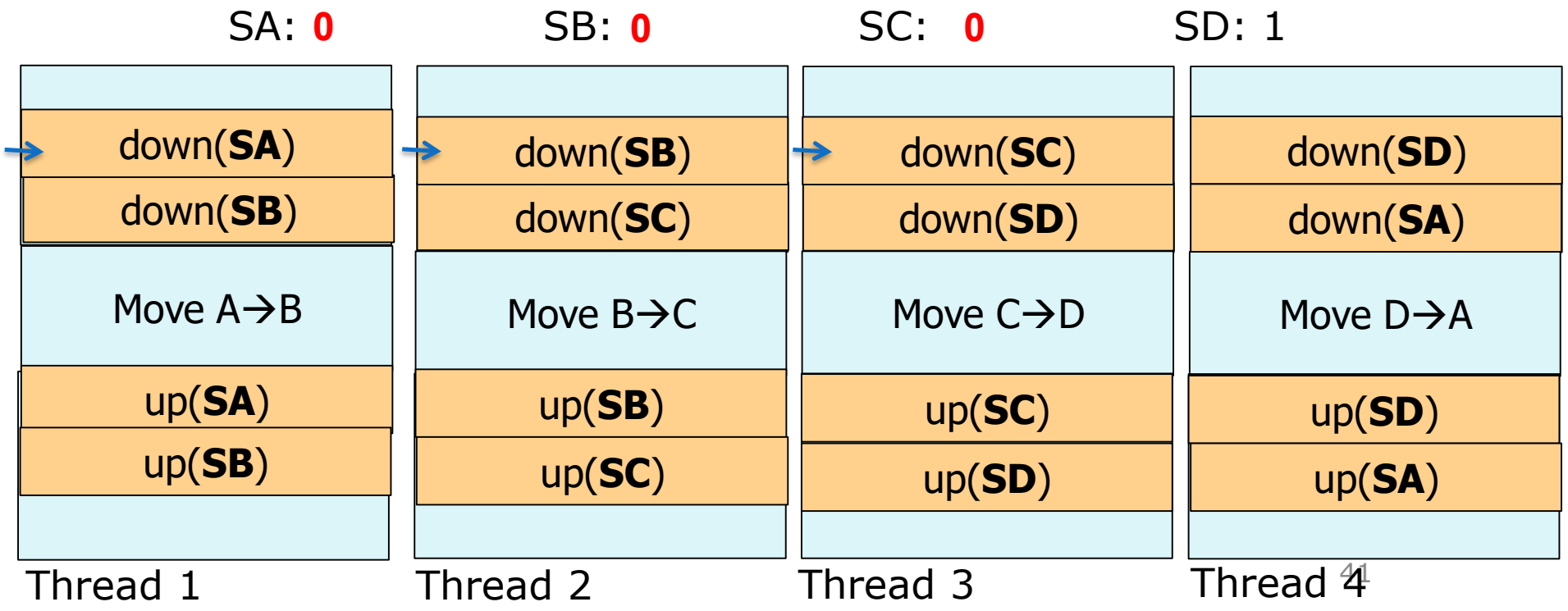
Semaphores are not Silver Bullets



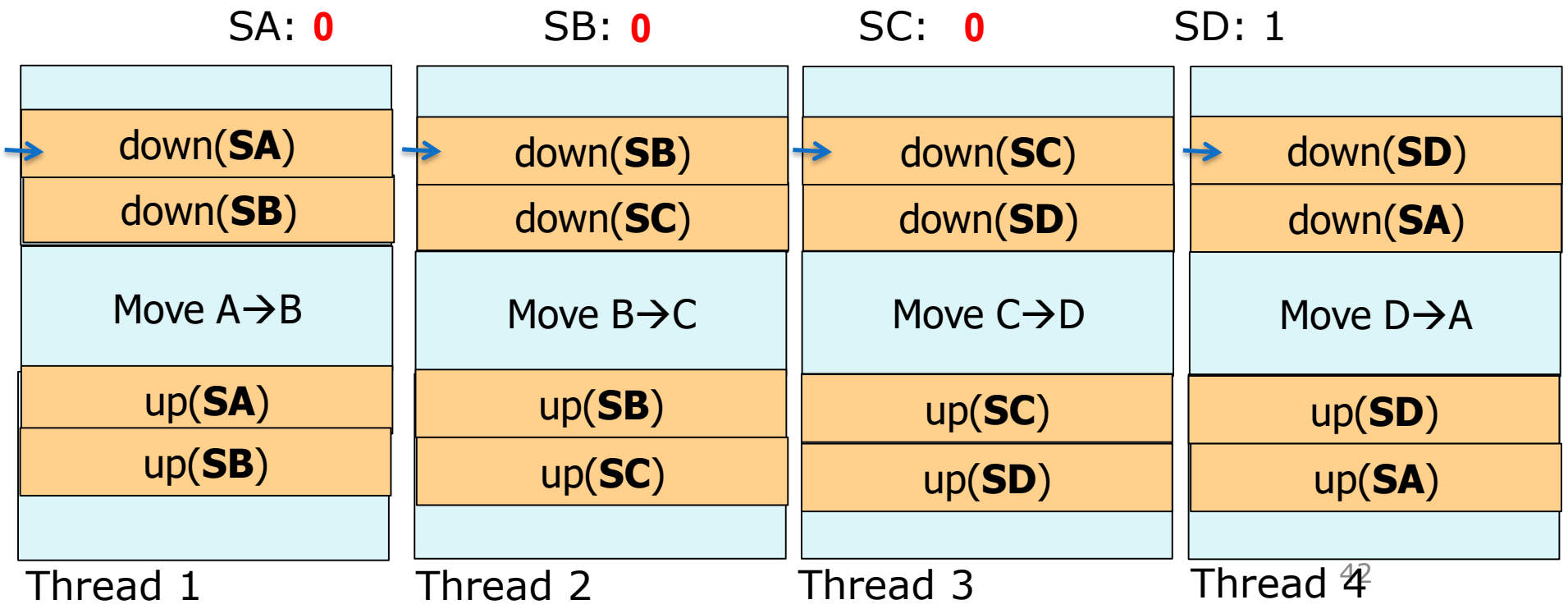
Semaphores are not Silver Bullets



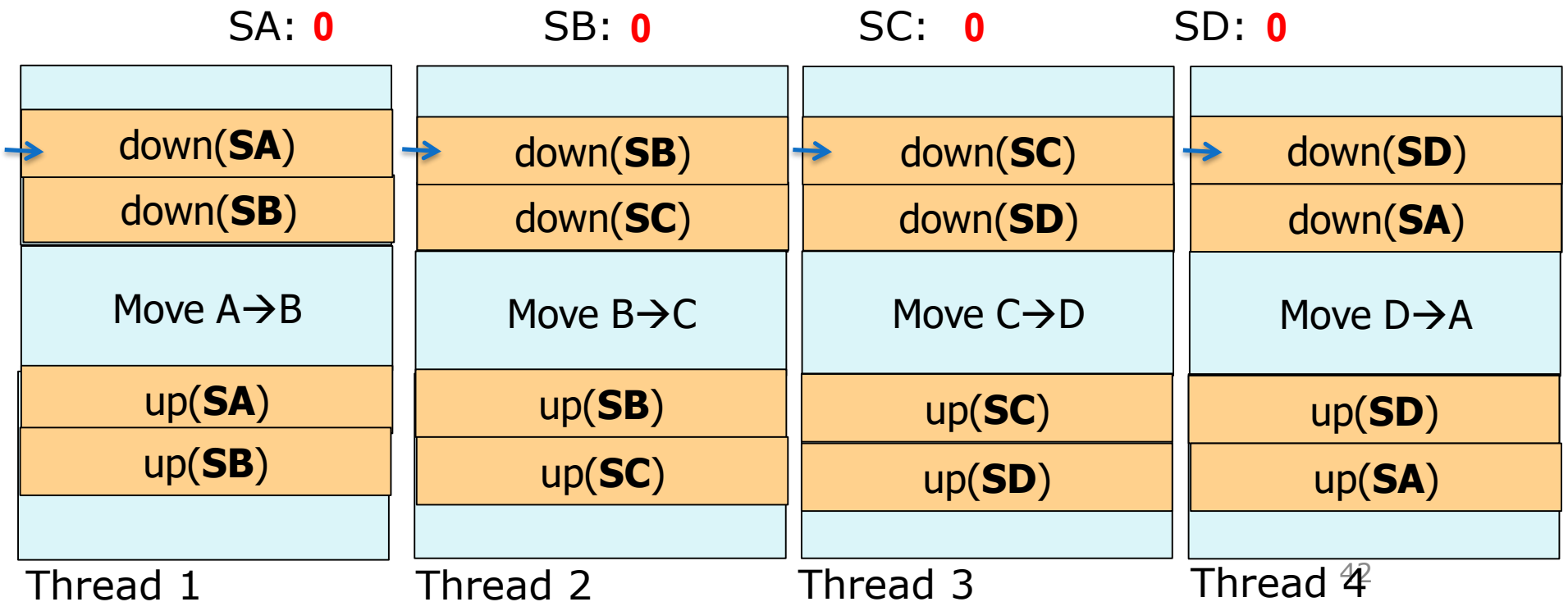
Semaphores are not Silver Bullets



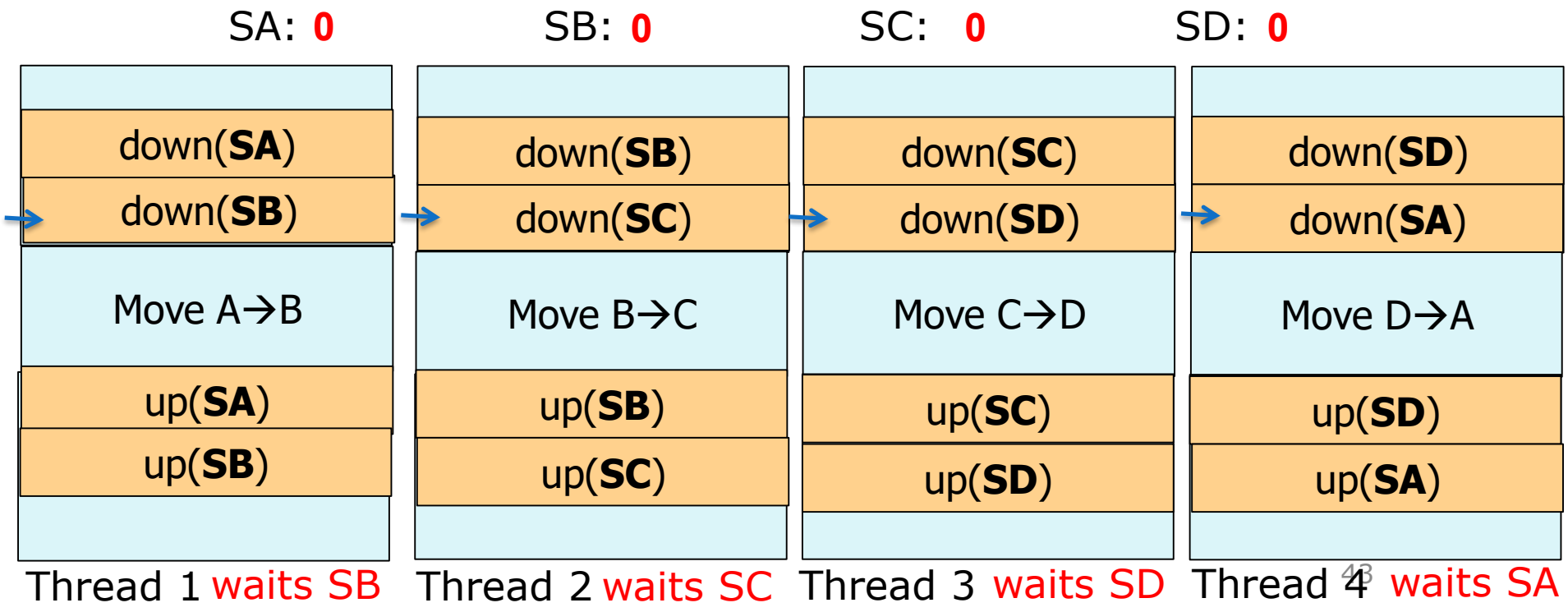
Semaphores are not Silver Bullets



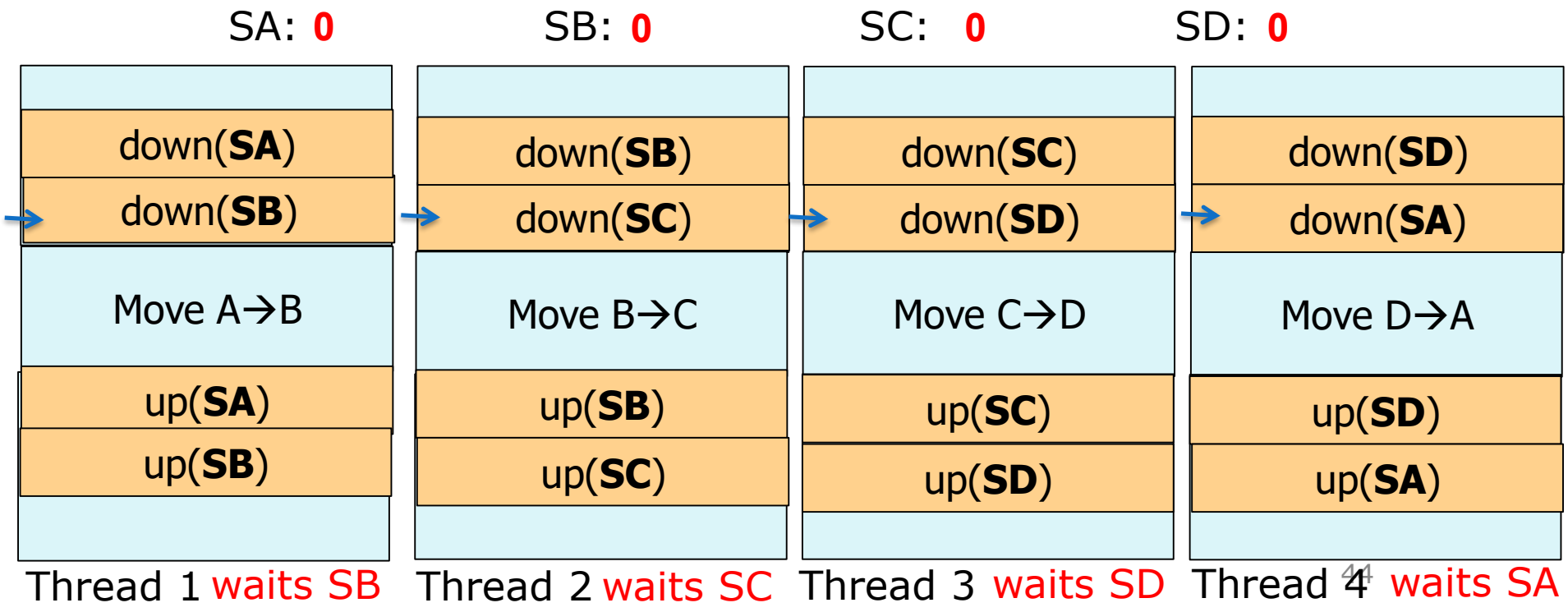
Semaphores are not Silver Bullets



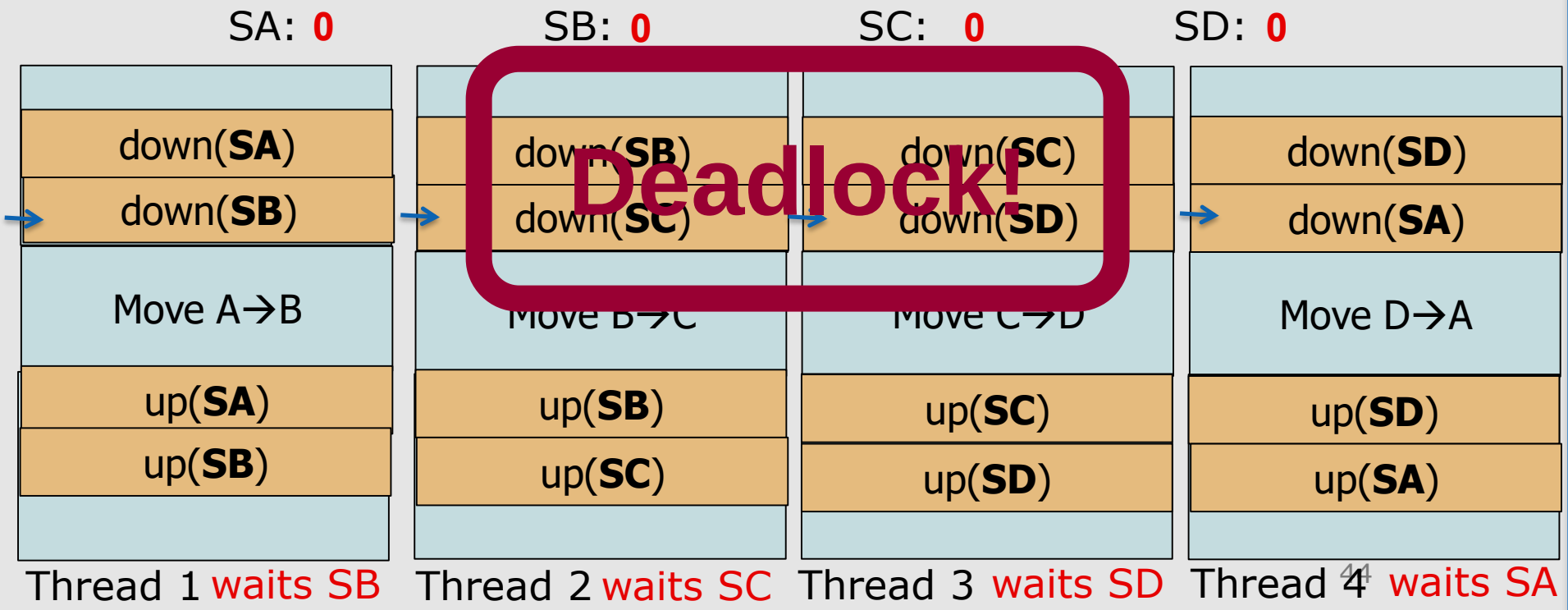
Semaphores are not Silver Bullets



Semaphores are not Silver Bullets



Semaphores are not Silver Bullets



Semaphores are not Silver Bullets

Although it is much easier to work with semaphores than read/write operations, it is still possible to cause problems (e.g., deadlocks, starvation, incorrectness)