

Algorithms for **Concurrent**  
Distributed Systems:  
The Mutual Exclusion problem

# Concurrent Distributed Systems

Changes to the model from the MPS:

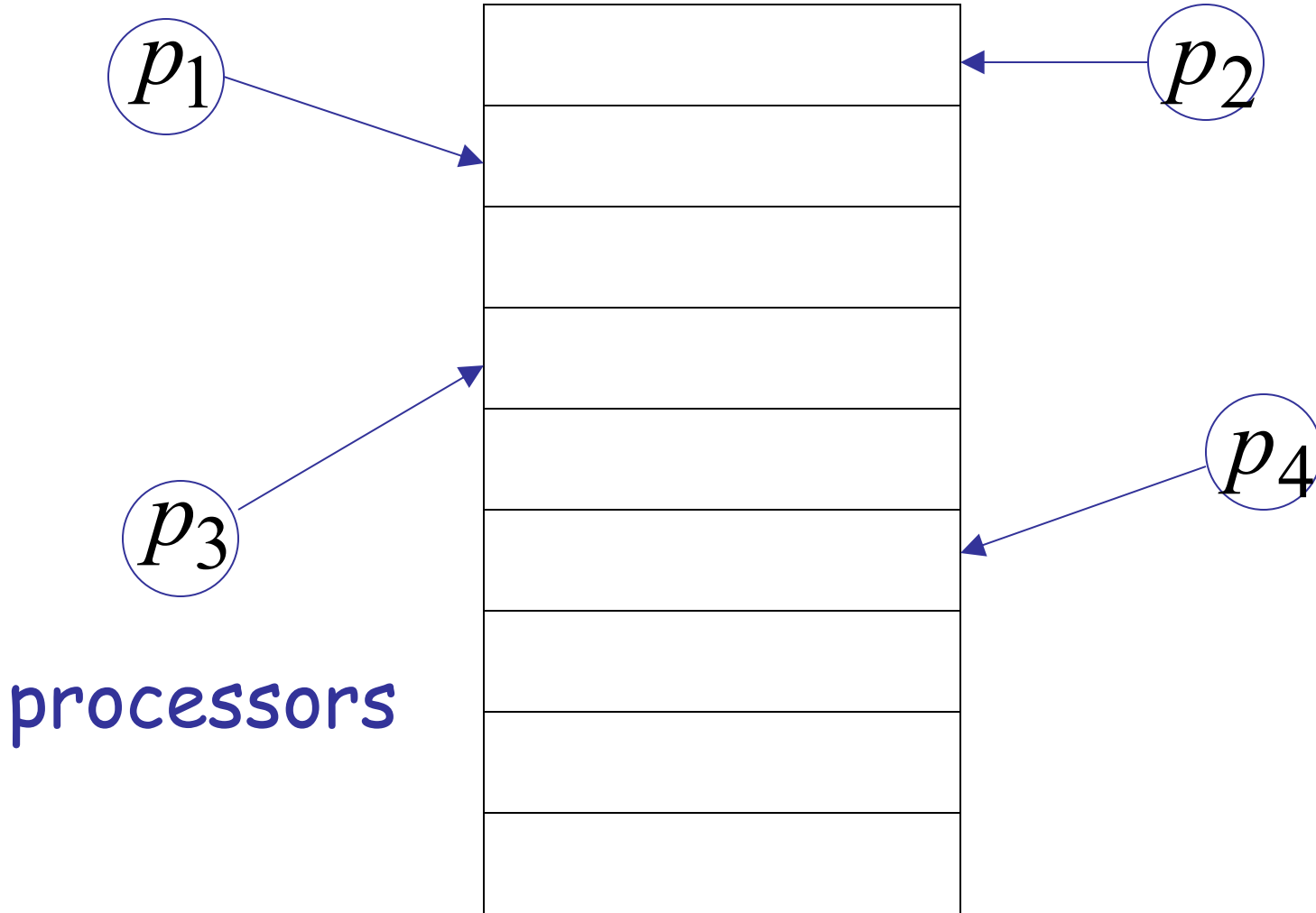
- MPS basically models a distributed system in which processors need to coordinate to perform a wide system goal (e.g., electing their leader)
- Now, we turn our attention to **concurrent systems**, where the processors run in parallel but without necessarily cooperating (for instance, they might just be a set of laptops in a LAN)

# Shared Memory System (SMS)

Changes to the model from the MPS:

- No cooperation  $\Rightarrow$  no **communication** channels between processors and no *inbuf* and *outbuf* state components
- Processors **notify** their status via a set of **shared variables**, instead of passing messages  $\Rightarrow$  no any communication graph!
- Each shared variable has a **type**, defining a set of operations that can be performed **atomically** (i.e., **instantaneously**, without interferences)

# Shared Memory



# Types of Shared Variables

1. Read/Write
2. Read-Modify-Write
3. Test & Set
4. Compare-and-swap
- ⋮

We will focus on the **Read/Write** type (the simplest one to be realized)

# Read/Write Variables

Read(v)

return(v);

Write(v,a)

v := a;

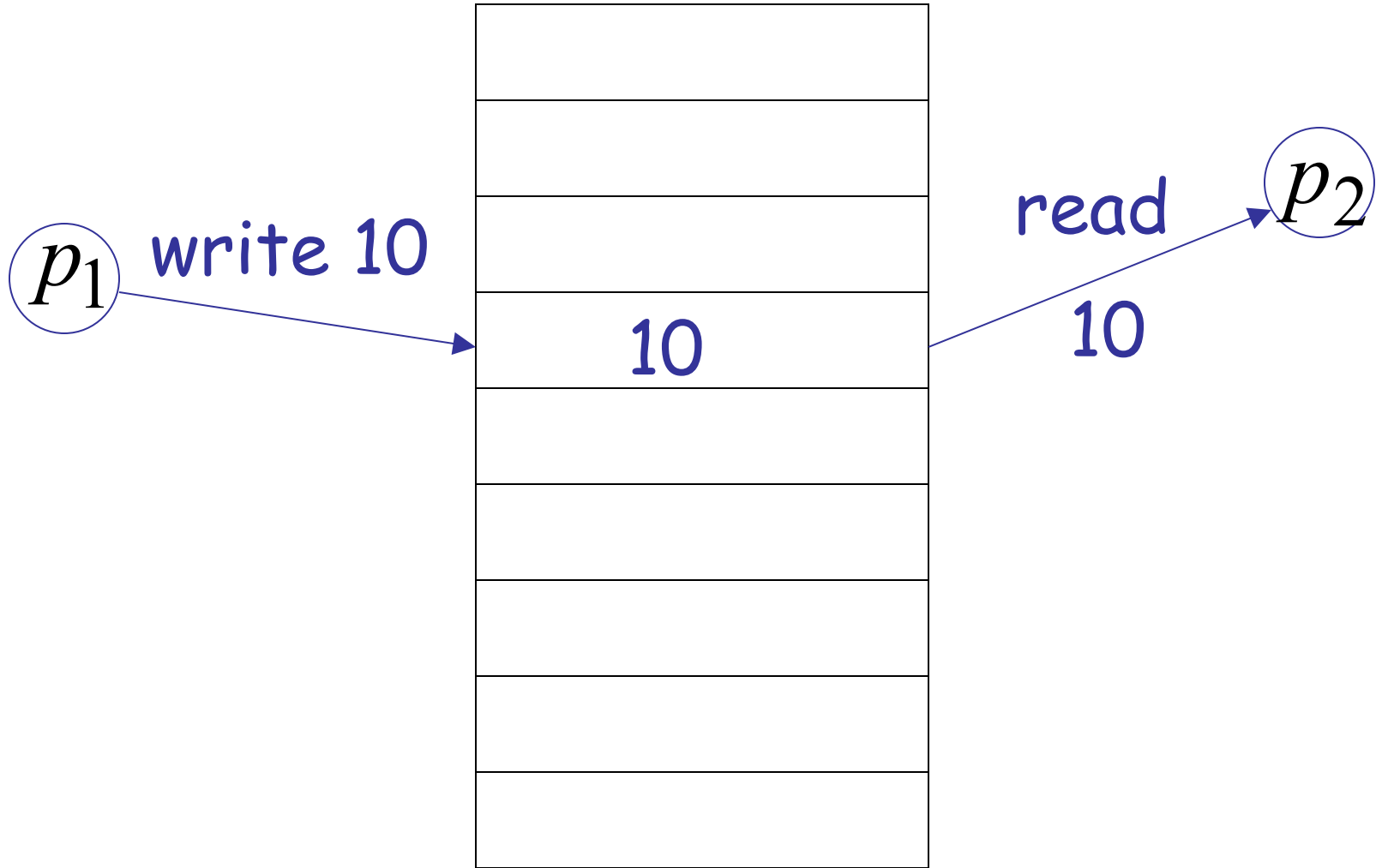
In one atomic step a processor can:

- **read** the variable, or
- **write** the variable

... but not both!

⇒ between a **read** and a **write** operation by a given processor (whose written value could be a function of the just read value), some other processor could have changed in the meantime the value of the variable!

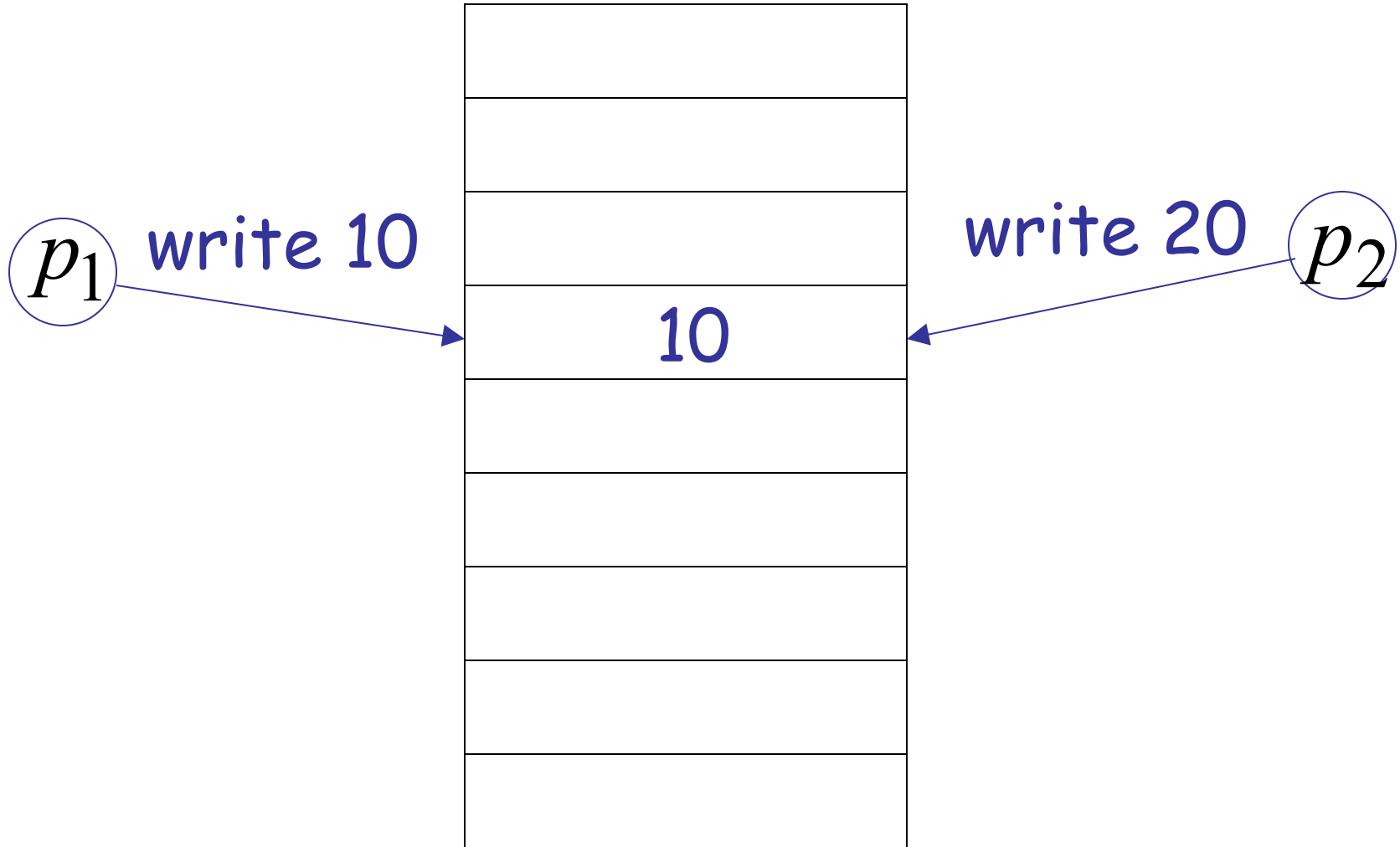
# An example



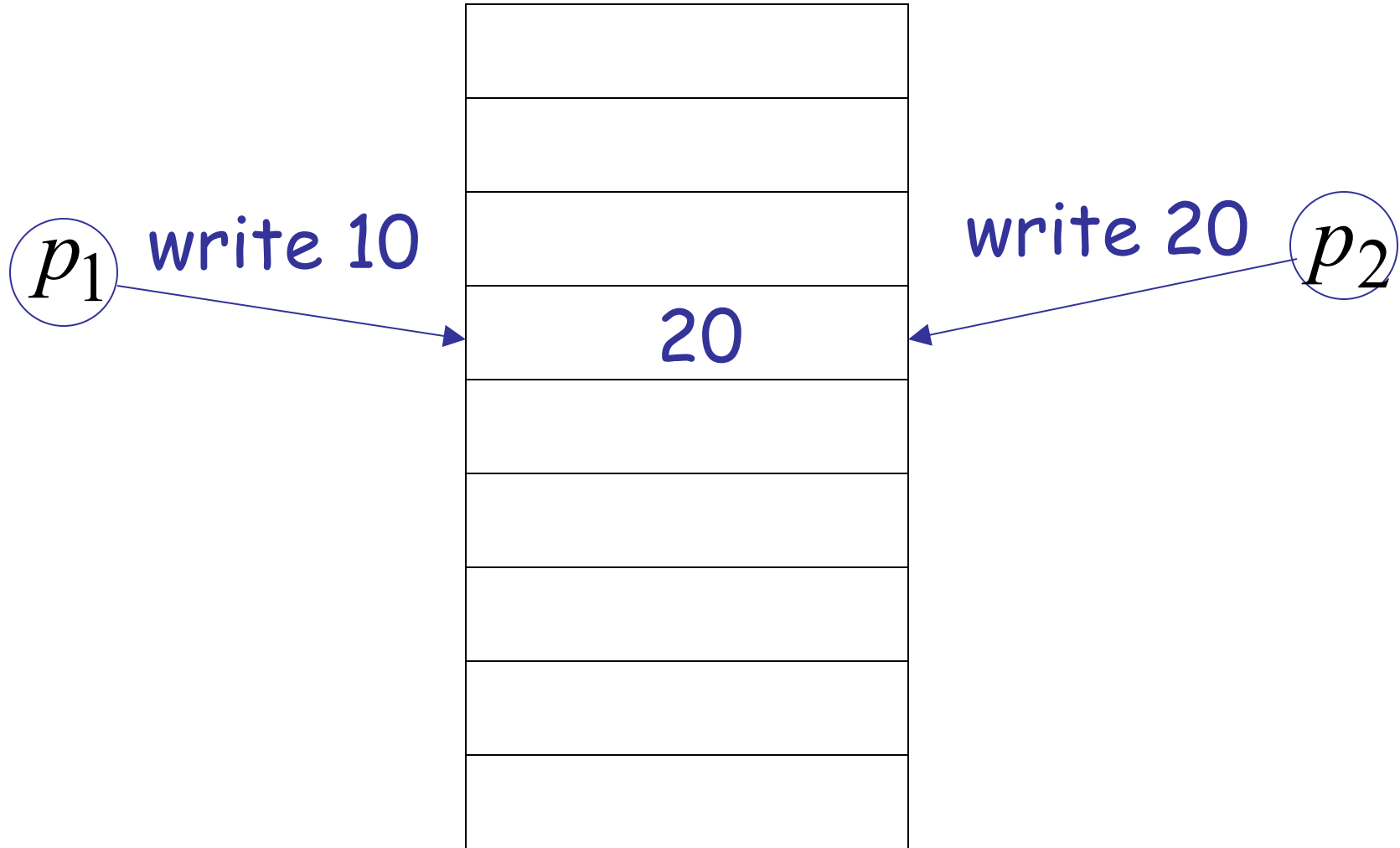




# Simultaneous writes are scheduled: Possibility 1

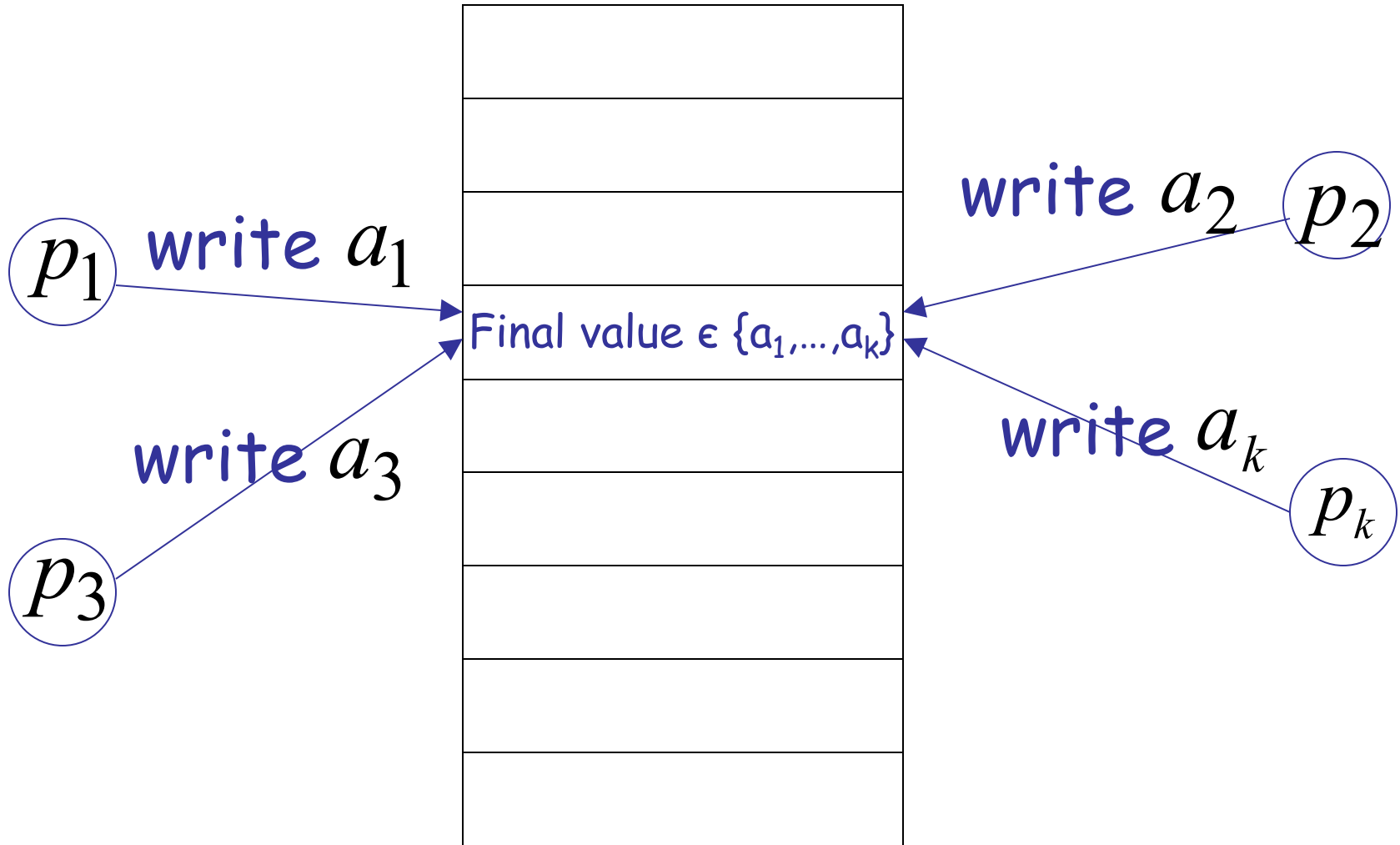


# Simultaneous writes are scheduled: Possibility 2

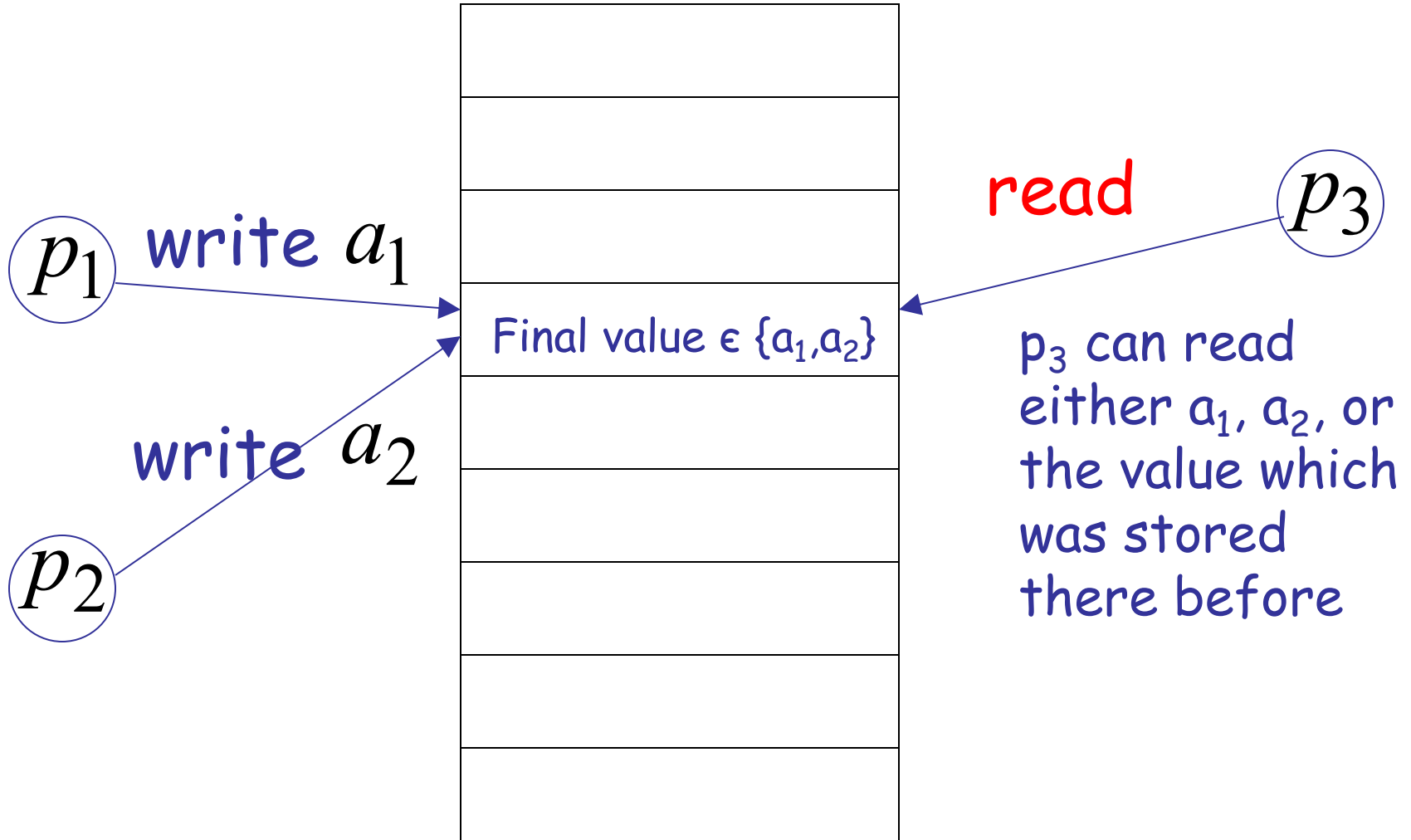


⇒ the surviving value is arbitrary!

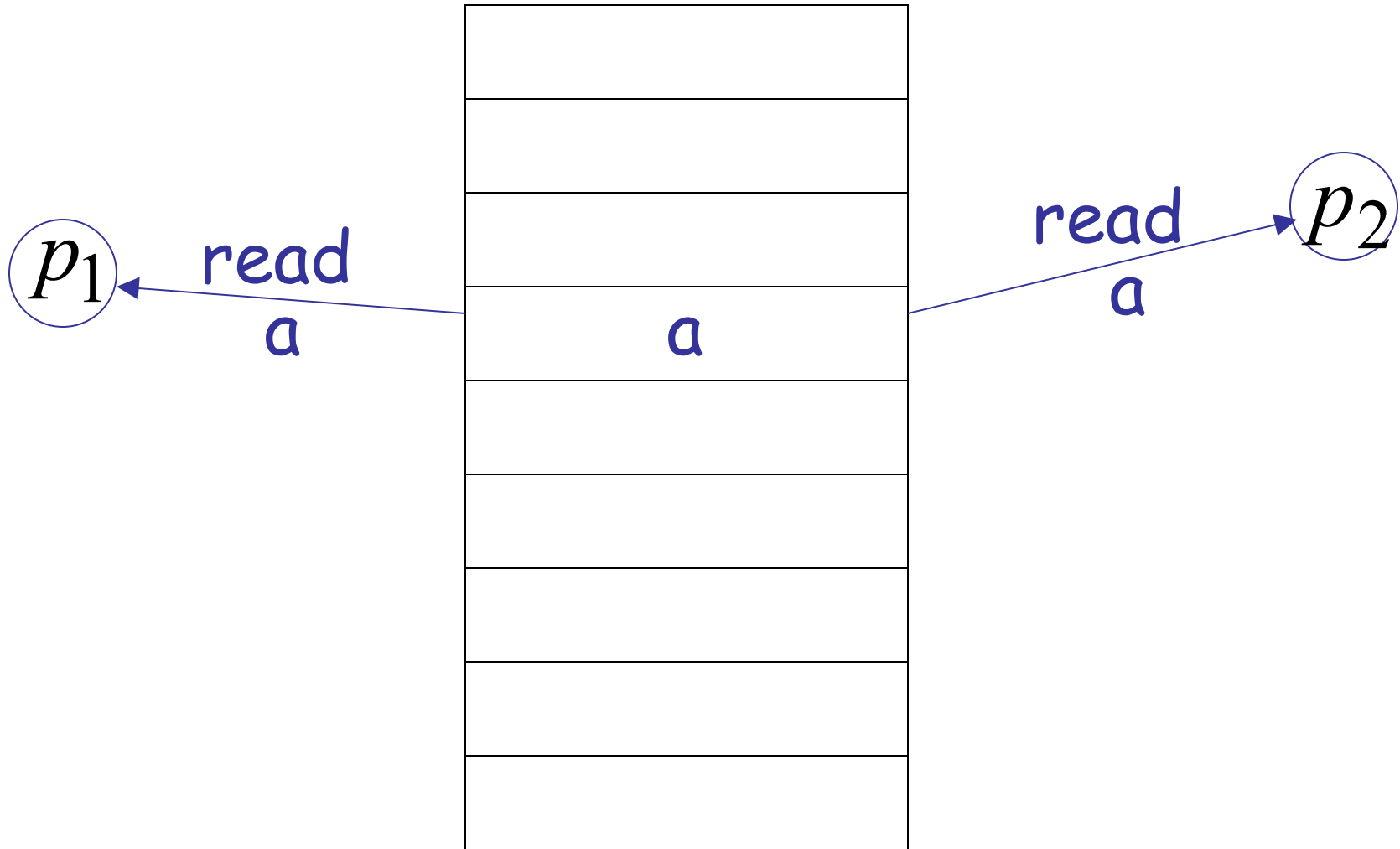
# Simultaneous writes are scheduled: In general:



# Simultaneous reads and writes are also scheduled



# Simultaneous Reads: no problem!



All read the same value

# Computation Step in the Shared Memory System

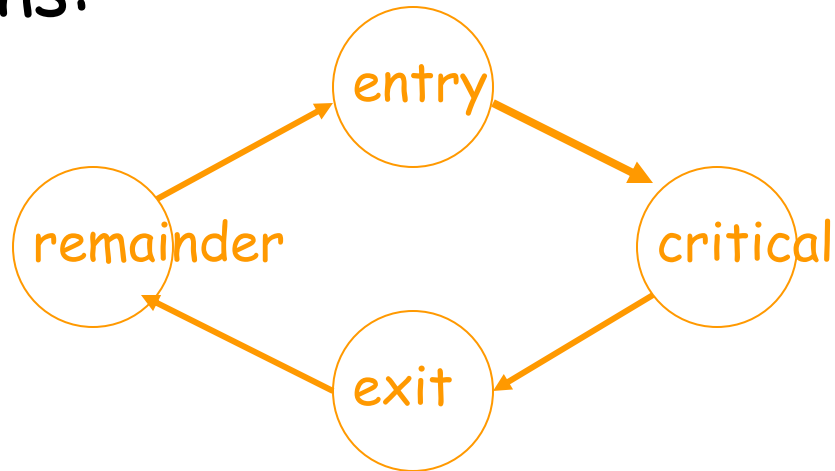
- When processor  $p_i$  takes a step:
  - $p_i$ 's state in the old configuration specifies which shared variable is to be accessed and with which operation
  - operation is done: shared variable's value in the new configuration (possibly) changes according to the operation
  - $p_i$ 's state in the new configuration changes according to its old state and the result of the operation

# The mutual exclusion (**mutex**) problem

- The main challenge in managing concurrent systems is **coordinating access** to resources that are shared among processes
- Assumptions on the SMS (similarly to the MPS):
  - Non-anonymous (ids are in **[0..n-1]**)
  - Non-uniform
  - Asynchronous

# Mutex code sections

- Each processor's code is divided into four sections:



- **entry**: synchronize with others to ensure mutually exclusive access to the ...
- **critical**: use some resource; when done, enter the...
- **exit**: clean up; when done, enter the...
- **remainder**: not interested in using the resource



# Mutex Algorithms

- *A mutual exclusion algorithm specifies code for entry and exit sections to ensure:*
  - **mutual exclusion**: at most one processor is within the critical section at any time, and
  - some kind of **liveness condition**, i.e., a guarantee on the use of the critical section (under the general assumption that no processor stays in its critical section forever). There are three commonly considered ones:

# Mutex Liveness Conditions

- **no deadlock**: if a processor is in its **entry section** at some point of time, then later **some** processor (i.e., maybe another one) is in its **critical section** (notice that a processor can be starved/locked in this situation)
- **no lockout**: if a processor is in its **entry section** at some point of time, then later the **same** processor is in its **critical section** (but maybe it will be overtaken an **unbounded number of times** by some other processor)
- **bounded waiting**: no lockout + while a processor is in its **entry section**, it can be overtaken in entering into the **critical section** only a **bounded number of times** by any other processor

These conditions are increasingly stronger: bounded waiting  $\Rightarrow$  no lockout  $\Rightarrow$  no deadlock

# Complexity Measure for Mutex

- Main complexity measure of interest for shared memory mutex algorithms is **amount of shared space** needed.
- Space complexity is affected by:
  - how powerful is the **type** of the shared variables (recall we only focus on **Read/Write** type)
  - how strong is the **liveness condition** to be satisfied (no deadlock/no lockout/bounded waiting)

# Mutex Results Using R/W

<i>Liveness Condition</i>	<i>upper bound</i>	<i>lower bound</i>
no deadlock		$n$ booleans
no lockout	$3(n-1)$ booleans (for $n=2^k$ ) (tournament algorithm)	
bounded waiting	$n$ booleans + $n$ <u>unbounded</u> integers (bakery algorithm)	

# The Bakery Algorithm (L. Lamport, 1974)

- Guaranteeing:
  - Mutual exclusion
  - Bounded waiting
- Using  $2n$  shared read/write variables
  - booleans `Choosing[i]`: initially false, written by  $p_i$  and read by others
  - (unbounded) integers `Number[i]`: initially 0, written by  $p_i$  and read by others

# Bakery Algorithm

Code for entry section of  $p_i$ :

```
Choosing[i] := true
Number[i] := max{Number[0], ...,
                Number[n-1]} + 1
Choosing[i] := false
for j := 0 to n-1 (except i) do
    wait until Choosing[j] = false
    wait until Number[j] = 0 or
        (Number[j], j) > (Number[i], i)
endfor
```

Doorway  
subsection  
(DS)

Bakery  
subsection  
(BS)

Semaphores

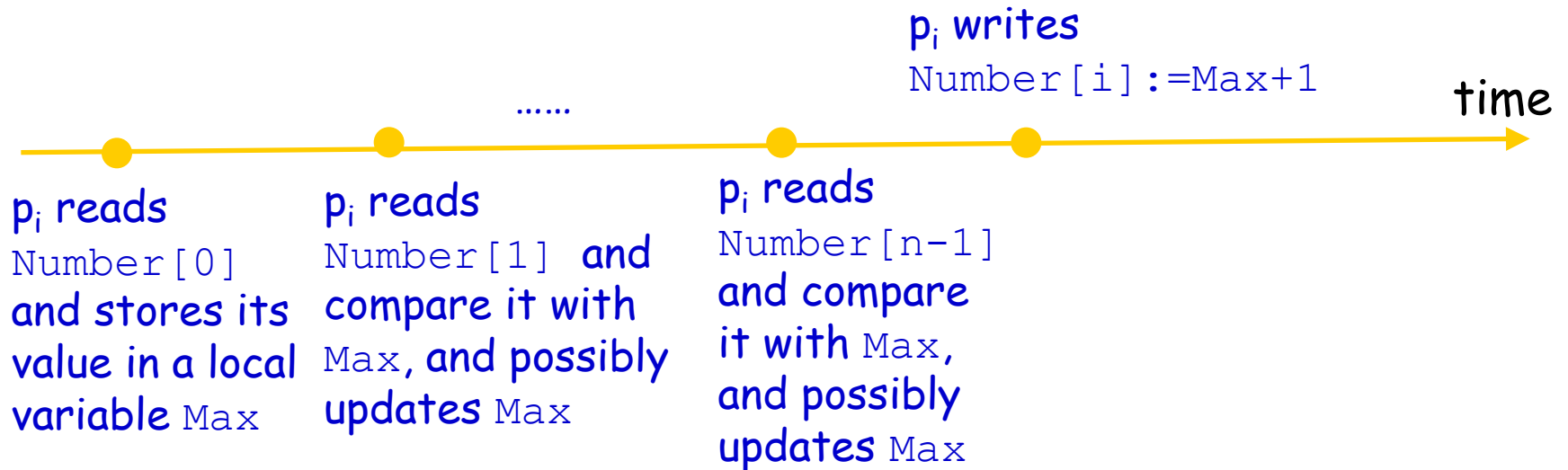
Ticket of  $p_i$

Code for exit section of  $p_i$ :

```
Number[i] := 0
```

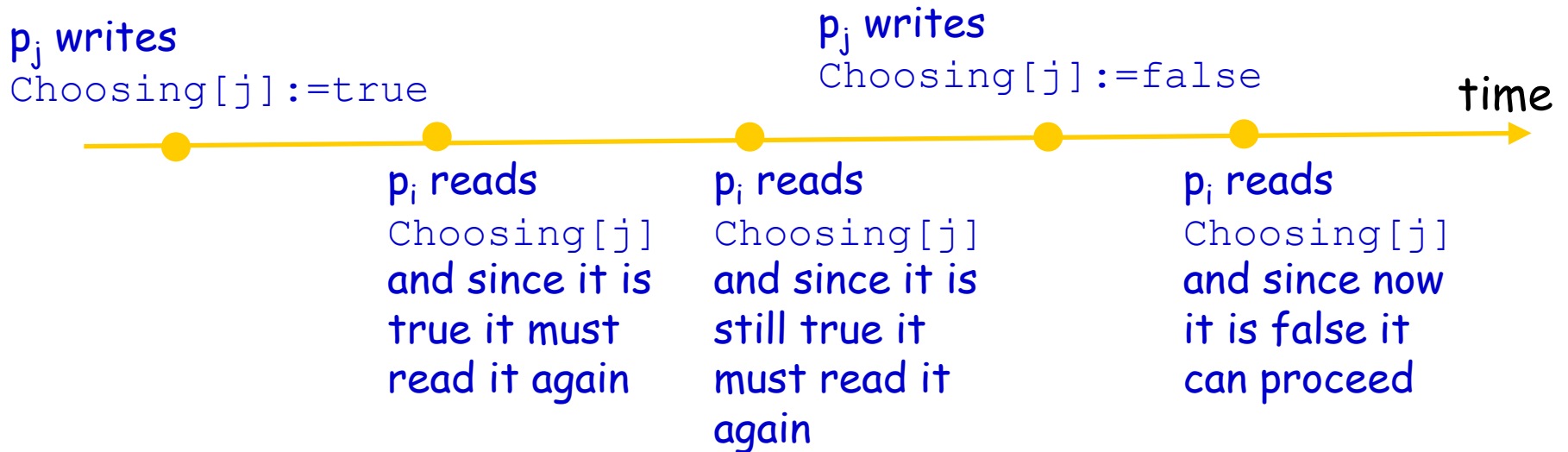
# The max operation

The **max** instruction will look like as follows on the timeline of  $p_i$ :



# The semaphore

The **wait until** instruction will look like as follows on the timeline of  $p_i$ :





# BA Provides Mutual Exclusion

**Lemma 1:** If  $p_i$  is in the **critical section (CS)**, then  $\text{Number}[i] > 0$ .

**Proof:** Trivial: in the doorway section it always takes a number  $> 0$ .

**Lemma 2:** If  $p_i$  is in the **CS** and  $\text{Number}[k] \neq 0$  ( $k \neq i$ ), then  $(\text{Number}[k], k) > (\text{Number}[i], i)$ .

**Proof:** Observe that a chosen number changes only after exiting from the **CS**, and that a number is  $\neq 0$  iff the corresponding processor is either in the entry (bakery) section or in the **CS**. Now, since  $p_i$  is in the **CS**, it passed the second wait statement for  $j=k$ .

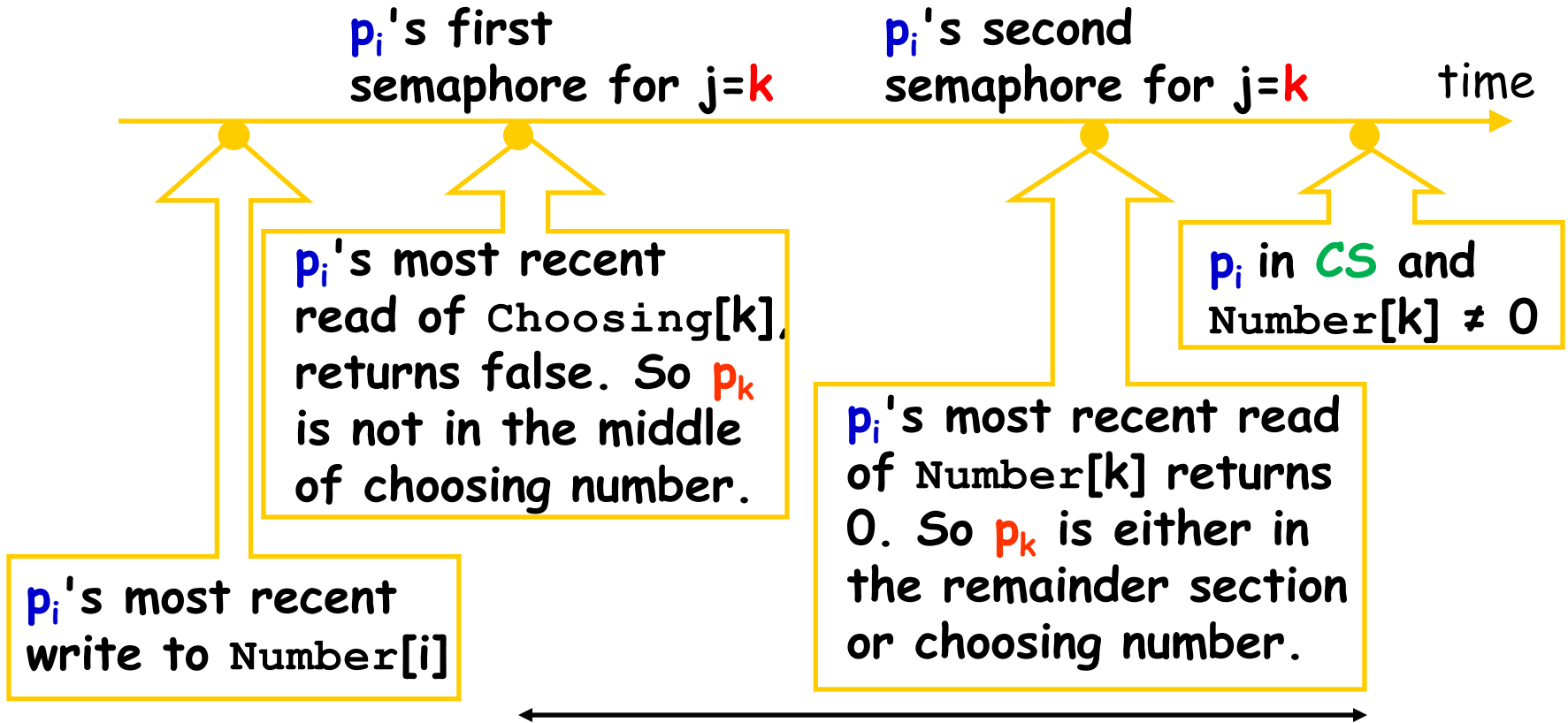
There are two cases:



Case 1: returns 0

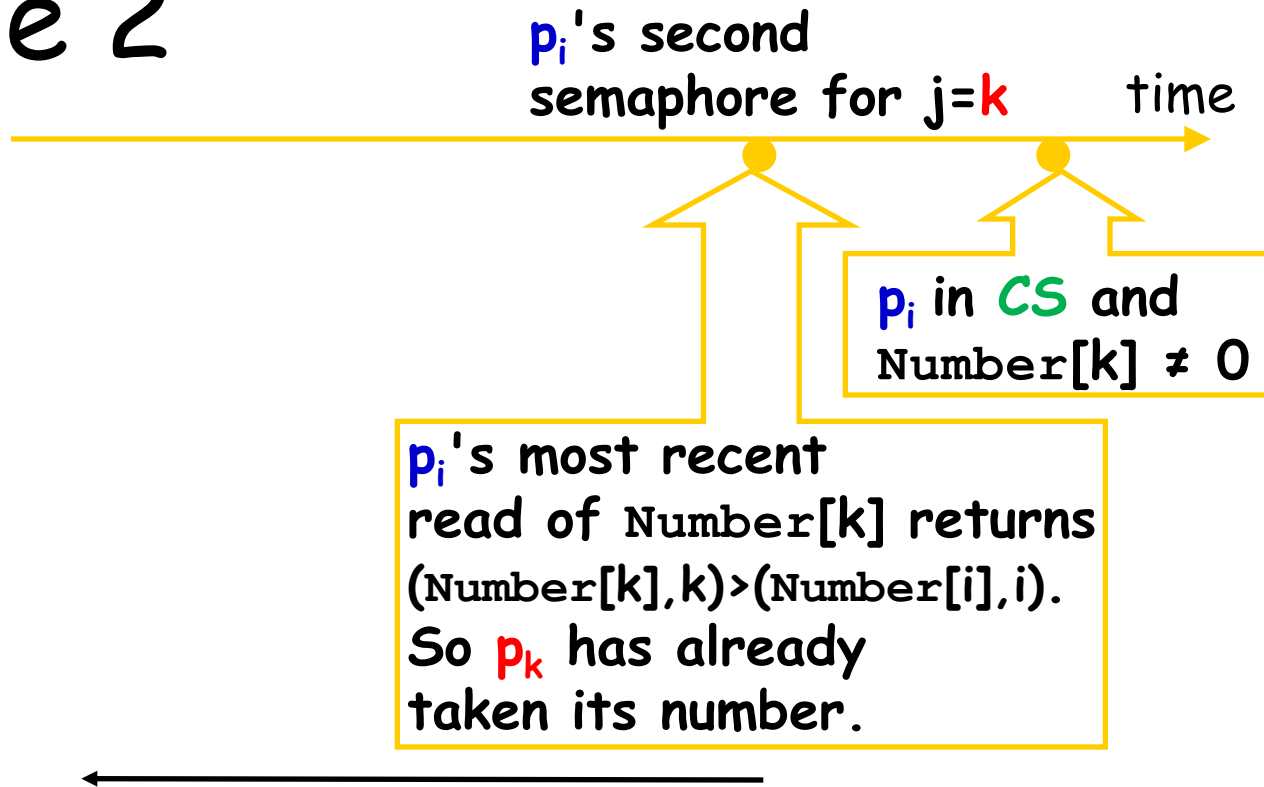
Case 2: returns  $(\text{Number}[k], k) > (\text{Number}[i], i)$

# Case 1



So  $p_k$  starts choosing its number in this interval, sees  $p_i$ 's number, and then will choose a **larger** one (i.e.,  $Number[k] > (Number[i])$ ); so, it will never enter in CS before than  $p_i$ , which means that its number does not change all over the time  $p_i$  is in the CS, and so the claim is true

# Case 2



So  $p_k$  chooses  $\text{Number}[k] \geq \text{Number}[i]$  in this interval, and does not change it until  $p_i$  exits from the CS, since it cannot overtake  $p_i$ . Indeed,  $p_k$  will be stopped by  $p_i$  either in the first wait statement (in case  $p_k$  finished its choice before than  $p_i$  and  $p_i$  is still choosing its number), or in the second one (since  $(\text{Number}[i], i) < (\text{Number}[k], k)$ ). Thus, it will remain  $(\text{Number}[i], i) < (\text{Number}[k], k)$  until  $p_i$  finishes its CS, and the claim follows.

END of PROOF

# Mutual Exclusion for BA

- **Mutual Exclusion:** Suppose  $p_i$  and  $p_k$  are simultaneously in  $CS$ ,  $i \neq k$ .
  - By Lemma 1, both have number  $> 0$ .
  - Since  $\text{Number}[k], \text{Number}[i] \neq 0$ , by Lemma 2
    - $(\text{Number}[k], k) > (\text{Number}[i], i)$  and
    - $(\text{Number}[i], i) > (\text{Number}[k], k)$

**Contradiction!**

# No Lockout for BA

- Assume in contradiction there is a **starved** processor.
- Starved processors must be stuck at the semaphores, *not* while choosing a number.
- Starved processors can be stuck only at the **second semaphore**, since sooner or later the `Choosing` variable of each processor will become false
- Let  $p_i$  be a starved processor with smallest `Number[i],i`.
- Any processor entering entry section after that  $p_i$  chose (i.e., wrote) its number, will choose a larger number, and therefore cannot overtake  $p_i$
- Every processor with a smaller ticket eventually enters **CS** (not starved) and exits, setting its number to 0. So, in the future, its number will be either 0 or larger than `Number[i]`
- Thus  $p_i$  cannot be stuck at the second semaphore forever by another processor.

**Contradiction!**

# What about bounded waiting?

**YES:** It is easy to see that any processor in the entry section can be overtaken at most **once** by any other processor (and so in total it can be overtaken at most  **$n-1$**  times).

# Space Complexity of BA

- Number of shared variables is  $2n$
- Choosing variables are booleans
- Number variables are **unbounded**: as long as the **CS** is occupied and some processor enters the entry section, the ticket number increases
- Is it possible for an algorithm to use less shared space?

# Bounded-space 2-Processor Mutex Algorithm with no deadlock (J.L. Peterson, 1981)

- Start with a bounded-variables algorithm for 2 processors with **no deadlock**, then extend to **no lockout**, then extend to  $n$  processors.
- Use 2 binary shared read/write variables (intuition: if  $p_i$  wants to enter into the CS, then it sets  $w[i]$  to 1):
  - $w[0]$ : initially 0, written by  $p_0$  and read by  $p_1$
  - $w[1]$ : initially 0, written by  $p_1$  and read by  $p_0$
- **Asymmetric** (or **non-homogenous**) code:  $p_0$  always has priority over  $p_1$



# Bounded-space 2-Processor Mutex Algorithm with no deadlock

Code for  $p_0$ 's entry section:

```
1  .  
2  .  
3  W[0] := 1  
4  .  
5  .  
6  →wait until W[1] = 0
```

Semaphore

Code for  $p_0$ 's exit section:

```
7  .  
8  W[0] := 0
```

# Bounded-space 2-Processor Mutex Algorithm with no deadlock

Code for  $p_1$ 's entry section:

```
1  W[1] := 0
2  wait until W[0] = 0
3  W[1] := 1
4  .
5      if (W[0] = 1) then goto Line 1
6  .
```

Semaphore



Code for  $p_1$ 's exit section:

```
7  .
8  W[1] := 0
```

# Analysis

- Satisfies **mutual exclusion**: processors use  $W$  variables to make sure of this (a processor **enters** only when its own  $W$  variable is set to 1 and the other  $W$  variable is **seen** to be 0; notice that when  $p_1$  is in the CS and  $p_0$  is waiting at Line 5 in the entry, then both  $W[0]$  and  $W[1]$  are equal to 1, while if  $p_0$  is in the CS and  $p_1$  is waiting at Line 2 in the entry, then  $W[0]=1$ , while  $W[1]=0$ )
  - Satisfies **no-deadlock**: if  $p_0$  wants to enter, it cannot be locked by  $p_1$  (since  $p_1$  will be forced to set  $W[1]:=0$ )
  - But unfair w.r.t.  $p_1$  (it can remain locked, if  $p_0$  sets  $W[0]$  to 1 continuously between line 3 and 5 of  $p_1$  execution)
- ⇒ Fix it by having the processors alternate in having the priority

# Bounded-space 2-Processor Mutex Algorithm with no lockout

Uses 3 binary shared read/write variables and is **symmetric**:

- $W[0]$ : initially 0, written by  $p_0$  and read by  $p_1$
- $W[1]$ : initially 0, written by  $p_1$  and read by  $p_0$
- `Priority`: initially 0, written and read by both

# Bounded-space 2-Processor Mutex Algorithm with no lockout

Code for  $p_i$ 's entry section:

```
1  W[i] := 0
2  wait until W[1-i] = 0 or Priority = i
3  W[i] := 1
4  if (Priority = 1-i) then
5      if (W[1-i] = 1) then goto Line 1
6  else wait until (W[1-i] = 0)
```

**Semaphores**



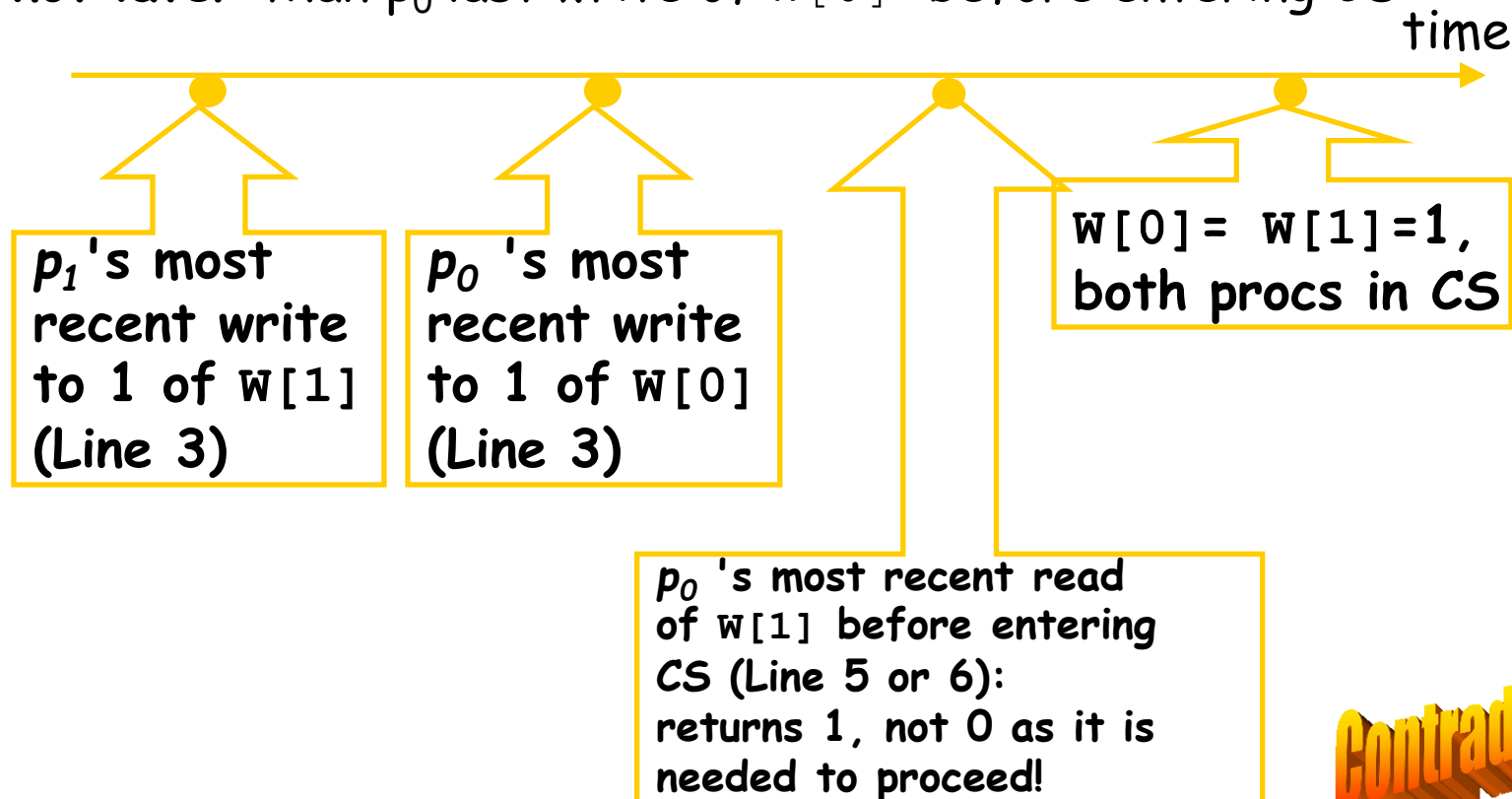
Code for  $p_i$ 's exit section:

```
7  Priority := 1-i
8  W[i] := 0
```

# Analysis: ME

## Mutual Exclusion:

- Suppose in contradiction  $p_0$  and  $p_1$  are simultaneously in CS, and then their  $w[]$  variables are set to 1.
- W.l.o.g., assume  $p_1$  last write of  $w[1]$  before entering CS happens not later than  $p_0$  last write of  $w[0]$  before entering CS

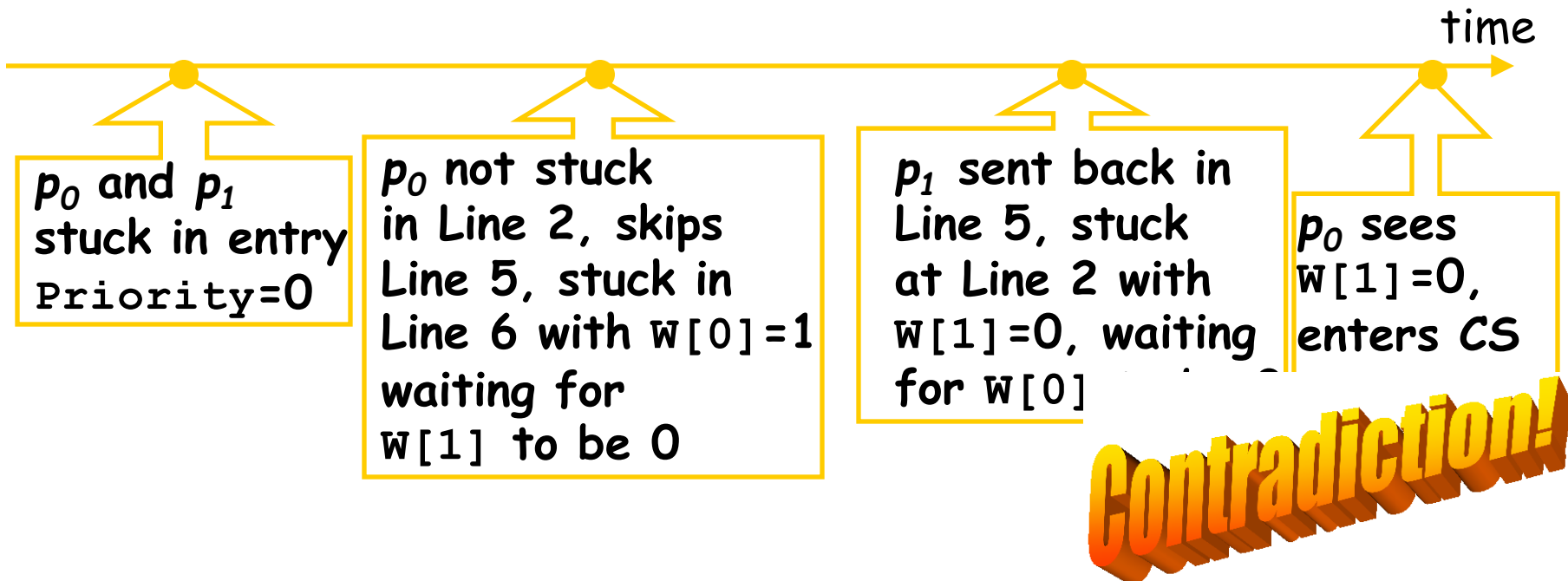


# Analysis: No-Deadlock

- Useful for showing **no-lockout**.
- If one processor ever stays in the **remainder** section forever, then its  $W[]$  variable will constantly be equal to 0, and so the other processor cannot be starved (it cannot be stuck at Line 5 or 6)
- So any deadlock would starve both processors in the **entry** section

# Analysis: No-Deadlock

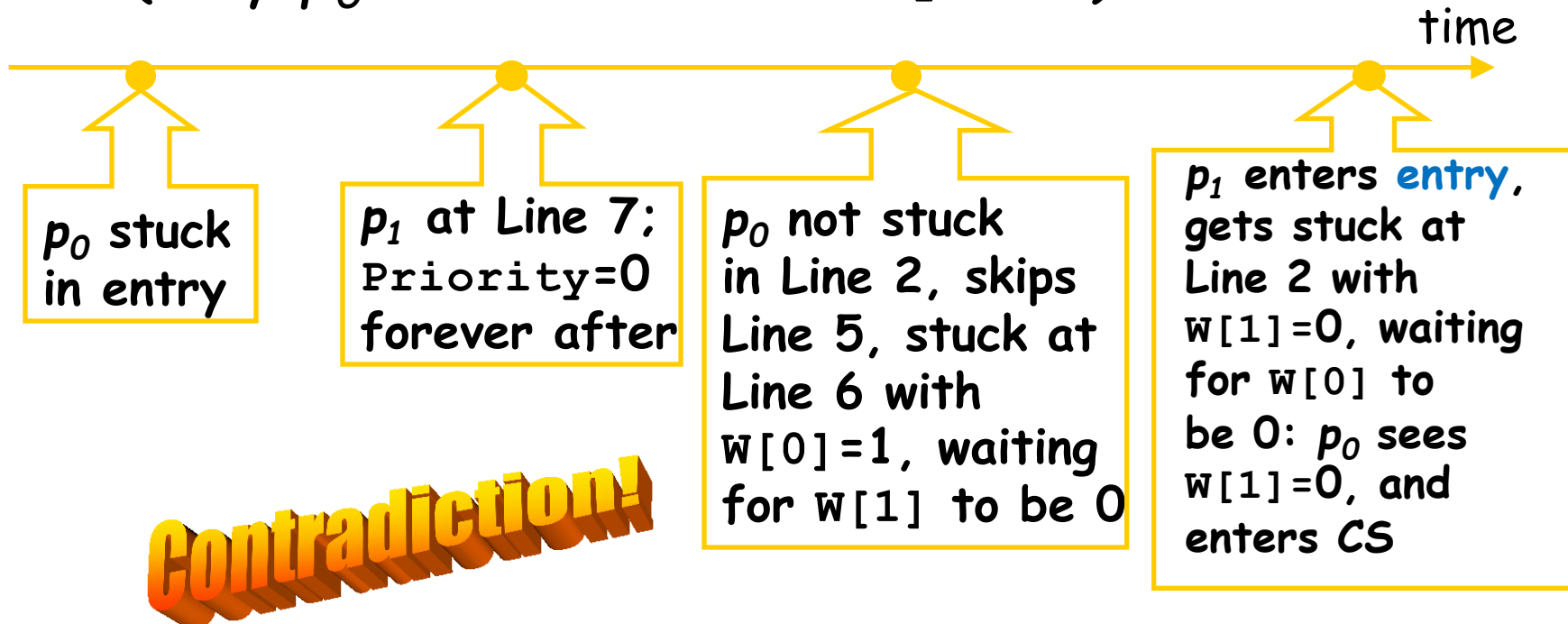
- Suppose in contradiction there is deadlock, and w.l.o.g., suppose `Priority` gets stuck at 0 after both processors are stuck in their entry sections (indeed `Priority` cannot be changed within the entry section):





# Analysis: No-Lockout

- Suppose in contradiction  $p_0$ , w.l.o.g., is starved.
- Since there is no deadlock,  $p_1$  enters CS infinitely often.
- The first time  $p_1$  executes Line 7 in **exit** section after  $p_0$  is stuck in entry, `Priority` gets stuck at 0 (only  $p_0$  can set `Priority` to 1)



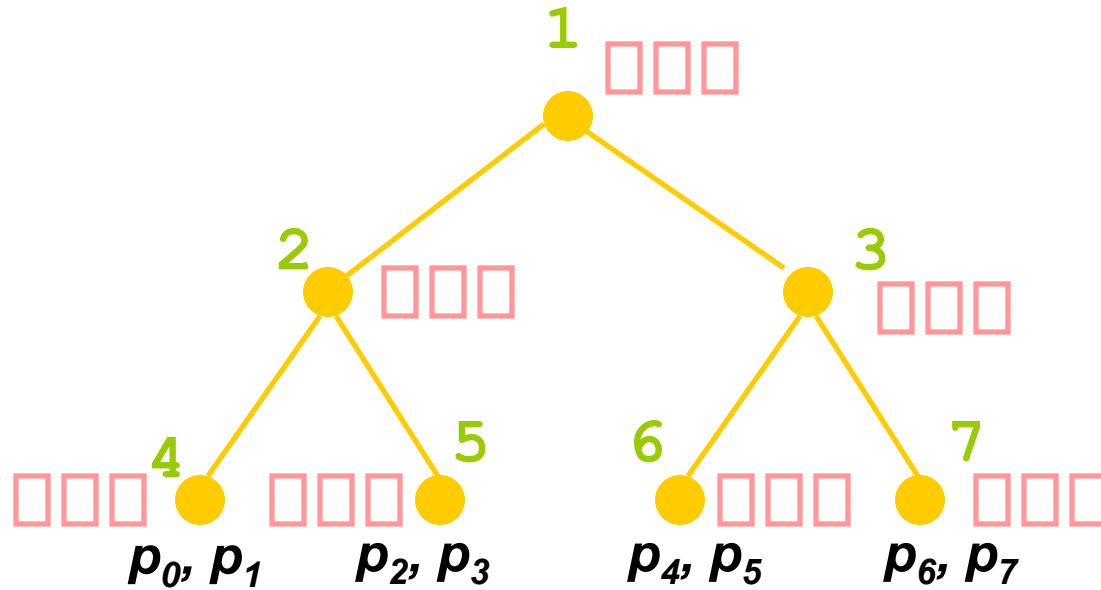
# Bounded Waiting?

- **NO:** A processor, even if having priority, might be overtaken repeatedly (in principle, an **unbounded** number of times) when it is in between Line 2 and 3.

# Bounded-space $n$ -Processor Mutex Algorithm with no lockout

- Can we get a bounded-space **no-lockout** mutex algorithm for  $n > 2$  processors?
- Yes! For the sake of simplicity, assume that  $n = 2^k$ , for some  $k > 1$ .
- Based on the notion of a **tournament tree**: complete binary tree with  $n-1$  nodes
  - tree is conceptual only! does not represent message passing channels
- A copy of the 2-processor algorithm is associated with each tree node
  - includes separate copies of the 3 shared variables

# Tournament Tree



We label the tree nodes from top to down and from left to right, from 1 to  $n-1$ ; then, associate processor  $p_i$ ,  $i=0, \dots, n-1$ , with the leaf node labelled  $2^{k-1} + \lfloor i/2 \rfloor$ , where  $k = \log n$  (recall that  $n=2^k$ ). Notice that, in general, if  $n \neq 2^k$ , then we complete the tree by adding "dummy" leaves

# Tournament Tree Mutex Algorithm

- Each processor begins entry section at the associated leaf (2 processors per leaf)
- A processor proceeds to next level in the tree by winning the 2-processor competition for current tree node:
  - on left side, plays role of  $p_0$
  - on right side, plays role of  $p_1$
- When a processor wins the 2-processor algorithm associated with the tree root, it enters CS.

# The entry code

```
procedure Node(v: integer; side: 0..1)
```

```
L: { wantsidev := 0  
    { wait until (want1-sidev = 0 or priorityv = side)  
    { wantsidev := 1  
    { if (priorityv = 1 - side) then  
      { if (want1-sidev = 1) then goto L  
    { else wait until (want1-sidev = 0)  
    { if (v = 1) then /* at the root */  
      { <Critical Section>  
    { else Node([v/2], v mod 2)  
    { wantsidev := 0  
    { priorityv := 1 - side  
end procedure
```

Entry

Exit

# More on TT Algorithm

- Code is recursive
- $p_i$  begins at tree node  $v$  labelled  $2^{k-1} + \lfloor i/2 \rfloor$ , playing role of  $p_{i \bmod 2}$ , where  $k = \log n$ .
- After winning at node  $v$ , "critical section" for node  $v$  is
  - entry code for all nodes on path from  $\lfloor v/2 \rfloor$  to root
  - real critical section
- Finally, executes exit code for all nodes on path from root to  $v$  (in each of these nodes, gives priority to the other side and sets its want variable to 0)

# Analysis

- **Correctness:** based on correctness of 2-processor algorithm and tournament structure:
  - **Mutual exclusion** for TT algorithm follows from ME for 2-processors algorithm at tree root.
  - **No-lockout** for tournament algorithm follows from no-lockout for the 2-processor algorithms at all nodes of tree
- **Space Complexity:**  $3(n-1)$  boolean read/write shared variables.
- **Bounded Waiting?** No, as for the 2-processor algorithm.



# Homework

Consider the mutex problem on a synchronous DS of 8 processors (with ids in 0..7). Show an execution of the tournament tree algorithm by assuming the following:

1. Initially, all the *want* and *priority* variables are equal to 0;
2. The system is **totally synchronous**, i.e., lines of code are executed simultaneously by all processors;
3. Throughout the **entry section**, a processor ends up a round either if it wins the competition (and possibly it enters the CS), or if it executes **7 lines** of codes;
4. If a node enters the CS at round **k**, then it exits at round **k+1**;
5. Throughout the **exit section**, a processor ends up a round after having executed the exit code for a node of the tree;
6.  $p_0, p_1, p_3, p_5$  and  $p_6$  decide to enter the CS in round 1, while the remaining processors decide to enter the CS in round 2.

**Hints:** 16 rounds until the last processor completes the exit section; entering sequence is  $p_0, p_5, p_3, p_6, p_1, p_4, p_2, p_7$