

# Operating Systems

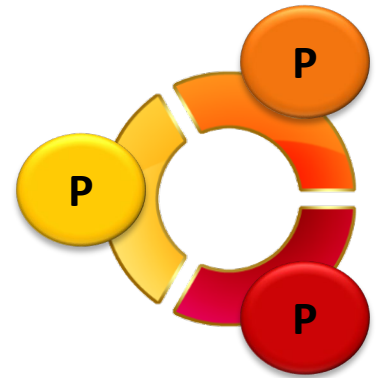
Prof. Yongkun Li

中科大-计算机学院 特任教授

<http://staff.ustc.edu.cn/~ykli>

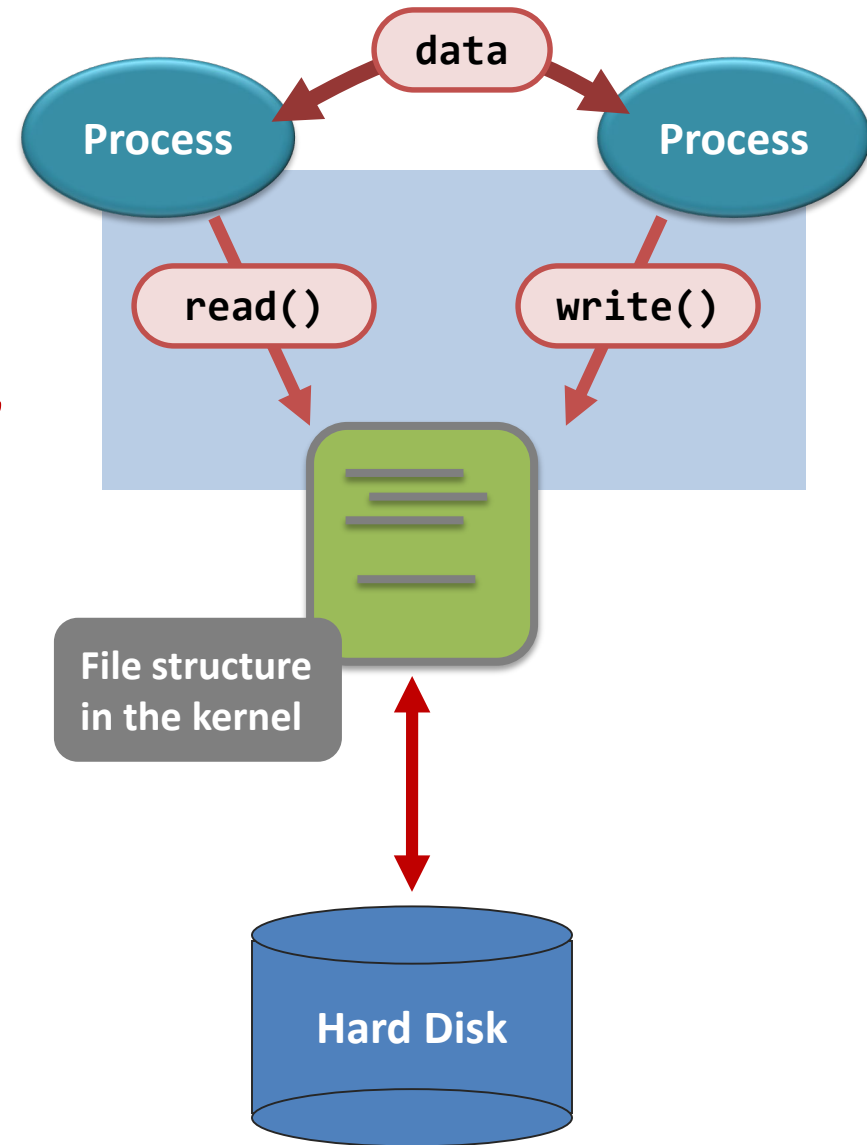
## Ch5 Process Communication & Synchronization -Part 2

# IPC problem: Race condition



# Evil source: the shared objects

- Pipe is implemented with the thought that **there may be more than one process accessing it “at the same time”**
- For shared memory and files, **concurrent access may yield unpredictable outcomes**



# Understanding the problem...

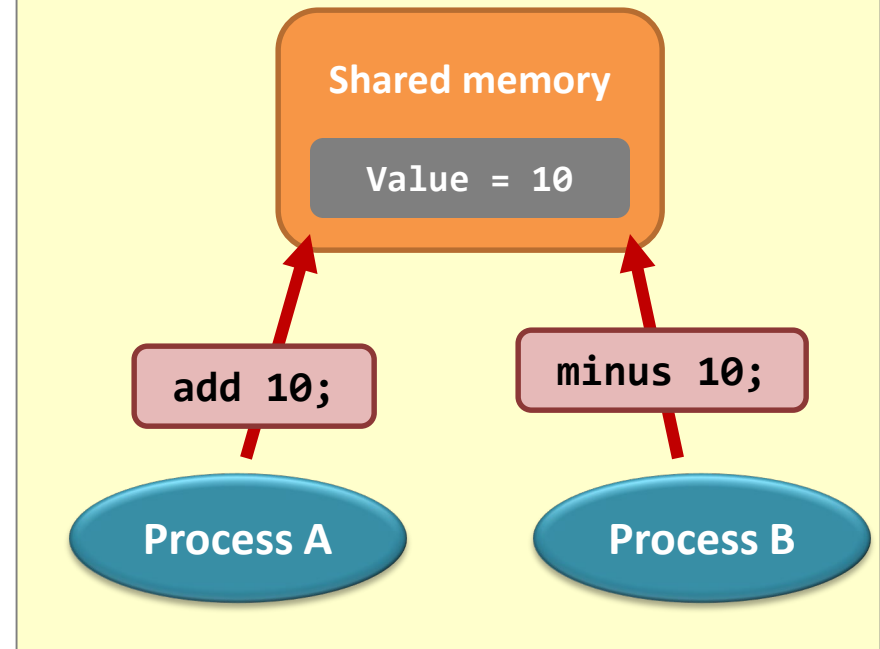
## High-level language for Program A

- 1 attach to the shared memory X;
- 2 add 10 to X;
- 3 exit;

## High-level language for Program B

- 1 attach to the shared memory X;
- 2 minus 10 to X;
- 3 exit;

## The Scenario



Guess what the final result should be?

It may be 10, 0 or 20, can you believe it?

# Understanding the problem...

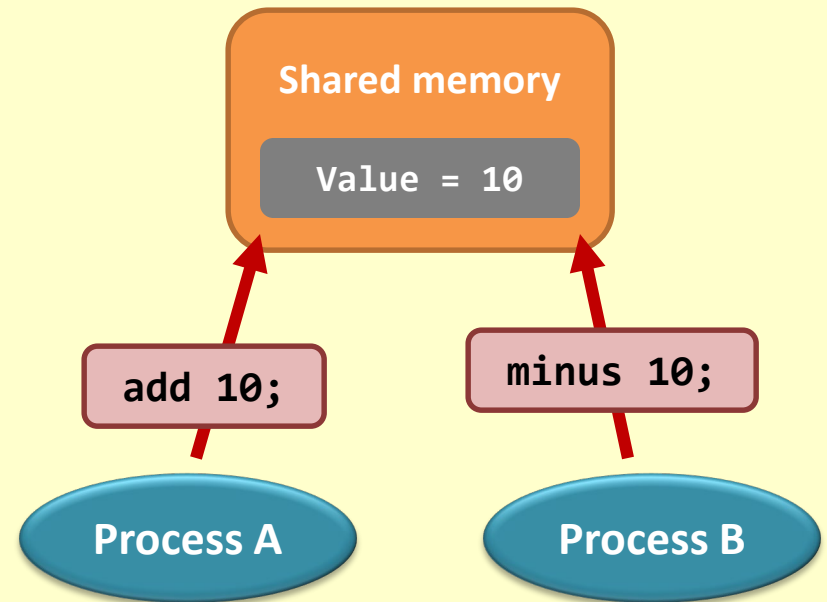
## High-level language for Program A

```
1 attach to the shared memory X;  
2 add 10 to X;  
3 exit;
```

## High-level language for Program B

```
1 attach to the shared memory X;  
2 minus 10 to X;  
3 exit;
```

## The Scenario



Remember the flow of executing a program and the system hierarchy?

# Understanding the problem...

## High-level language for Program A

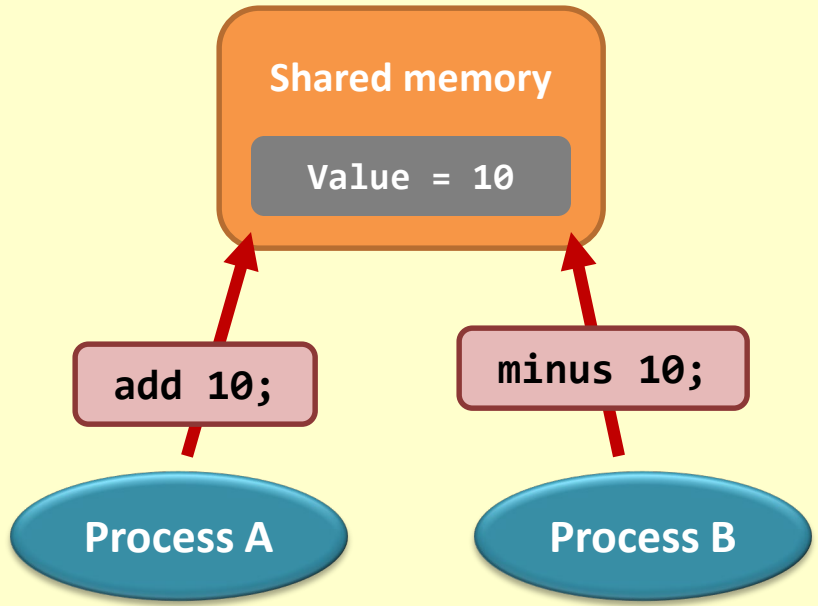
```
1 attach to the shared memory X;  
2 add 10 to X;  
3 exit;
```

This operation is not atomic

## Partial low-level language for Program A

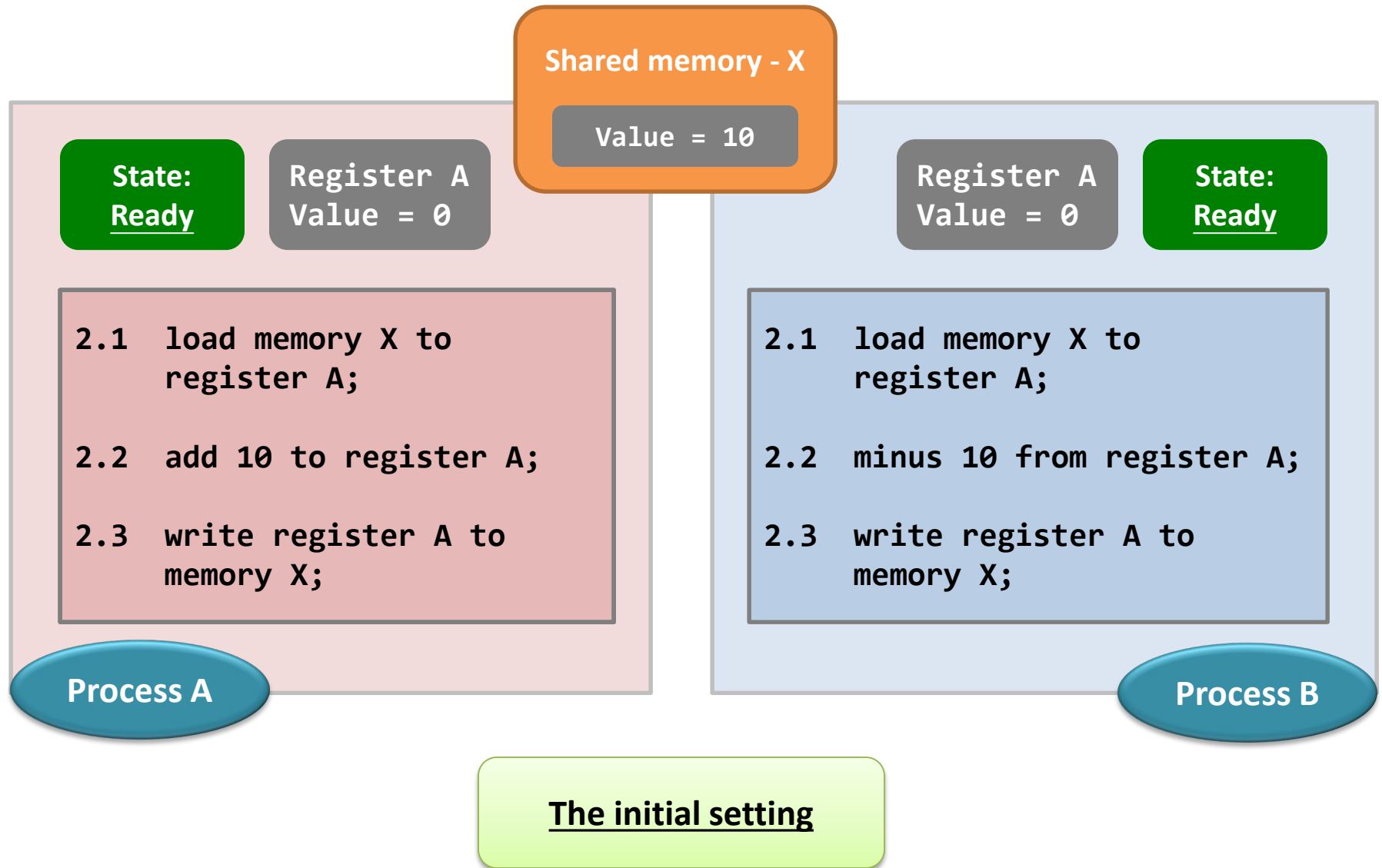
```
1 attach to the shared memory X;  
.....  
2.1 load memory X to register A;  
2.2 add 10 to register A;  
2.3 write register A to memory X;  
.....  
3 exit;
```

## The Scenario



Guess what? This code block is evil!

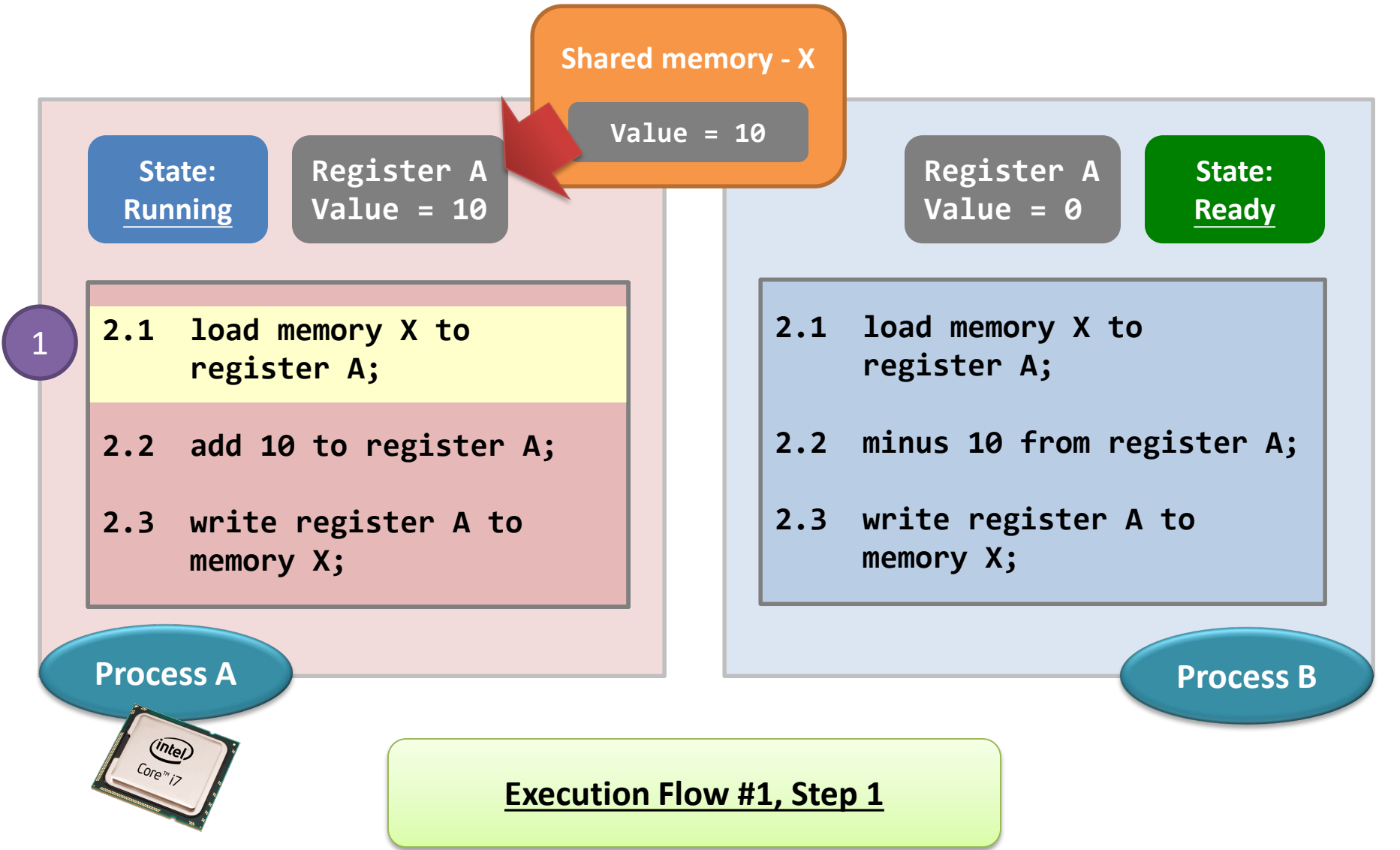
# Understanding the problem...



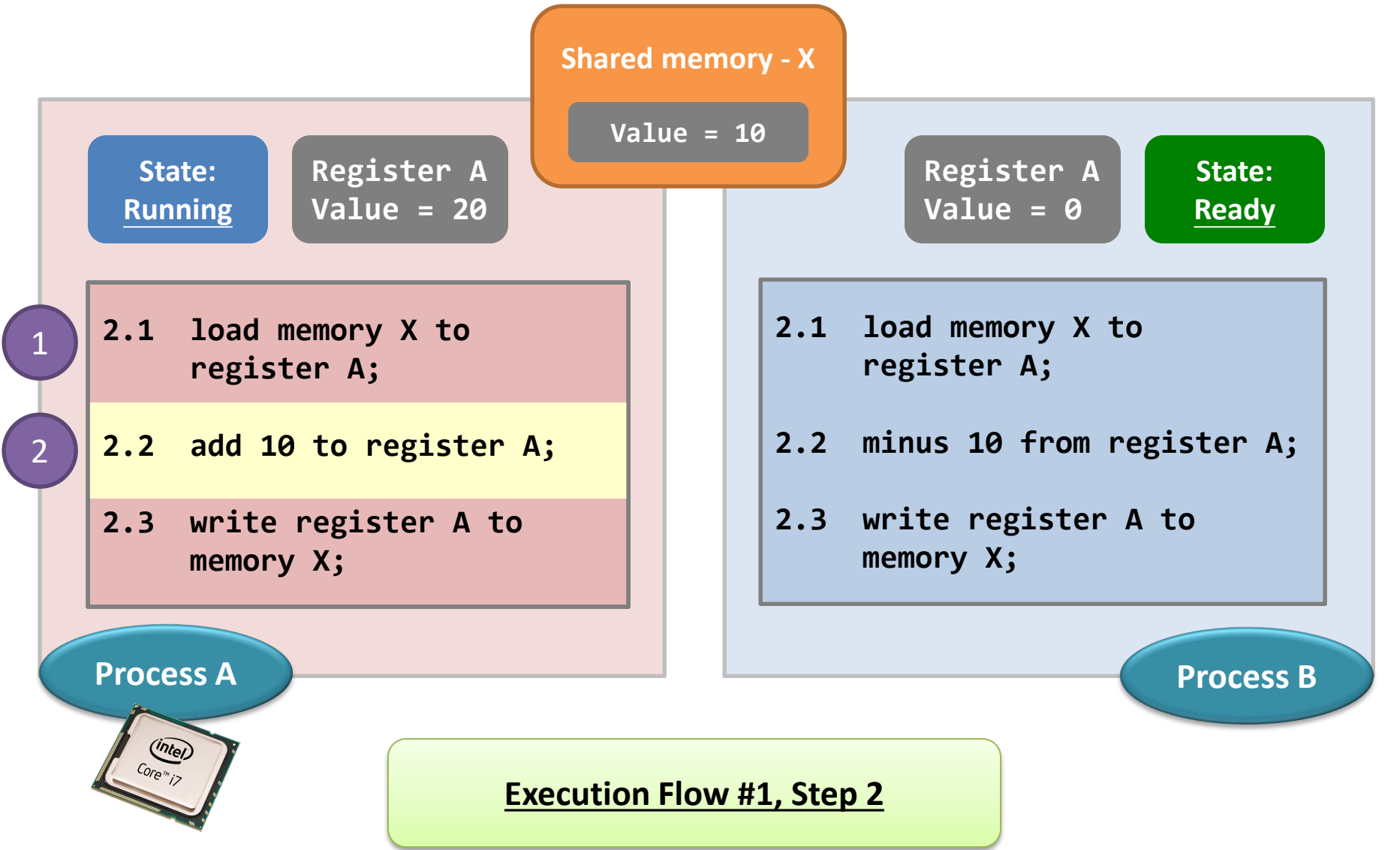
# Execution Flow #1



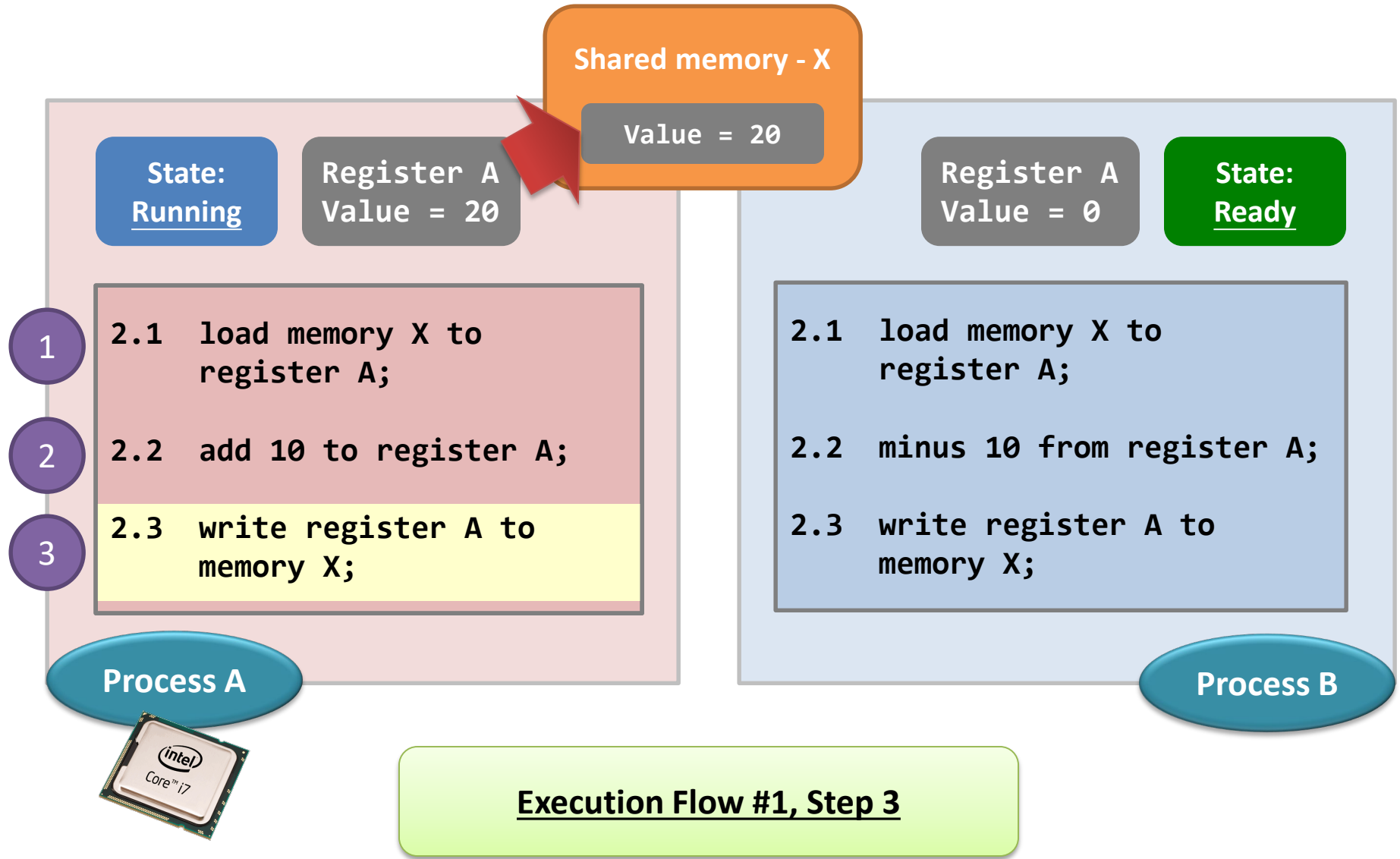
# Problem not yet arise...



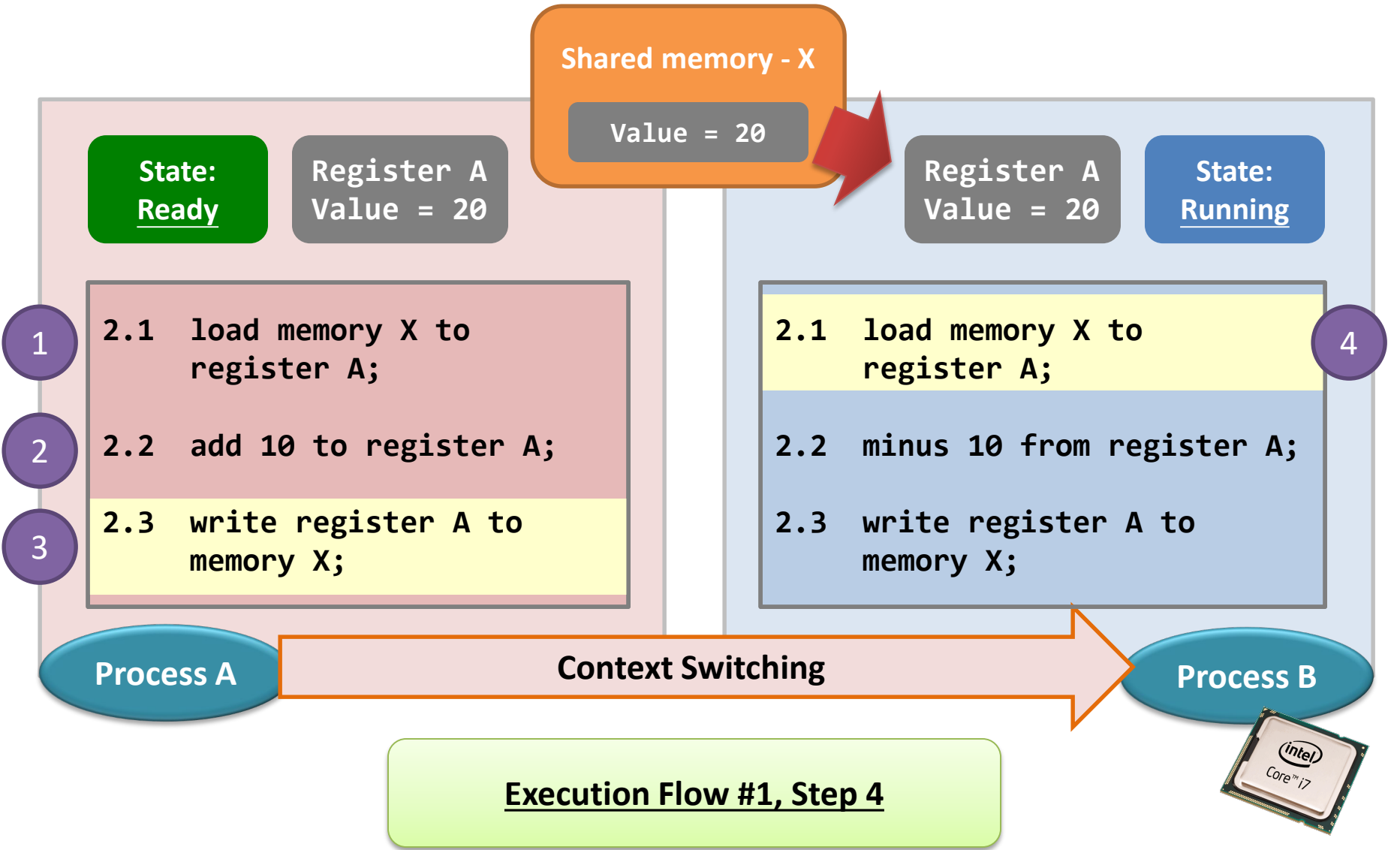
# Problem not yet arise...



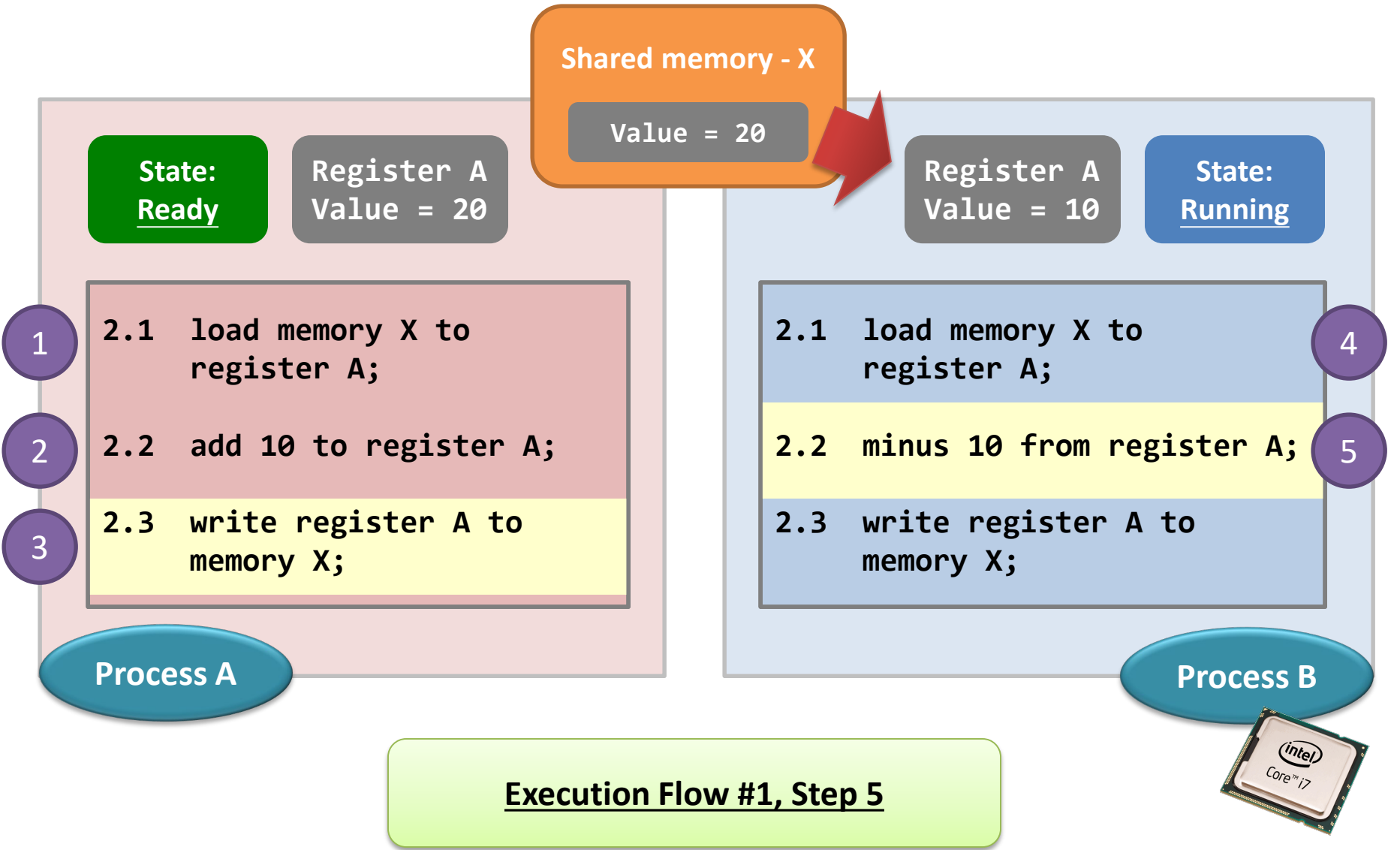
# Problem not yet arise...



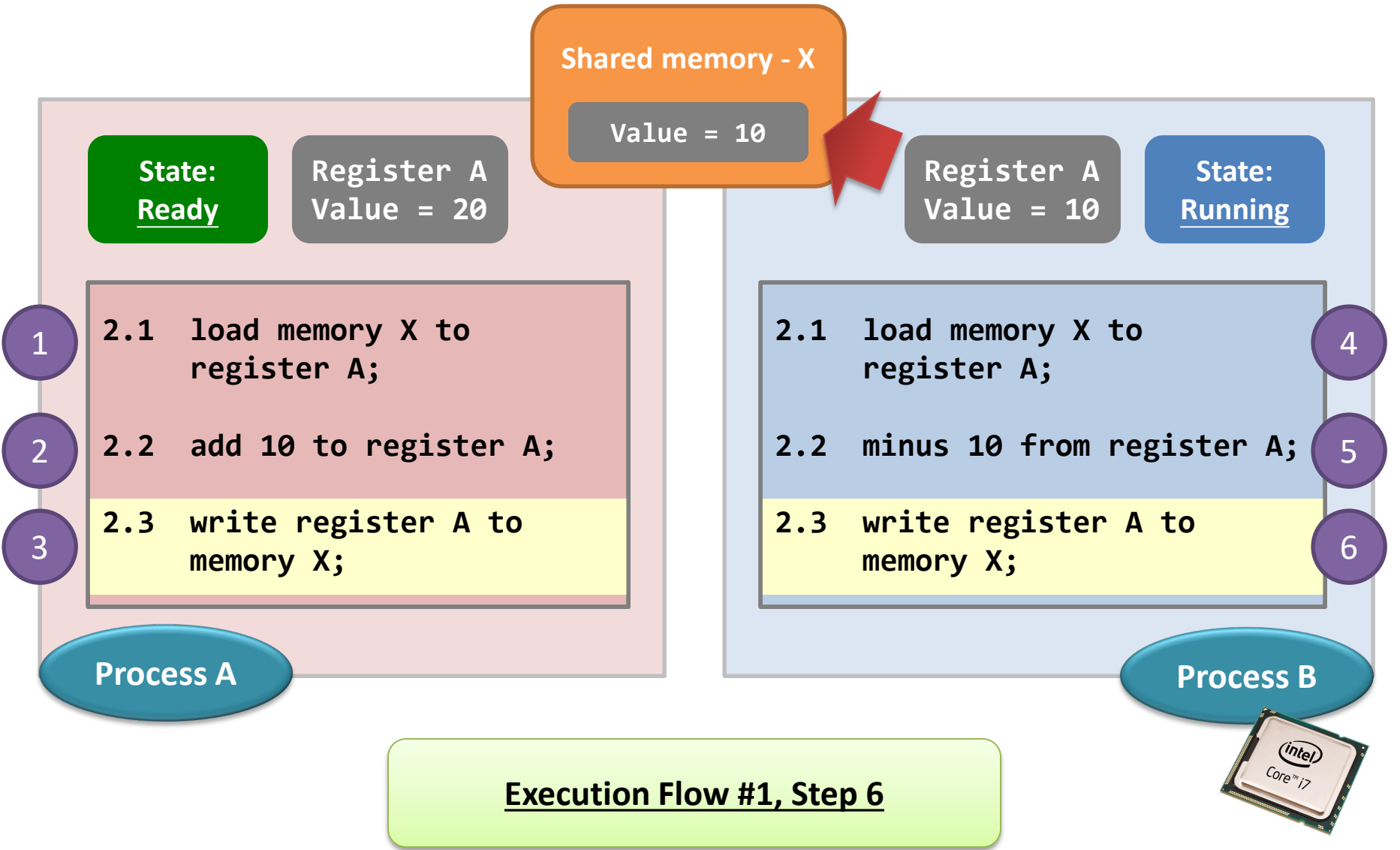
# Problem not yet arise...



# Problem not yet arise...

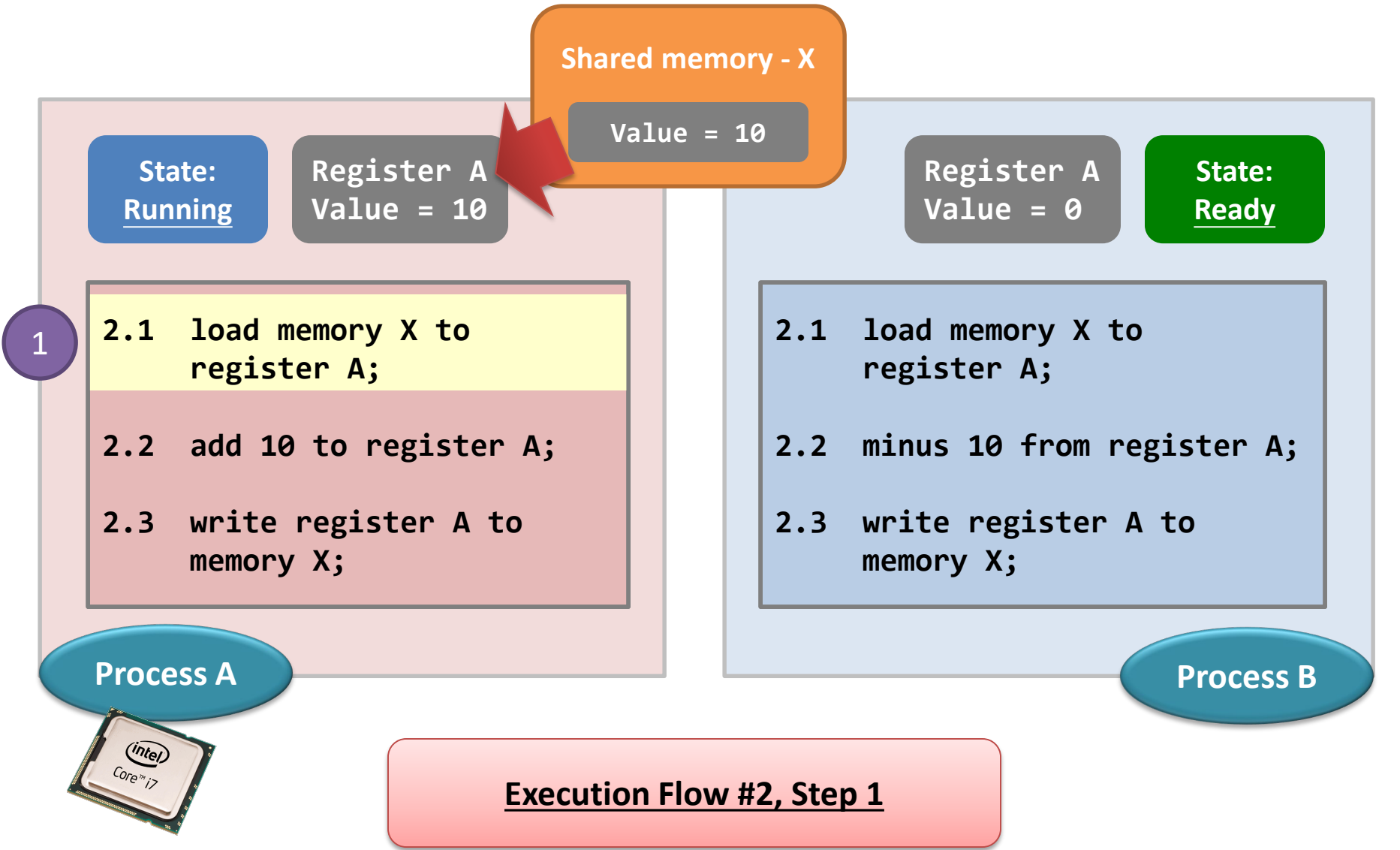


# Problem not yet arise...



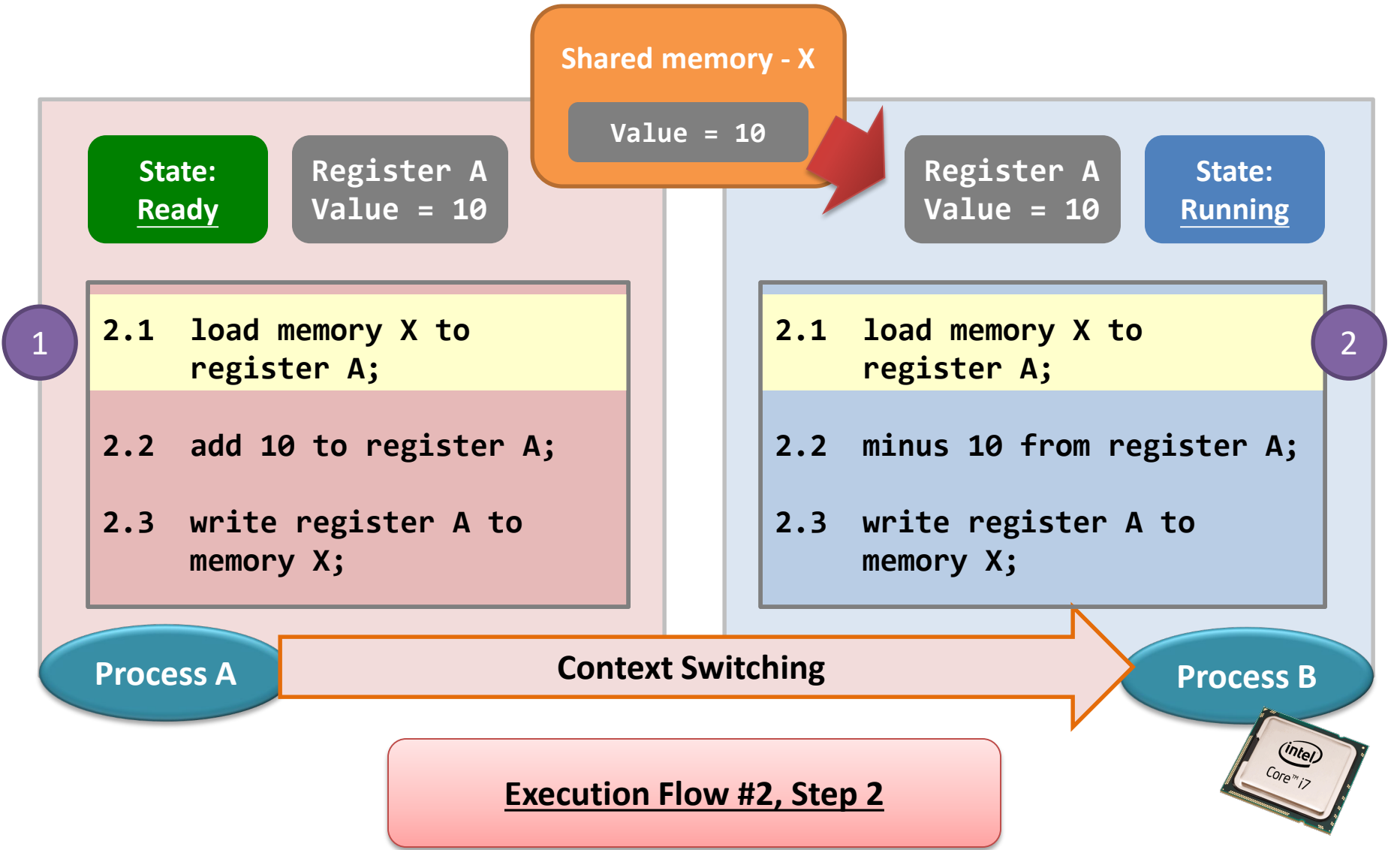
## **Execution Flow #2**

# Problem arise...

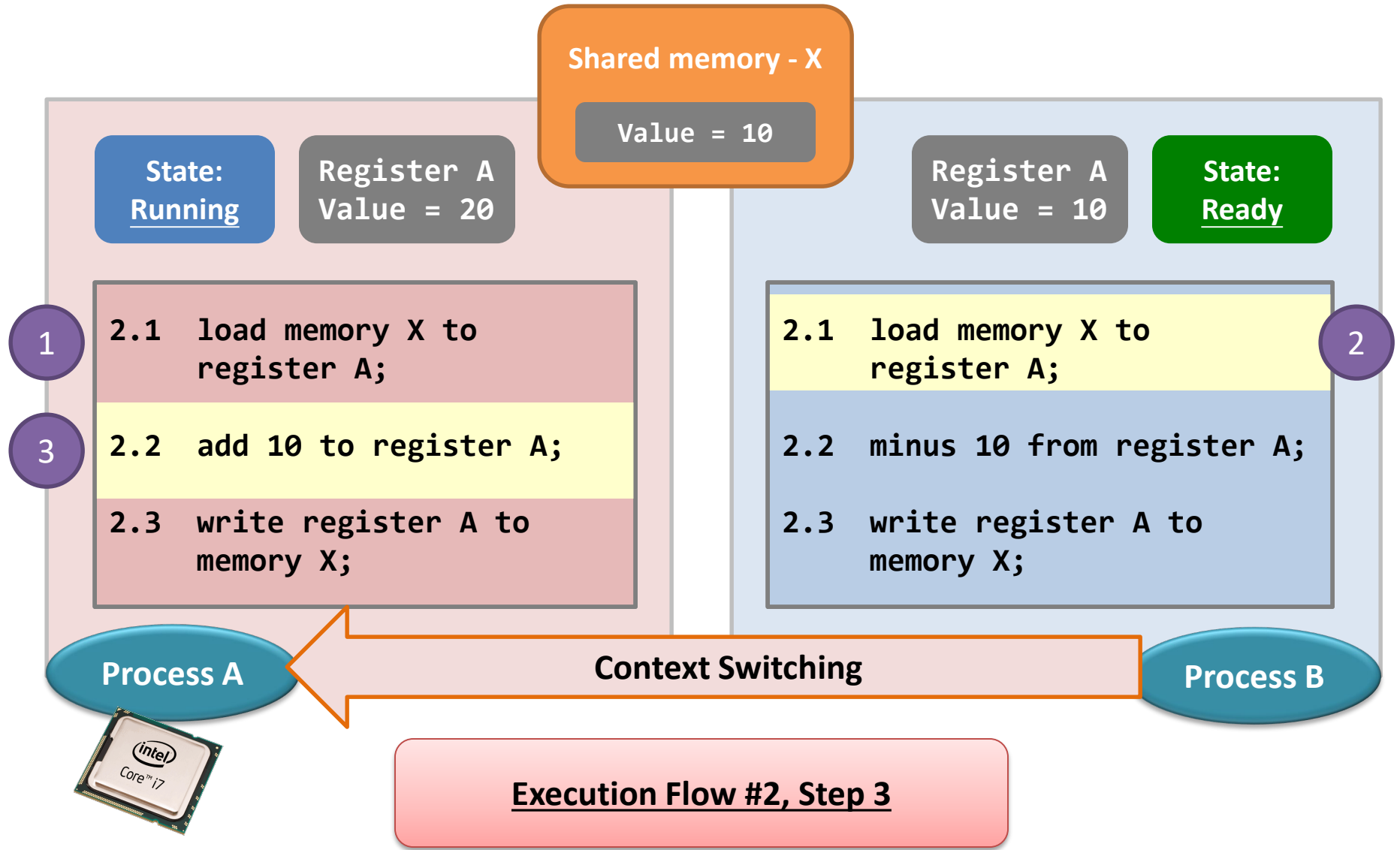




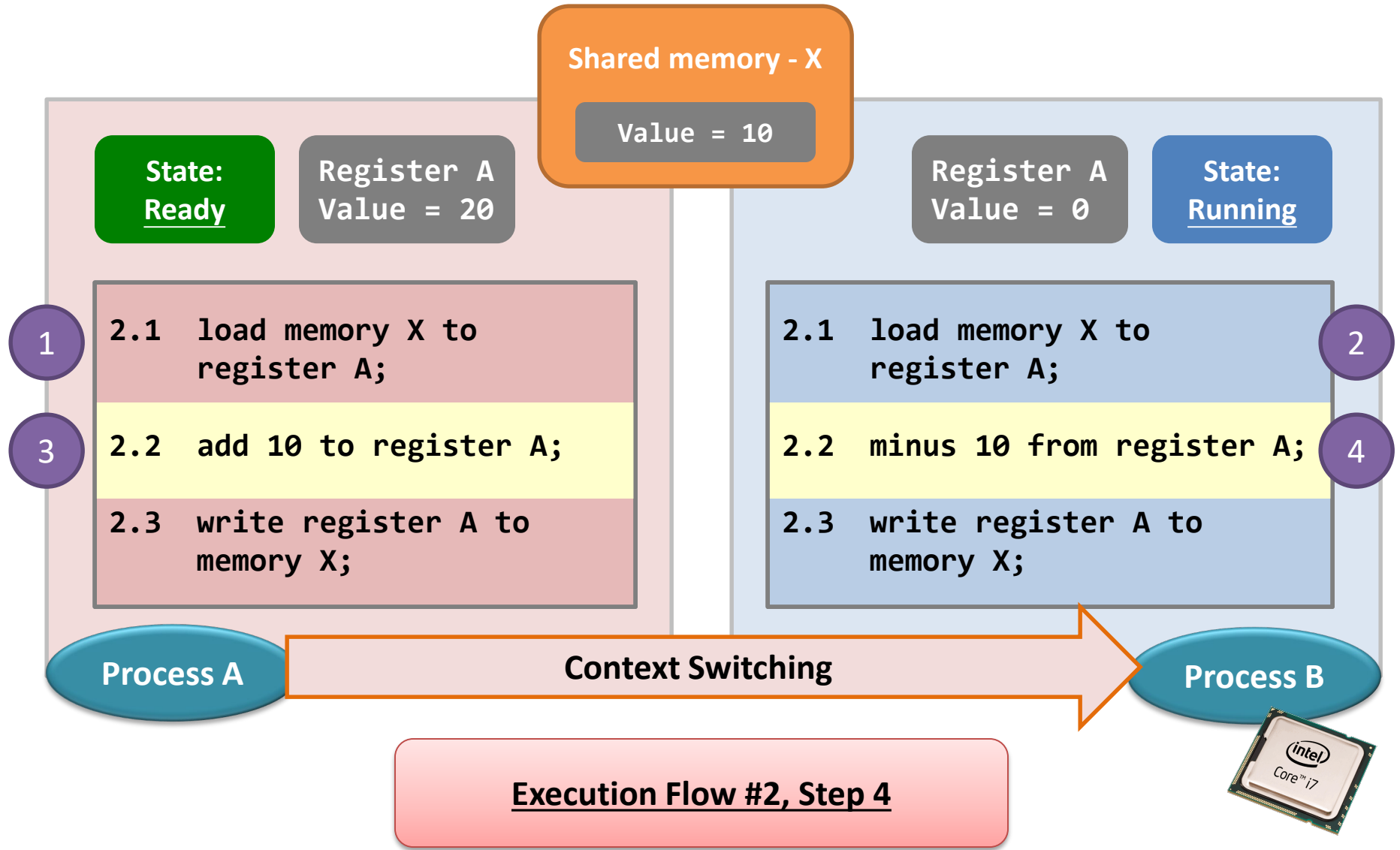
# Problem arise...



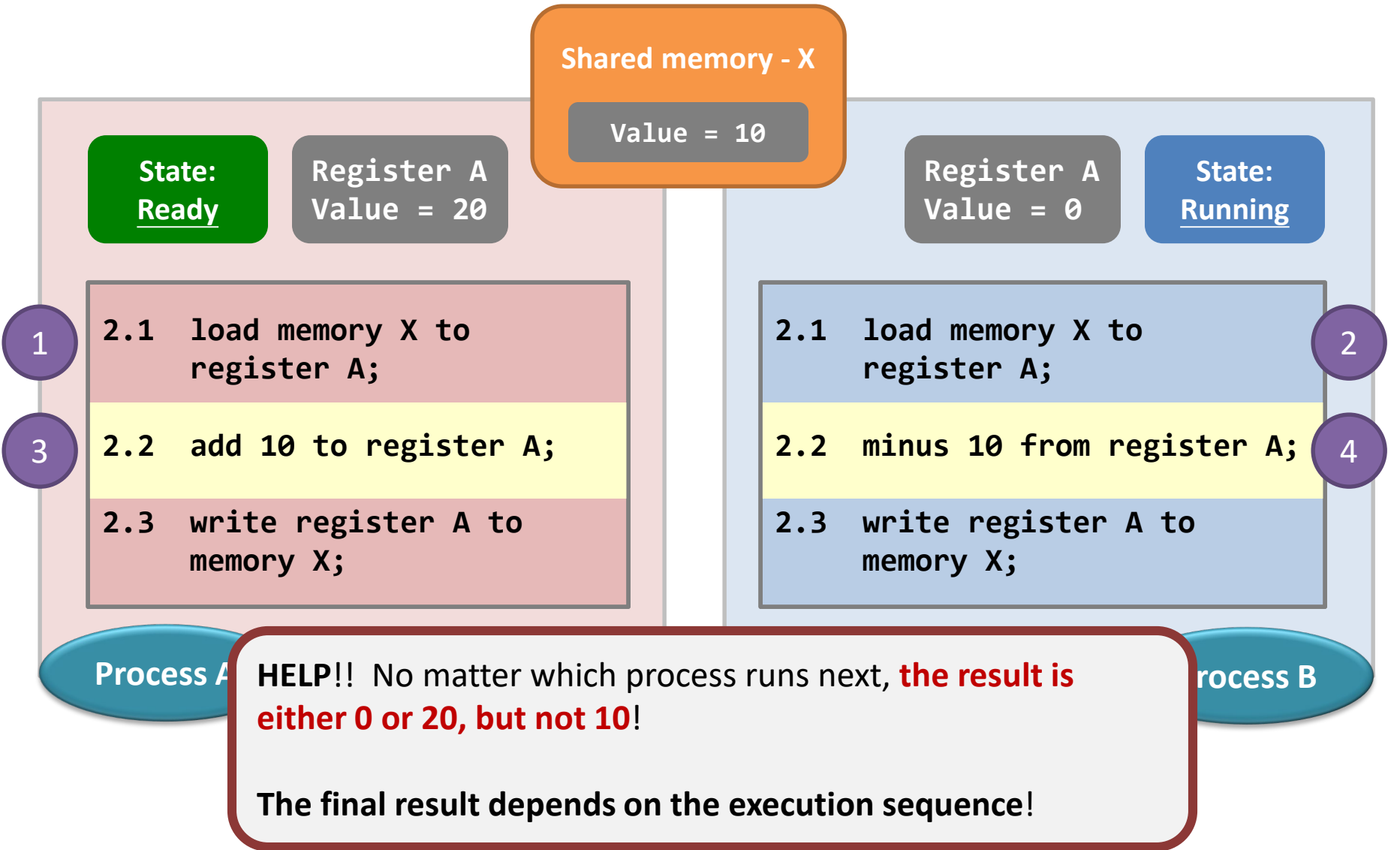
# Problem arise...



# Problem arise...



# Problem arise...

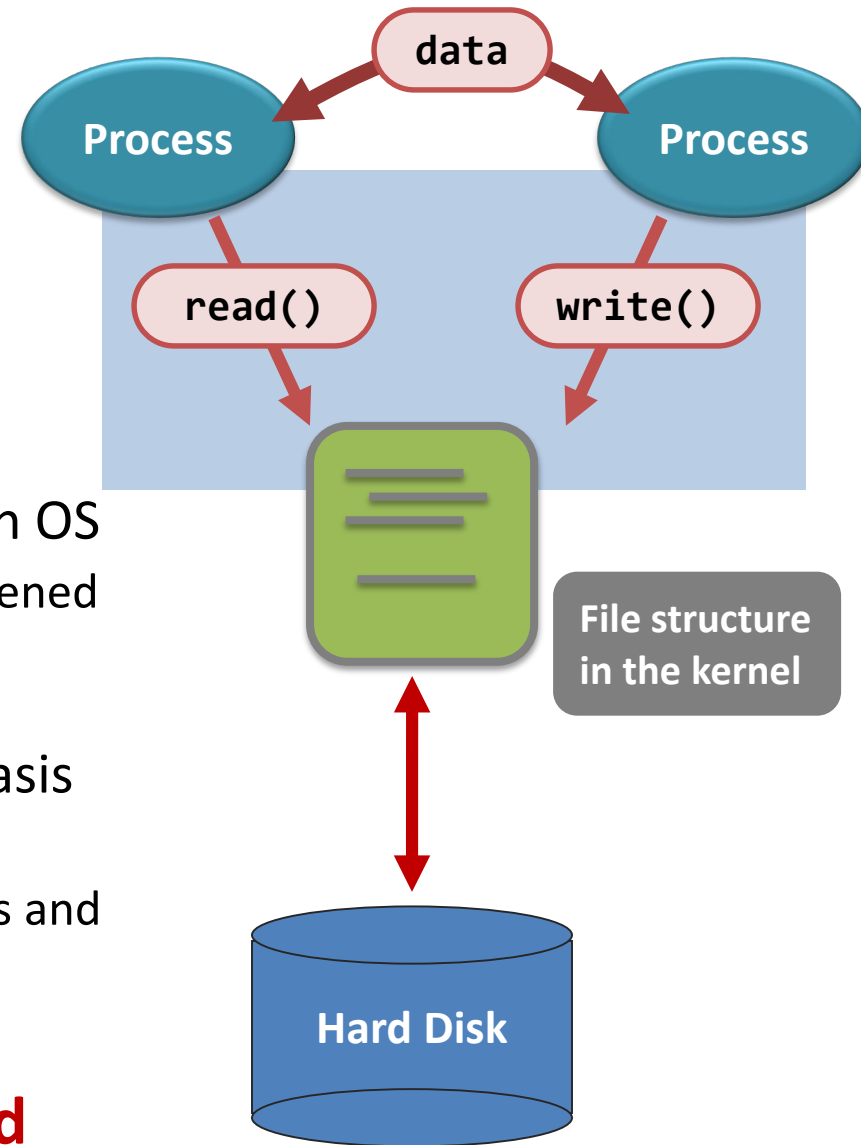


# Race condition – the curse

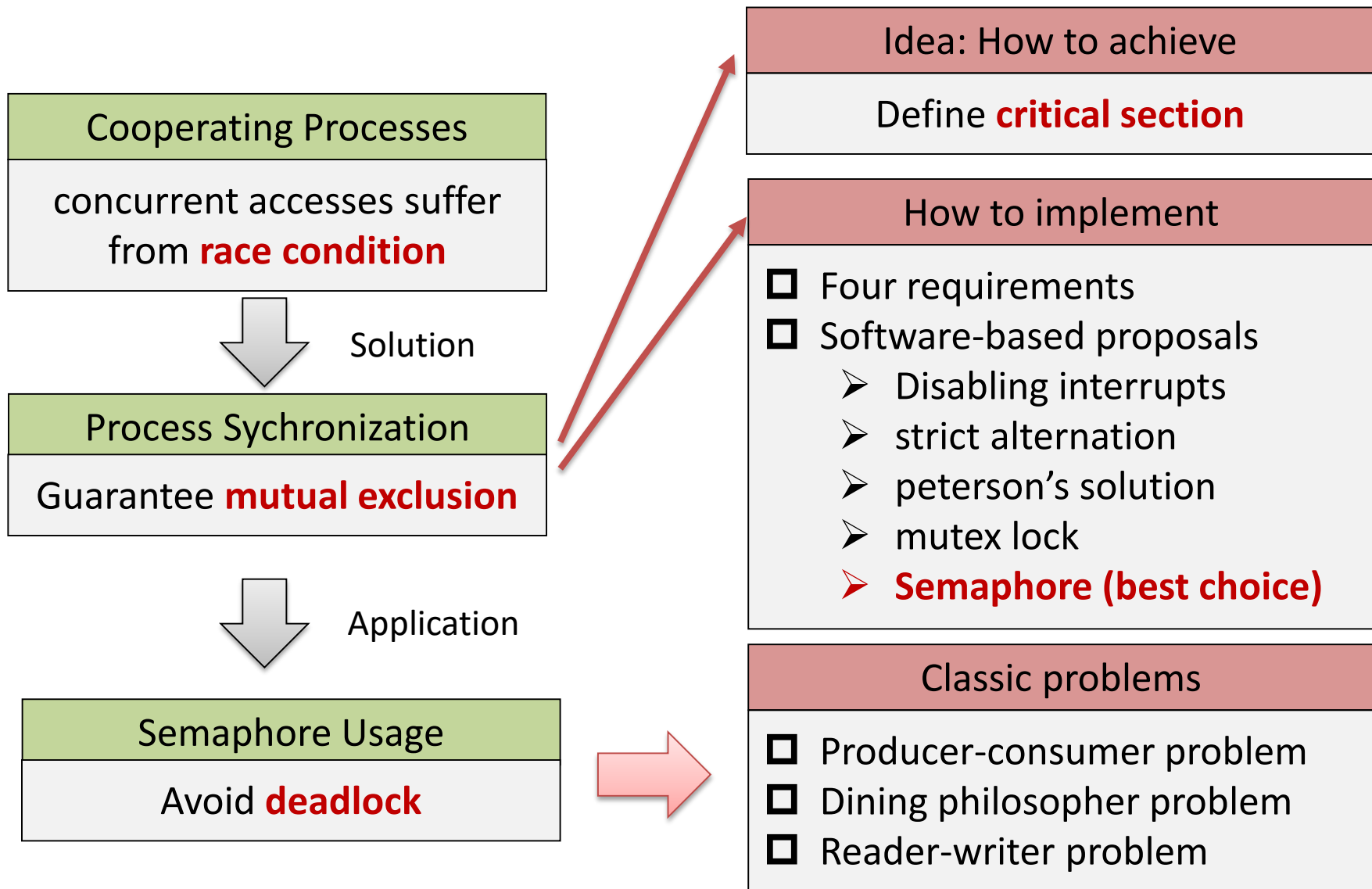
- The above scenario is called the **race condition**.
- A **race condition** means
  - the outcome of an execution depends on a particular order in which the shared resource is accessed.
- Remember: race condition is always a bad thing and debugging race condition has no fun at all!
  - It may end up ...
    - 99% of the executions are fine.
    - 1% of the executions are problematic.

# Race condition – the curse

- For shared memory and files, **concurrent access may yield unpredictable outcomes**
  - **Race condition**
- Common situation
  - Resource sharing occurs frequently in OS
    - EXP: Kernel DS maintaining a list of opened files, maintaining memory allocation, process lists...
  - Multicore brings an increased emphasis on multithreading
    - Multiple threads share global variables and dynamically allocated memory
- **Process synchronization is needed**



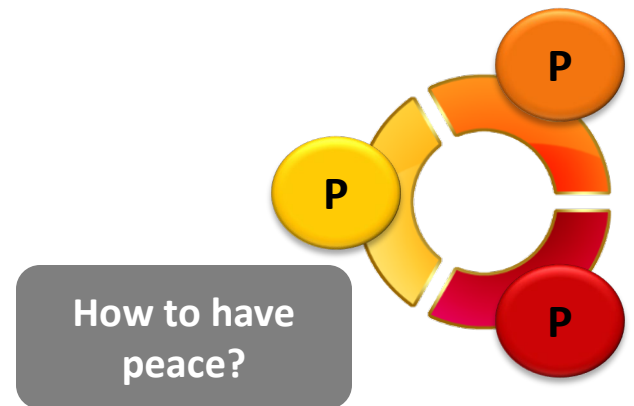
# Topics in Process Synchronization



# Inter-process communication (IPC)

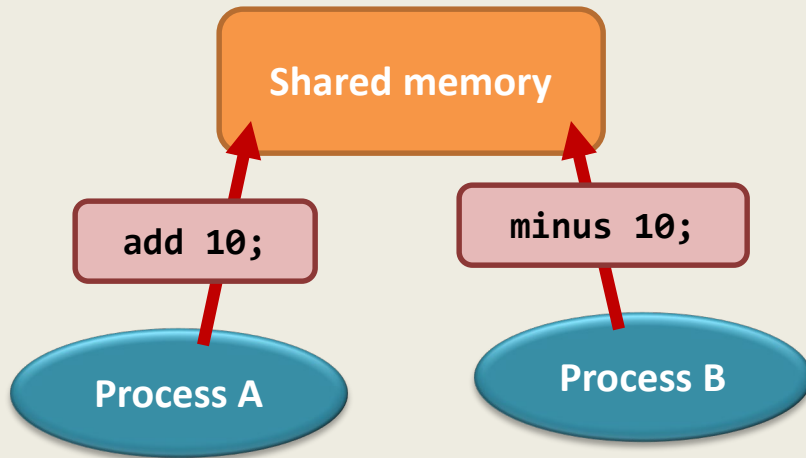
- Mutual exclusion

- what & how to achieve?





# Mutual Exclusion



Two processes playing with the same shared memory is dangerous.

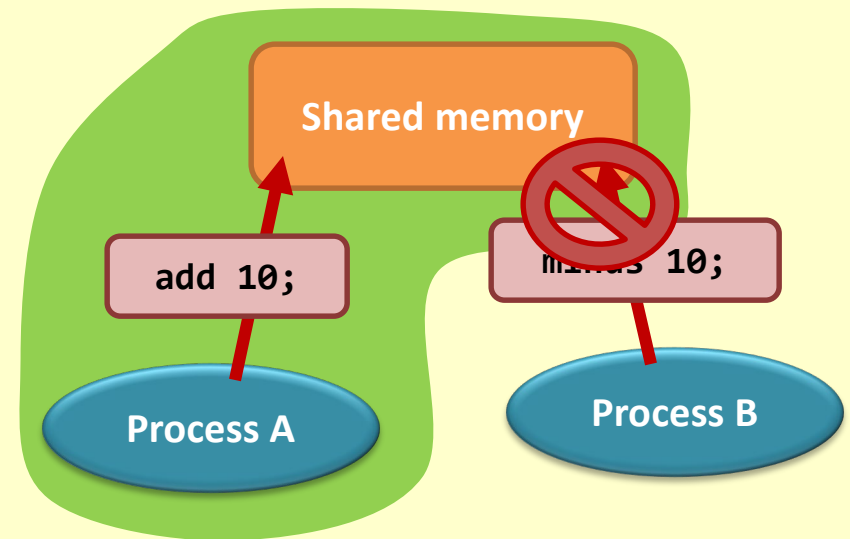
We will face the curse - **race condition**.

The solution can be simple:

When I'm playing with the shared memory, no one could touch it.

This is called **mutual exclusion**.

A set of processes would not have the problem of race condition *if mutual exclusion is guaranteed*.

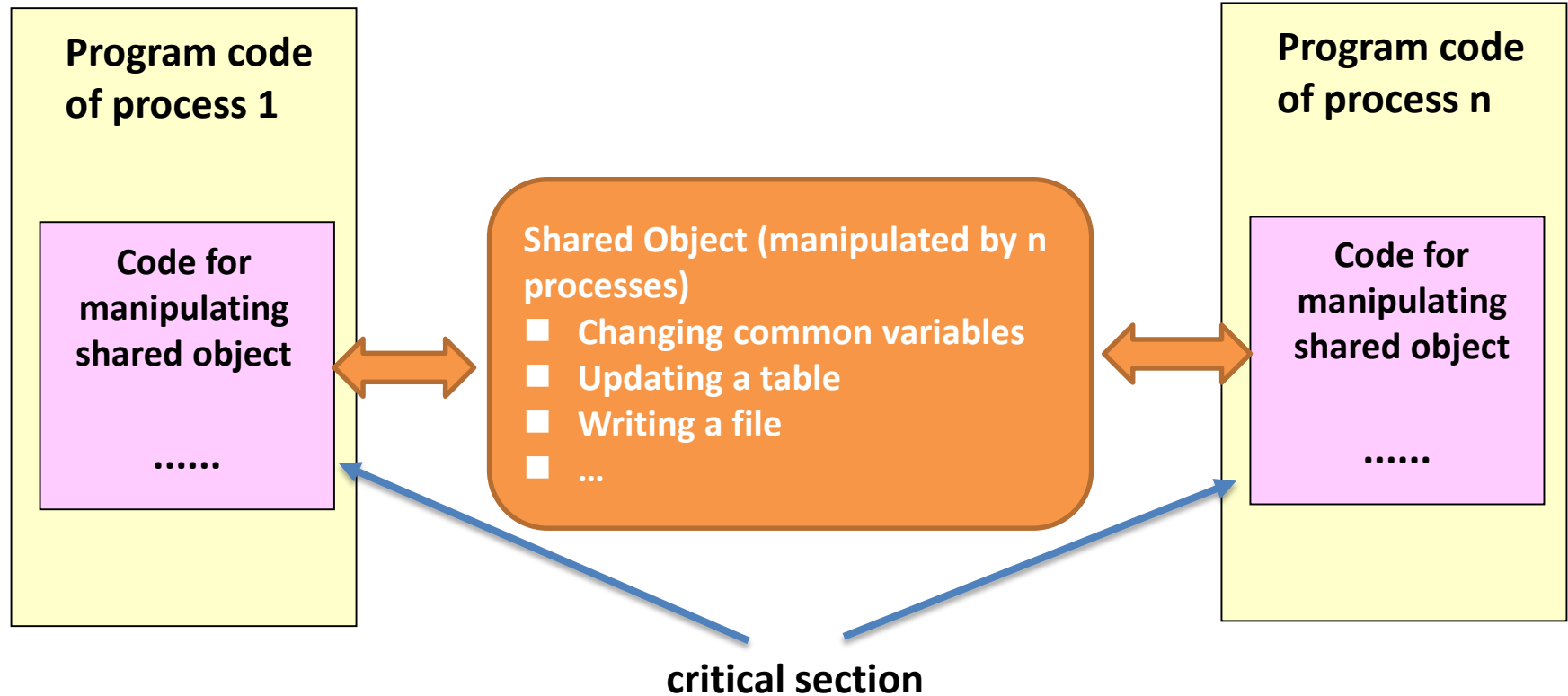


# How to realize mutual exclusion?

- Kernel
  - Preemptive kernels and nonpreemptive kernels
    - Allows (not allow) a process to be preempted while it is running in kernel mode
  - A nonpreemptive kernel is essentially free from race conditions on kernel data structures, and also easy to design (especially for SMP architecture)
  - Why would anyone favor a preemptive kernel
    - More responsive
    - More suitable for real-time programming

# Mutual Exclusion

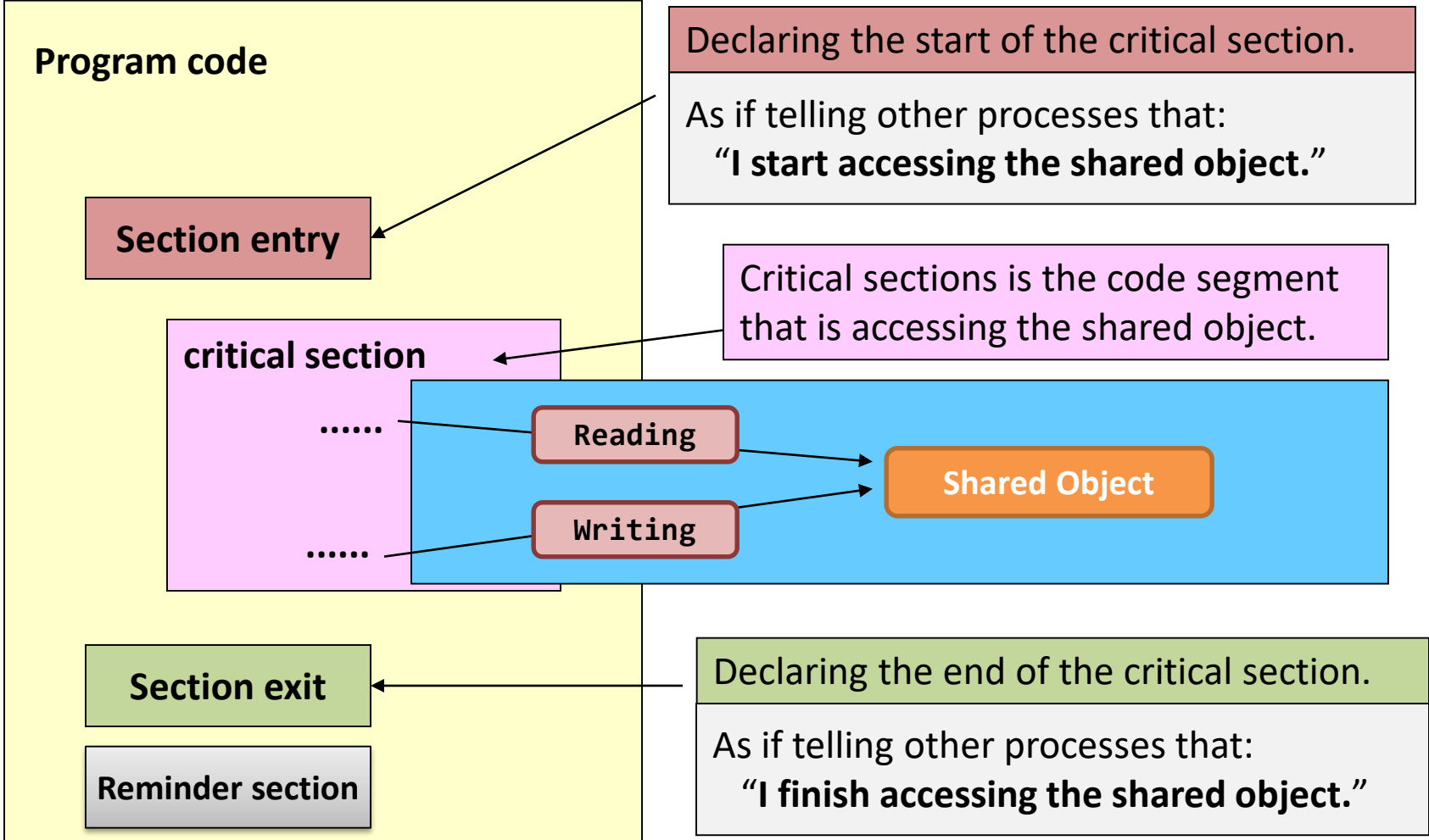
- More generally, how to realize?



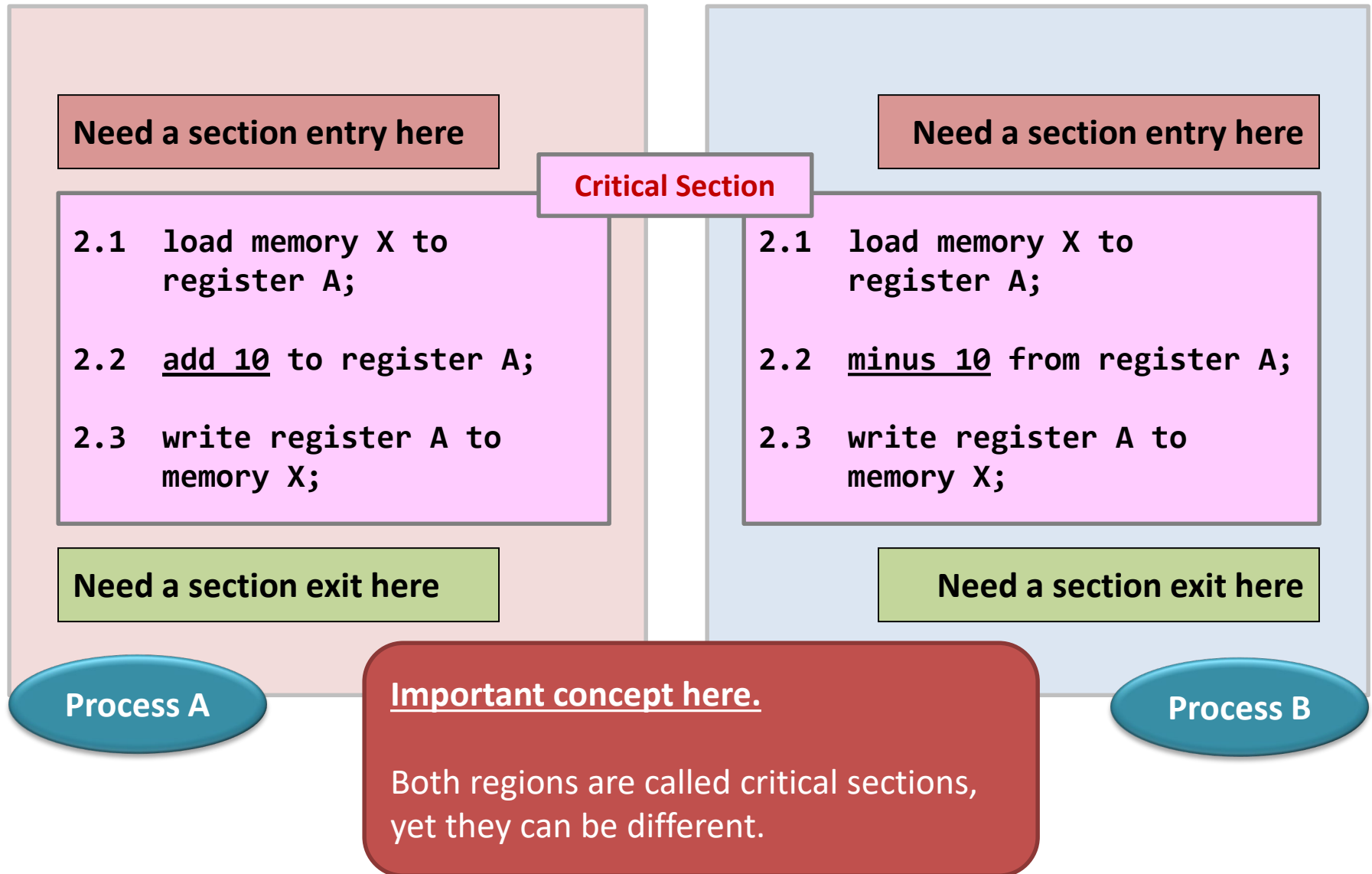
**Solution:** To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.

# Critical Section – General Structure

To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.



# Critical Section – Example



# Summary...for the content so far...

- **Race condition** is a problem.
  - It makes a concurrent program producing **unpredictable** results if you are using shared objects as the communication medium.
  - The outcome of the computation **totally depends on the execution sequences** of the processes involved.
- **Mutual exclusion** is a requirement.
  - If it could be achieved, then the problem of the race condition would be gone.
  - Mutual exclusion hinders the performance of parallel computations.

# Summary...for the content so far...

- **Defining critical sections** is a solution.
  - They are code segments that access shared objects.
  - Critical section must be **as tight as possible**.
    - Well, you can declare the entire code of a program to be a big critical section.
    - But, the program will be a very high chance to block other processes or to be blocked by other processes.
  - Note that one critical section can be designed for **accessing more than one shared objects**.

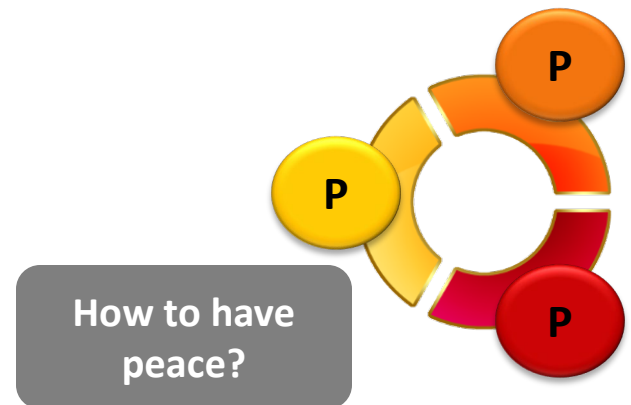
# Summary...for the content so far...

- **Implementing section entry and exit** is a challenge.
  - The entry and the exit are **the core parts that guarantee mutual exclusion**, but not the critical section.
  - Unless they are correctly implemented, race condition would appear.



# Inter-process communication (IPC)

- **Mutual exclusion:**
  - how to achieve?
  - how to implement?  
(section entry and exit)



# Entry and exit implementation - requirements

- **Requirement #1: Mutual Exclusion**. No two processes could be simultaneously inside their critical sections.

**Implication**: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

- **Requirement #2**. Each process is executing at a nonzero speed, but no assumptions should be made about the relative speed of the processes and the number of CPUs.

**Implication**: the solution **cannot depend on the time spent inside the critical section**, and the solution cannot assume the number of CPUs in the system.

# Entry and exit implementation - requirements

- **Requirement #3: progress.** No process running outside its critical section should block other processes.

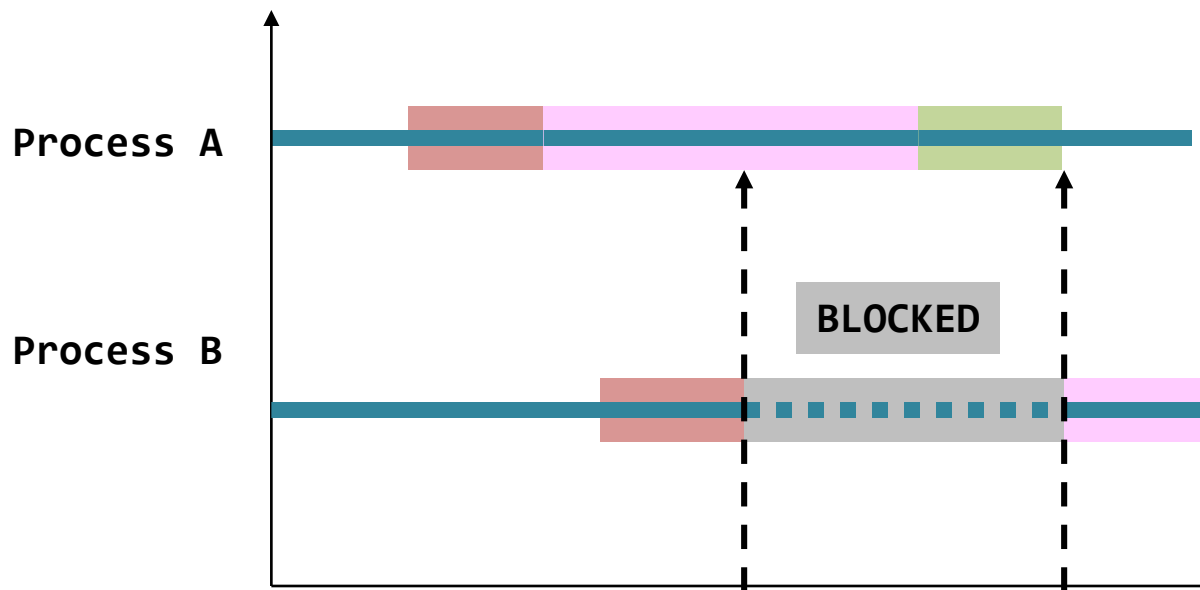
**Implication:** Only processes that are **not executing in their remainder sections** can participate in deciding which will enter its critical section.

- **Requirement #4: Bounded waiting.** No process would have to wait forever in order to enter its critical section.





**Implication:** There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section (no processes should be **starved to death**).

# A typical mutual exclusion scenario

Remember, it is always the entry blocks other processes, but not the critical section.



## Keys

-  Critical section entry
-  Inside Critical section
-  Critical section exit
-  Shared object (if any)

We will be using this coloring scheme throughout this part.

B tries to enter its critical section but A is in its critical section.

A leaves its critical section and B resumes execution accordingly.

# Mutual Exclusion Implementation

- Challenges of Implementing **section entry** & **exit**
  - Both operations must be atomic
  - Also need to satisfy the above requirements
  - Performance consideration
- Hardware solution
  - Rely on atomic instructions
  - `test_and_set()`
  - `compare_and_swap`

# Example: test\_and\_set()

- Definition

```
boolean test_and_set(boolean *target) {  
    boolean rv = *target;  
    *target = true;  
  
    return rv;  
}
```

- Mutual exclusion implementation

```
do {  
    while (test_and_set(&lock))  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = false;  
  
    /* remainder section */  
} while (true);
```

# Example: compare\_and\_swap()

- Definition

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;

    return temp;
}
```

- Mutual exclusion implementation

How to satisfy bounded waiting?

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
} while (true);
```

# Enhanced version

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

    /* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    /* remainder section */
} while (true);
```

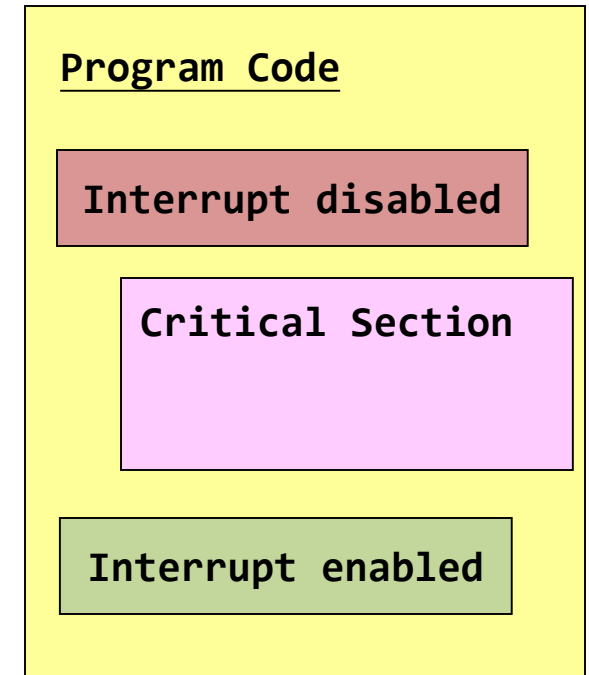
lock is initialized as false





# Proposal #1 – disabling interrupt.

- **Method**
  - Similar idea as nonpreemptive kernels
  - To **disable context switching** when the process is inside the critical section.
- **Effect**
  - When a process is in its critical section, no other processes could be able to run.
- **Implementation**
  - A new system call should be provided.
- **Correctness?**
  - **Correct**, but it is not an attractive solution.
  - Not as feasible in a multiprocessor environment
  - Performance issue (may sacrifice concurrency)



# Proposal #2: Mutex Locks

- **Idea**

- A process must acquire the lock before entering a critical section, and release the lock when it exits the critical section
- Using a new shared object to detect the status of other processes, and “**lock**” the shared object

Shared object: “available” (lock)

```
1 acquire(){
2     while(!available)
3         ; /* busy waiting */
4     available = false;
5 }
```

```
1 release(){
2     available = true;
3 }
```

# Proposal #2: Mutex Locks

- **Implementation**
  - Calls to acquire and release locks must be performed **atomically**
  - Often use hardware instructions
- **Issue**
  - Busy waiting: Waste CPU resource
    - **Spinlock**
- **Applications**
  - Multiprocessor system
    - When locks are expected to be held for short times

Note that: all processes run the following same code.

## Program Code

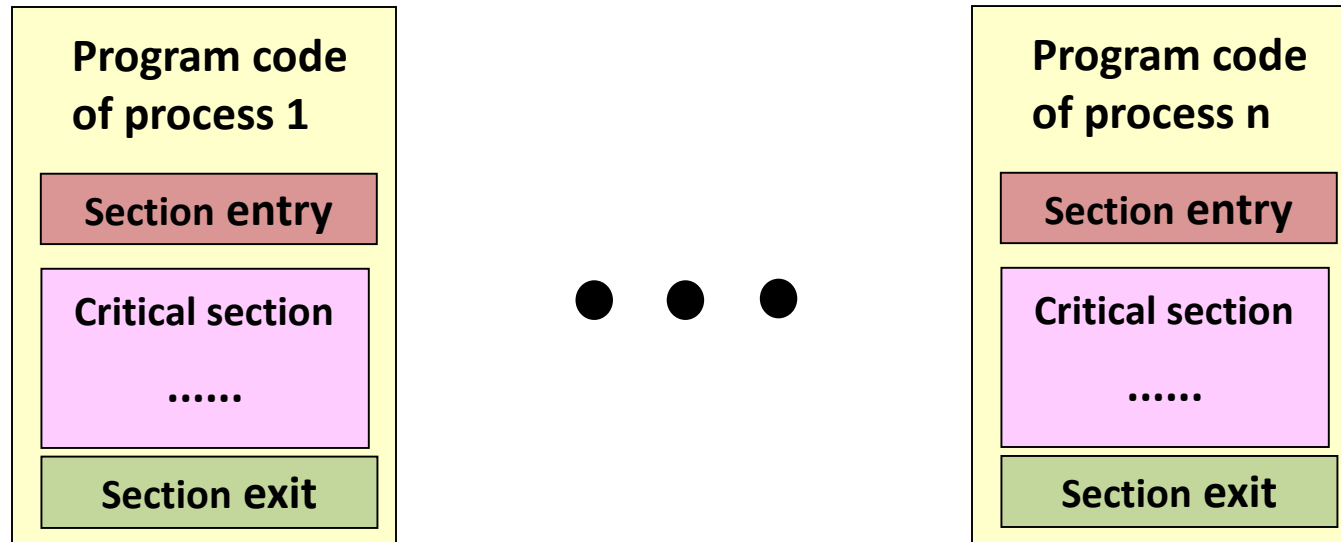
```
acquire();
```

```
Critical Section
```

```
release();
```

# Other software-based solutions

- Aim
  - To decide which process could go into its critical section

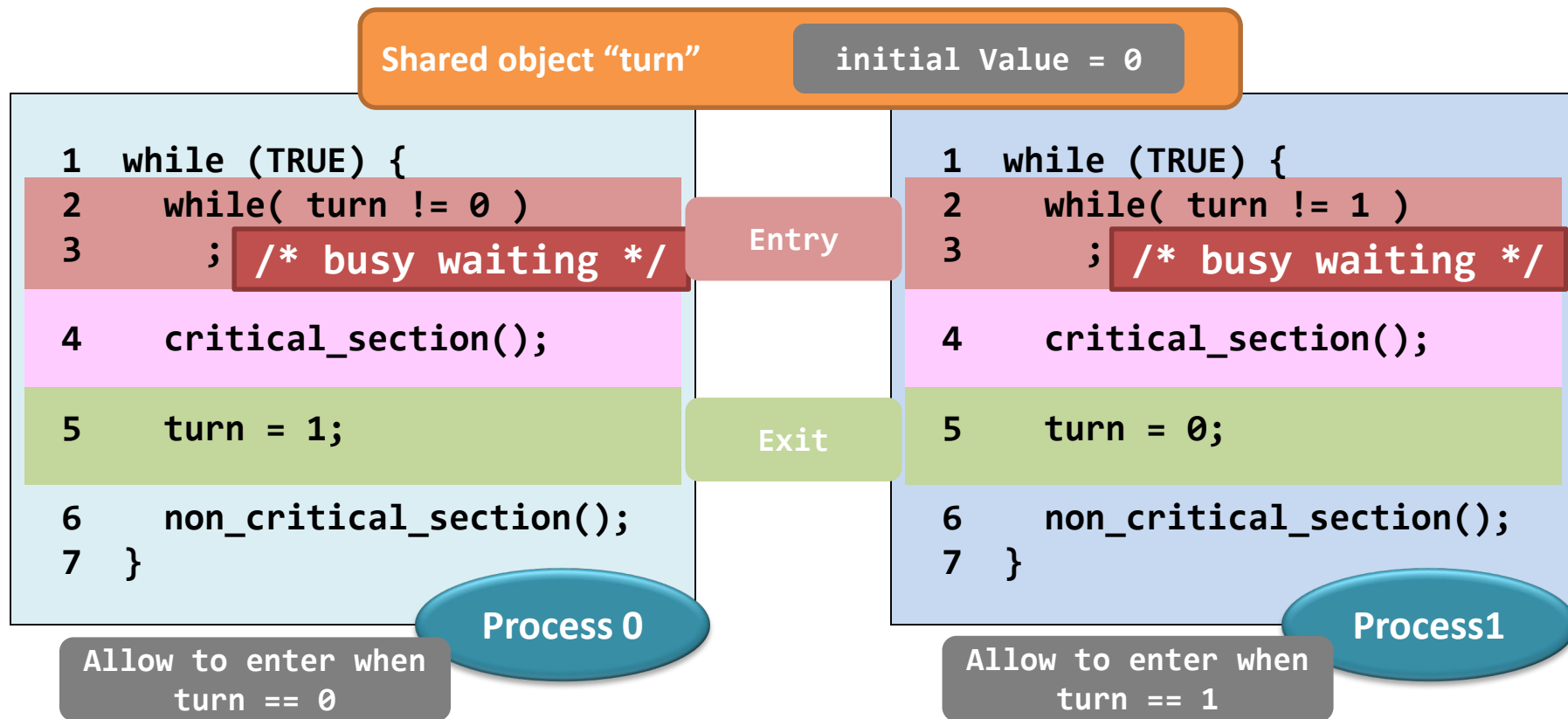


- Key Issue
  - Detect the status of processes (section entry)
    - Need other shared variables

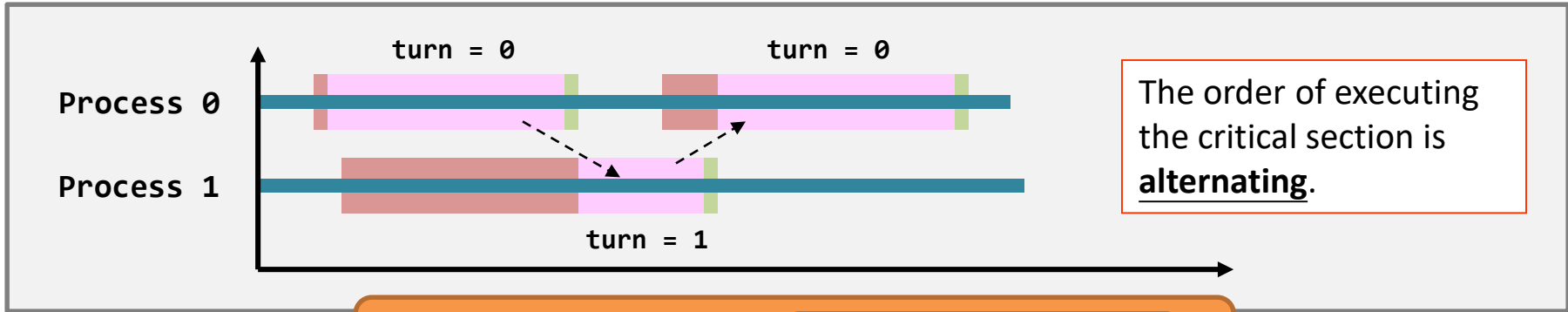
# Proposal #3: Strict alternation

- **Method**

- Using a new shared object to detect the status of other processes



# Proposal #3: Strict alternation



Shared object "turn"      initial Value = 0

```
1 while (TRUE) {
2   while( turn != 0 )
3     ; /* busy waiting */
4   critical_section();
5   turn = 1;
6   non_critical_section();
7 }
```

Process 0

```
1 while (TRUE) {
2   while( turn != 1 )
3     ; /* busy waiting */
4   critical_section();
5   turn = 0;
6   non_critical_section();
7 }
```

Process1

# Proposal #3: Strict alternation - Cons

- Strict alternation seems good, yet, it is **inefficient**.
  - Busy waiting wastes CPU resources.
- In addition, the alternating order is **too strict**.
  - What if Process 0 wants to enter the critical section **twice in a row**? **NO WAY!**
  - Violate any requirement?

**Requirement #3.** No process running outside its critical section should block other processes.

# Proposal #4: Peterson's solution

- How to improve the strict alternation proposal?
- The Peterson's solution:
  - Processes would act as a gentleman: if you want to enter, I'll let you first
  - No alternation is there
  - Share two data items
    - **int turn;** //whose turn to enter its critical section
    - **Boolean interested[2];** //if a process wants to enter



# Proposal #4: Peterson's solution

Shared object: "turn" &  
"interested[2]"

```
1  int turn;                                /* who can enter critical section */
2  int interested[2] = {FALSE,FALSE};      /* wants to enter critical section*/
3
4  void enter_region( int process ) {      /* process is 0 or 1 */
5      int other;                          /* number of the other process */
6      other = 1-process;                  /* other is 1 or 0 */
7      interested[process] = TRUE;         /* want to enter critical section */
8      turn = other;
9      while ( turn == other &&
              interested[other] == TRUE )
10         ; /* busy waiting */
11 }
12
13 void leave_region( int process ) {      /* process: who is leaving */
14     interested[process] = FALSE;        /* I just left critical region */
15 }
```

Entry

Exit

# Proposal #4: Peterson's solution

```
1 int turn;
2 int interested[2] = {FALSE,FALSE};
3
4 void enter_region( int process ) {
5     int other;
6     other = 1-process;
7     interested[process] = TRUE;
8     turn = other;
9     while ( turn == other &&
10           interested[other] == TRUE
11           ; /* busy waiting */
12 )
13 void leave_region( int process ) {
14     interested[process] = FALSE;
15 }
```

Line 8 takes the other one the turn to run.

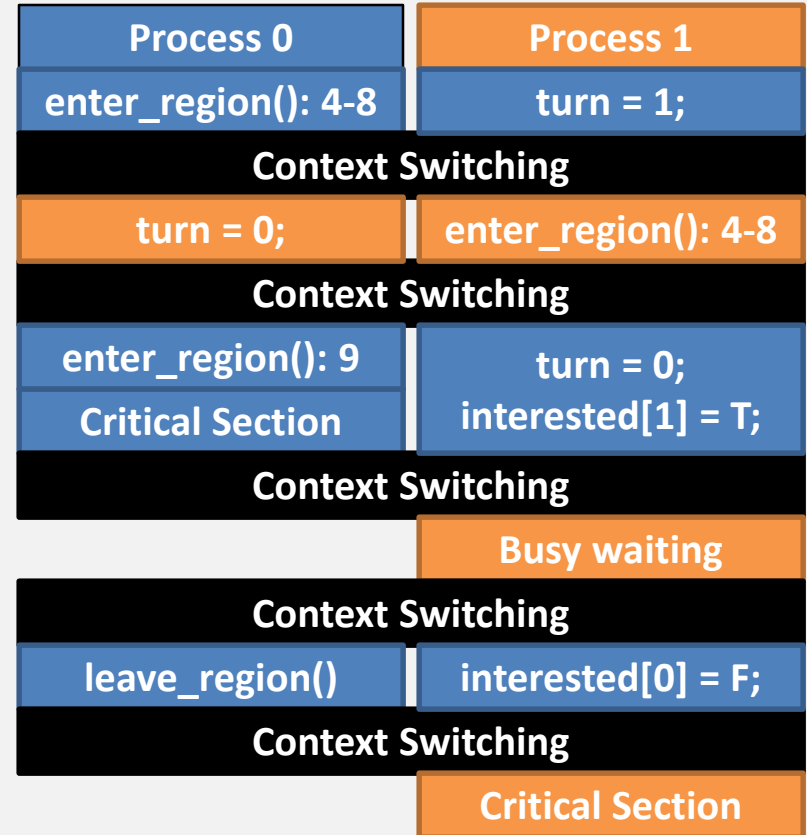
Of course, the process is willing to wait when she wants to enter the critical section.

*"I'm a gentleman!"*

The process always let another process to enter the critical region first although she wants to enter too.

# Proposal #4: Peterson's solution

```
1 int turn;
2 int interested[2] = {FALSE, FALSE};
3
4 void enter_region( int process ) {
5     int other;
6     other = 1-process;
7     interested[process] = TRUE;
8     turn = other;
9     while ( turn == other &&
10           interested[other] == TRUE )
11         ; /* busy waiting */
12 }
13 void leave_region( int process ) {
14     interested[process] = FALSE;
15 }
```

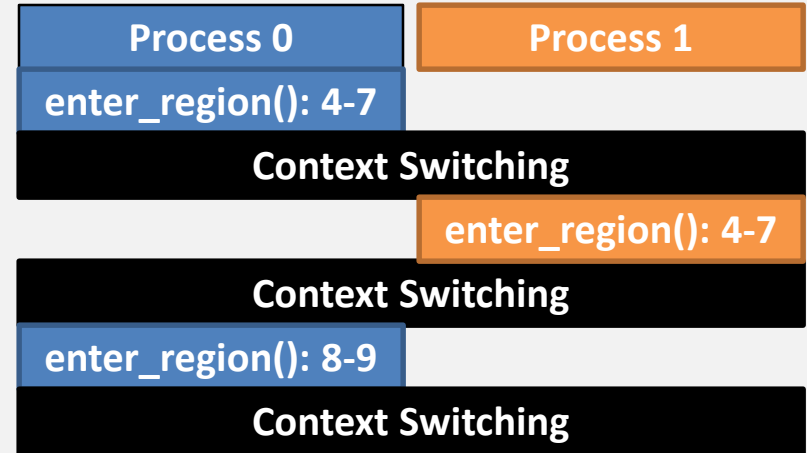


and the story goes on...

Can you show that the requirements are satisfied?

# Proposal #4: Peterson's solution

```
1 int turn;
2 int interested[2] = {FALSE,FALSE};
3
4 void enter_region( int process ) {
5     int other;
6     other = 1-process;
7     interested[process] = TRUE;
8     turn = other;
9     while ( turn == other &&
10           interested[other] == TRUE )
11         ; /* busy waiting */
12
13 void leave_region( int process ) {
14     interested[process] = FALSE;
15 }
```

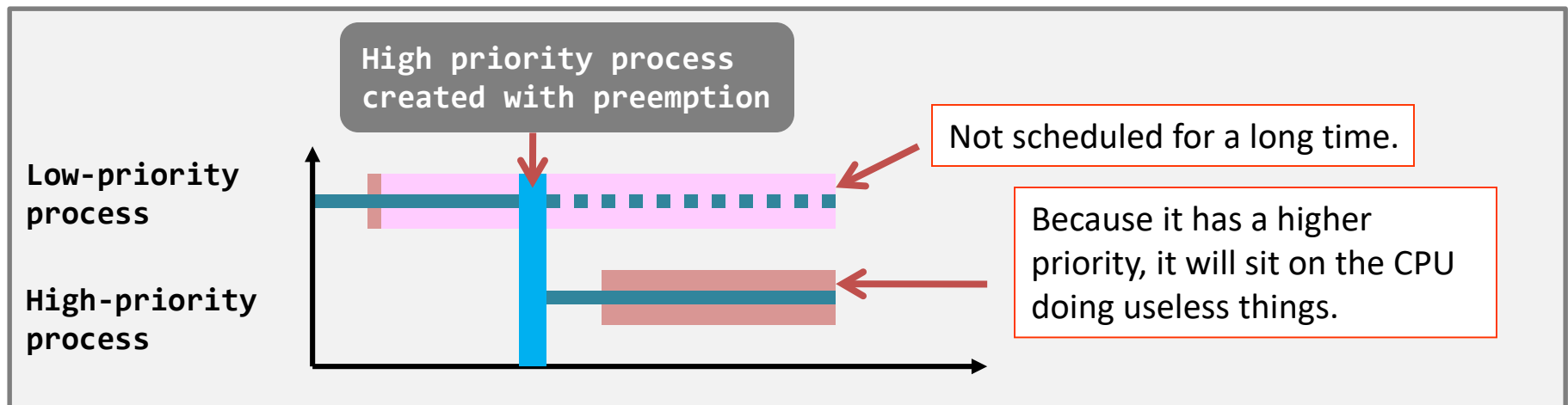


Can you complete the flow?  
(what is the difference?)

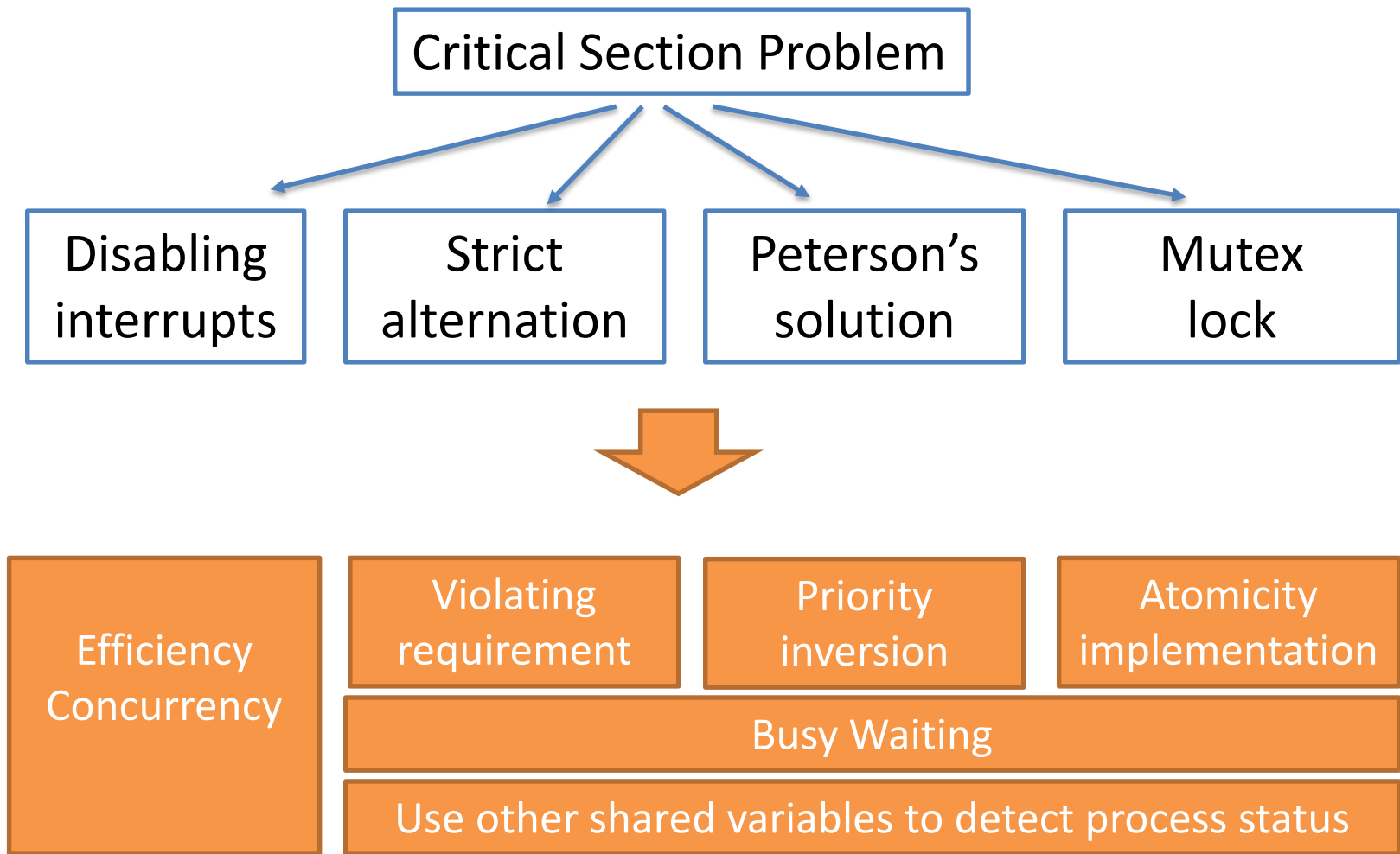
Can both processes progress?

# Proposal #4: Peterson's solution – issues

- Busy waiting has its own problem...
  - An apparent problem: wasting CPU time.
  - A hidden, serious problem: **priority inversion problem**.
    - A low priority process is inside the critical region, but ...
    - A high priority process wants to enter the critical region.
    - Then, the high priority process will perform busy waiting for a long time or even forever.

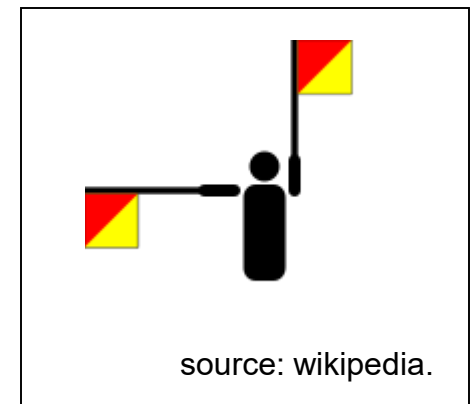


# Story so far...



# Final proposal: Semaphore

- In real life, semaphore is a flag signaling system.
  - It tells a train driver (or a plane pilot) when to stop and when to proceed.
- When it comes to programming...
  - A semaphore is a data type.
  - You can imagine that it is an integer (but it is certainly not an integer when it comes to real implementation).



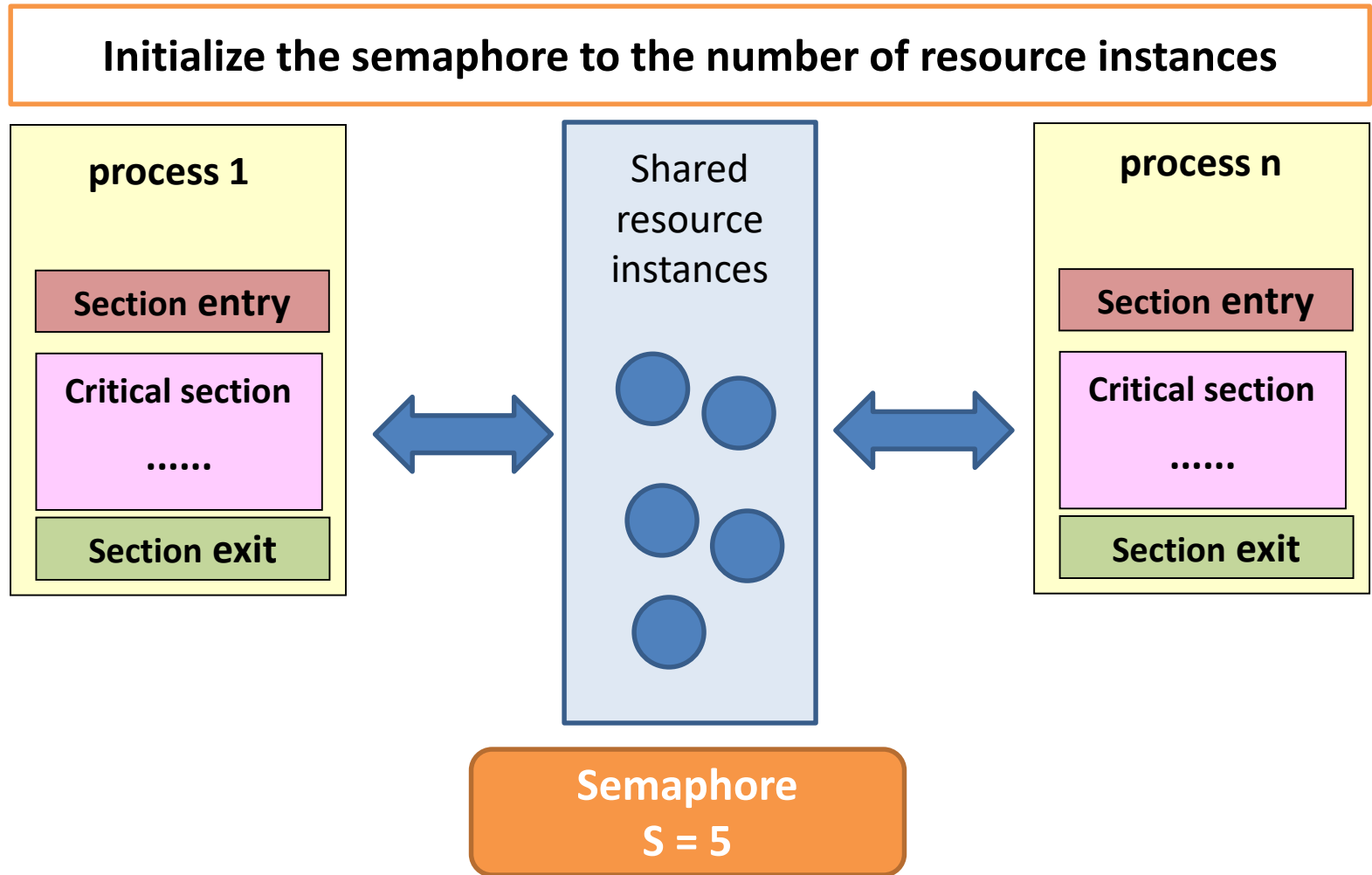
# Final proposal: Semaphore

- Semaphore is a data type (**additional shared object**)
  - Denote the status or the number of resources
  - Two types
    - **Binary semaphore**: 0 or 1 (similar to mutex lock)
    - **Counting semaphore**: control finite number of resources
- Accessed through two standard **atomic** operations
  - **down()**: originally termed P (from Dutch *proberen*, “to test”), **wait()** in textbook
    - Decrementing the count
  - **up()**: originally termed V (from *verhogen*, “to increment”), **signal()** in textbook
    - Incrementing the count



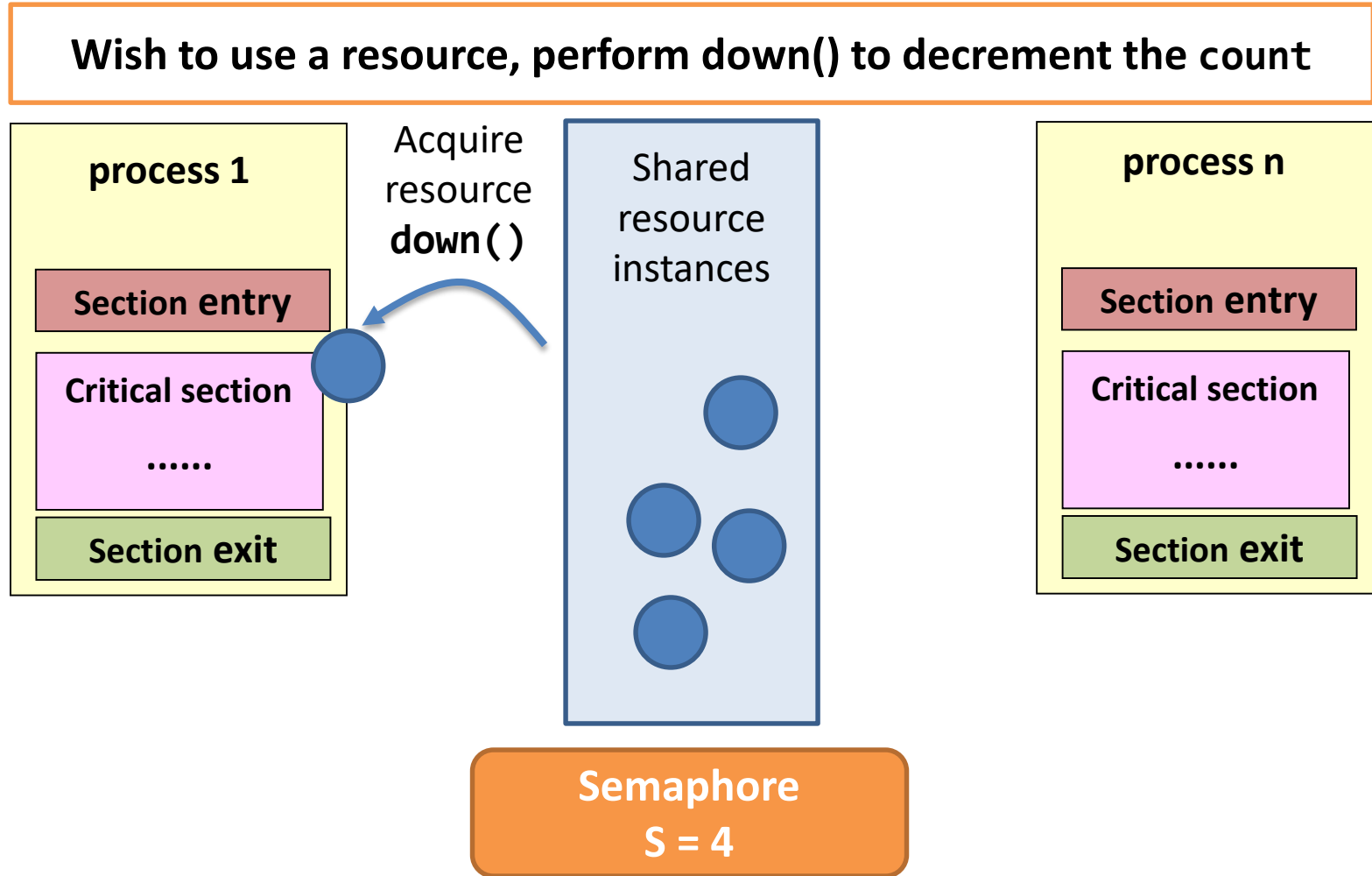
# Final proposal: Semaphore

- Idea



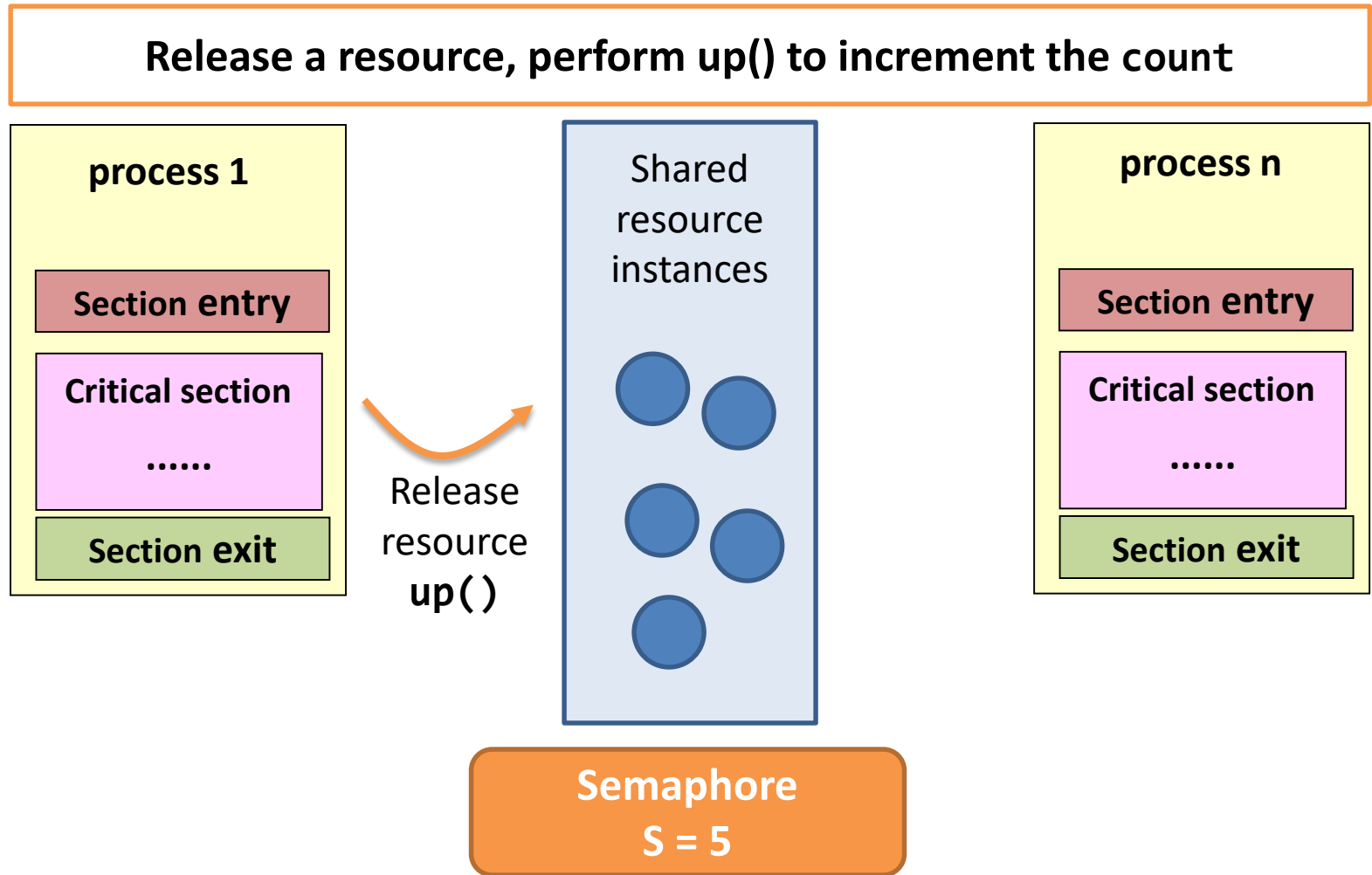
# Final proposal: Semaphore

- Idea



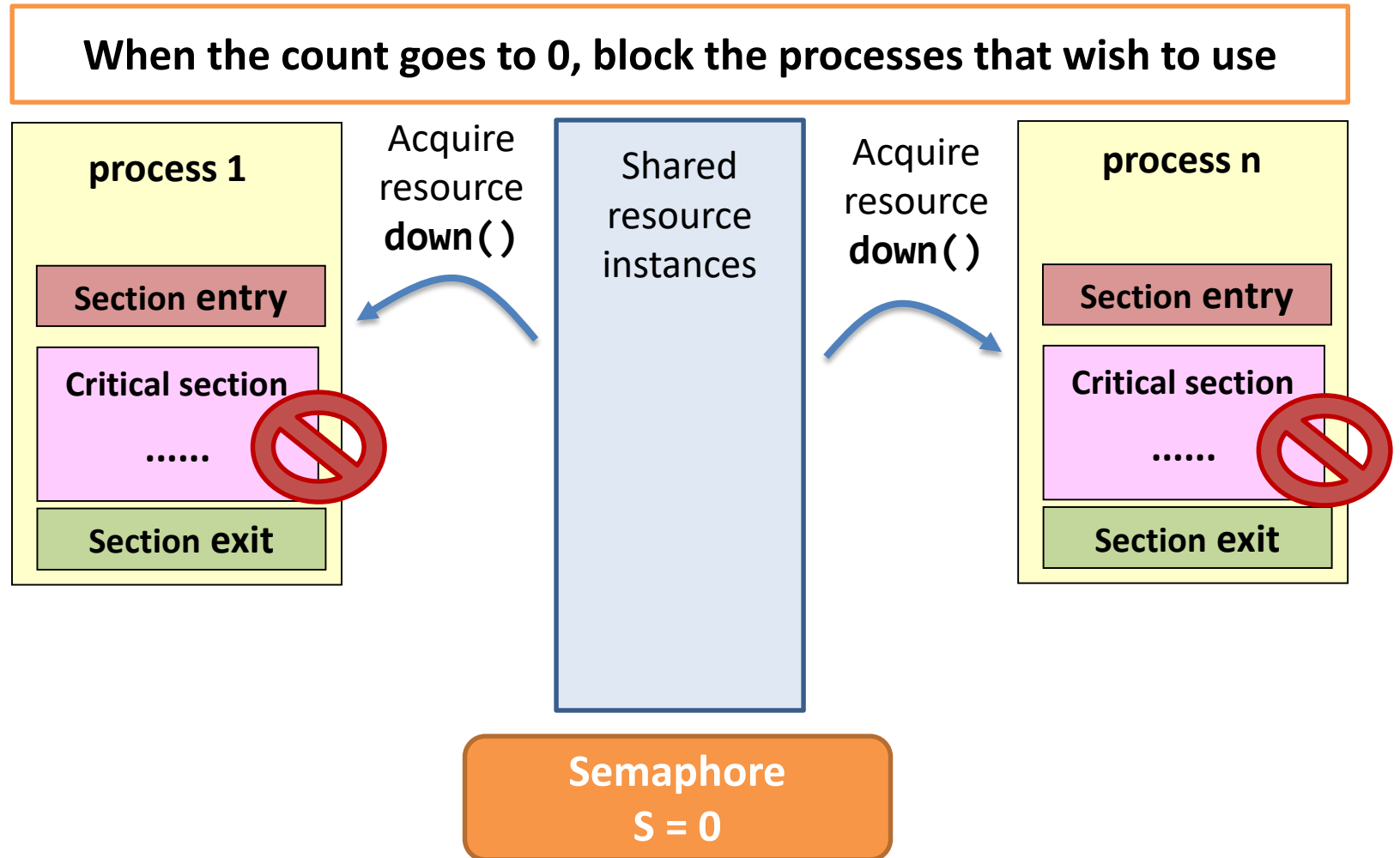
# Final proposal: Semaphore

- Idea



# Final proposal: Semaphore

- Idea



# Semaphore – Simple Implementation

## Data Type definition

```
typedef int semaphore;
```

**Counting Semaphore:** initialized to be the number of resources available

## Section Entry: down()

```
1 void down(semaphore *s) {
2
3     while ( *s == 0 ) {
4
5         ;//busy waiting
6
7     }
8     *s = *s - 1;
9
10 }
```

## Section Exit: up()

```
1 void up(semaphore *s) {
2
3
4
5     *s = *s + 1;
6
7 }
```

# Semaphore – Address busy waiting

## Data Type definition

```
typedef int semaphore;
```

## First issue: Busy waiting

**Solution:** block the process instead of busy waiting (place the process into a waiting queue)

## Section Entry: down()

```
1 void down(semaphore *s) {
2
3     while ( *s == 0 ) {
4
5         special_sleep();
6
7     }
8     *s = *s - 1;
9
10 }
```

## Section Exit: up()

```
1 void up(semaphore *s) {
2
3     if ( *s == 0 )
4         special_wakeup();
5     *s = *s + 1;
6
7 }
```

# Semaphore – Address busy waiting

## Data Type definition

```
typedef int semaphore;
```

## First issue: Busy waiting

**Solution:** block the process instead of busy waiting (place the process into a waiting queue)

```
typedef struct{  
  
    int value;  
    struct process * list;  
  
}semaphore;
```

## Note

**Implementation:** The waiting queue may be associated with the semaphore, so a semaphore is not just an integer

# Semaphore – Atomicity

## Data Type definition

```
typedef int semaphore;
```

## Section Entry: down()

```
1 void down(semaphore *s) {  
2  
3     while ( *s == 0 ) {  
4  
5         special_sleep();  
6  
7     }  
8     *s = *s - 1;  
9  
10 }
```

**Second issue: Atomicity** (both operations must be atomic)

**Solution:** Disabling interrupts

## Section Exit: up()

```
1 void up(semaphore *s) {  
2  
3     if ( *s == 0 )  
4         special_wakeup();  
5     *s = *s + 1;  
6  
7 }
```



# Semaphore – Atomicity

## Data Type definition

```
typedef int semaphore;
```

## Section Entry: down()

```
1 void down(semaphore *s) {
2     disable_interrupt();
3     while ( *s == 0 ) {
4         enable_interrupt();
5         special_sleep();
6         disable_interrupt();
7     }
8     *s = *s - 1;
9     enable_interrupt();
10 }
```

**Second issue: Atomicity** (both operations must be atomic)

**Solution:** Disabling interrupts

Also, only one process can invoke “**disable\_interrupt()**”. Later processes would be blocked until “**enable\_interrupt()**” is called.

## Section Exit: up()

```
1 void up(semaphore *s) {
2     disable_interrupt();
3     if ( *s == 0 )
4         special_wakeup();
5     *s = *s + 1;
6     enable_interrupt();
7 }
```

# Semaphore – The code

## Data Type definition

```
typedef int semaphore;
```

## Section Entry: down()

```
1 void down(semaphore *s) {
2     disable_interrupt();
3     while ( *s == 0 ) {
4         enable_interrupt();
5         special_sleep();
6         disable_interrupt();
7     }
8     *s = *s - 1;
9     enable_interrupt();
10 }
```

Why need these two statements?

Disabling interrupts may sacrifice concurrency, so it is essential to keep the critical section as short as possible

## Section Exit: up()

```
1 void up(semaphore *s) {
2     disable_interrupt();
3     if ( *s == 0 )
4         special_wakeup();
5     *s = *s + 1;
6     enable_interrupt();
7 }
```

# Semaphore – details

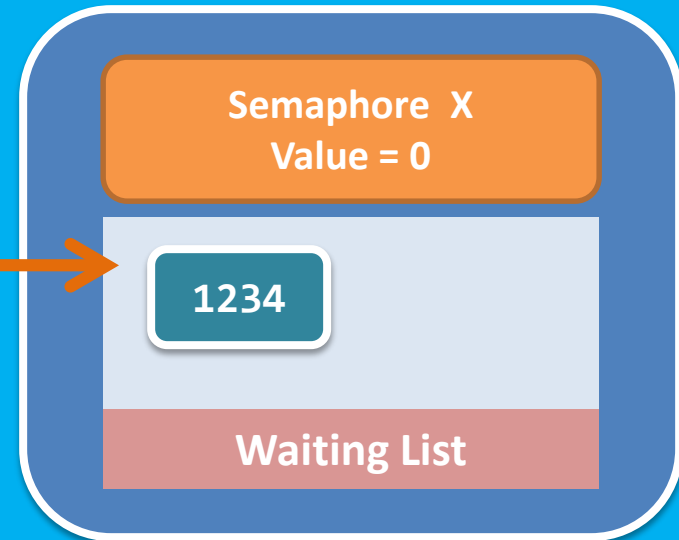
Process 1234

down(X)

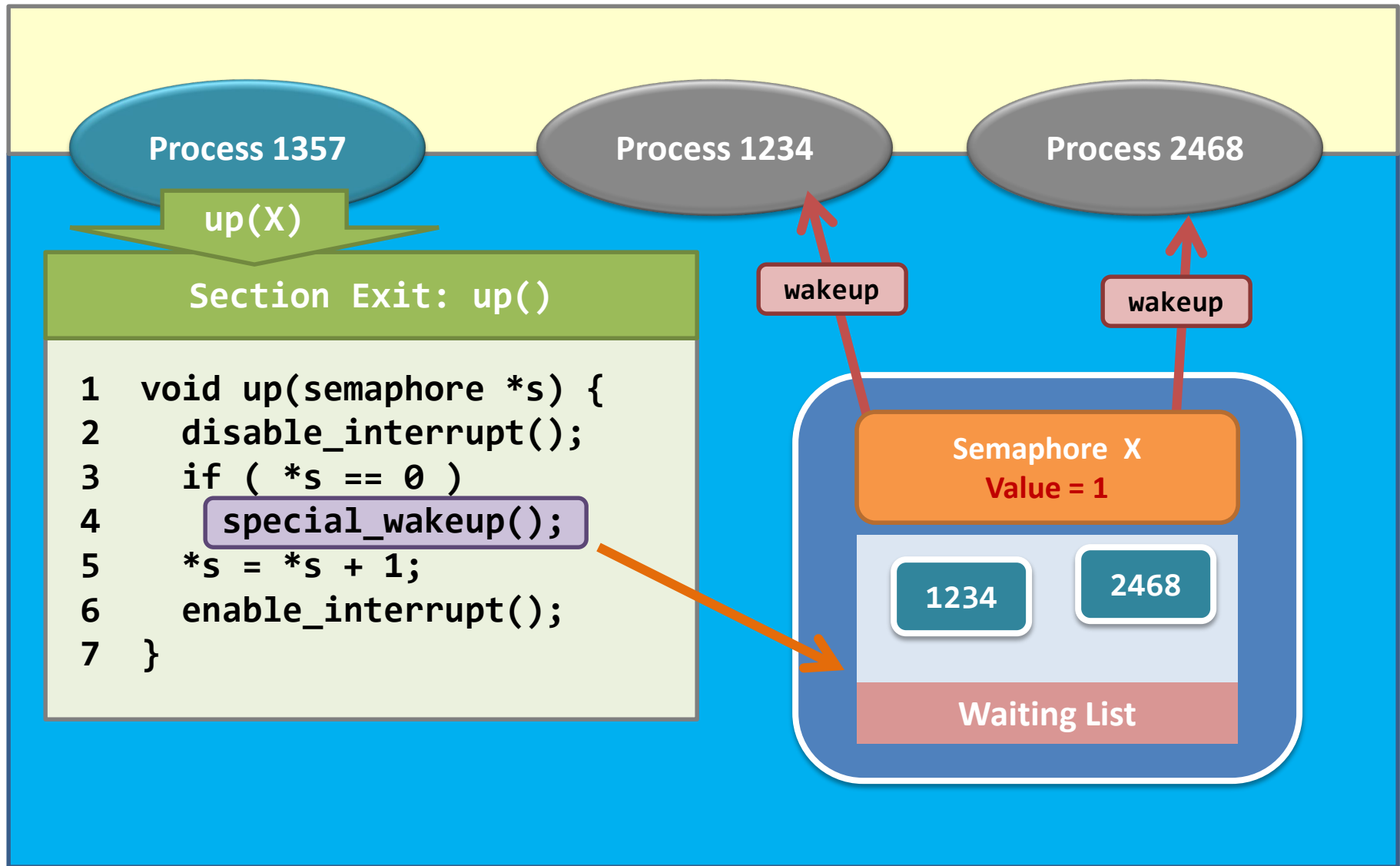
Section Entry: down()

Suppose that process 1234 is willing to access the shared resource (enter its critical section), but no resource is available

```
1 void down(semaphore *s) {  
2     disable_interrupt();  
3     while ( *s == 0 ) {  
4         enable_interrupt();  
5         special_sleep();  
6         disable_interrupt();  
7     }  
8     *s = *s - 1;  
9     enable_interrupt();  
10 }
```



# Semaphore – details



# Semaphore – details

Process 1234

Process 2468

down(X)

down(X)

Note that it is impossible for **two blocked processes to get out of the down() simultaneously.**

Why?

Only one process can invoke **disable\_interrupt()**

Only one process can manipulate this shared variable

Section Entry: down()

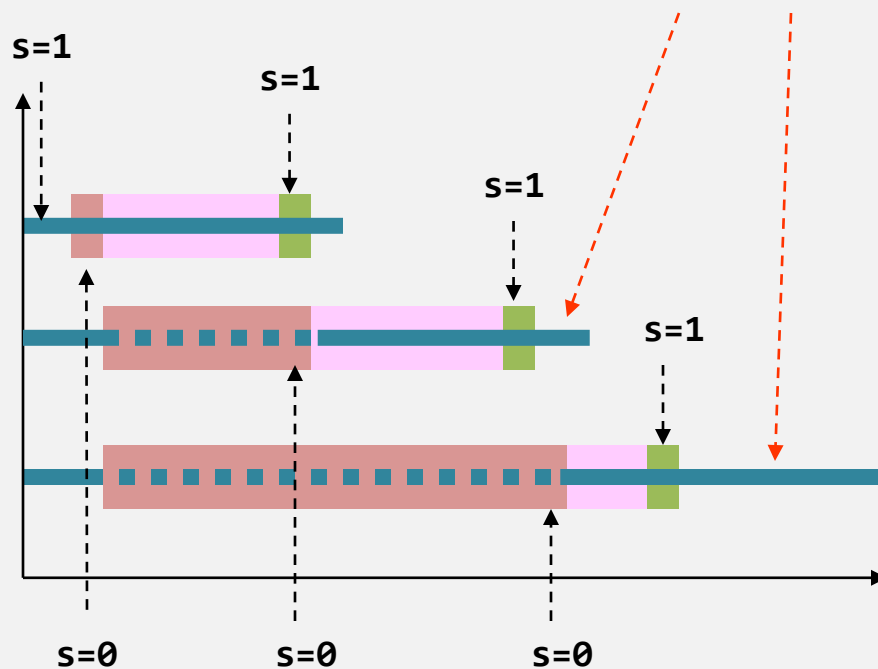
```
1 void down(semaphore *s) {
2     disable_interrupt();
3     while ( *s == 0 ) {
4         enable_interrupt();
5         special_sleep();
6         disable_interrupt();
7     }
8     *s = *s - 1;
9     enable_interrupt();
10 }
```

here

# Semaphore – in action

- Add them together...

Either one of the processes can enter the critical section when the first process calls “up(s)”.



```
semaphore *s;  
*s = 1;      /* initial value */
```

```
1 while(TRUE) {  
2   down(s);  
3   critical_section();  
4   up(s);  
5 }
```

entry

exit

# Summary...on semaphore

- More on semaphore...it demonstrates an important kind of operations – **atomic operations**.

## Definition of atomic operation

- Either none of the instructions of an atomic operation were completed, or
- All instructions of an atomic operation are completed.

- In other words, the entire **up()** and **down()** are indivisible.
  - If it returns, the change must have been made;
  - If it is aborted, no change would be made.

# Summary...on critical section problem

- What happened is just the implementation of mutual exclusion (section entry and section exit).

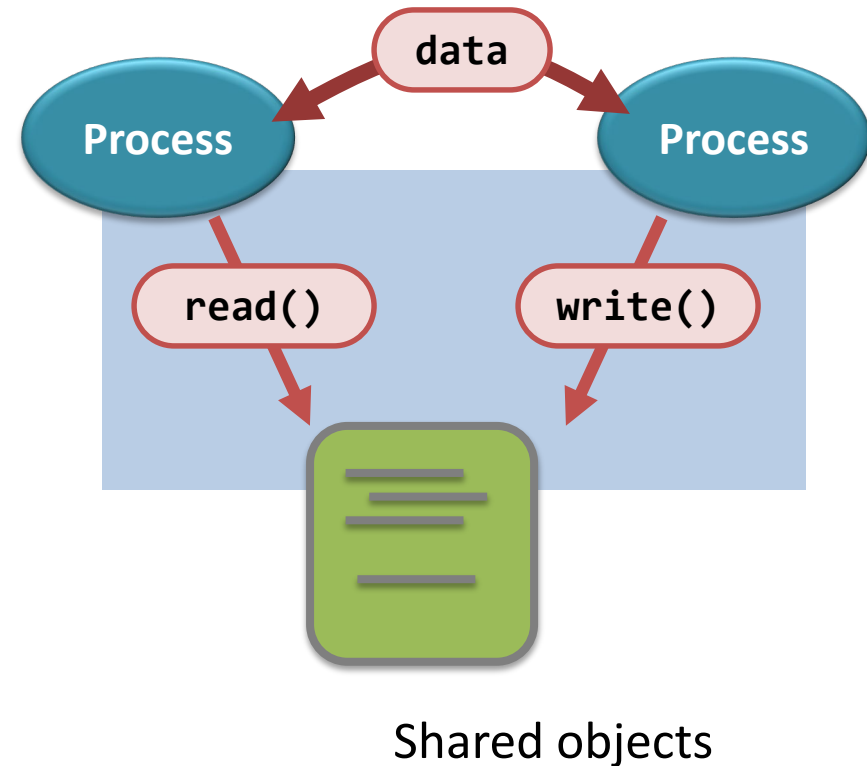
	Comments
Disabling interrupts	Time consuming for multiprocessor systems, sacrifices concurrency.
Strict alternation	Not a good one, busy waiting & violating one requirement.
Peterson's solution	Busy waiting & has a potential " <i>priority inversion problem</i> ".
Mutex lock	Busy waiting, often relies on hardware instructions.
Semaphore	<b>BEST CHOICE.</b>

- What is next?
  - How to use semaphore to solve classic IPC problems
  - Deadlock



# Story so far...

- For shared memory and files, concurrent access may yield unpredictable outcomes
  - **Race condition**
- To avoid race condition, **mutual exclusion** must be guaranteed
  - Critical section
  - Implementations (entry/exit)
    - Hardware instructions
    - Disabling interrupts
    - Strict alternation
    - Peterson's solution
    - Mutex lock
    - **Semaphore**



# Semaphore Usage

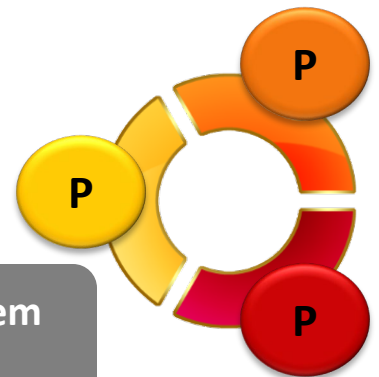
- Semaphore can be used for
  - Mutual exclusion (binary semaphore)
  - Process synchronization (counting semaphore may be needed)
- How to do **process synchronization** w/ semaphore?
  - Mutual exclusion + coordination (multiple semaphores)
  - Careless design may lead to other issues
    - Deadlock

# The Deadlock Problem

## Classic IPC problems

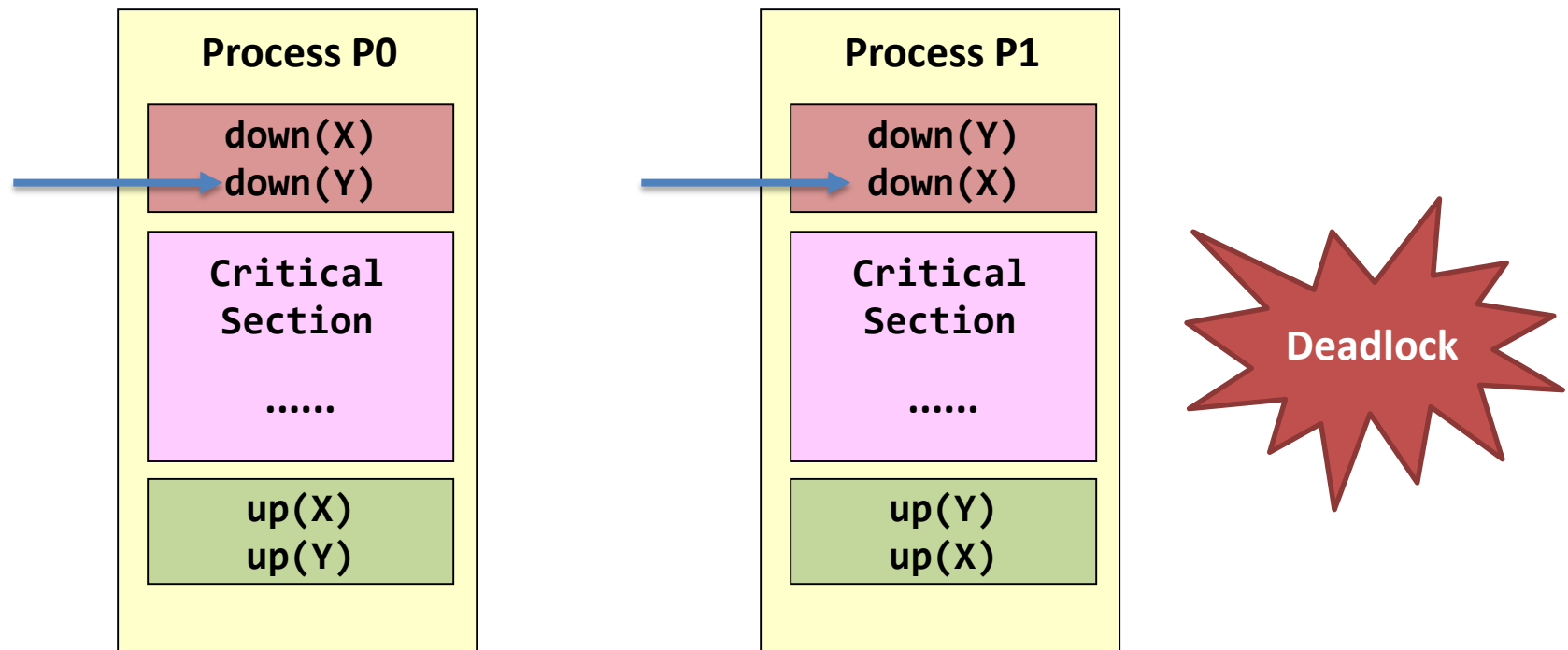
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them  
not to fight.



# Deadlock Example

- Problems when using semaphore



**Scenario:** P0 must wait until P1 executes  $up(Y)$ , P1 must wait until P0 executes  $up(X)$

# Deadlock Requirements

- **Requirement #1: Mutual Exclusion.**
  - Only one process at a time can use a resource
- **Requirement #2. Hold and wait.**
  - A process must be holding at least one resource and waiting to acquire additional resources held by other processes

# Deadlock Requirements

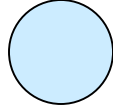

- **Requirement #3: No preemption.**

- A resource can be released only voluntarily by the process holding it after that process has completed its task

- **Requirement #4. Circular wait.**

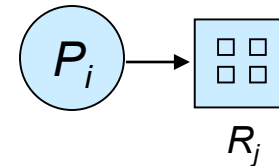
- There exists a set  $\{P_0, P_1, \dots, P_n\}$  of waiting processes such that  $P_0$  waits for  $P_1$ ,  $P_1$  waits for  $P_2$ , ...,  $P_{n-1}$  waits for  $P_n$ ,  $P_n$  waits for  $P_0$

# How to Handle Deadlocks

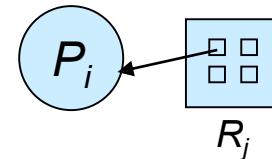
- Deadlock characterization: Deadlocks can be described using **resource-allocation graph**
  - Set  $V$  is partitioned into two types:
    - $P = \{P_1, P_2, \dots, P_n\}$ : processes 
    - $R = \{R_1, R_2, \dots, R_m\}$ : all resource types (each type may have multiple instances) 

– Set  $E$

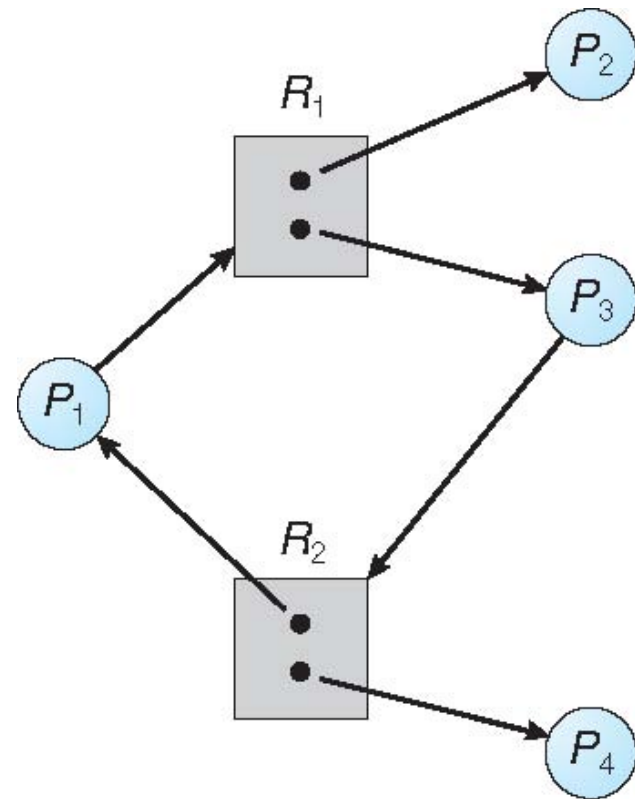
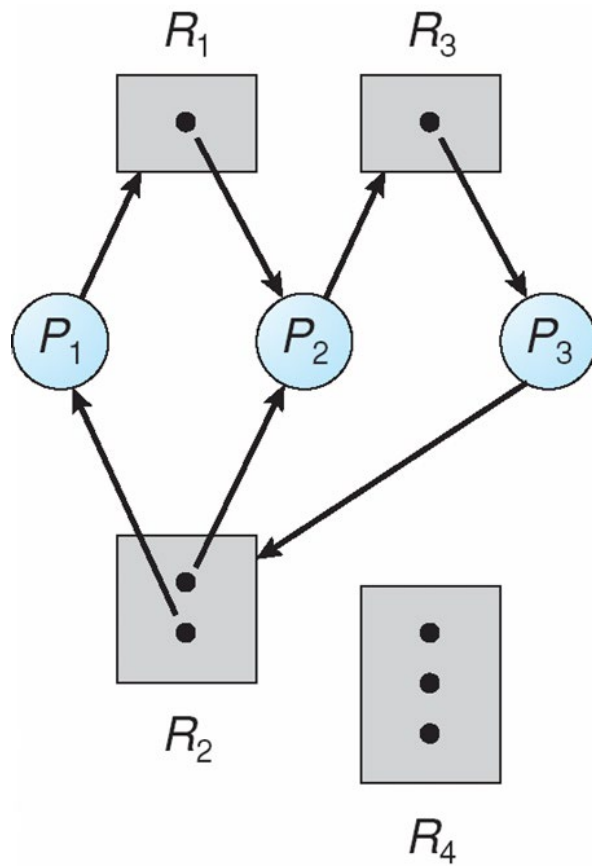
- **request edge** – directed edge  $P_i \rightarrow R_j$



- **assignment edge** – directed edge  $R_j \rightarrow P_i$



# Examples



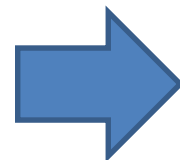
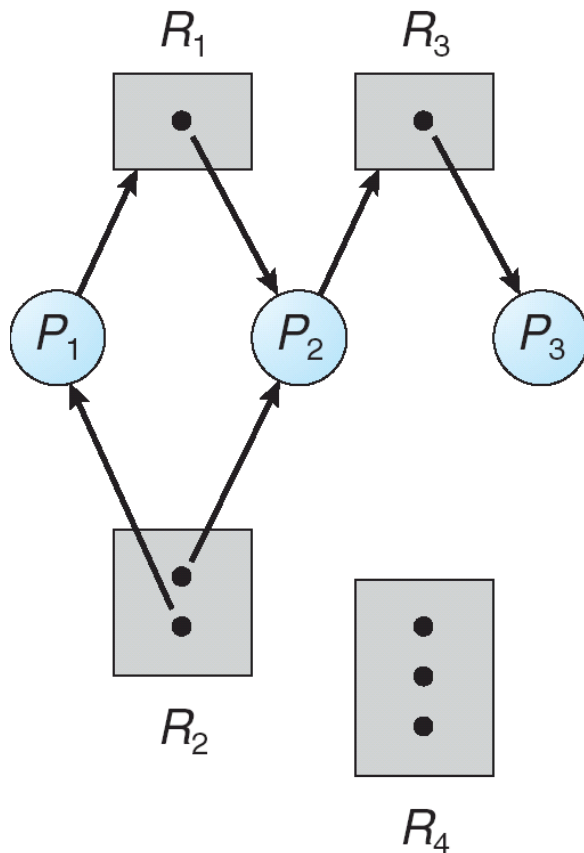


# How to Handle Deadlocks

- **Detect** deadlock and recover

- Case 1: Each resource has one instance

- Resource-allocation graph: detect the existence of a cycle

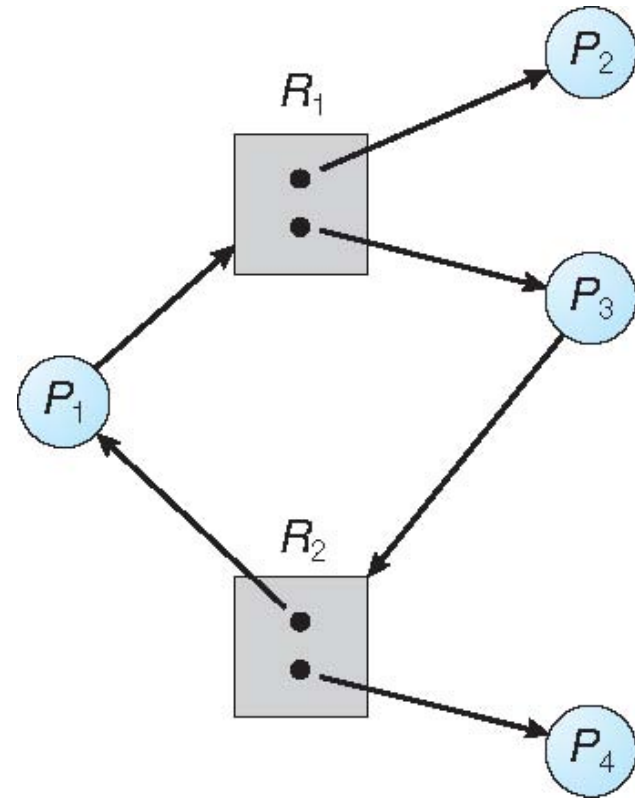
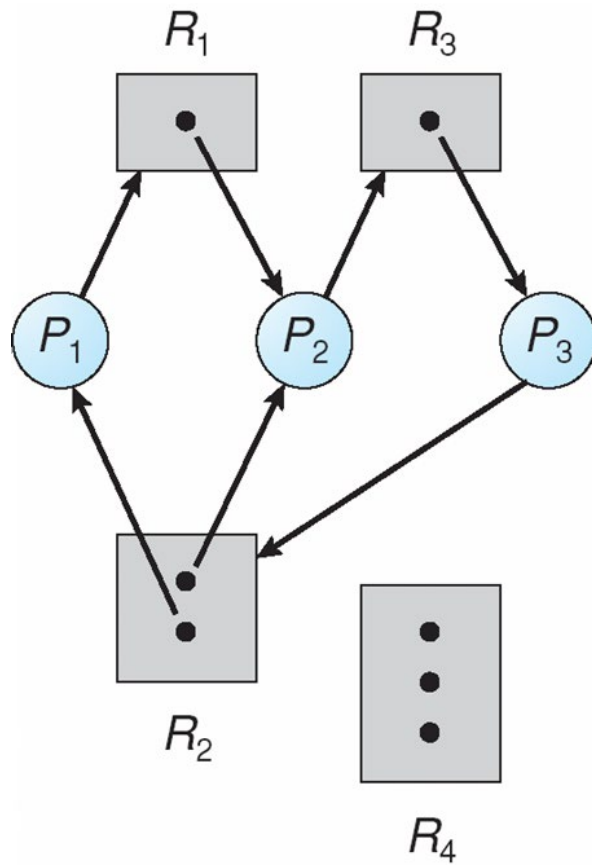


<b>No cycles</b>
No deadlock

<b>Contains a cycle</b>
Case 1: only one instance per resource
Case 2: several instances per resource type: possible deadlock

# Examples



# How to Handle Deadlocks

- **Detect** deadlock and recover

- Case 2: Each resource has multiple instances

- Matrix method: four data structures

- Existing (total) resources ( $m$  types):  $(E_1, E_2, \dots, E_m)$

- Available resources:  $(A_1, A_2, \dots, A_m)$

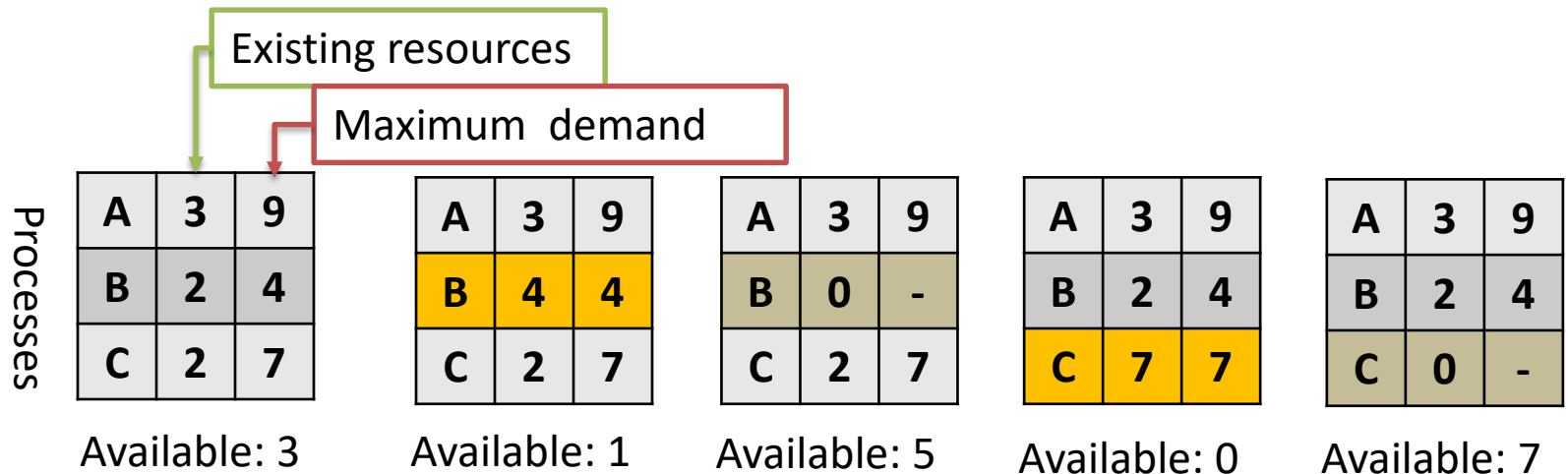
- Allocation matrix:  $\begin{bmatrix} C_{11} & \cdots & C_{1m} \\ \vdots & \ddots & \vdots \\ C_{n1} & \cdots & C_{nm} \end{bmatrix}$  ( $C_{ij}$ : # of type- $j$  resources held by process  $i$ )

- Request matrix:  $\begin{bmatrix} R_{11} & \cdots & R_{1m} \\ \vdots & \ddots & \vdots \\ R_{n1} & \cdots & R_{nm} \end{bmatrix}$  ( $R_{ij}$ : # of type- $j$  resources requested by process  $i$ )

- Repeatedly check  $P_i$  s.t.  $R_i \leq A$ ? ( $P_i$  can be satisfied)
  - ✓ Yes:  $A = A + C_i$  (release resources)
  - ✓ No: End (remaining processes are deadlocked)

# How to Handle Deadlocks

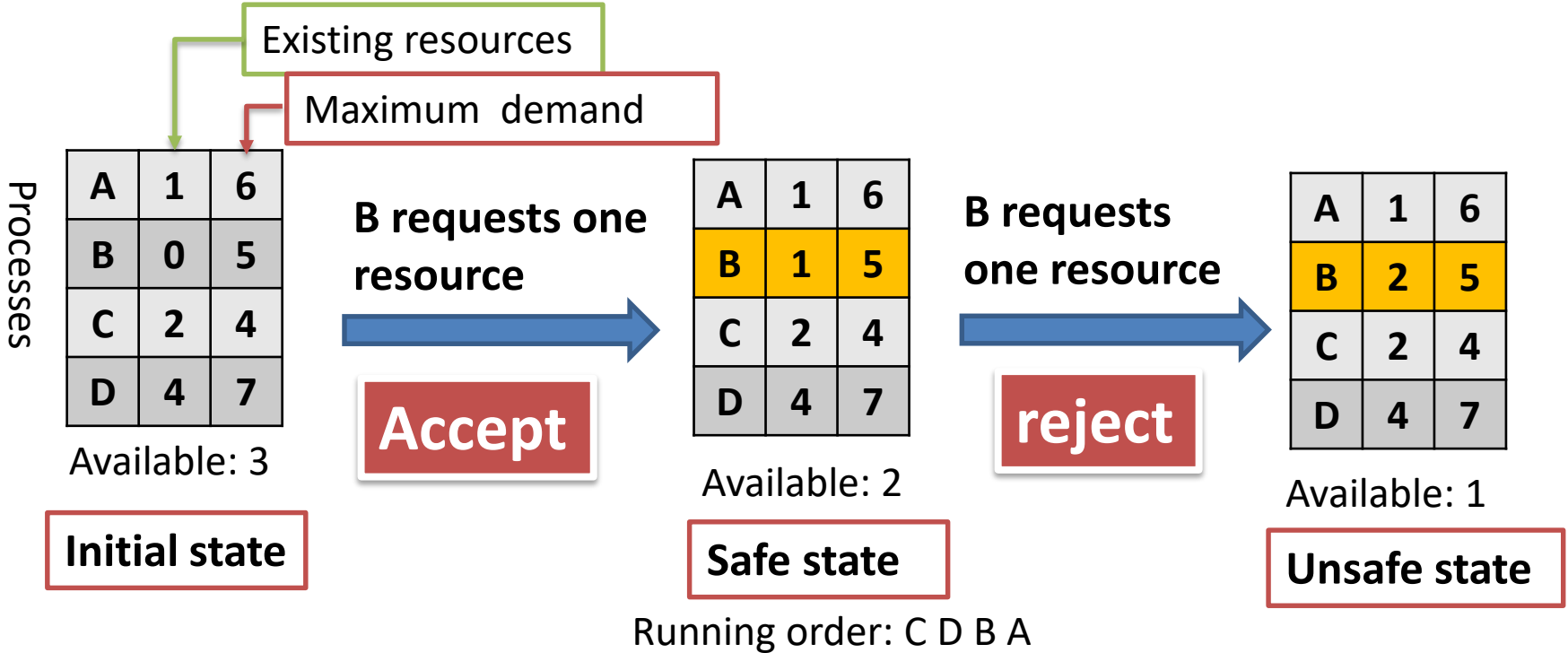
- **Prevent/avoid** deadlocks: Banker's algorithm
  - Idea: check system state defined by  $(E, A, C, R)$ 
    - **Safe state**: exist one running sequence to guarantee that all processes' demand can be satisfied



- **Unsafe state**: Not exist any sequence to guarantee the demand
  - It is not deadlock (it can still run for some time/processes may release some resources)

# How to Handle Deadlocks

- **Prevent/avoid** deadlocks: Banker's algorithm
  - For each request: safe (accept), unsafe (reject)



The algorithm can also be extended to the case of multiple resources, but it needs to know the demand

# How to Handle Deadlocks

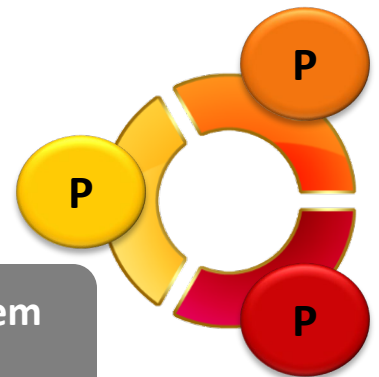
- **Ignore** the problem and pretend that deadlocks never occur (stop functioning and restart manually)
  - 鸵鸟算法（假装没发生）
  - Used by most operating systems, including UNIX and windows
  - Deadlocks occur infrequently, avoiding/detecting it is expensive
- A deadlock-free solution does not eliminate **starvation**

# The Deadlock Problem

## Classic IPC problems

- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them  
not to fight.



# What are the problems?

- All the IPC classical problems use **semaphores** to fulfill the synchronization requirements.

	Properties	Examples
Producer-Consumer Problem	Two classes of processes: <u>producer</u> and <u>consumer</u> ; At least one producer and one consumer.	FIFO buffer, such as pipe.
Dining Philosophy Problem	They are all running the same program; At least two processes.	Cross-road traffic control.
Reader-Writer Problem	Two classes of processes: <u>reader</u> and <u>writer</u> . No limit on the number of the processes of each class.	Database.

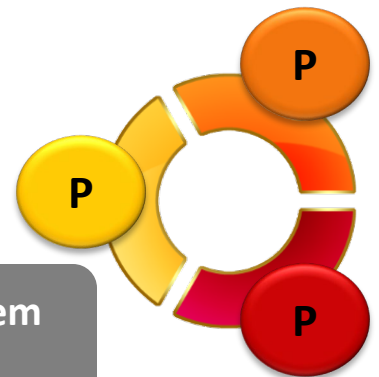


# The Deadlock Problem

## Classic IPC problems

- **Producer-consumer problem**
- Dining philosopher problem
- Reader-writer problem

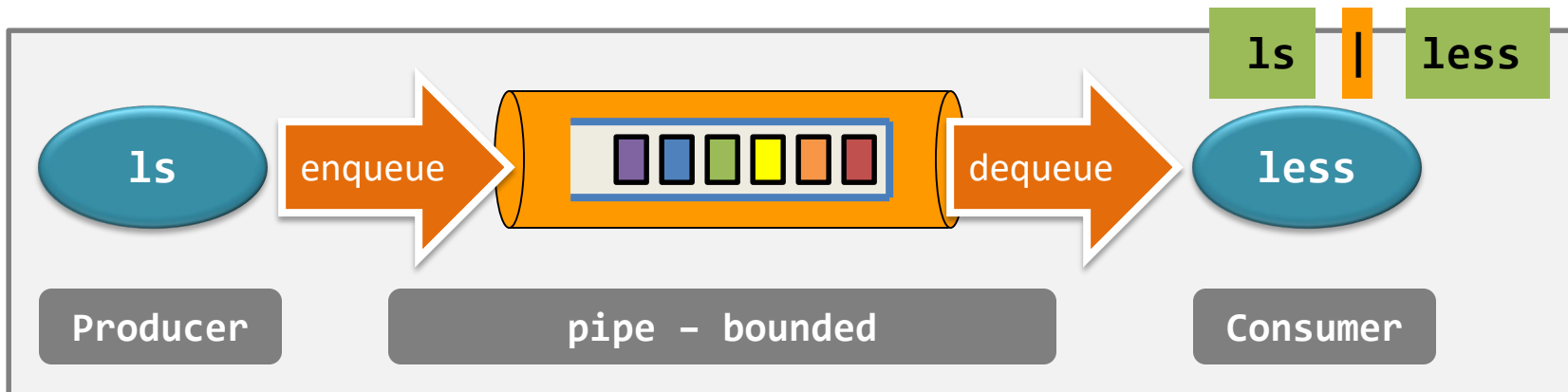
Let's teach them  
not to fight.



# Producer-consumer problem – recall

- Also known as the **bounded-buffer problem**.

<b>A bounded buffer</b>	-It is a shared object; -Its size is bounded, say N slots. -It is a queue (imagine that it is an array implementation of queue).
<b>A producer process</b>	-It produces a unit of data, and -writes that a piece of data to the tail of the buffer at one time.
<b>A consumer process</b>	-It removes a unit of data from the head of the bounded buffer at one time.



# Producer-consumer problem – recall

## Producer-consumer requirement #1

When the producer wants to  
(a) put a new item in the buffer, but  
(b) **the buffer is already full...**

Then,

- (1) **The producer should be suspended**, and
- (2) **The consumer should wake the producer up** after she has dequeued an item.

## Producer-consumer requirement #2

When the consumer wants to  
(a) consumes an item from the buffer, but  
(b) **the buffer is empty...**

Then,

- (1) **The consumer should be suspended**, and
- (2) **The producer should wake the consumer up** after she has enqueued an item.

# Producer-consumer problem

- Pipe is working fine. Is it enough?
  - What if we cannot use pipes?
    - Say, there are 2 producers and 2 consumers without any parent-child relationships?
  - Then, **the kernel can't protect you with a pipe.**
- In the following, we revisit the producer-consumer problem with the use of shared objects and semaphores, instead of pipe.

# Design – Semaphores

- **ISSUE #1: Mutual Exclusion.**

**Solution:** one binary semaphore (mutex)

- **ISSUE #2: Synchronization (coordination).**

- Remember the two requirements:

- Insert an item when it is not FULL
- Consume an item when it is not EMPTY

- Can we use a binary semaphore?

**Solution:** two counting semaphores (full & empty)

# Producer-consumer problem – solution

## Note

The functions “`insert_item()`” and “`remove_item()`” are accessing the bounded buffer (codes in critical section).

The size of the bounded buffer is “`N`”.

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7
8         insert_item(item);
9
10
11     }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6
7         item = remove_item();
8
9
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – solution

## Note

Mutual exclusion requirement

Synchronization requirement

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7
8         insert_item(item);
9
10
11     }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6
7         item = remove_item();
8
9
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7
8         insert_item(item);
9
10
11     }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6
7         item = remove_item();
8
9
10        consume_item(item);
11    }
12 }
```



# Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

**mutex:**

What is its purpose?

Why is the initial value of mutex 1?

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10    }
11 }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

**mutex:**

what is its purpose?

Why is the initial value of mutex 1?

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10
11     }
12 }
```

The “**mutex**” stands for mutual exclusion.

- **down()** and **up()** statements are the entry and the exit of the critical section, respectively.

What is the meaning of the initial value 1?

# Producer-consumer problem – Understanding

Why we need three semaphores, “empty”, “full”, “mutex”?

How about “full” and “empty”?

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – Understanding

- The two variables are not for mutual exclusion, but for **process synchronization**.
  - “*Process synchronization*” means **to coordinate** the set of processes so as to produce meaningful output.

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – Understanding

For “empty”,

- Its initial value is N;
- It decrements by 1 in each iteration.
- When it reaches 0, the producers sleeps.

So, does it sound like one of the requirements?

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

The consumer wakes the producer up when it finds “empty” is 0.

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – Understanding

- Semaphore can be more than mutual exclusion!

**empty**

It represents the number of empty slots.

**full**

It represents the number of occupied slots.

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         down(&empty);
7         down(&mutex);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – question

## Question.

Can we swap Lines 6 & 7 of the producer?

Let us simulate what will happen with the modified code!

## Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

## Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex); ←
7*      down(&empty); ←
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```

# Producer-consumer problem – question

Producer

running until Line  
10

Consumer

We are showing the value of the semaphores before the producer is suspended.

mutex = 1

empty = 0

full = N

## Producer function

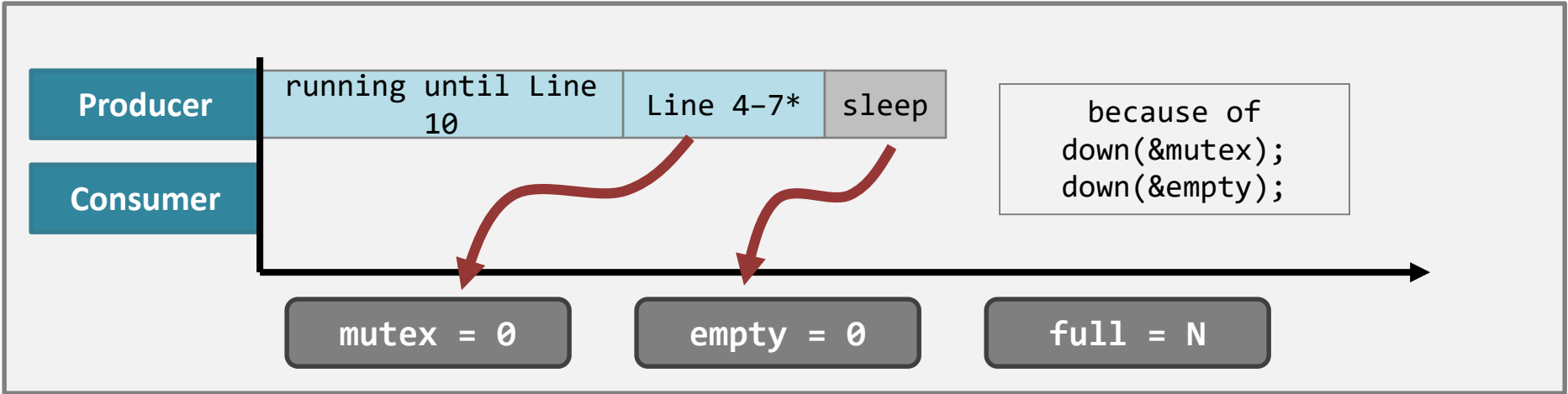
```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      down(&mutex);
7*      down(&empty);
8         insert_item(item);
9         up(&mutex);
10        up(&full);
11    }
12 }
```

## Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         down(&full);
6         down(&mutex);
7         item = remove_item();
8         up(&mutex);
9         up(&empty);
10        consume_item(item);
11    }
12 }
```



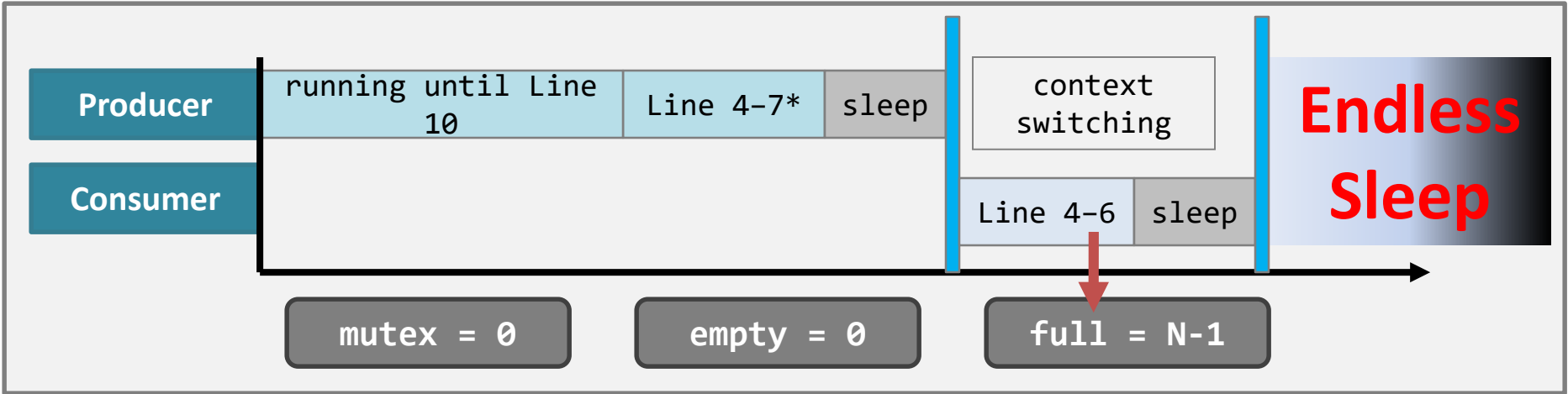
# Producer-consumer problem – question



```
Producer function  
1 void producer(void) {  
2     int item;  
3  
4     while(TRUE) {  
5         item = produce_item();  
6*      down(&mutex);  
7*      down(&empty);  
8         insert_item(item);  
9         up(&mutex);  
10        up(&full);  
11    }  
12 }
```

```
Consumer Function  
1 void consumer(void) {  
2     int item;  
3  
4     while(TRUE) {  
5         down(&full);  
6         down(&mutex);  
7         item = remove_item();  
8         up(&mutex);  
9         up(&empty);  
10        consume_item(item);  
11    }  
12 }
```

# Producer-consumer problem – question

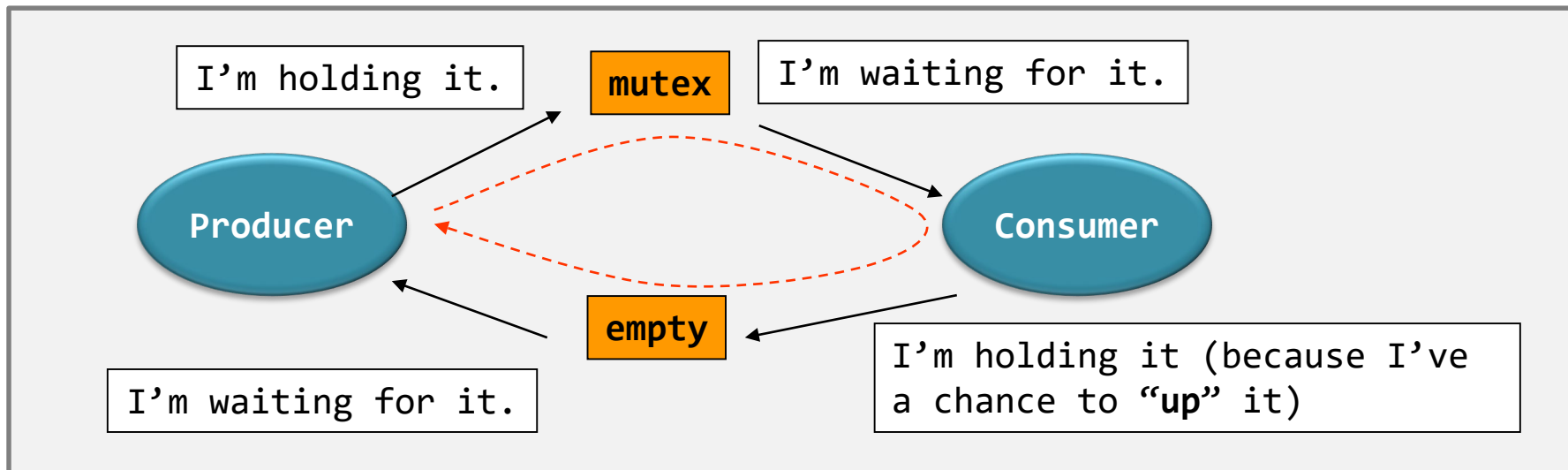


```
Producer function
1 void producer(void) {
2   int item;
3
4   while(TRUE) {
5     item = produce_item();
6*    down(&mutex);
7*    down(&empty);
8     insert_item(item);
9     up(&mutex);
10    up(&full);
11  }
12 }
```

```
Consumer Function
1 void consumer(void) {
2   int item;
3
4   while(TRUE) {
5     down(&full);
6     down(&mutex);
7     item = remove_item();
8     up(&mutex);
9     up(&empty);
10    consume_item(item);
11  }
12 }
```

# Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
  - The producer is waiting for the consumer to “up( )” the “empty” semaphore, and
  - the consumer is waiting for the producer to “up( )” the “mutex” semaphore.

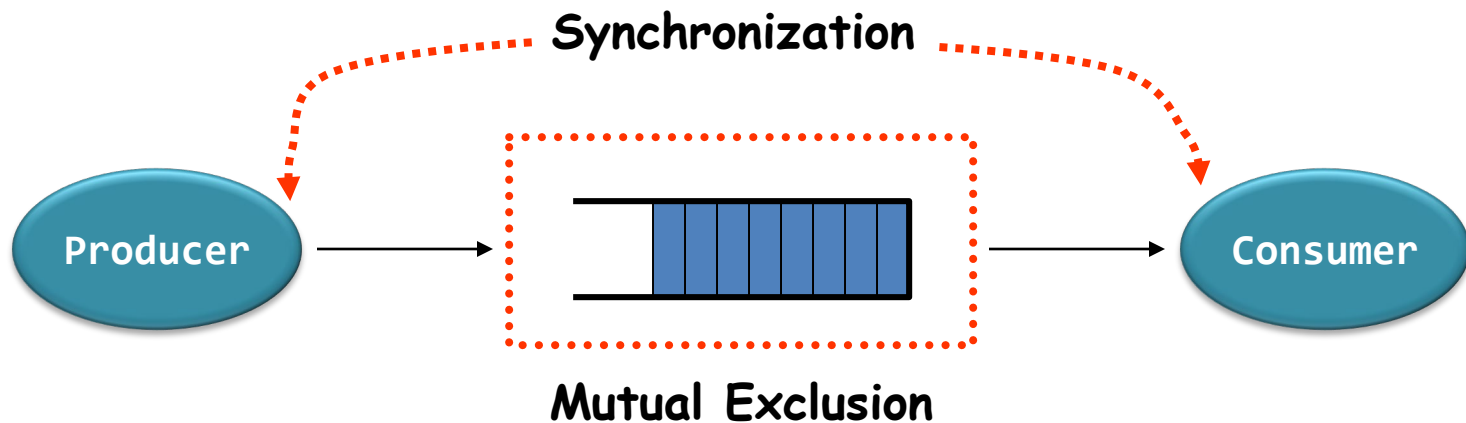


# Producer-consumer problem

- **Deadlock** happens when a **circular wait** appears
  - The producer is waiting for the consumer to “**up()**” the “**empty**” semaphore, and
  - the consumer is waiting for the producer to “**up()**” the “**mutex**” semaphore.
- **No progress could be made by all processes + All processes are blocked.**
  - **Implication:** careless implementation of the producer-consumer solution can be disastrous.

# Summary on producer-consumer problem

- The problem can be divided into two sub-problems.
  - Mutual exclusion.
    - The buffer is a shared object. Mutual exclusion is needed.
  - Synchronization.
    - Because the buffer's size is bounded, coordination is needed.



# Summary on producer-consumer problem

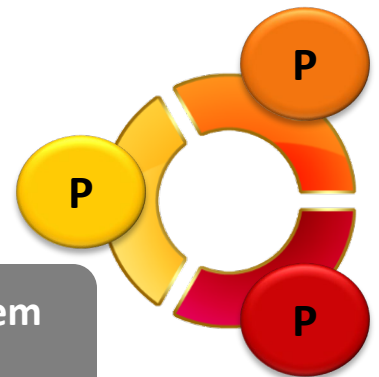
- How to guarantee mutual exclusion?
  - A **binary semaphore** is used as the entry and the exit of the critical sections.
- How to achieve synchronization?
  - Two semaphores are used as **counters** to monitor the status of the buffer.
  - Two semaphores are needed because the two suspension conditions are different.

# The Deadlock Problem

## Classic IPC problems

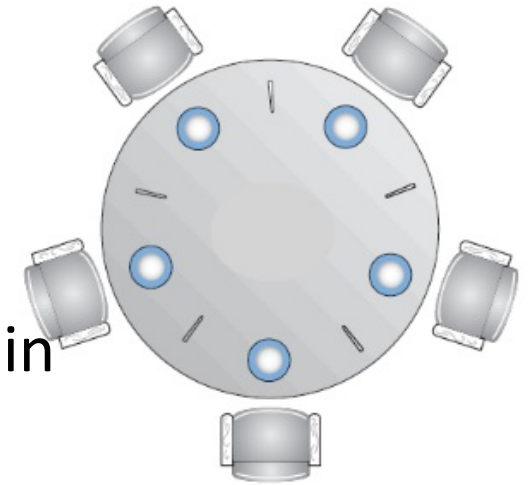
- Producer-consumer problem
- **Dining philosopher problem**
- Reader-writer problem

Let's teach them  
not to fight.



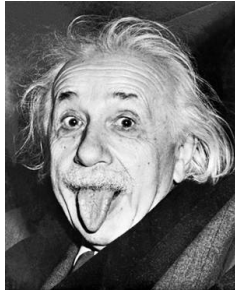
# Dining philosopher – introduction

- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks.
- The jobs of each philosopher are
  - to think and
  - to eat: They **need exactly two chopsticks** in order to eat the spaghetti.
- Question: how to construct a synchronization protocol such that
  - they will not result in any **deadlocking scenarios**, and
  - they will not be **starved to death**

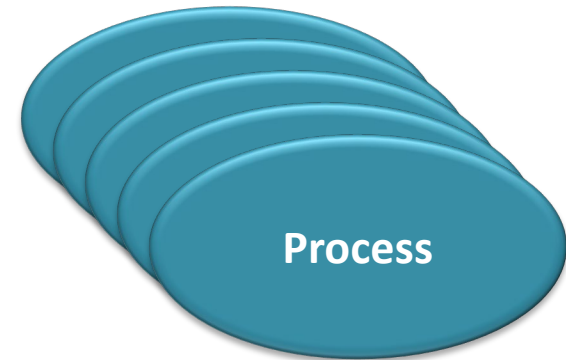




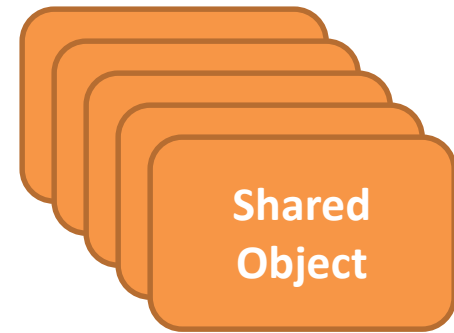
# Dining philosopher – introduction



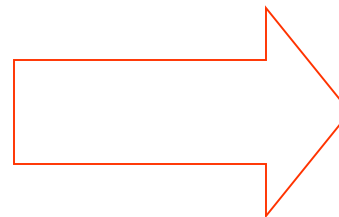
Philosophers



Chopsticks



Spaghetti

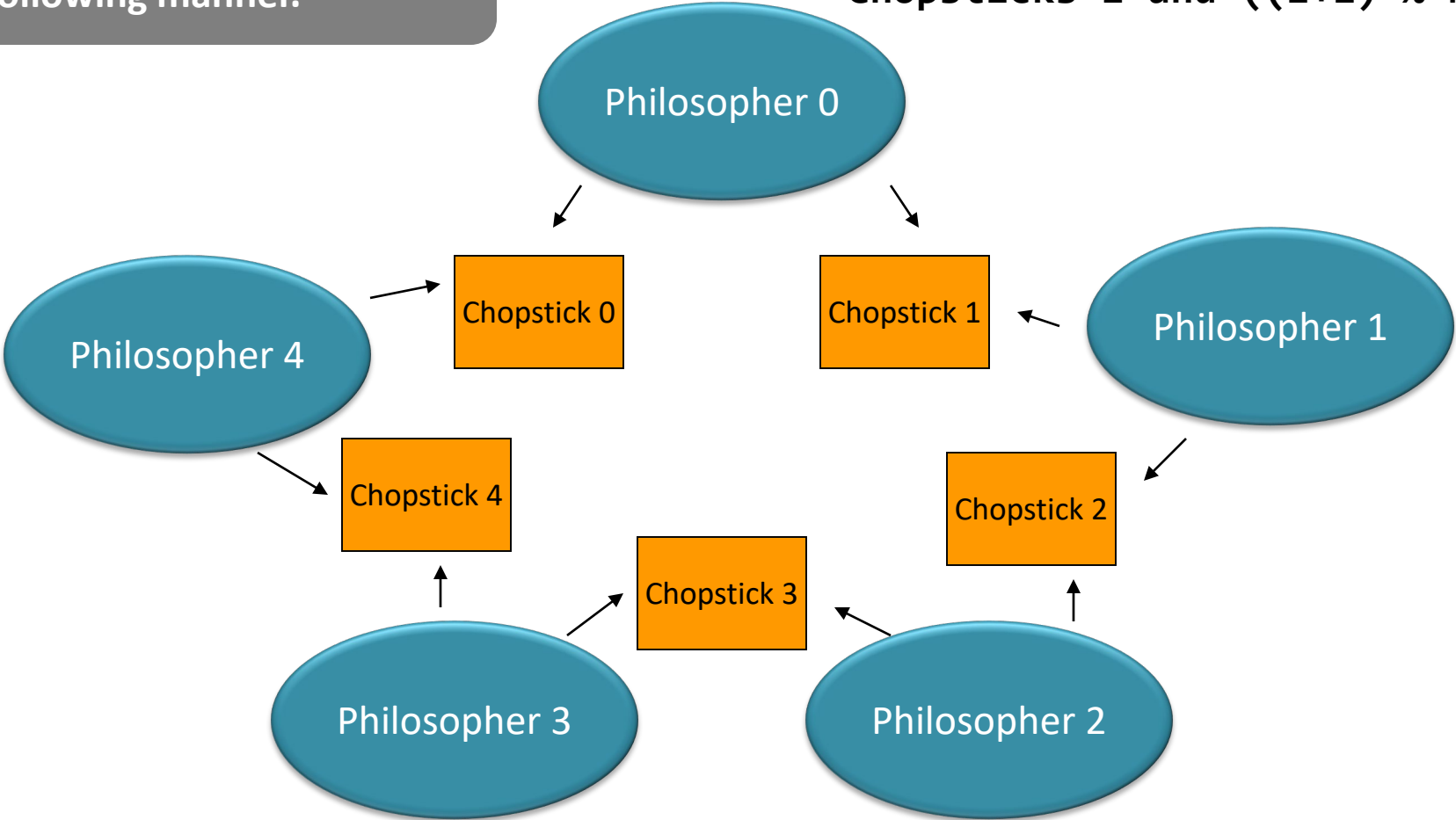


Consider to have  
infinite supply.

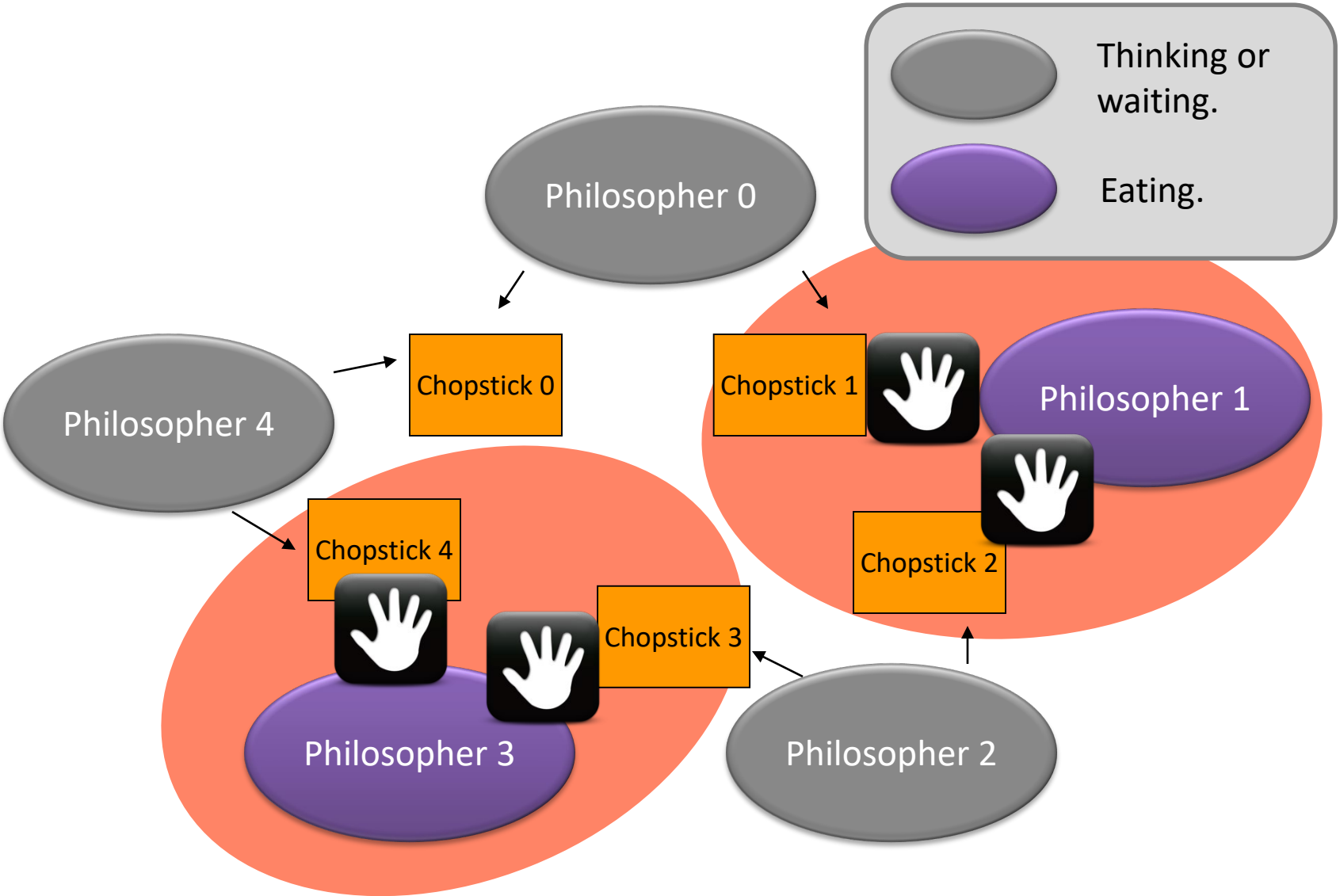
# Dining philosopher – introduction

The chopsticks are arranged in the following manner.

Philosopher  $i$  needs  
Chopsticks  $i$  and  $((i+1) \% N)$ ;





# Dining philosopher – introduction

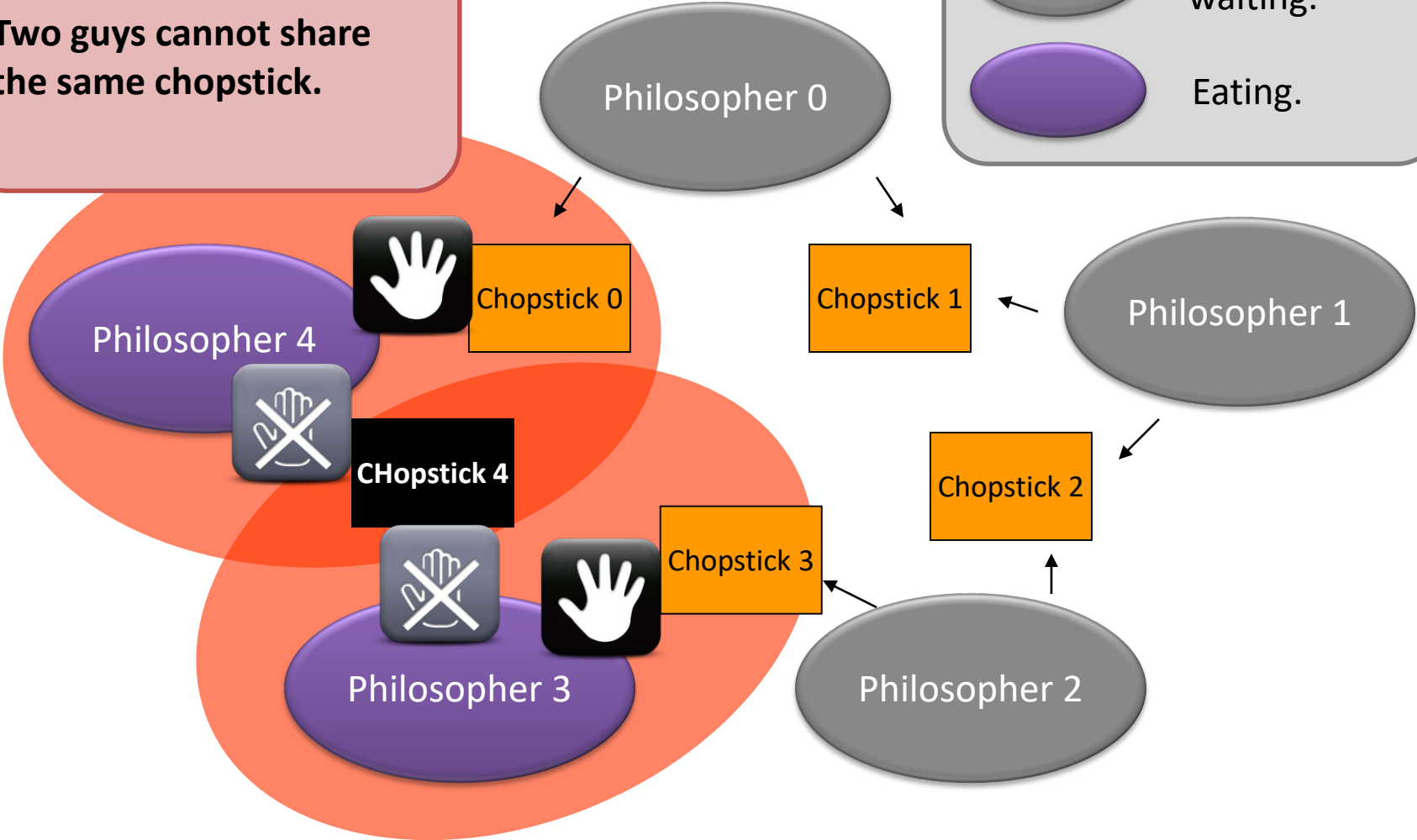


# Dining philosopher – introduction

Two guys cannot share the same chopstick.

Thinking or waiting. 

Eating. 



# Dining philosopher – requirement #1

- Mutual exclusion

- What if there is no mutual exclusion?

- Then: while you're eating, the two men besides you will and must **steal all your chopsticks!**

- Let's propose the following solution:

- When you are hungry, you have to check if anyone is using the chopstick that you need.

- If yes, you have to wait.

- If no, **seize both chopsticks.**

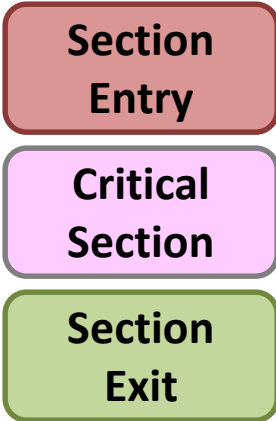
- After eating, put down all your chopsticks.

# Dining philosopher – meeting requirement #1?

**A quick question:** what should be initial values?

```
Shared object  
#define N 5  
semaphore chop[N];
```

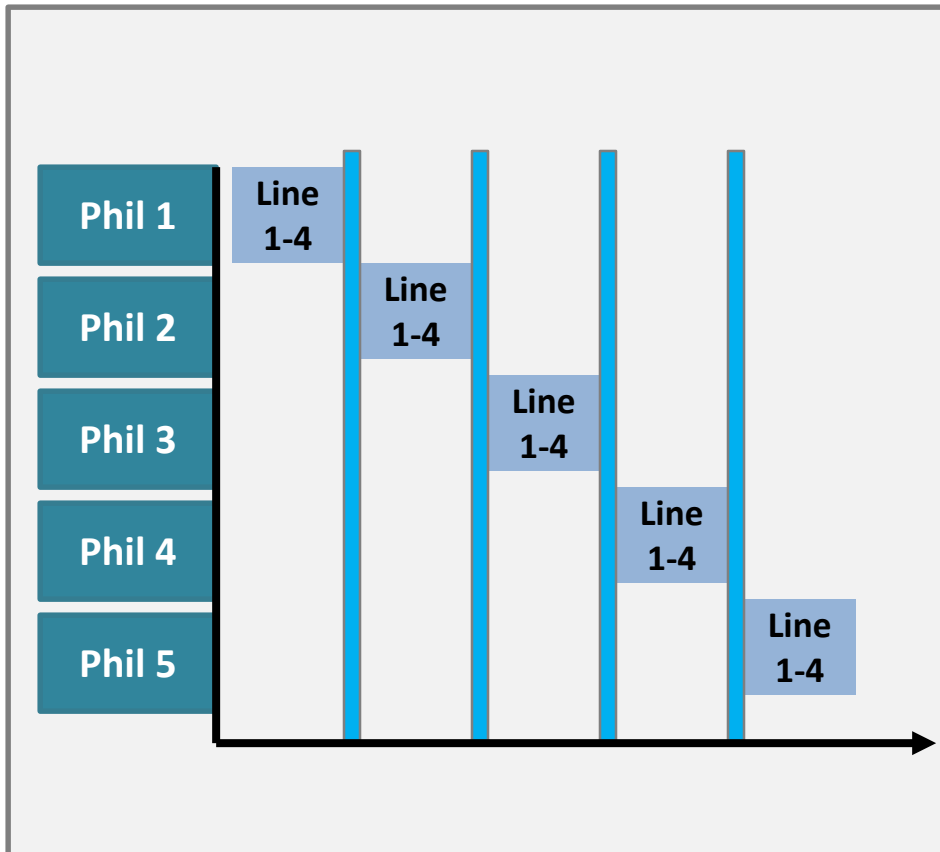
```
Helper Functions  
void take(int i) {  
    down(&chop[i]);  
}  
  
void put(int i) {  
    up(&chop[i]);  
}
```



```
Main Function  
  
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         take((i+1) % N);  
6         eat();  
7         put(i);  
8         put((i+1) % N);  
9     }  
10 }
```

# Dining philosopher – meeting requirement #1?

Final Destination: Deadlock!



## Main Function

```
1 void philosopher(int i) {  
2     while (TRUE) {  
3         think();  
4         take(i);  
5         take((i+1) % N);  
6         eat();  
7         put(i);  
8         put((i+1) % N);  
9     }  
10 }
```

# Dining philosopher – requirement #2

- **Synchronization**

- Should avoid any **potential deadlocking execution order.**

- How about the following suggestions:

- First, a philosopher **takes a chopstick.**

- If a philosopher finds that he cannot take the second one, then he should **put down the first chopstick.**

- Then, the philosopher **goes to sleep** for a while.

- Again, the philosopher tries to get both chopsticks until both ones are seized.



# Dining philosopher – meeting requirement #2?

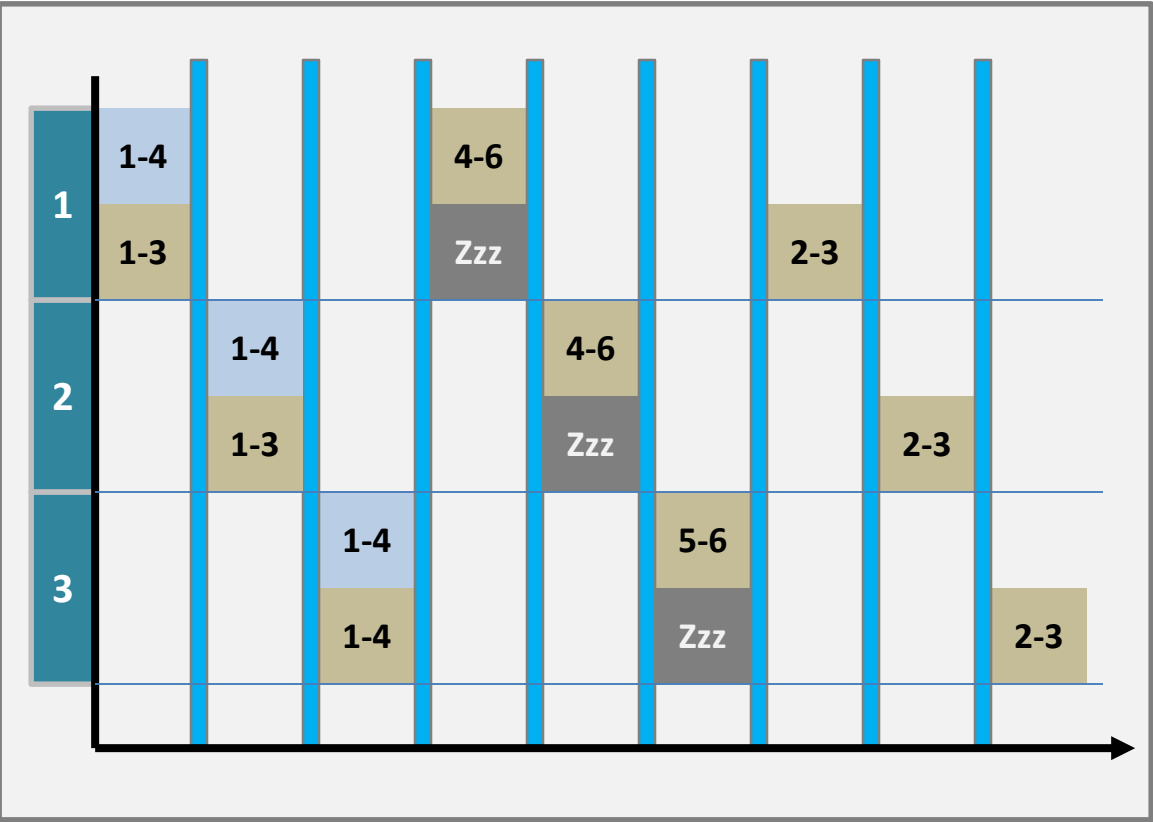
## The code: meeting requirement #2?

```
1 void philosopher(int i) {
2     while (TRUE) {
3         think();
4         take(i);
5         eat();
6         up(&chop[i]);
7         up(&chop[(i+1)%N]);
8     }
9 }
```

```
1 void take(int i) {
2     while(TRUE) {
3         down(&chop[i]);
4         if (isUsed((i+1)%N)) {
5             up(&chop[i]);
6             sleep(1);
7         }
8         else {
9             down(&chop[(i+1)%N]);
10            break;
11        }
12    }
13 }
```

# Dining philosopher – meeting requirement #2?

**Potential Problem: Philosophers are all busy but no progress were made!**



Assume N = 3 (because the space is limited)

```

1 void take(int i) {
2     while(TRUE) {
3         down(&chop[i]);
4         if (isUsed((i+1)%N)) {
5             up(&chop[i]);
6             sleep(1);
7         }
8     }
9     else {
10        down(&chop[(i+1)%N]);
11        break;
12    }
13 }
    
```

```

1 void philosopher(int i) {
2     while (TRUE) {
3         think();
4         take(i);
5         eat();
6         up(&chop[i]);
7         up(&chop[(i+1)%N]);
8     }
9 }
    
```

# Dining philosopher – before the final solution.

- Before we present the final solution, let's see what are the problems that we have.

## Problems

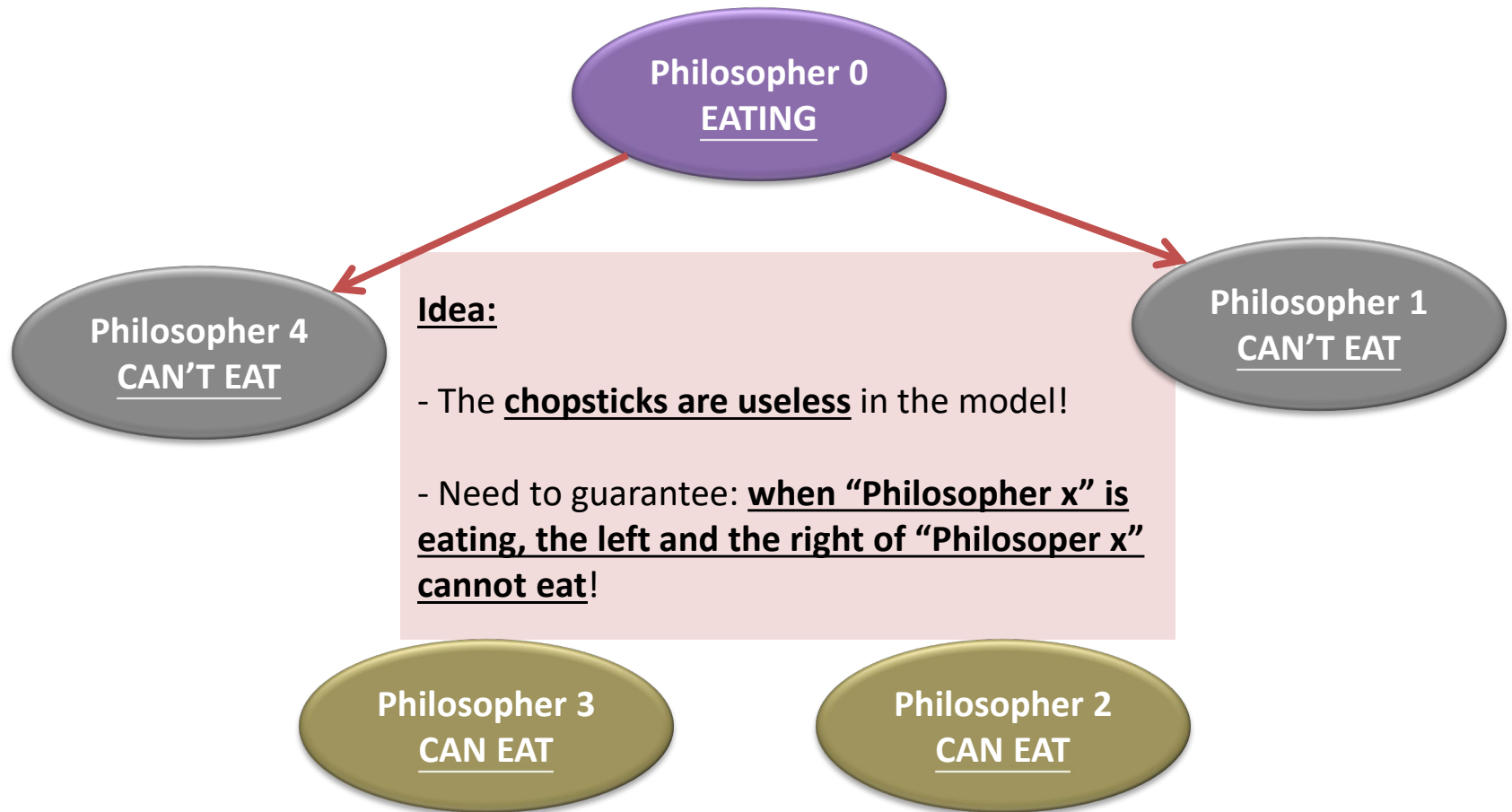
Model a chopstick as a semaphore is intuitive, but is not working.

The problem is that we are afraid to “`down()`”, as that may lead to a deadlock.

Using `sleep()` to avoid deadlock is effective, yet bringing another problem.

We can always create an execution order that keeps all the philosophers busy, but without useful output.

# Dining philosopher – before the final solution.



# Dining philosopher – the final solution.

## Shared object

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

## Main function

```
1 void philosopher(int i) {
2     think();
3     take(i);
4     eat();
5     put(i);
6 }
```

## Section entry

```
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

## Section exit

```
1 void put(int i) {
2     down(&mutex);
3     state[i] = THINKING;
4     test(LEFT);
5     test(RIGHT);
6     up(&mutex);
7 }
```

I will explain the  
code later.

## Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

# Dining philosopher – the final solution.

```
Shared object
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

Going “left” and “right” in a circular manner.

The states of the philosophers, including “EATING”, “THINKING”, and “HUNGRY”.  
Remember, this is shared array.

To guarantee mutual exclusive access to the “state[N]” array.

Guess:  
What is the meaning of the semaphore s[N]?

To fulfill the synchronization requirement.  
Question. What are the initial values of the “s[N]” array?

# Dining philosopher – the final solution.

```
Shared object
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N

int state[N];
semaphore mutex = 1;
semaphore s[N];
```

```
Section entry
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

Question. What are they doing?

If both chopsticks are available, I eat. Else, I sleep.

```
Extremely important helper function
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

If they are eating, I can't be eating.

# Dining philosopher – the final solution.

Try to let the one on the **left of the caller** to eat.

Try to let the one on the **right of the caller** to eat.

## Section exit

```
1 void put(int i) {
2     down(&mutex);
3     state[i] = THINKING;
4     test(LEFT);
5     test(RIGHT);
6     up(&mutex);
7 }
```

## Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

Wake up the one who can eat!



# Dining philosopher – the final solution.

An illustration: How can  
Philosopher 1 start eating?

Philosopher 0  
THINKING

Philosopher 4  
THINKING

**Note: no chopsticks objects  
will be shown in this  
illustration because we  
don't need them now.**

Philosopher 1  
THINKING

Philosopher 3  
THINKING

Philosopher 2  
THINKING

# Dining philosopher – the final solution.

```
Section entry
1 void take(int i) {
2   down(&mutex);
3   state[i] = HUNGRY;
4   test(i);
5   up(&mutex);
6   down(&s[i]);
7 }
```

Call take();

Philosopher 0  
HUNGRY

To LEFT:  
are you "EATING"?

To RIGHT:  
are you "EATING"?

Philosopher 4  
THINKING

Philosopher 1  
THINKING

Philosopher 3  
THINKING

Philosopher 2  
THINKING

# Dining philosopher – the final solution.

```
Section entry
1 void take(int i) {
2   down(&mutex);
3   state[i] = HUNGRY;
4   test(i);
5   up(&mutex);
6   down(&s[i]);
7 }
```

Call take();

Philosopher 0  
HUNGRY

To LEFT:  
are you "EATING"?

To RIGHT:  
are you "EATING"?

Philosopher 4  
THINKING

Philosopher 1  
THINKING

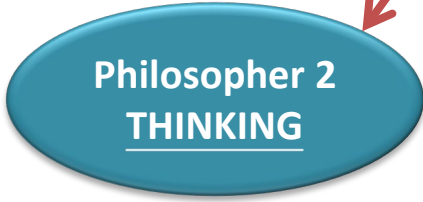
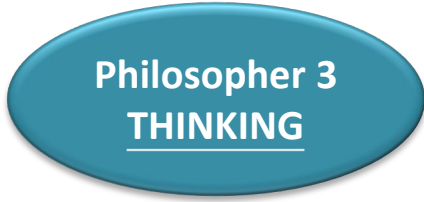
Calling take().  
but, it is blocked.  
Why?

Philosopher 3  
THINKING

Philosopher 2  
THINKING

# Dining philosopher – the final solution.

```
Section entry
1 void take(int i) {
2   down(&mutex);
3   state[i] = HUNGRY;
4   test(i);
5   up(&mutex);
6   down(&s[i]);
7 }
```



To LEFT:  
are you "EATING"?

Now, it is  
freed from  
blocking.

To RIGHT:  
are you  
"EATING"?

# Dining philosopher – the final solution.

```
Section entry
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

Philosopher 0  
EATING

Philosopher 1  
HUNGRY

Blocked;  
because of  
`down(&s[1]);`

Philosopher 4  
THINKING

To RIGHT:  
are you  
"EATING"?

Philosopher 3  
HUNGRY

To LEFT:  
are you  
"EATING"?

Philosopher 2  
THINKING

# Dining philosopher – the final solution.

## Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Philosopher 0  
EATING

Philosopher 4  
THINKING

Philosopher 1  
HUNGRY

Blocked;  
because of  
`down(&s[1]);`

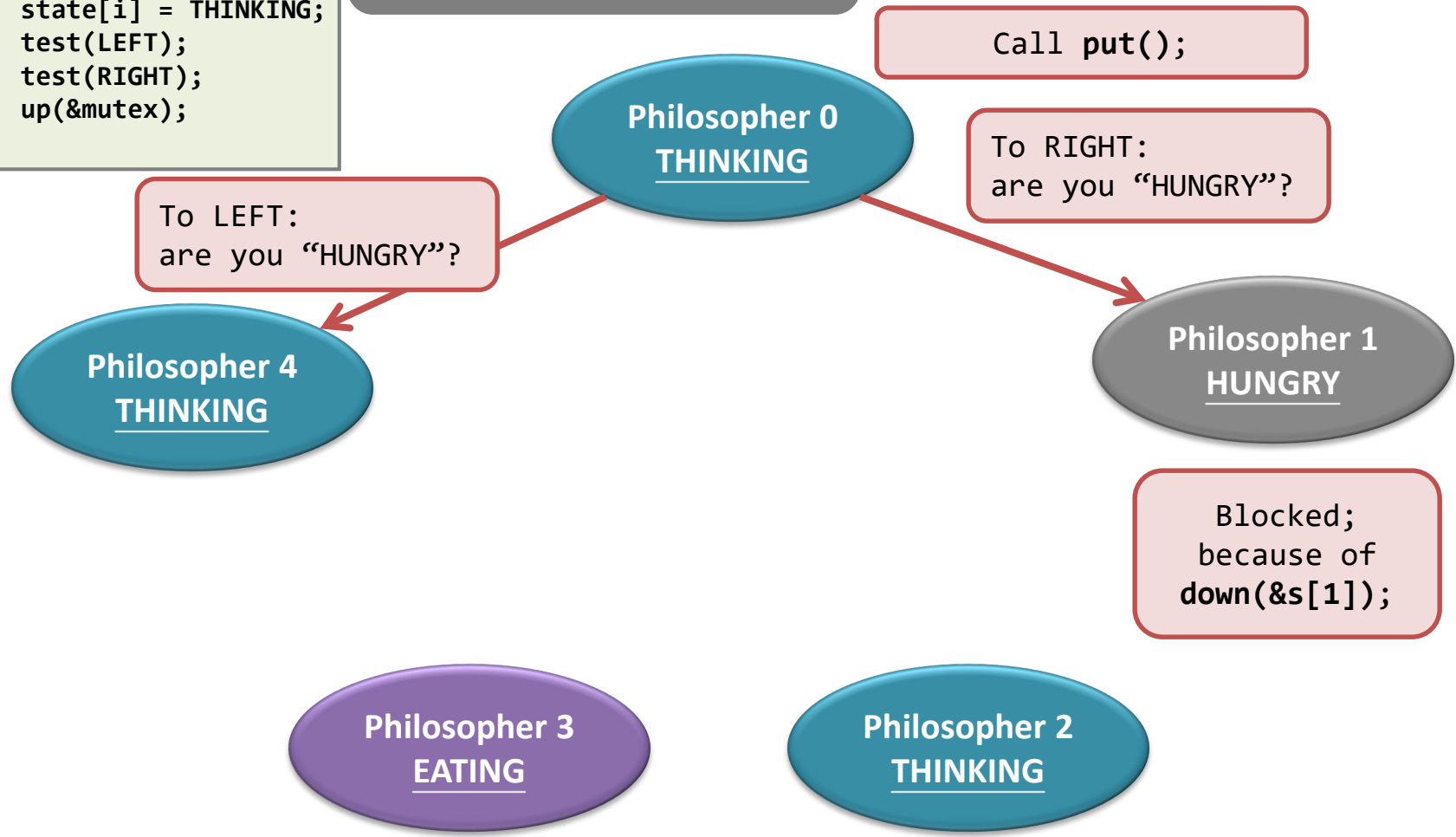
Philosopher 3  
EATING

Philosopher 2  
THINKING

# Dining philosopher – the final solution.

```
Section exit
1 void put(int i) {
2   down(&mutex);
3   state[i] = THINKING;
4   test(LEFT);
5   test(RIGHT);
6   up(&mutex);
7 }
```

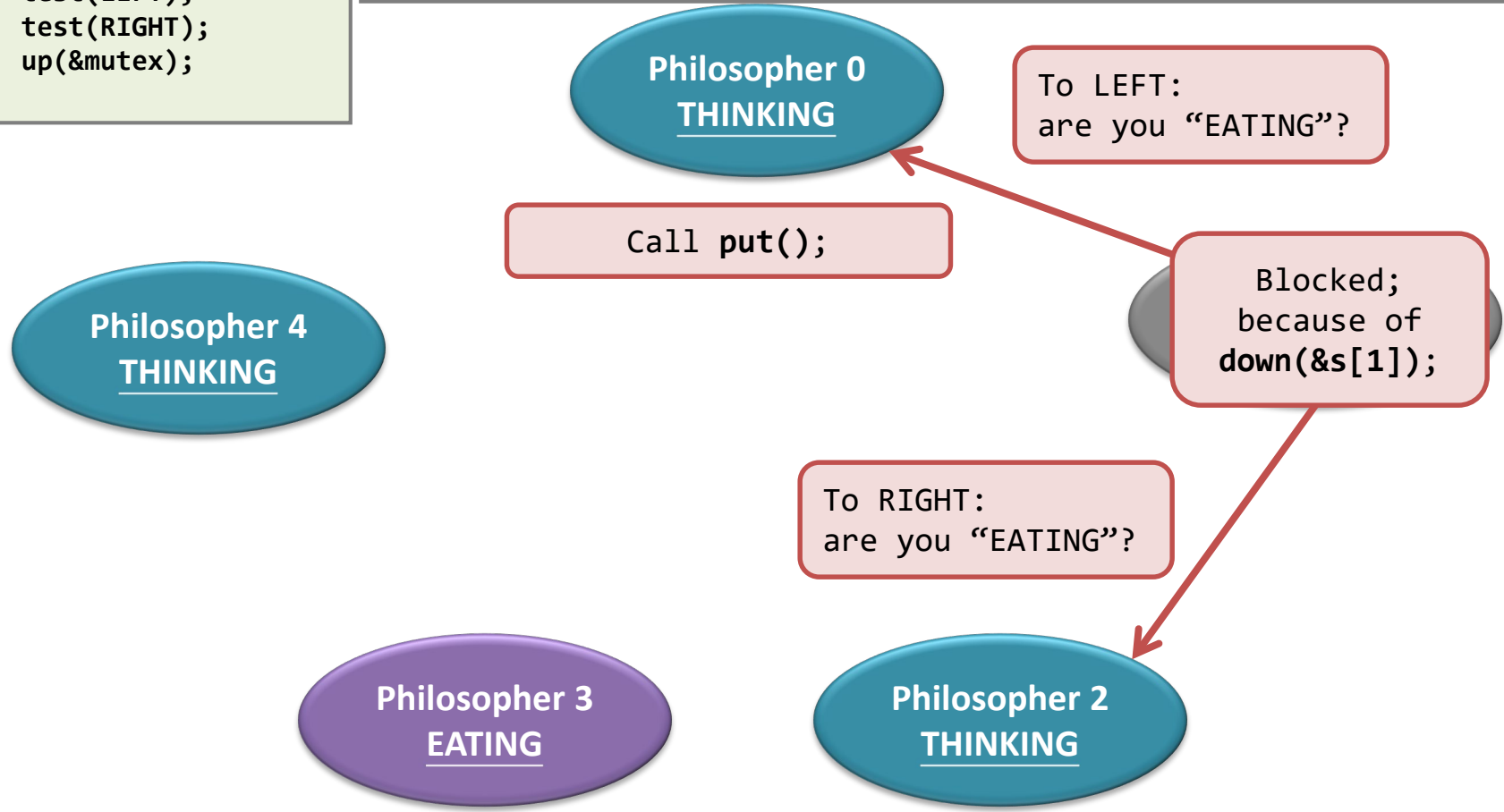
An illustration: How can Philosopher 1 start eating?



# Dining philosopher – the final solution.

```
Section exit
1 void put(int i) {
2   down(&mutex);
3   state[i] = THINKING;
4   test(LEFT);
5   test(RIGHT);
6   up(&mutex);
7 }
```

```
1 void test(int i) {
2   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3     state[i] = EATING;
4     up(&s[i]);
5   }
6 }
```

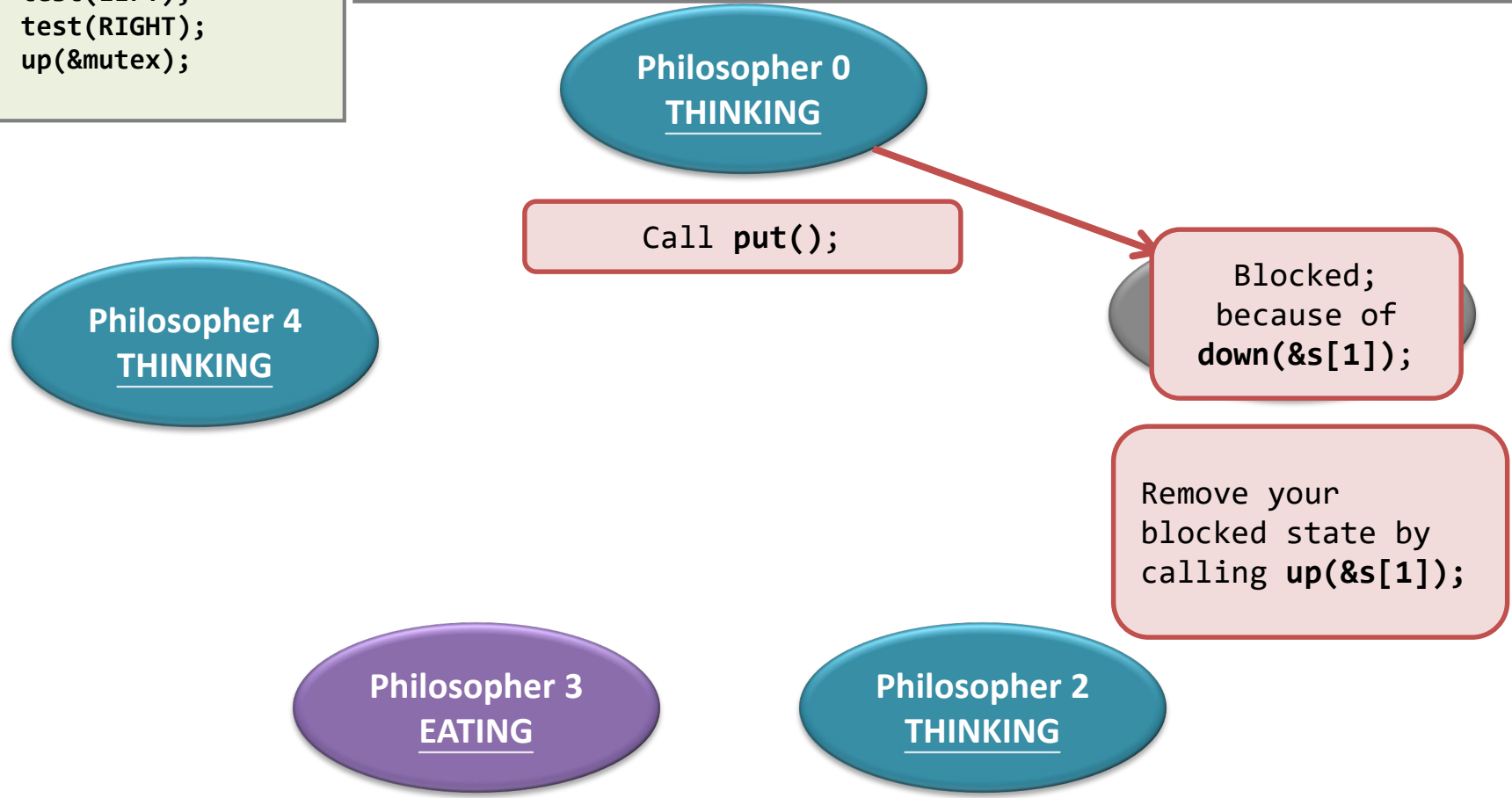




# Dining philosopher – the final solution.

```
Section exit
1 void put(int i) {
2   down(&mutex);
3   state[i] = THINKING;
4   test(LEFT);
5   test(RIGHT);
6   up(&mutex);
7 }
```

```
1 void test(int i) {
2   if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3     state[i] = EATING;
4     up(&s[i]);
5   }
6 }
```



# Dining philosopher – the final solution.

## Section entry

```
1 void take(int i) {  
2     down(&mutex);  
3     state[i] = HUNGRY;  
4     test(i);  
5     up(&mutex);  
6     down(&s[i]);  
7 }
```

Philosopher 0  
THINKING

Philosopher 4  
THINKING

Philosopher 1  
EATING

Eventually...

Philosopher 3  
EATING

Philosopher 2  
THINKING

# Dining philosopher - summary

- What is the shared object in the final solution?
  - How to guarantee the mutual exclusion

Section entry	Section exit
<pre>1 void take(int i) { 2     down(&amp;mutex); 3     state[i] = HUNGRY; 4     test(i); 5     up(&amp;mutex); 6     down(&amp;s[i]); 7 }</pre>	<pre>1 void put(int i) { 2     down(&amp;mutex); 3     state[i] = THINKING; 4     test(LEFT); 5     test(RIGHT); 6     up(&amp;mutex); 7 }</pre>

# Dining philosopher - summary

- Think:
  - Why the semaphore  $s[N]$  is needed
  - How to set its initial value

## Section entry

```
1 void take(int i) {
2     down(&mutex);
3     state[i] = HUNGRY;
4     test(i);
5     up(&mutex);
6     down(&s[i]);
7 }
```

## Extremely important helper function

```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

# Dining philosopher - summary

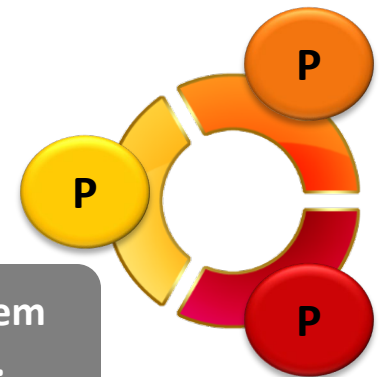
- Solution to IPC problem can be difficult to comprehend.
  - Usually, intuitive methods failed.
  - Depending on time, e.g., `sleep(1)`, does not guarantee a useful solution.
- As a matter of fact, dining philosopher **is not restricted to 5 philosophers.**

# The Deadlock Problem

## Classic IPC problems

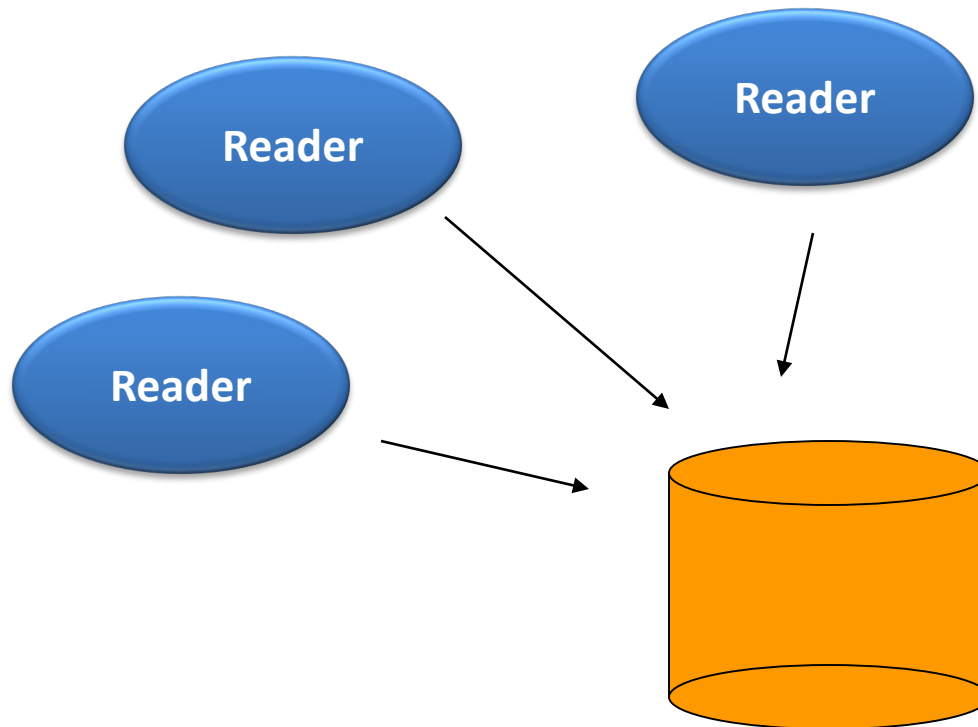
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

Let's teach them  
not to fight.



# Reader-writer problem – introduction

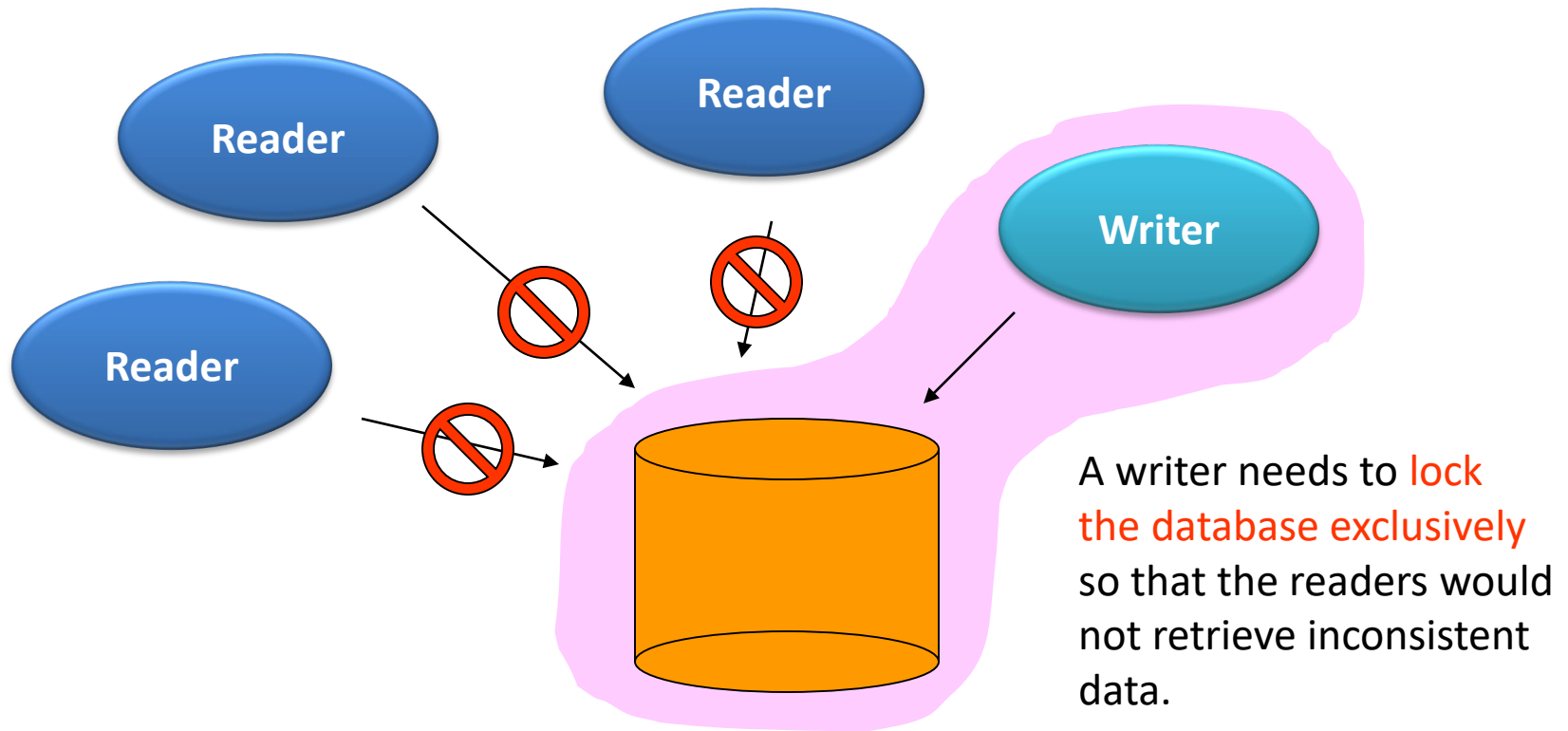
- It is a concurrent database problem.



Readers are allowed to read the content of the database concurrently.

# Reader-writer problem – introduction

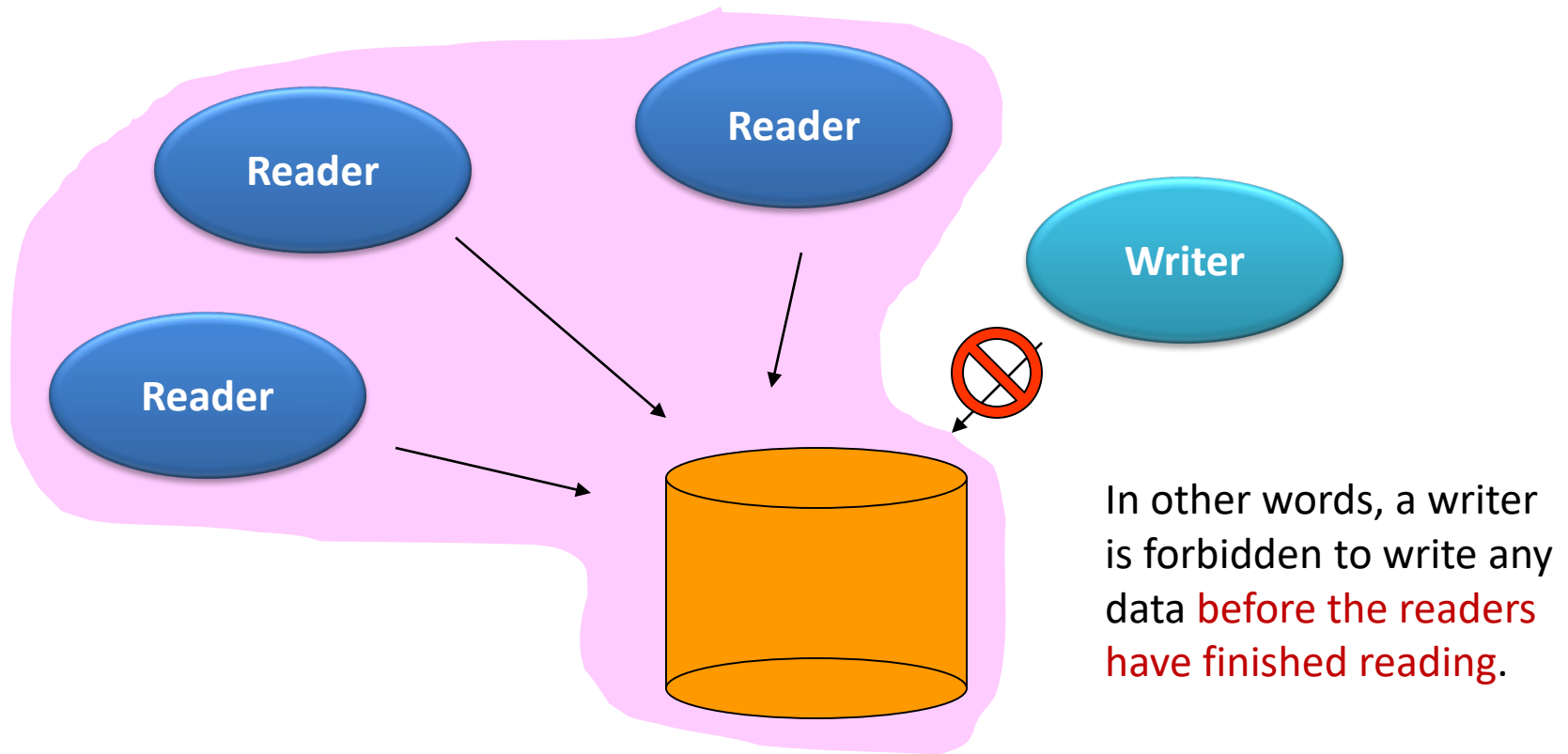
- It is a concurrent database problem.





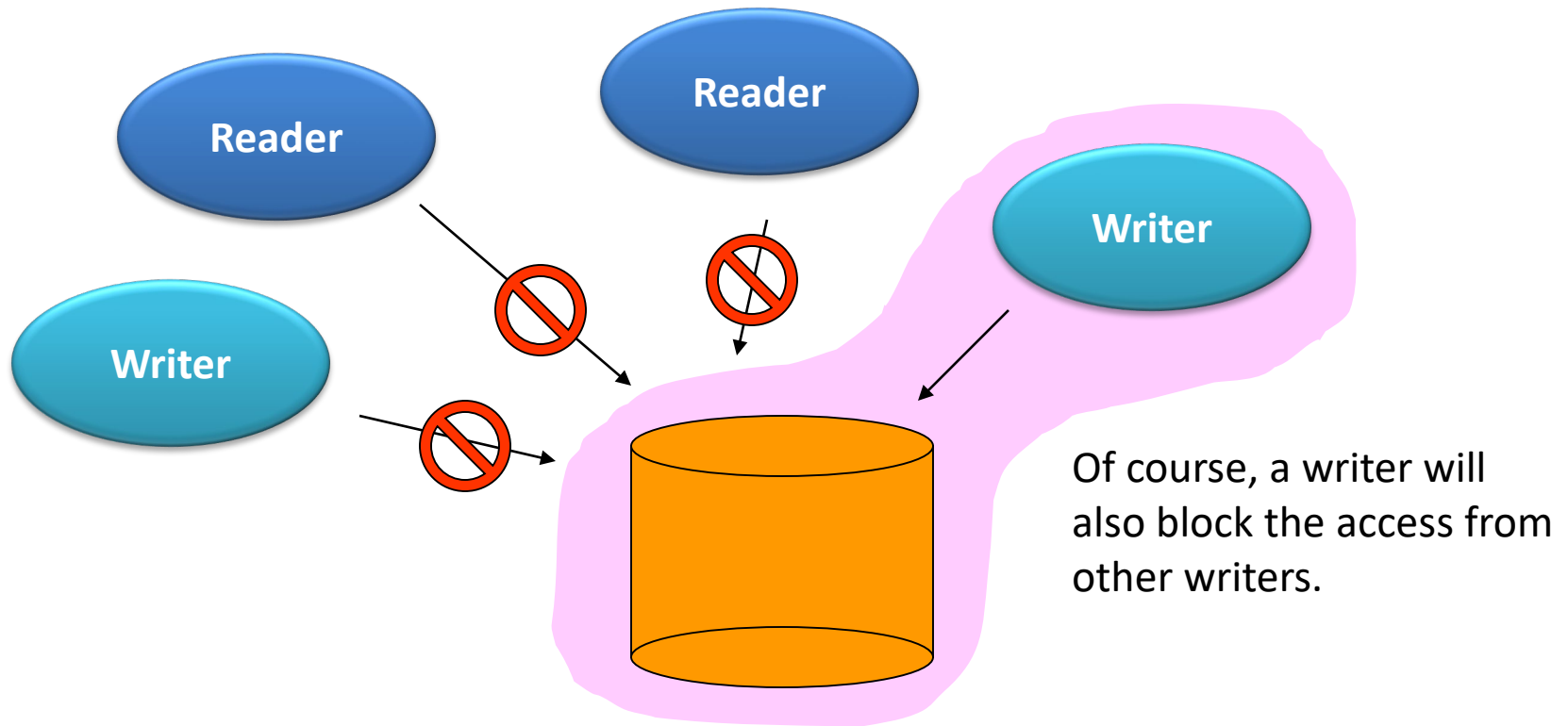
# Reader-writer problem – introduction

- It is a concurrent database problem.



# Reader-writer problem – introduction

- It is a concurrent database problem.

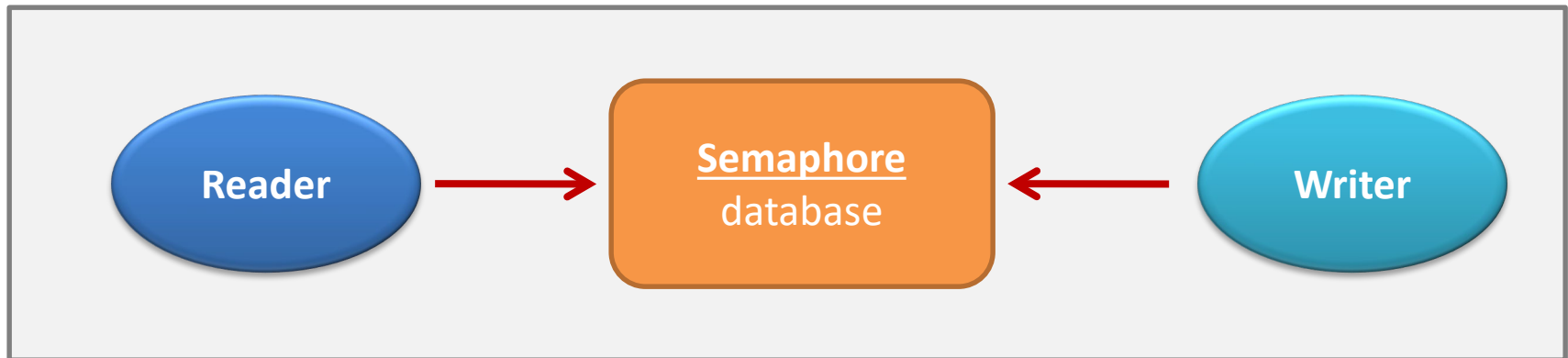


# Reader-writer problem – subproblems

- A mutual exclusion problem.
  - The database is a shared object.
- A synchronization problem.
  - **Rule 1.** While a reader is reading, other readers is allowed to read the database.
  - **Rule 2.** While a reader is reading, no writers is allowed to write to the database.
  - **Rule 3.** While a writer is writing, no writers and readers are allowed to access the database.
- A concurrency problem.
  - **Simultaneous access for multiple readers** is allowed and must be guaranteed.

# Reader-writer problem – solution outline

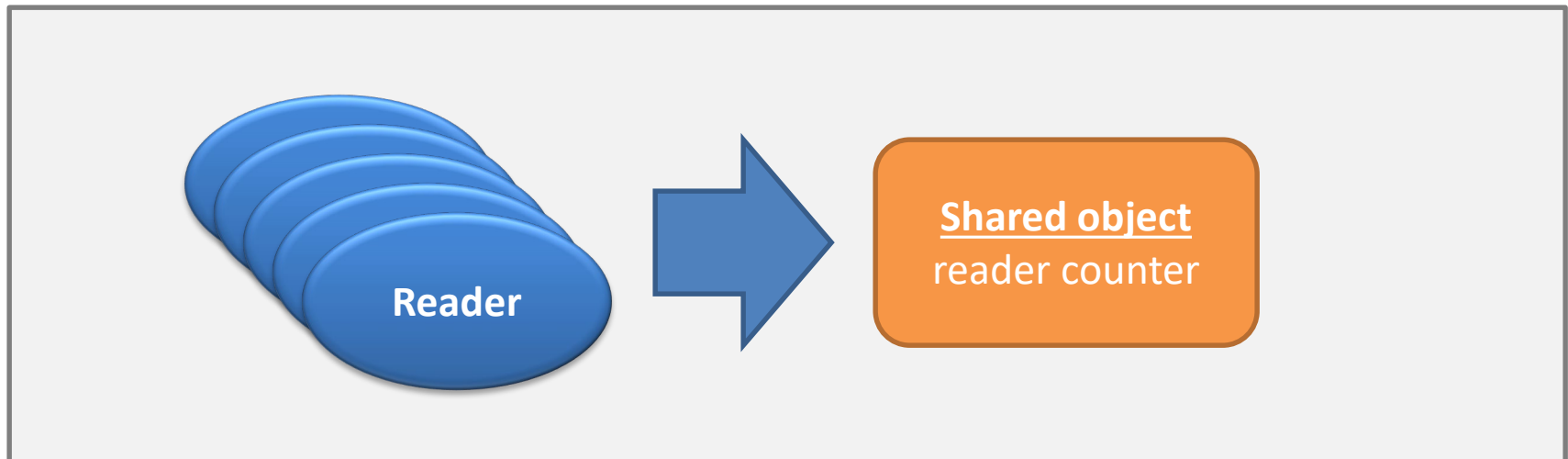
- **Mutual exclusion**: relate the readers and the writers to one semaphore.
  - This guarantees **no readers and writers** could proceed to their critical sections at the same time.
  - This also guarantees **no two writers** could proceed to their critical sections at the same time.



# Reader-writer problem – solution outline

- Readers' concurrency

- The **first reader coming** to the system “**down()**” the “**database**” semaphore.
- The **last reader leaving** the system “**up()**” the “**database**” semaphore.



# Reader-writer problem – final solution

## Shared object

```
semaphore db      = 1;  
semaphore mutex  = 1;  
int read_count   = 0;
```

## Writer function

```
1 void writer(void) {  
2   while(TRUE) {  
3     prepare_write();  
4     down(&db);  
5     write_database();  
6     up(&db);  
7   }  
8 }
```

Section Entry  
prepare\_write();  
down(&db);

Critical Section  
write\_database();

Section Exit  
up(&db);

## Reader Function

```
1 void reader(void) {  
2   while(TRUE) {  
3     down(&mutex);  
4     read_count++;  
5     if(read_count == 1)  
6       down(&db);  
7     up(&mutex);  
8     read_database();  
9     down(&mutex);  
10    read_count--;  
11    if(read_count == 0)  
12      up(&db);  
13    up(&mutex);  
14    process_data();  
15  }  
16 }
```

Section Entry  
down(&mutex);  
read\_count++;  
if(read\_count == 1)  
 down(&db);  
up(&mutex);

Critical Section  
read\_database();

Section Exit  
down(&mutex);  
read\_count--;  
if(read\_count == 0)  
 up(&db);  
up(&mutex);  
process\_data();  
}

# Reader-writer problem – final solution

```
Shared object
semaphore db      = 1;
semaphore mutex   = 1;
int read_count    = 0;
```

Guarantee the mutual exclusion between the readers and the writers.

Protect the “**read\_count**” variable.

Keep track of the number of readers in the system.

# Reader-writer problem – final solution

## Shared object

```
semaphore db    = 1;  
semaphore mutex = 1;  
int read_count = 0;
```

## Writer function

```
1 void writer(void) {  
2   while(TRUE) {  
3     prepare_write();  
4     down(&db);  
5     write_database();  
6     up(&db);  
7   }  
8 }
```

Section Entry

```
prepare_write();  
down(&db);
```

Critical Section

```
write_database();
```

Section Exit

```
up(&db);
```

The writer is allowed to enter its critical section when no other process is in its critical section (protected by the “**db**” semaphore)



# Reader-writer problem – final solution

## Shared object

```
semaphore db    = 1;  
semaphore mutex = 1;  
int read_count  = 0;
```

The first reader “**down()**” the “**db**” semaphore so that no writers would be allowed to enter their critical sections.

The last reader “**up()**” the “**db**” semaphore so as to let the writers to enter their critical section.

## Reader Function

```
1 void reader(void) {  
2     while(TRUE) {  
3         down(&mutex);  
4         read_count++;  
5         if(read_count == 1)  
6             down(&db);  
7         up(&mutex);  
  
8         read_database();  
  
9         down(&mutex);  
10        read_count--;  
11        if(read_count == 0)  
12            up(&db);  
13        up(&mutex);  
14        process_data();  
15    }  
16 }
```

# Reader-writer problem – summary

- This solution does not limit the number of readers and the writers admitted to the system.
  - A realistic database needs this property.
- This solution gives readers a higher priority over the writers.
  - Whenever there are readers, writers must be blocked, not the other way round.
- **What if a writer should be given a higher priority?**

# Summary on IPC problems

- The problems have the following properties in common:
  - Multiple processes;
  - Shared and limited resources;
  - Processes have to be synchronized in order to generate useful output;
- The synchronization algorithms have the following requirements in common:
  - Guarantee mutual exclusion;
  - Uphold the correct synchronization among processes;
  - Deadlock-free.

# Summary on Ch5

