3150 - Operating Systems

Dr. WONG Tsz Yeung

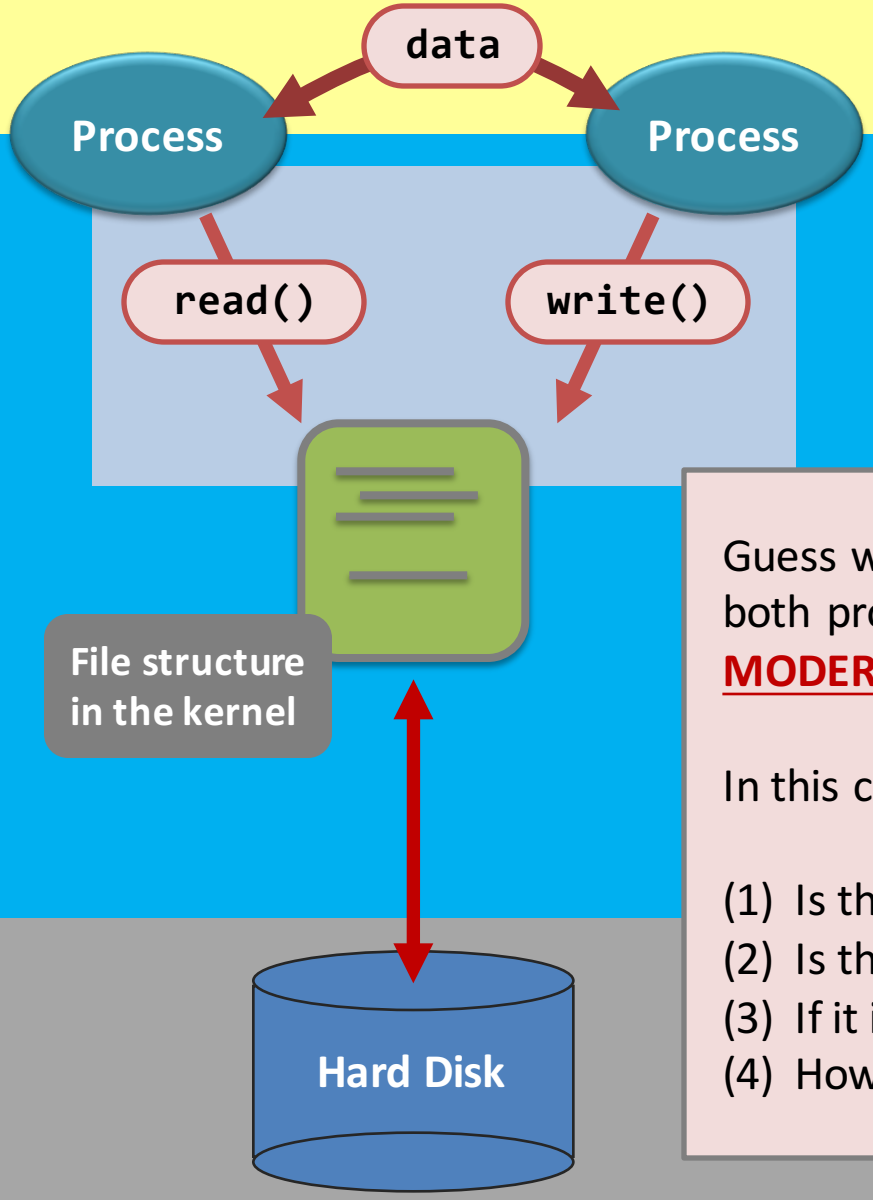
Animation warning.
p.15-19, p.21-24, p.104-112

Don't print them out.

Chapter 2, part 5
Inter-Process Communication (IPC) and Process Synchronization

-things become CRAZY when two guys need to work together…

Outline



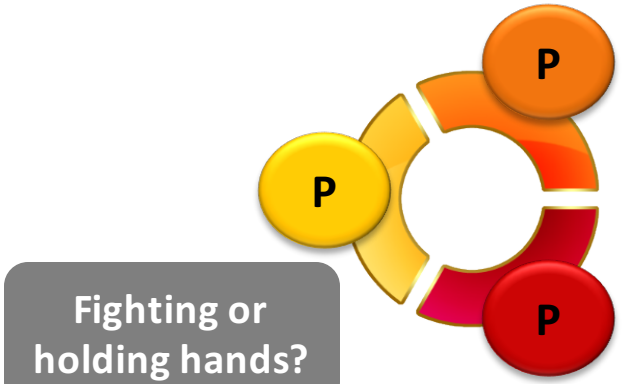
Do you think that the processes can call the two different system calls on the same file simultaneously?

Guess what? The file will be accessed **simultaneously** by both processes **without any coordinations**, under **ALL MODERN OPERATING SYSTEMS**!

In this chapter, we'll learn:

(1) Is the example the instructor's dream or the reality?
(2) Is there any problems?
(3) If it is problematic, what is the big deal?
(4) How can the problem be solved?

Inter-process communication (IPC)
- What, why, and how?



Fighting or holding hands?

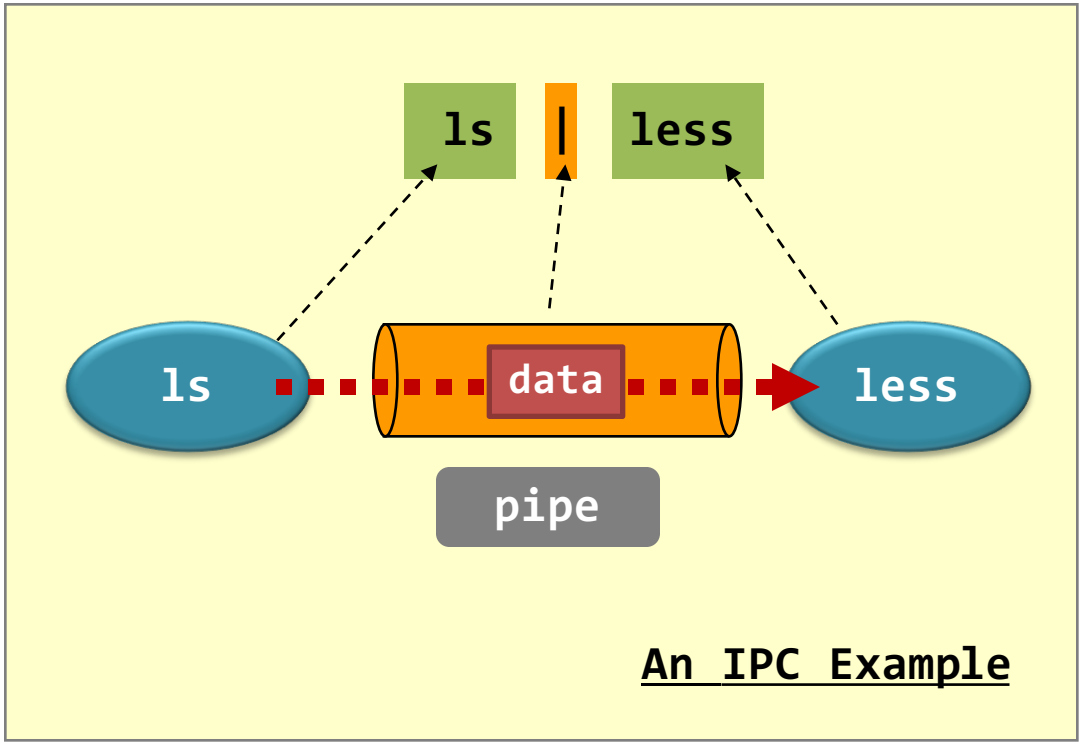
What is IPC?

- You've already tried inter-process communication (IPC) in the lectures: **we used pipe**!

IPC is a paradigm: ways or models that allow processes to communicate.

Pipe is not IPC; pipe is a **shared object** between two processes.

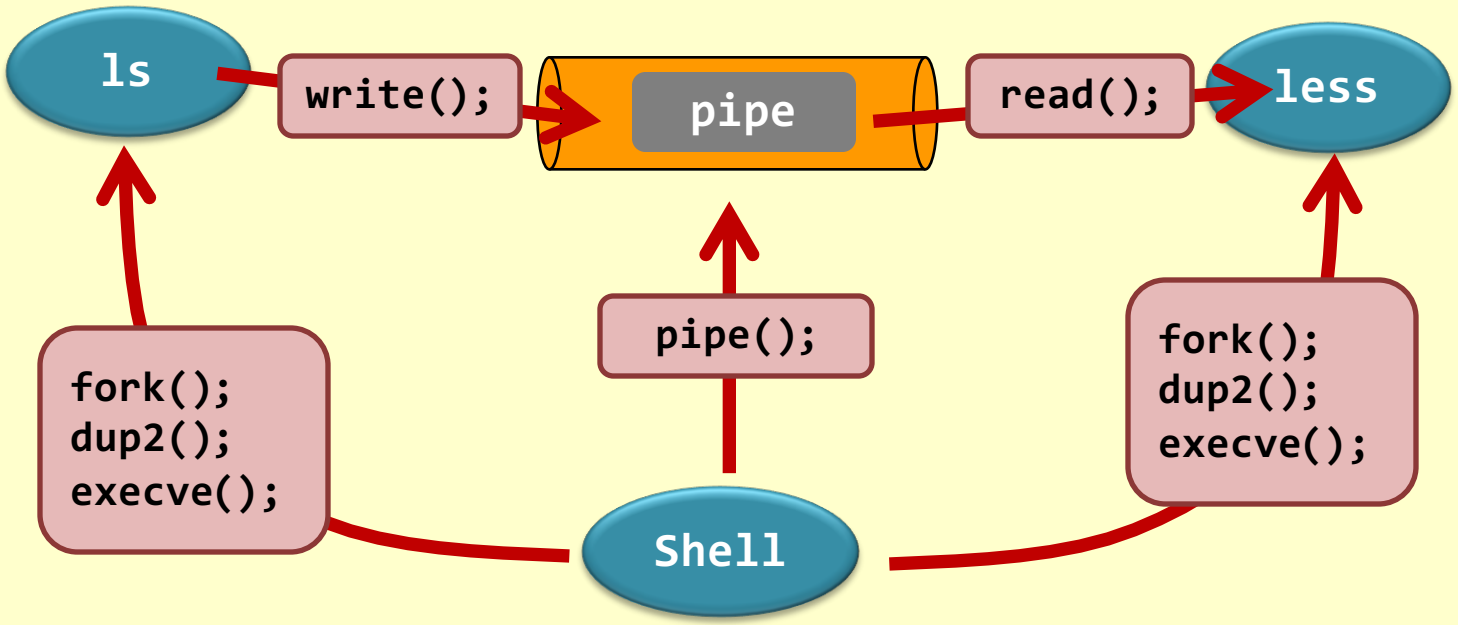
Using pipe is a way to realize IPC.



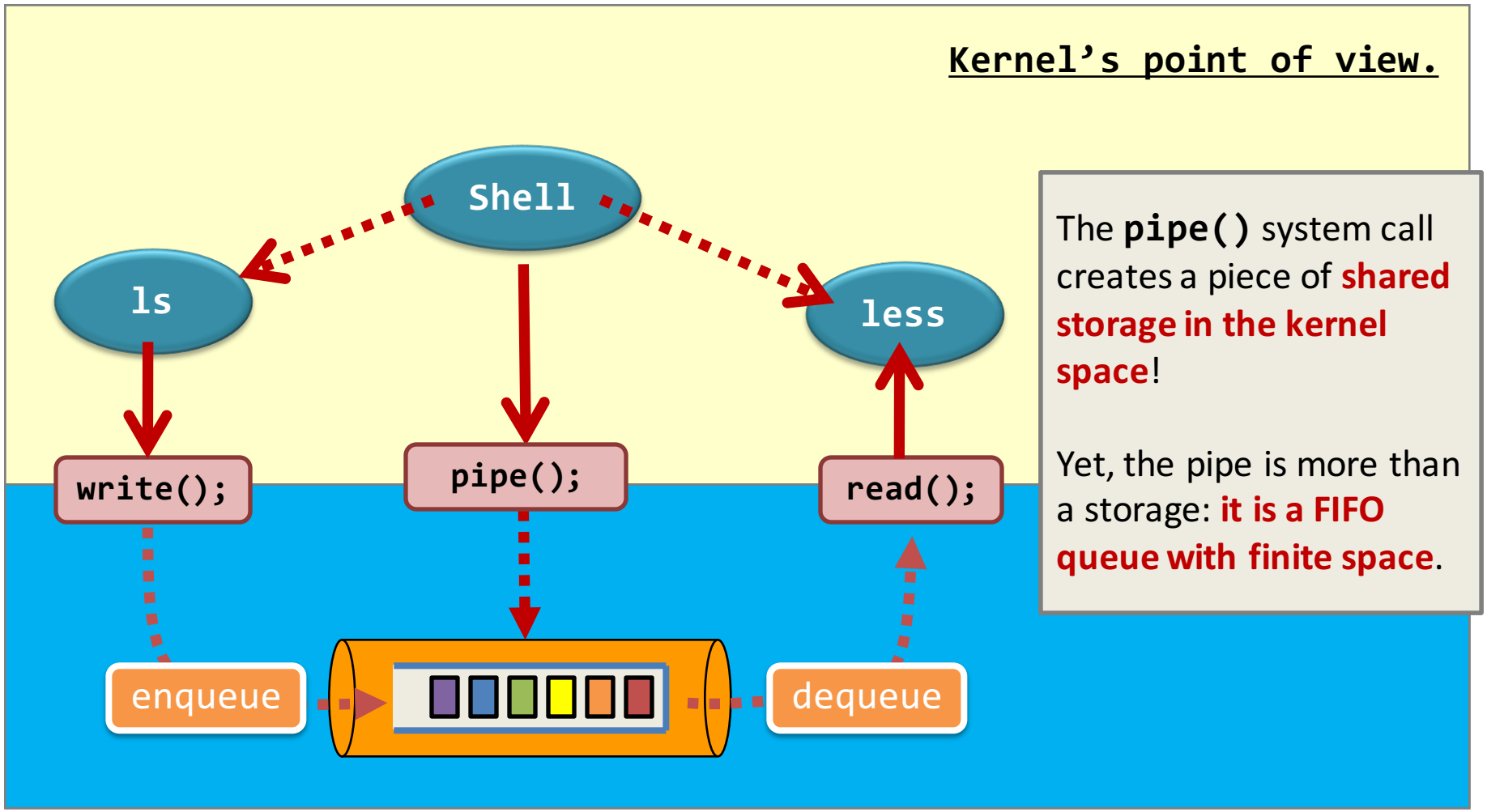
Pipe – your first IPC experience

dup2() is a system call that a file descriptor (or the index to the array of opened file) points to some other opened file, instead of the originally opened file.

Programmer's point of view.



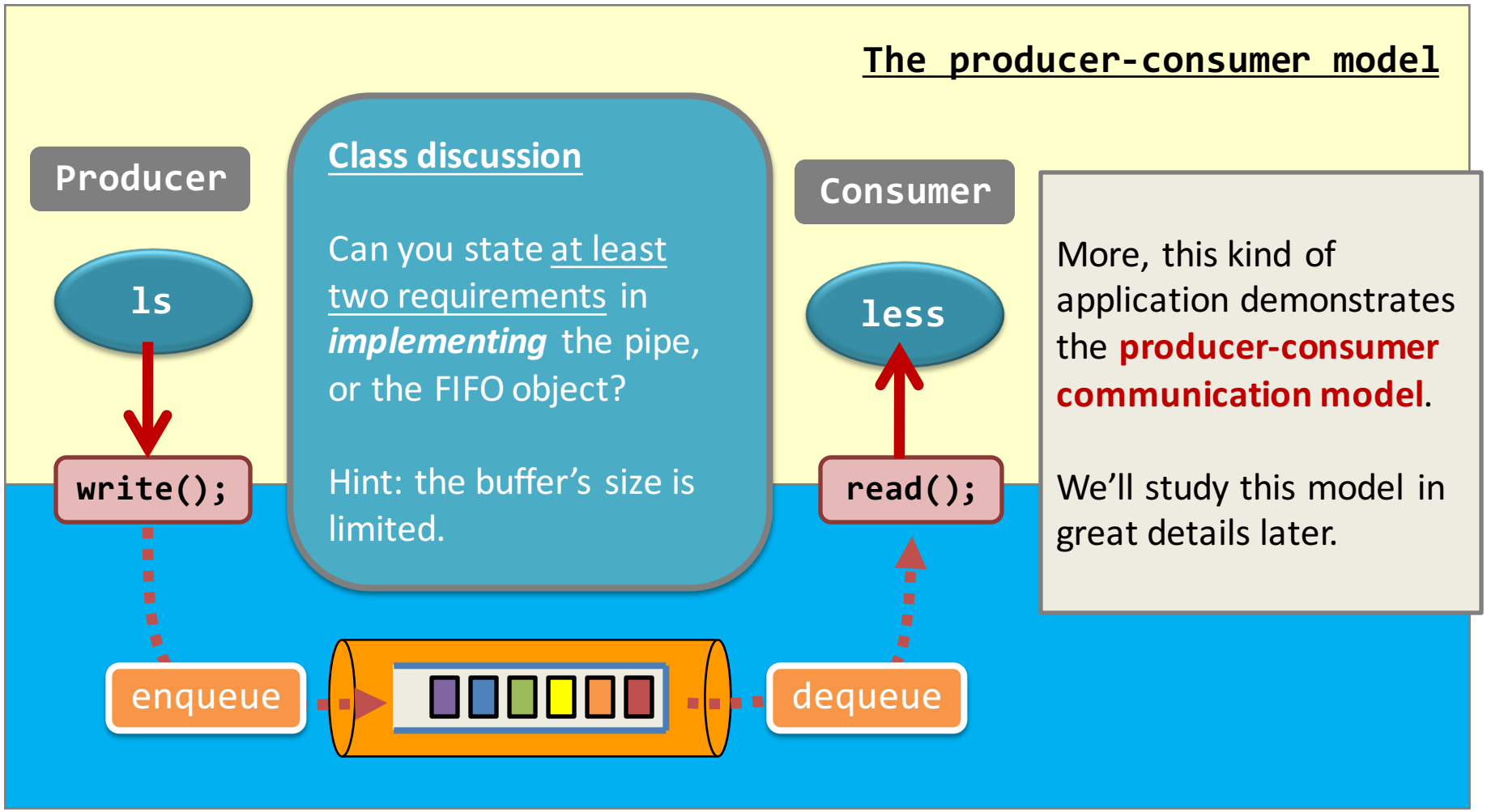
Pipe – your first IPC experience



The **pipe()** system call creates a piece of **shared storage in the kernel space**!

Yet, the pipe is more than a storage: **it is a FIFO queue with finite space**.

Pipe – your first IPC experience

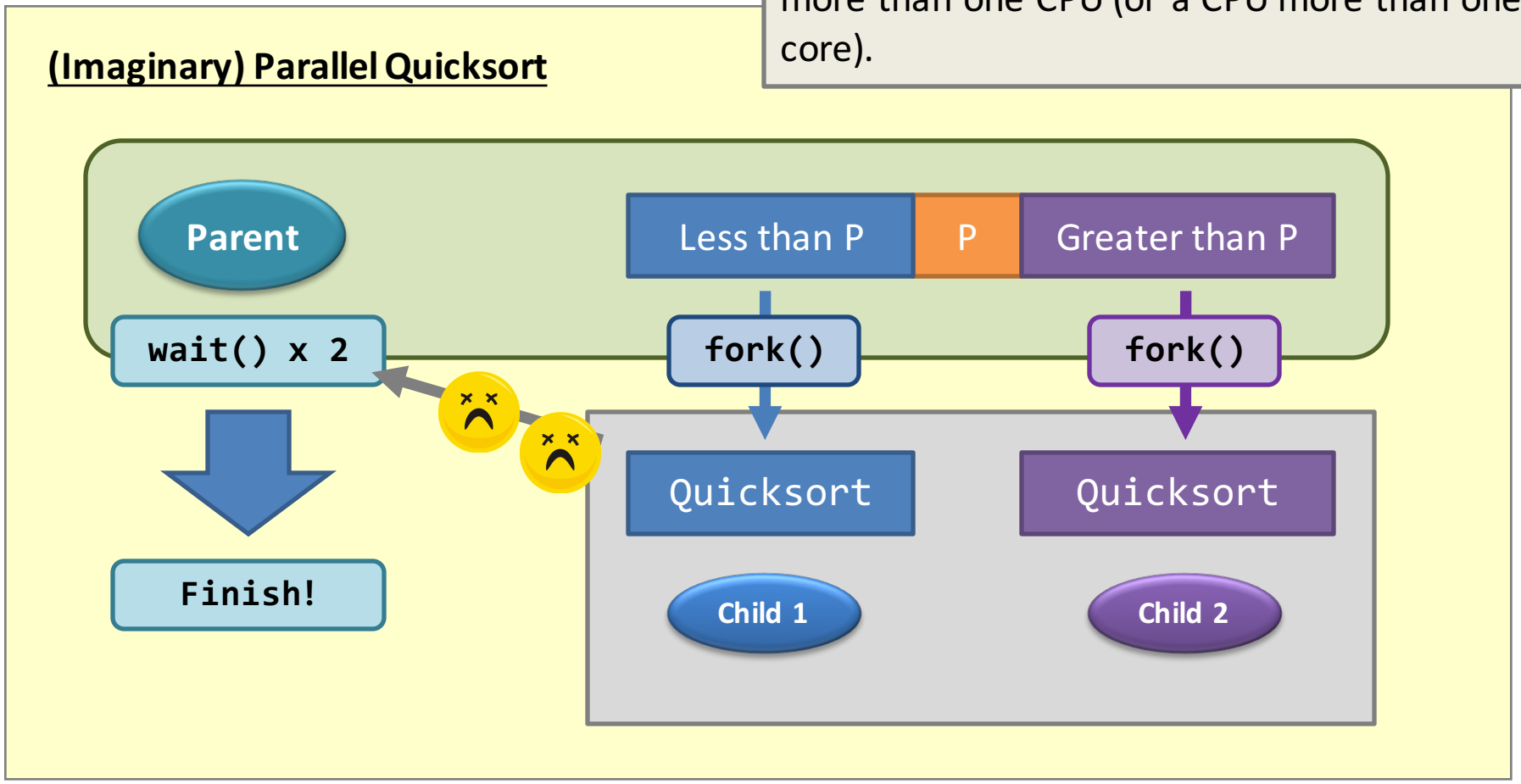


Parallelization – your second experience

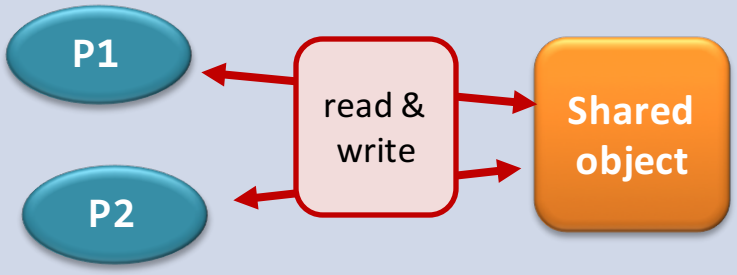
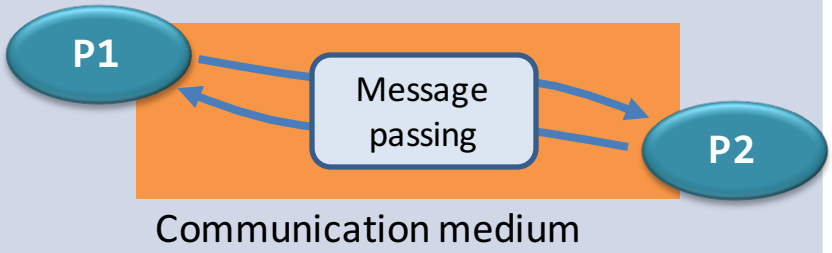
EXTRA

- The **`fork()`** system call is very handy to build parallel programs.

Parallelization is possible only when there are more than one CPU (or a CPU more than one core).

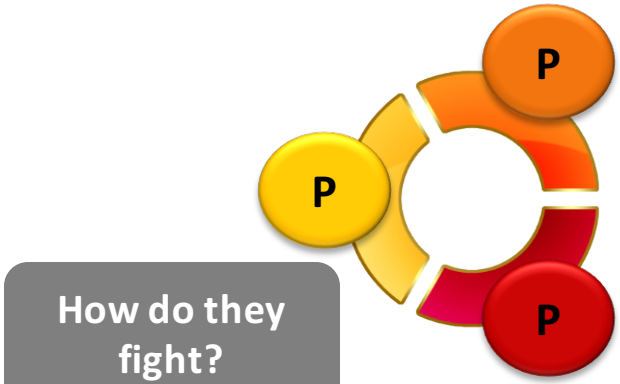


IPC models

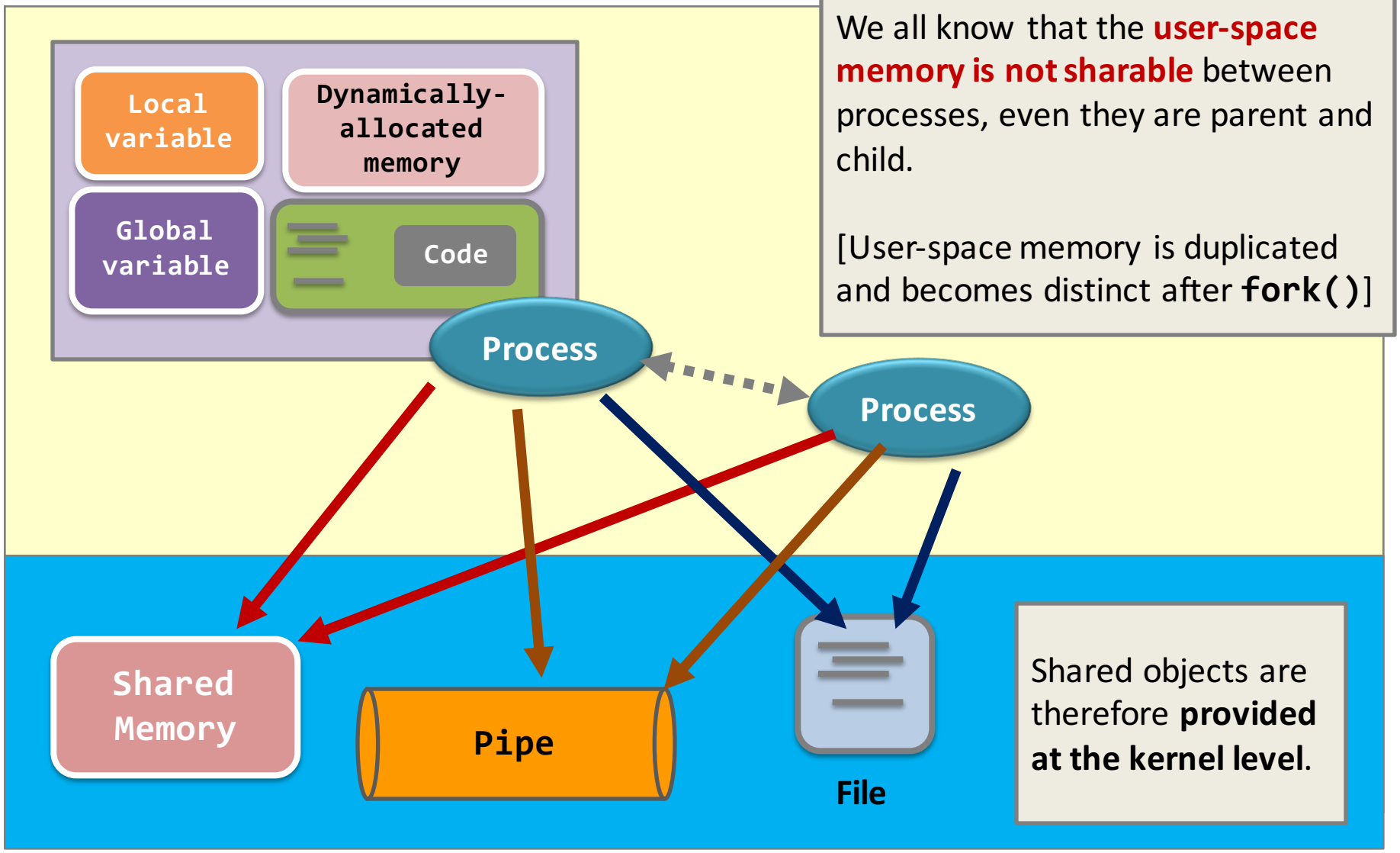
Shared Objects	Message Passing
	
Challenge. Coordination can only be done by detecting the status of the shared object. *E.g., is the pipe empty / full?*	**Challenges.** Coordination relies on the reliability and the efficiency of the communication medium (and protocol).
E.g., pipes, shared memory, and regular files. This is the **focus of this part** because you'll find a lot of real-life examples.	E.g., socket programming (in CSCI4430), message passing interface (MPI) library for computing clusters.

Inter-process communication (IPC)

- What, why, and how?
- The problem: race condition.



Evil source: the shared objects.



We all know that the **user-space memory is not sharable** between processes, even they are parent and child.

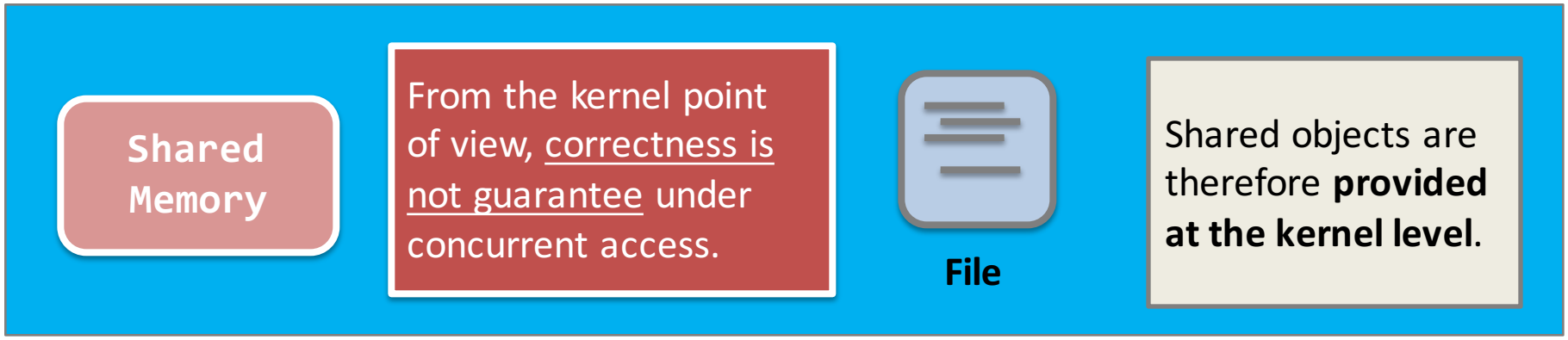
[User-space memory is duplicated and becomes distinct after **`fork()`**]

Shared objects are therefore **provided at the kernel level**.

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…

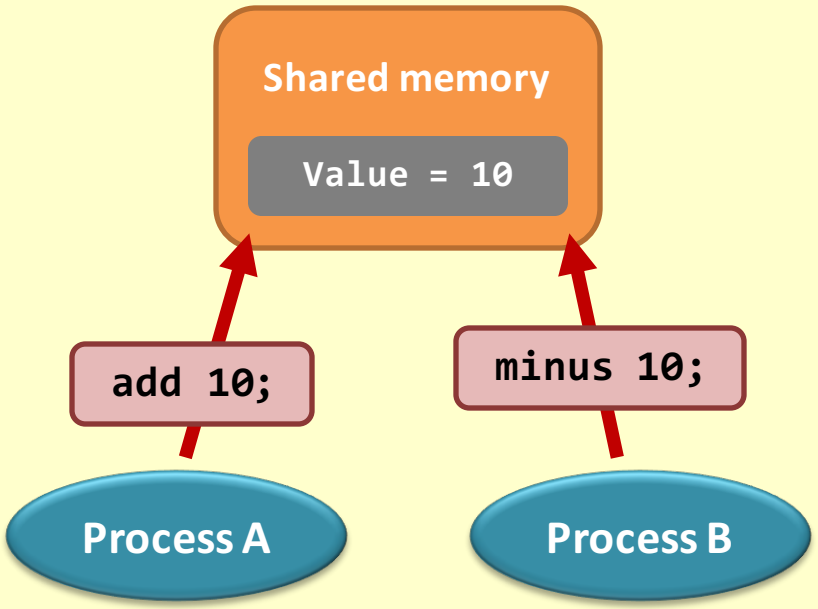
High-level language for Program A

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

Partial low-level language for Program A

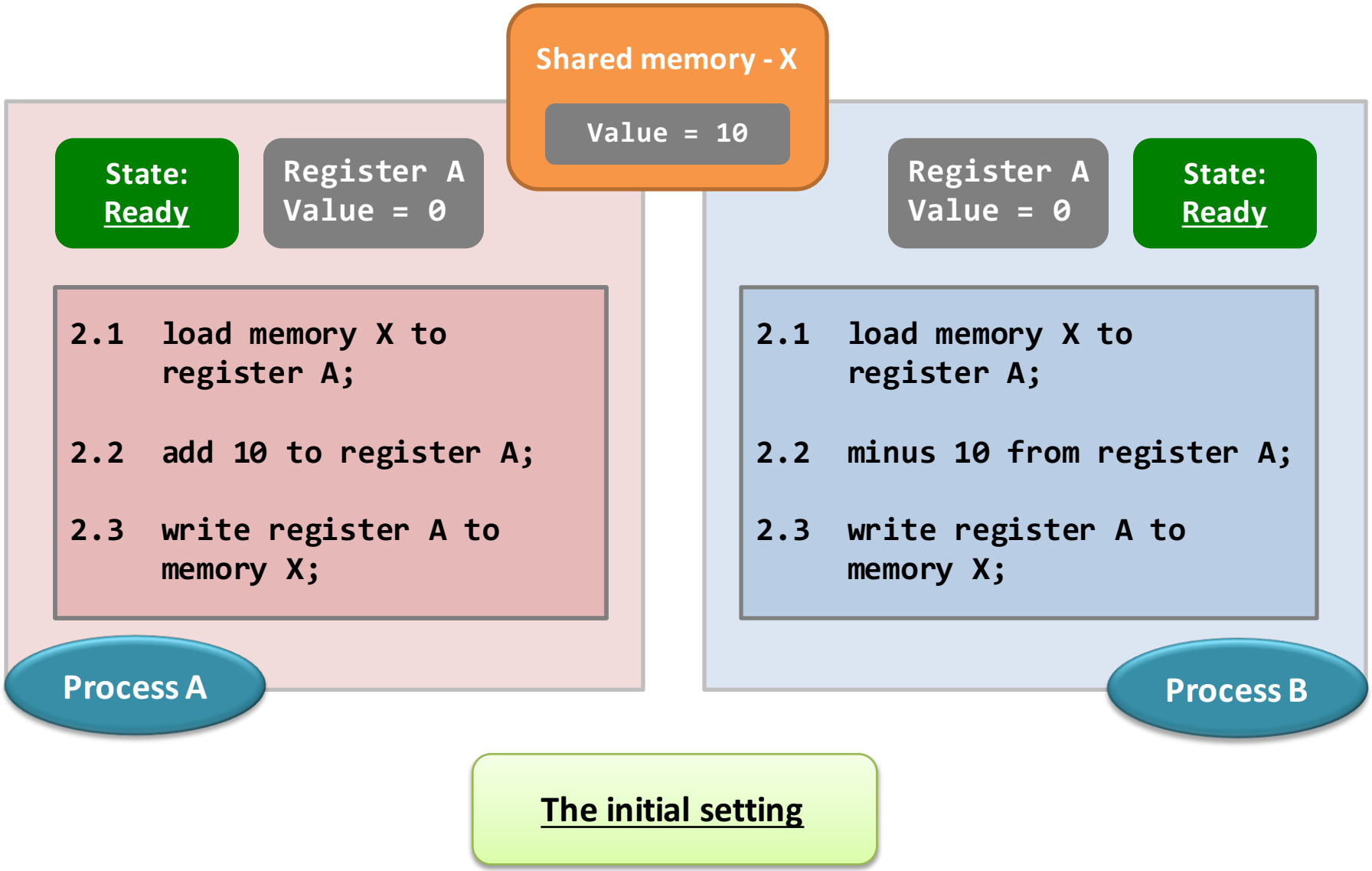
```
1     attach to the shared memory X;
......
2.1   load memory X to register A;
2.2   add 10 to register A;
2.3   write register A to memory X;
......
3     exit;
```

The Scenario



Guess what? This code block is evil!

Problem arise…



Problem not yet arise…

Don't print

Shared memory - X

Value = 10

State: Running

Register A Value = 10

Register A Value = 0

State: Ready

1

```
2.1  load memory X to
     register A;

2.2  add 10 to register A;

2.3  write register A to
     memory X;
```

```
2.1  load memory X to
     register A;

2.2  minus 10 from register A;

2.3  write register A to
     memory X;
```

Process A

Process B

Execution Flow #1, Step 1

Problem not yet arise…

Don't print

Shared memory - X

Value = 10

State: Running

Register A Value = 20

1 2.1 load memory X to register A;

2 2.2 add 10 to register A;

2.3 write register A to memory X;

Process A

Register A Value = 0

State: Ready

2.1 load memory X to register A;

2.2 minus 10 from register A;

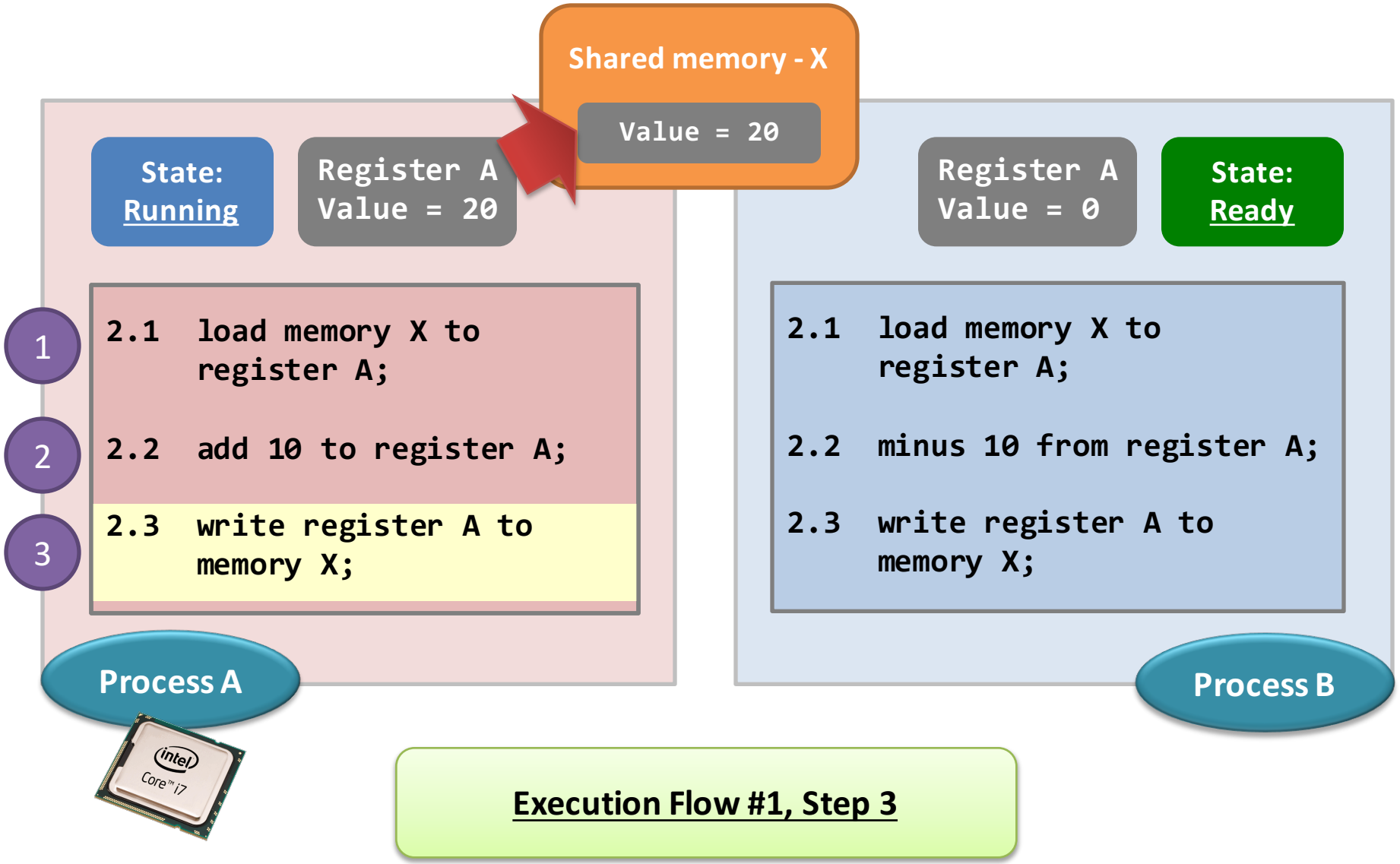
2.3 write register A to memory X;

Process B

Execution Flow #1, Step 2

Problem not yet arise…

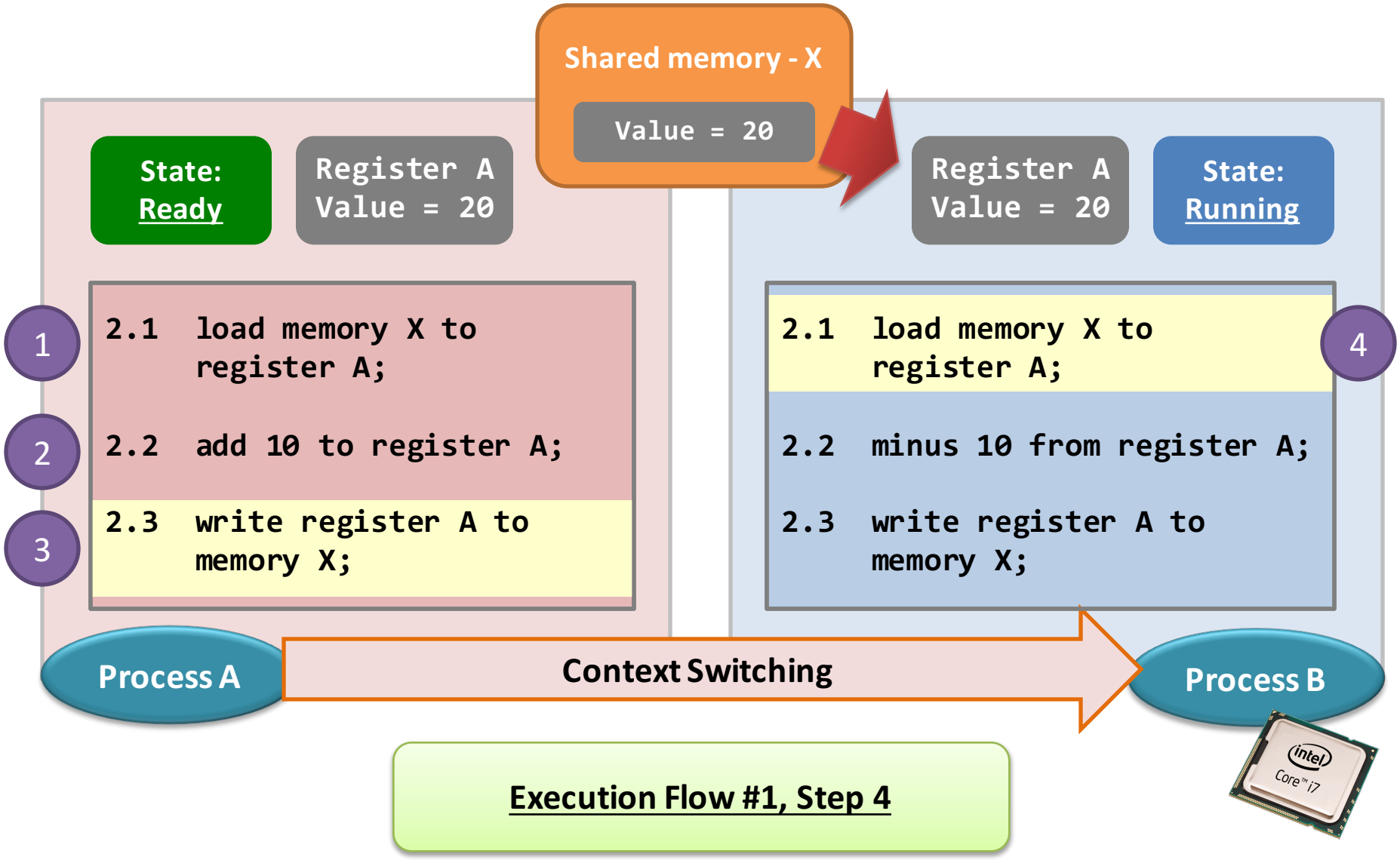
Don't print



Execution Flow #1, Step 3

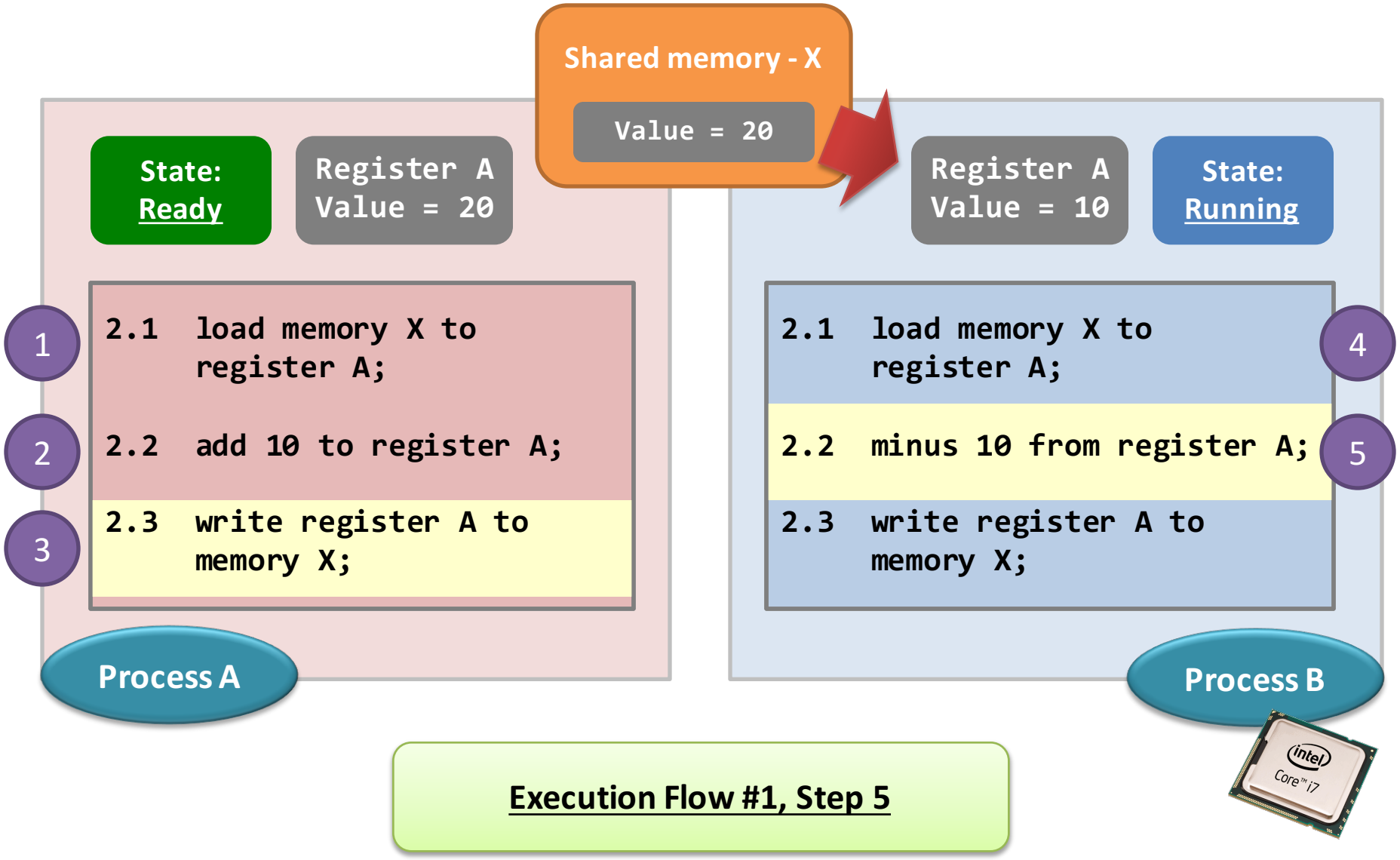
Problem not yet arise…

Don't print

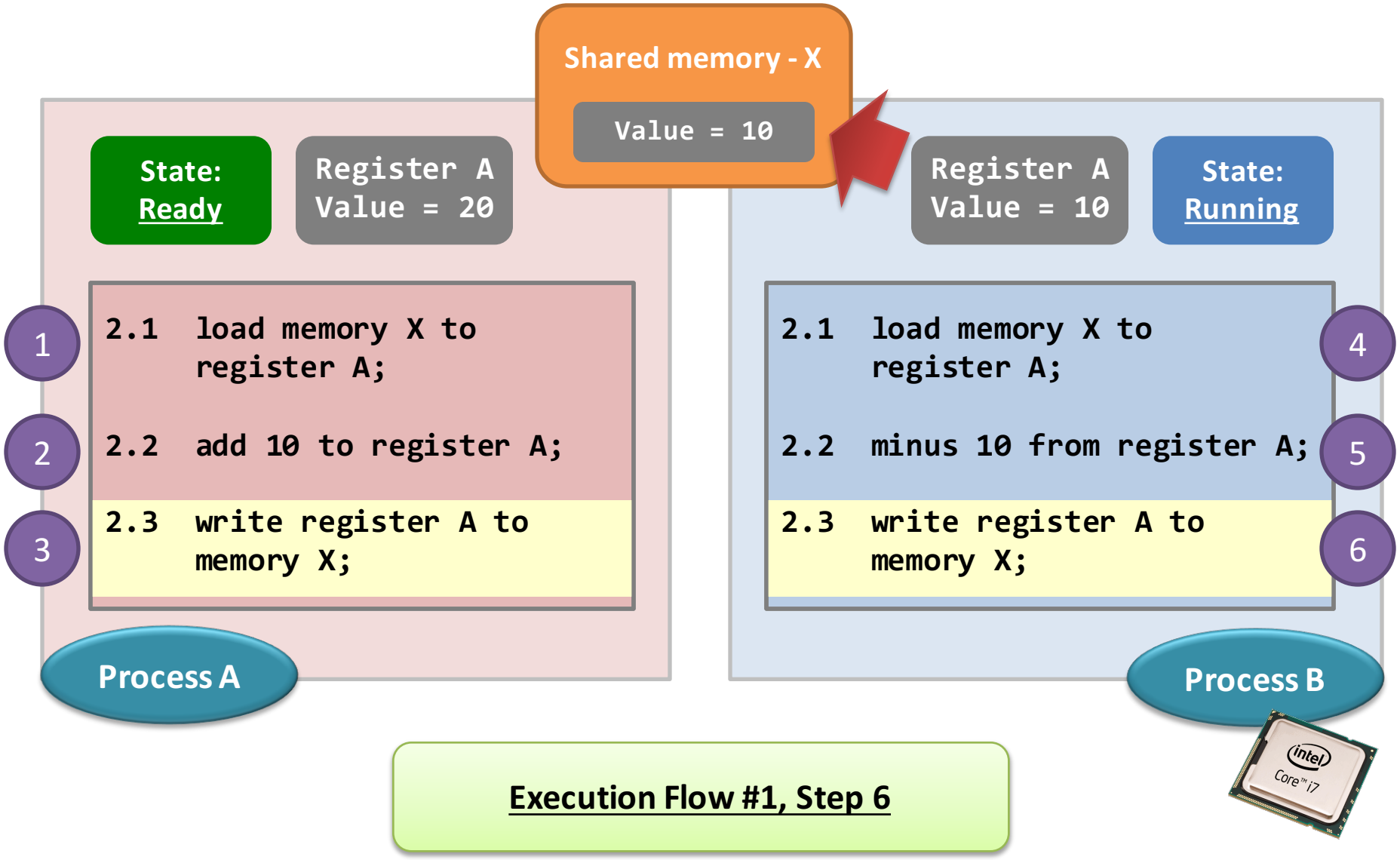


Problem not yet arise…

Don't print

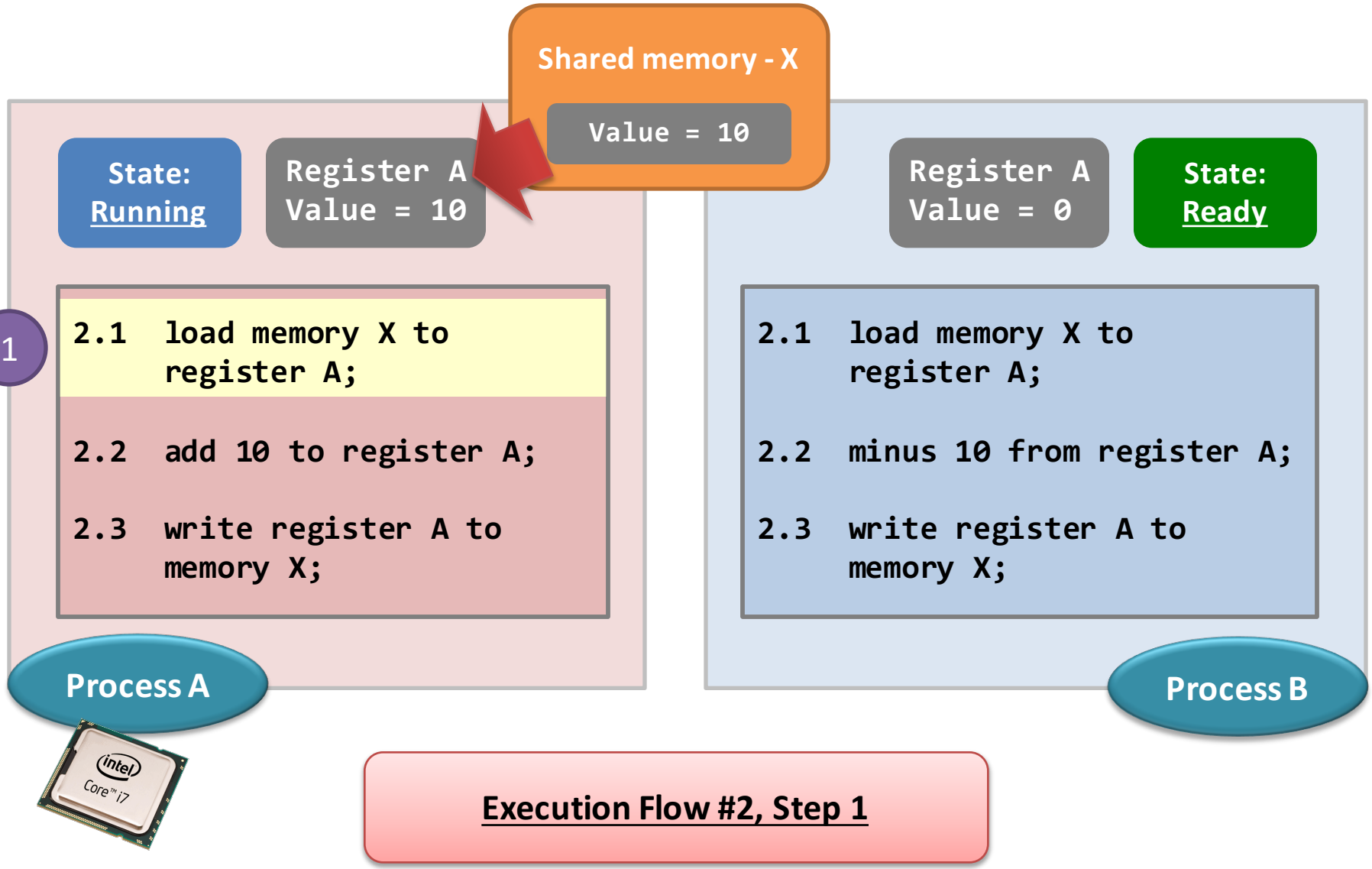


Problem not yet arise…



Problem arise…

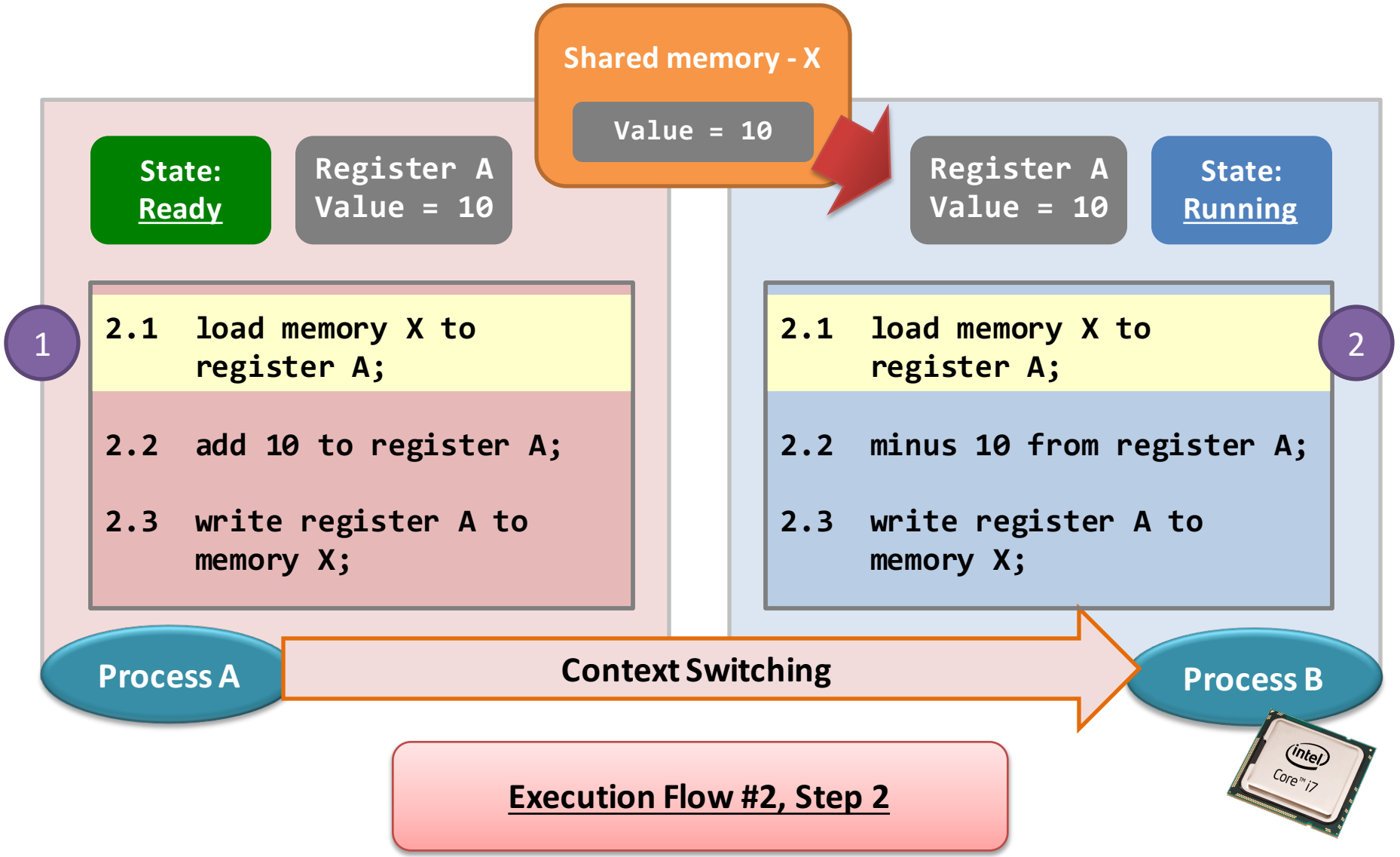
Don't print



Execution Flow #2, Step 1

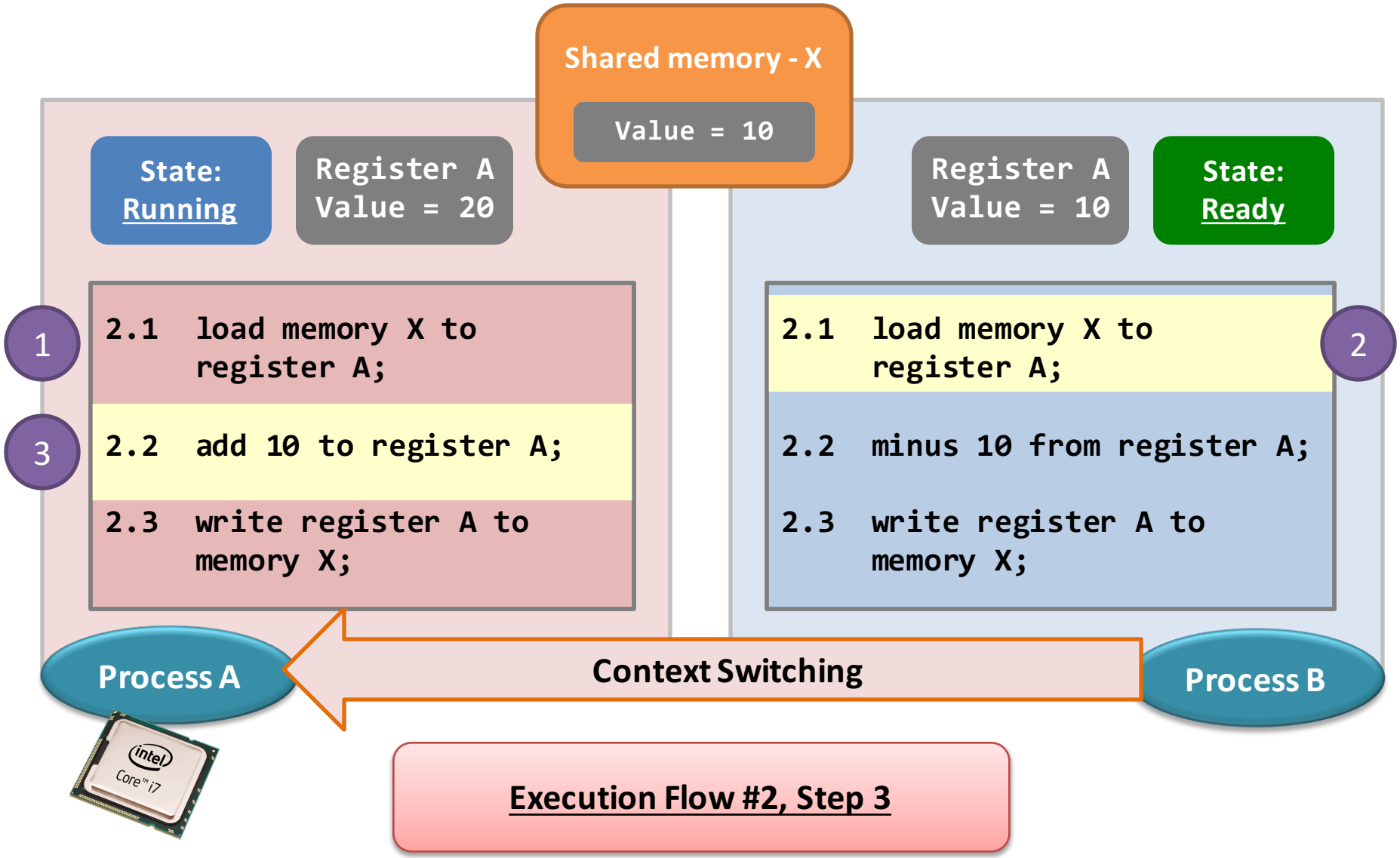
Problem arise…

Don't print



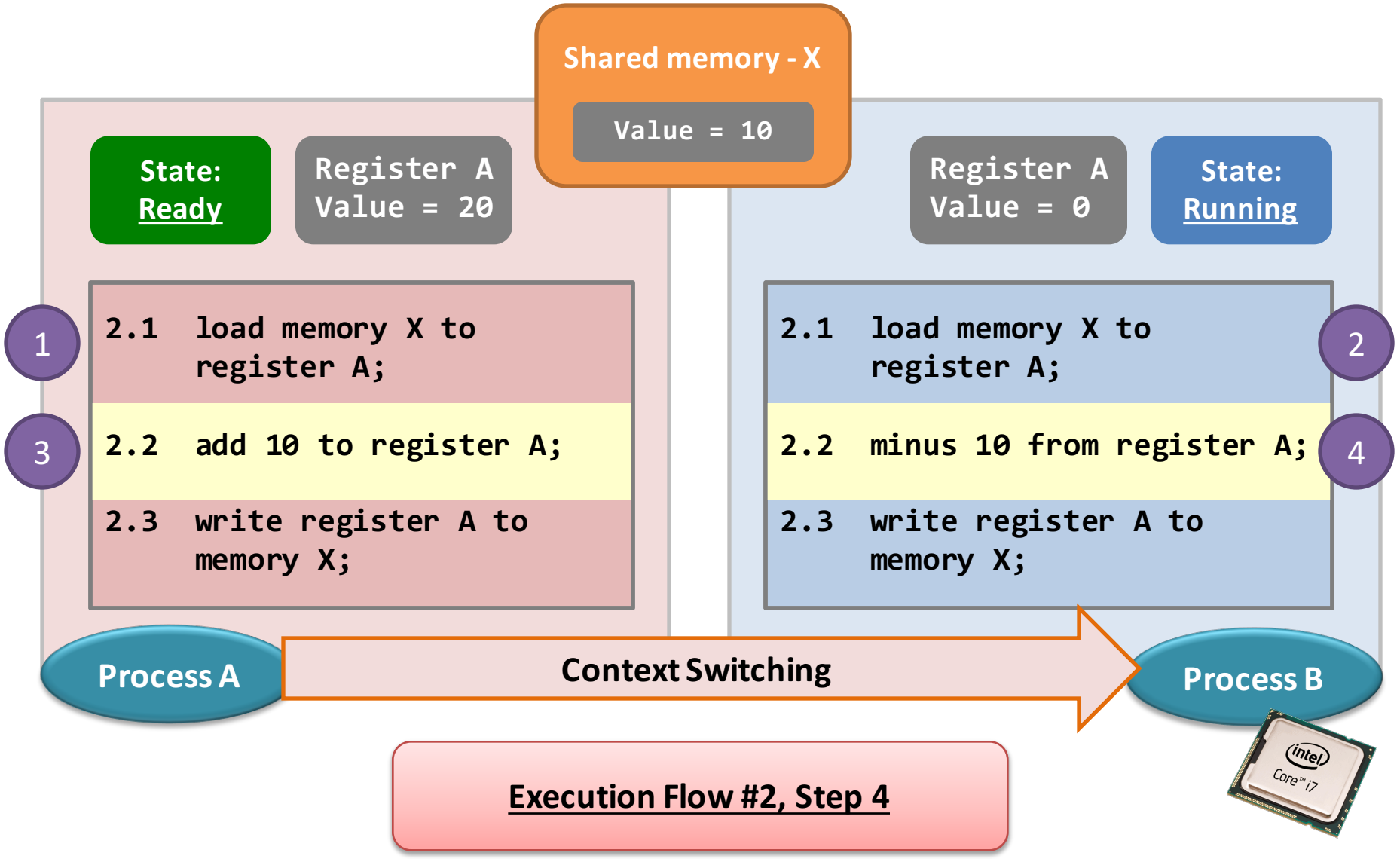
Problem arise…

Don't print

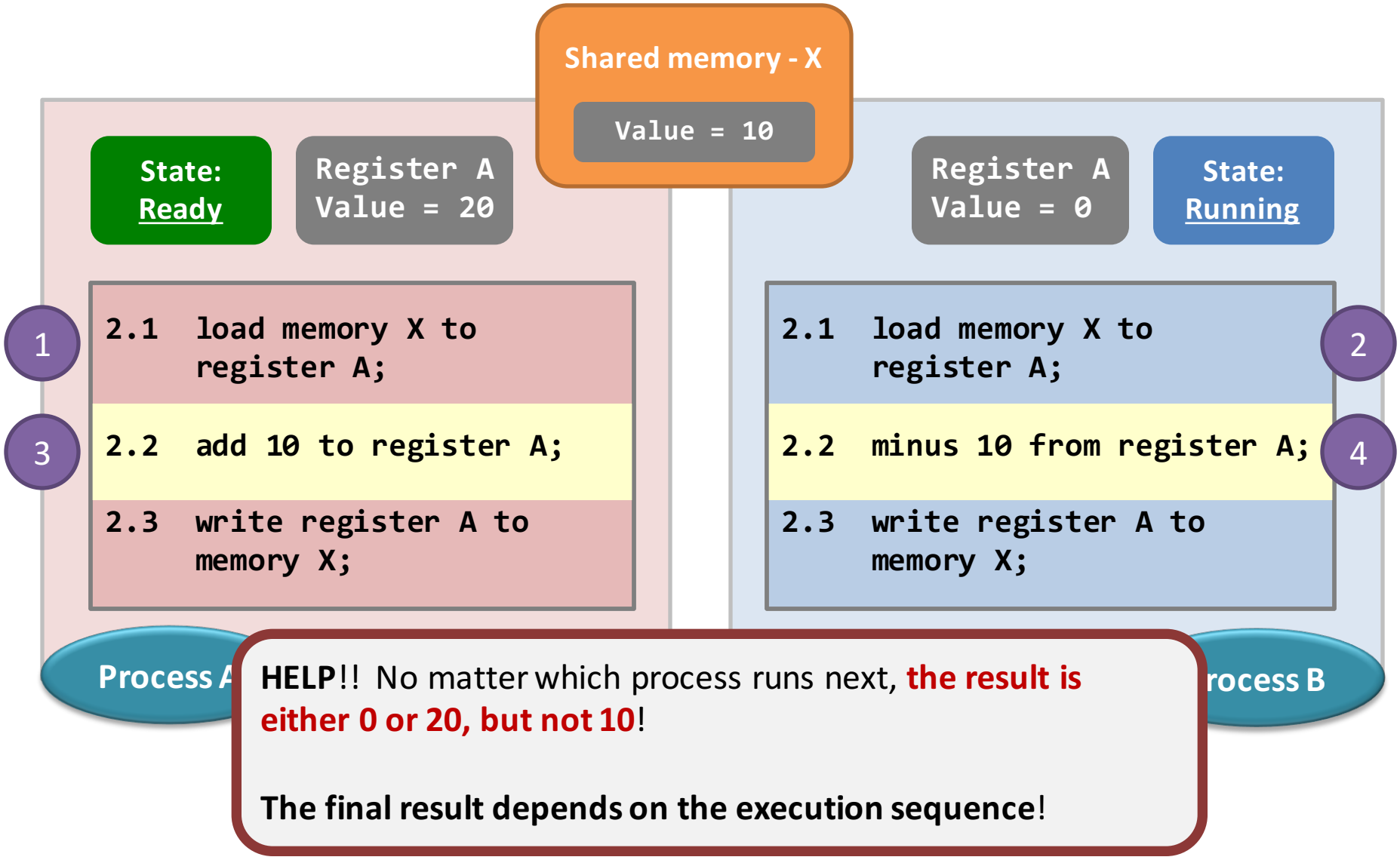


Problem arise…

Don't print



Problem arise…



Race condition – the curse

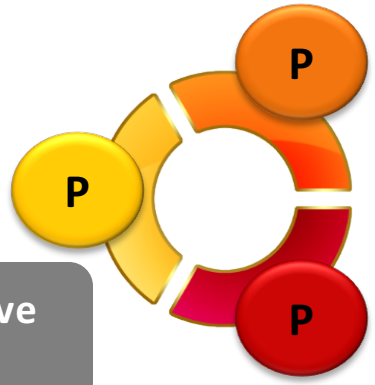
- The above scenario is called the **race condition**.

- 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.

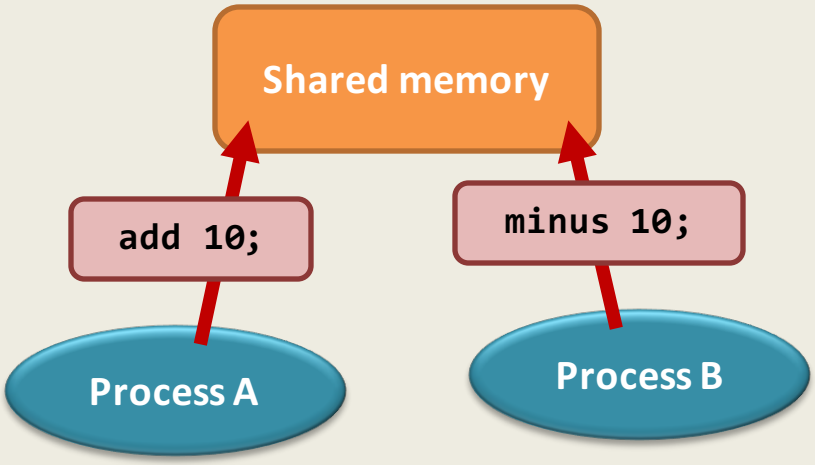
Inter-process communication (IPC)

- What, why, and how?
- The problem: race condition.
- **Mutual exclusion:**
  - **how to achieve?**



How to have peace?

Mutual Exclusion – the cure



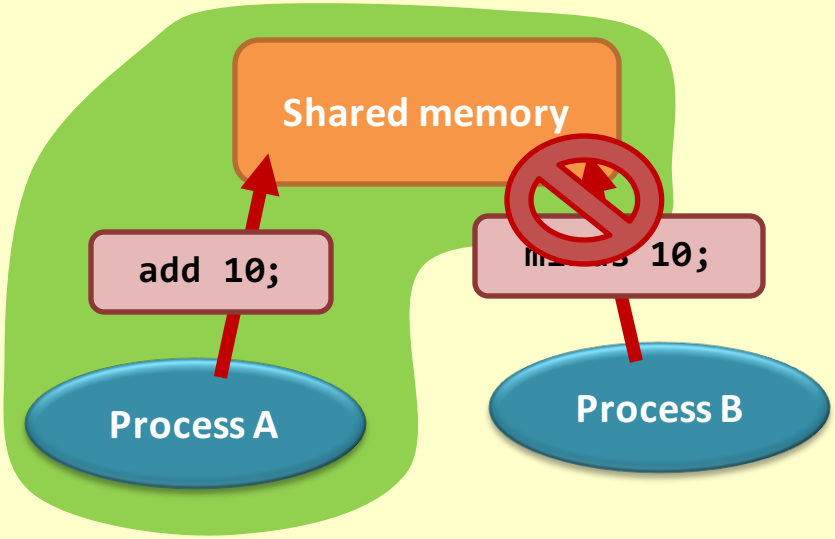
Two processes playing with **the same shared memory** is dangerous.

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

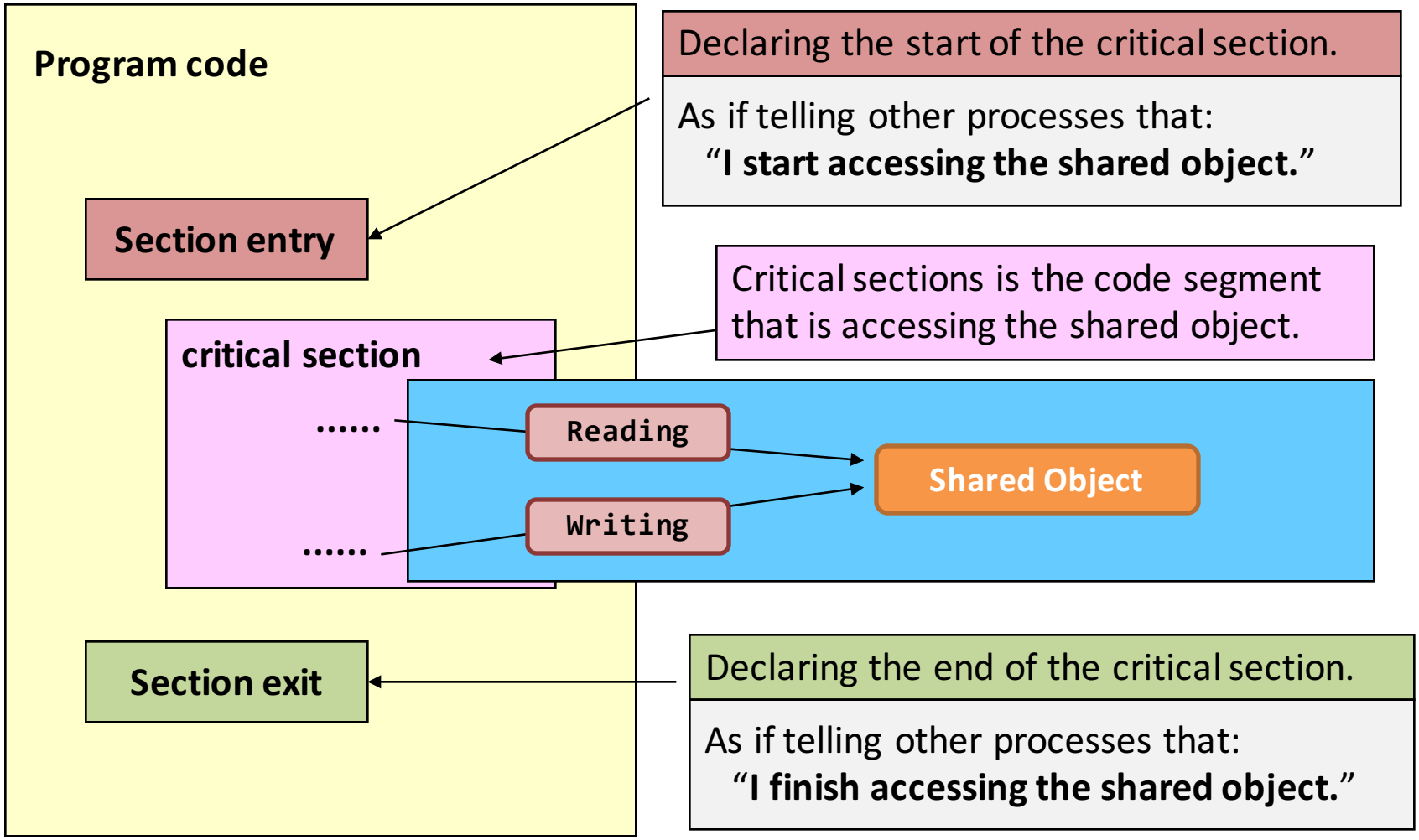
The solution can be simple:

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

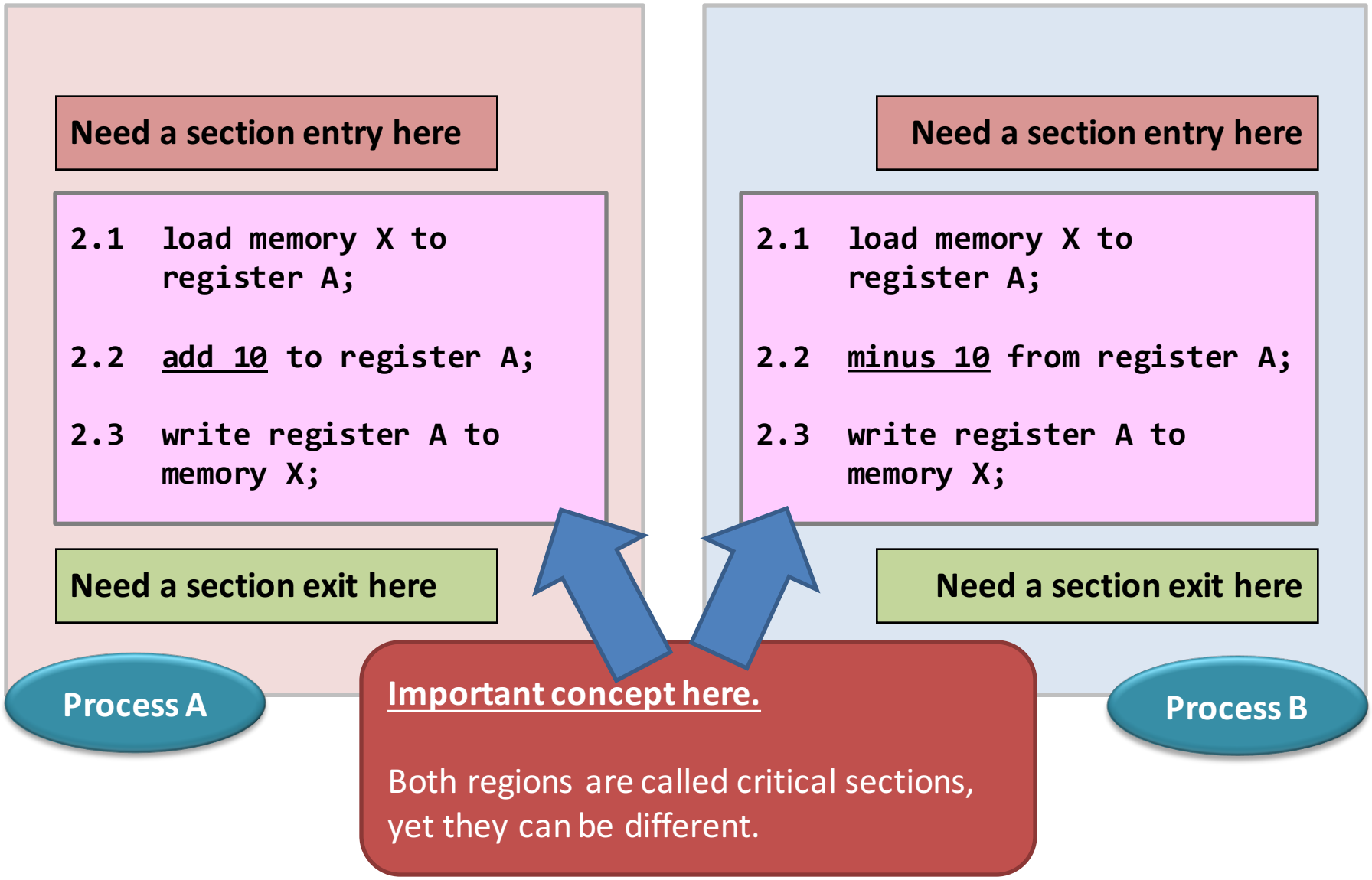
This is called **mutual exclusion**.
A set of processes would not have the problem of race condition *if mutual exclusion is guaranteed*.



Critical Section – the realization



Critical Section – the realization



In case you still don't understand…

Shared Object

Certainly, you want mutual exclusion!

Set of blocked processes, waiting to proceed.

Section Entry

Section Exit

Of course, the critical sections **are different** from process to process.

Before going into the critical section, a process turns this lock to "**occupied**".

Before leaving the critical section, a process turns this lock to "**vacant**".

Implication: This lock is actually tell other processes what to do.

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.

Summary…for the content so far…

- **Defining critical sections** is an implementation.
  - 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**.

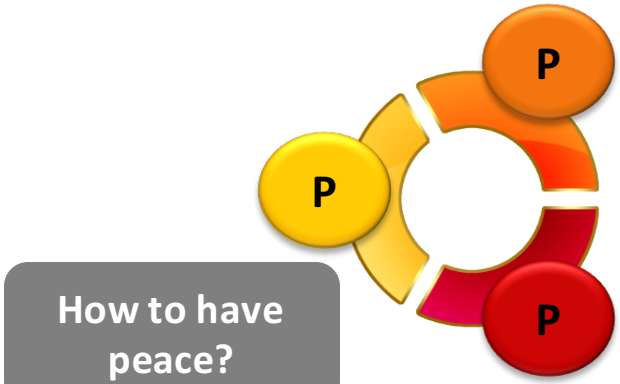
Summary…for the content so far…

- **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.
  - Recall the toilet example: *what if the lock is broken*?

- **Mutual exclusion hinder the performance of parallel computations.**

Inter-process communication (IPC)

- What, why, and how?
- The problem: race condition.
- **Mutual exclusion:**
  - how to achieve?
  - **how to implement?**



Entry and exit implementation - requirements

- **Requirement #1**. 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**. No assumptions should be made about the speeds and the numbers 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**. No process running outside its critical section should block other processes.

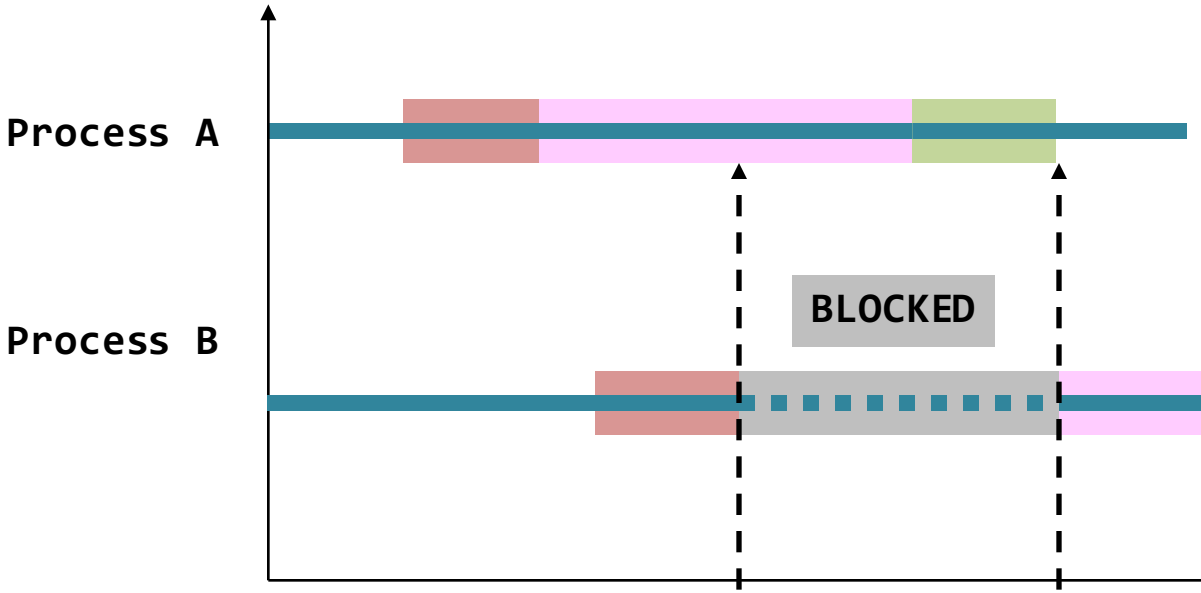
> **Implication**: otherwise, it may end up with a scenario that **all the processes were blocked** and no process would be inside the critical section.

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

> **Implication**: 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)

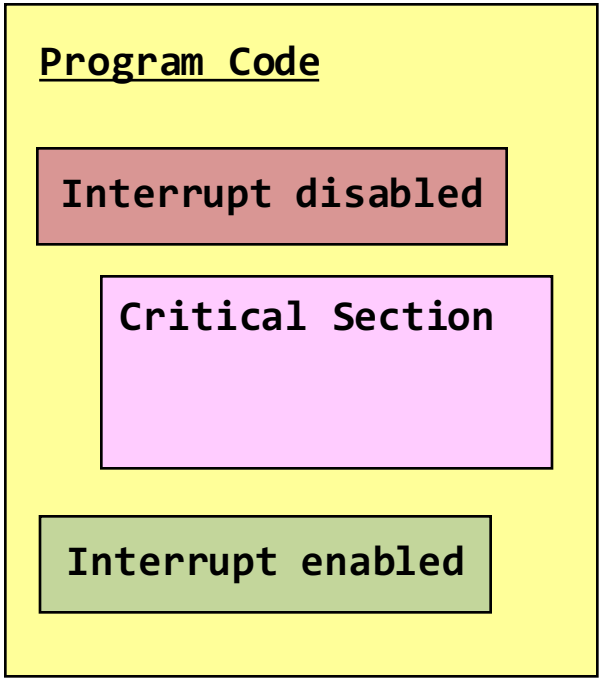
We will be using this coloring scheme throughout this part.

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

A leaves its critical section and B resumes execution accordingly.

Proposal #1 – disabling interrupt.

- **Aim**
  - 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 is not an *attractive* solution.
  - ***Isn't it a terrible solution*** that allows user programs to disable or enable interrupts freely?
  - What if there are careless programmers?



Proposal #2: the “lock” shared object.

- **Aim.**
  - Using a new shared object to detect the status of other processes.
- **Pros.**
  - Simple to implement.
  - Language independent;
  - Can work on any platform.
- **Cons.**
  - Waste CPU resource with the busy waiting implementation.
- **Correctness?**

FAILED!

Note that: all processes run the following same code.

Program Code

```
1  while( lock == 1 )
2    ; /* busy waiting */
3  lock = 1;
```

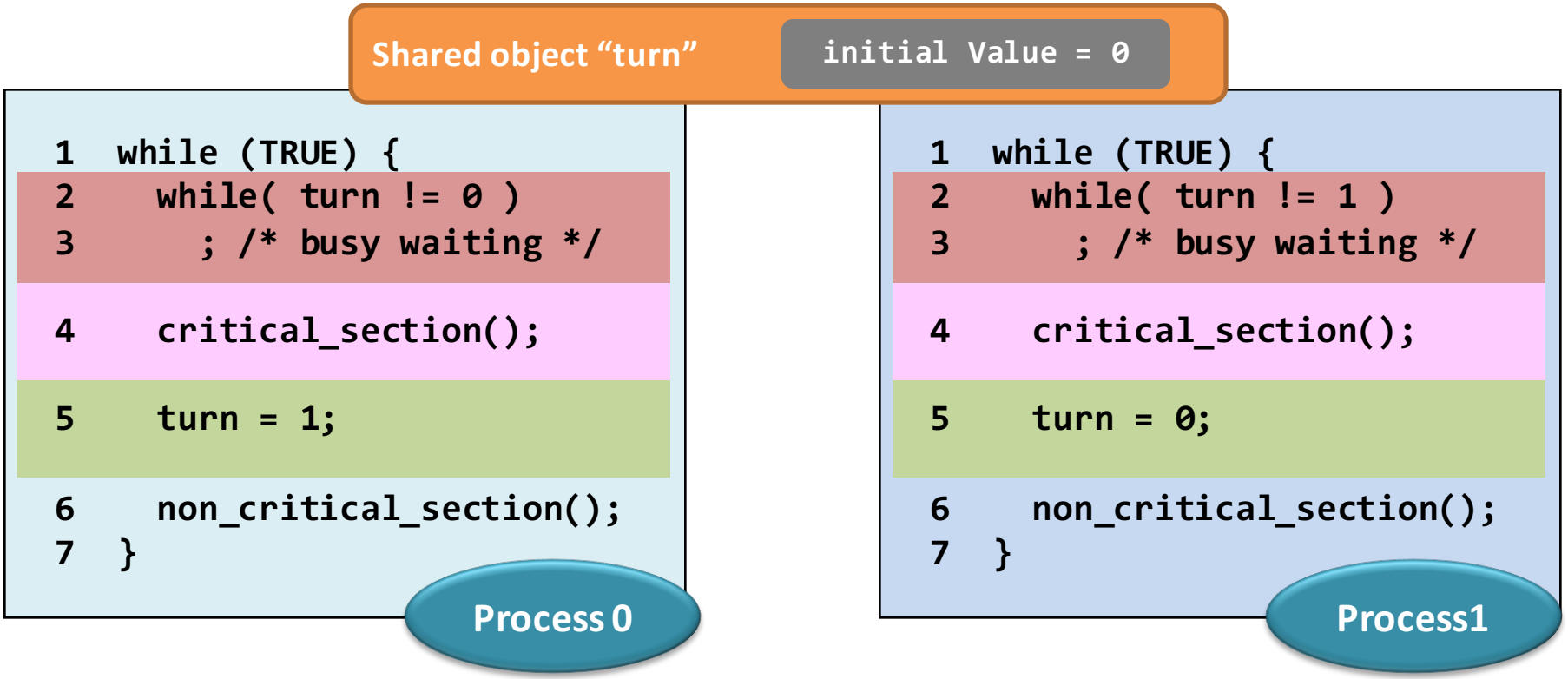
```
Critical Section
```

```
lock = 0;
```

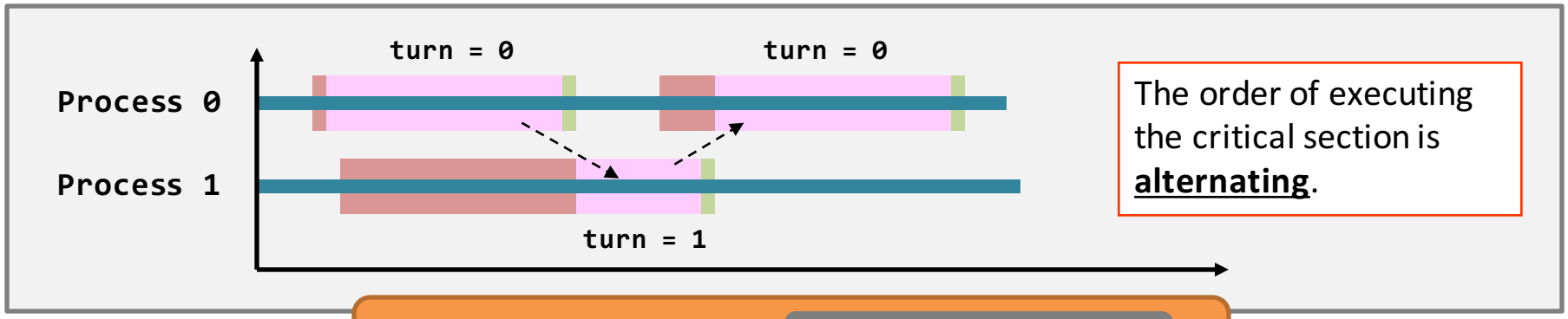
The reason is left as a class discussion.

Proposal #3: Strict alternation

- **Aim.**
  - Using a new shared object to detect the status of other processes (again?!)



Proposal #3: Strict alternation



The order of executing the critical section is **alternating**.

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!**
  - Hence, violating Requirement 3 of being a good mutual exclusion solution.

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

Proposal #4: Peterson's solution

- (The code of the Peterson's solution will occupy a whole page, so I put the thing I want to say early)

- The Peterson's solution is an improved version of the strict alternation proposal.

- Highlights:
  - No alternation is there;
  - Processes would act as a gentleman: if you want to enter, I'll let you first.

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 = process;
 9    while ( turn == process &&
               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  }
```

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 = process;
 9    while ( turn == process &&
              interested[other] == TRUE )
10      ;    /* busy waiting */
11  }
12
13  void leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

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

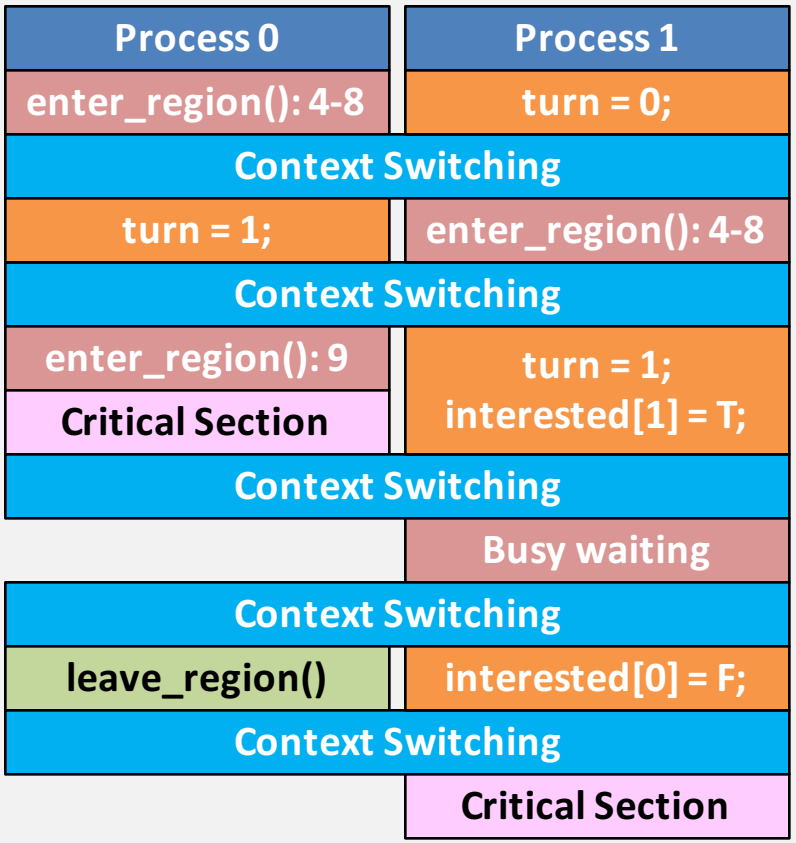
Line 8 therefore try to make herself the one to run.

"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

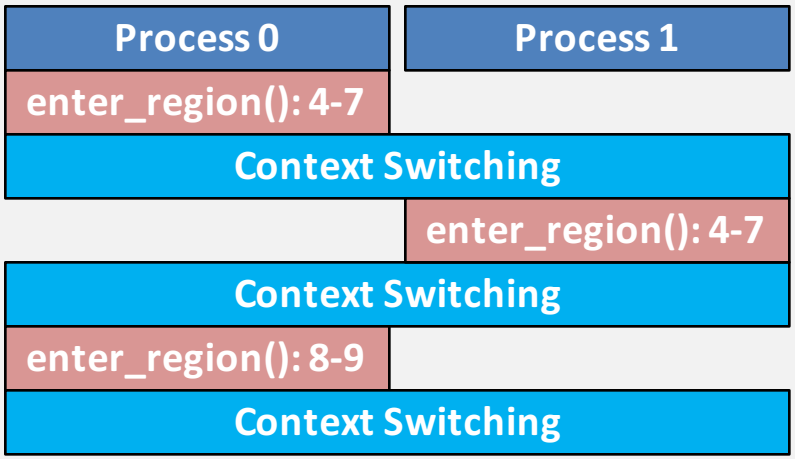
```
 1  int turn;
 2  int interested[2] = {FALSE,FALSE};
 3
 4  void enter_region( int process ) {
 5    int other;
 6    other = 1-process;
 7    interested[process] = TRUE;
 8    turn = process;
 9    while ( turn == process &&
               interested[other] == TRUE )
10      ;    /* busy waiting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



and the story goes on…

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 = process;
 9    while ( turn == process &&
               interested[other] == TRUE )
10      ;    /* busy waiting */
11  }
12
13  void leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

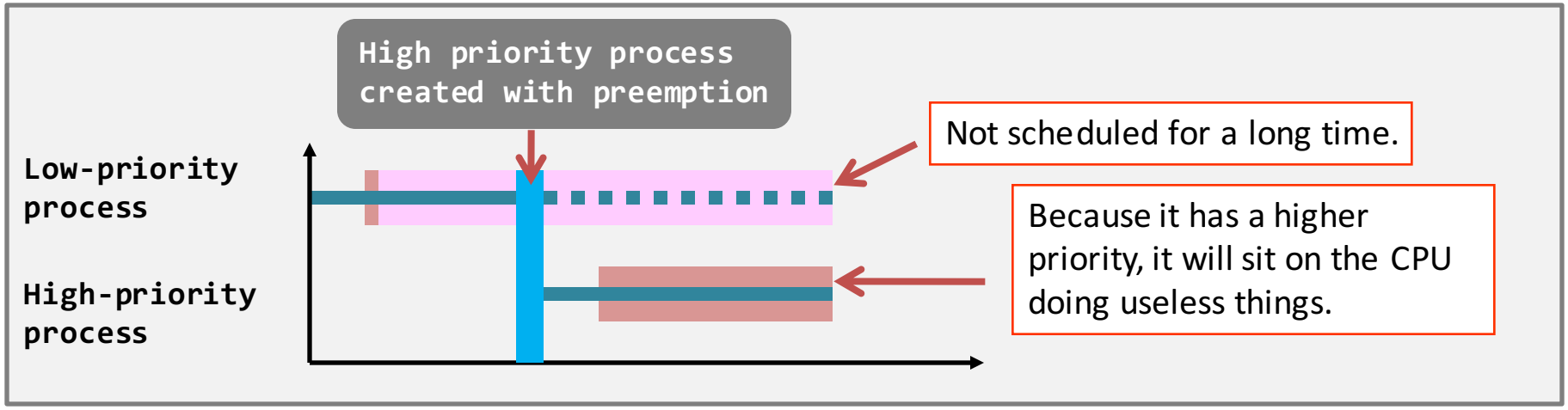


Can you complete the flow?

Can both processes progress?

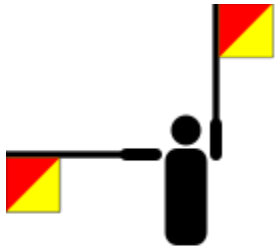
Proposal #4: Peterson's solution – issues

- 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.



Final proposal: Semaphore very important

- 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.



source: wikipedia.

- 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 – the code

Data Type definition

```
typedef int semaphore;
```

semaphore up down c

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  }
```

The calls "**up()**" and "**down()**" are implemented inside the kernel.

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  }
```

Final proposal: Semaphore – details

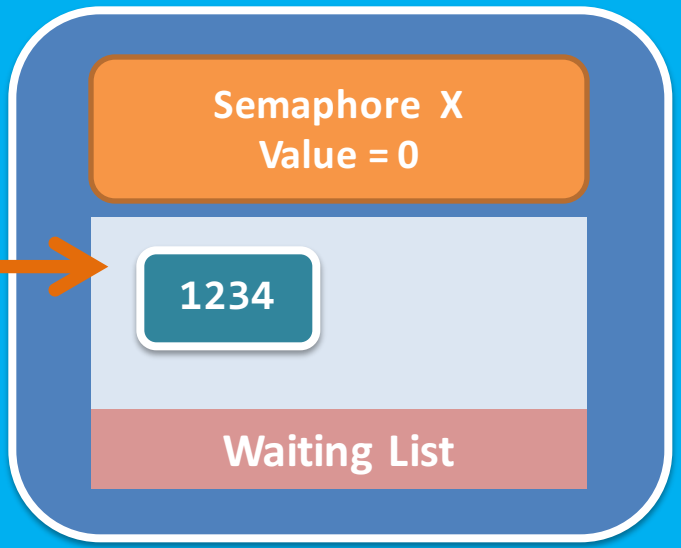
Keep in mind that **every shared object must be allocated from and be kept in the kernel space** such that every process is able to attach to.

Process 1234

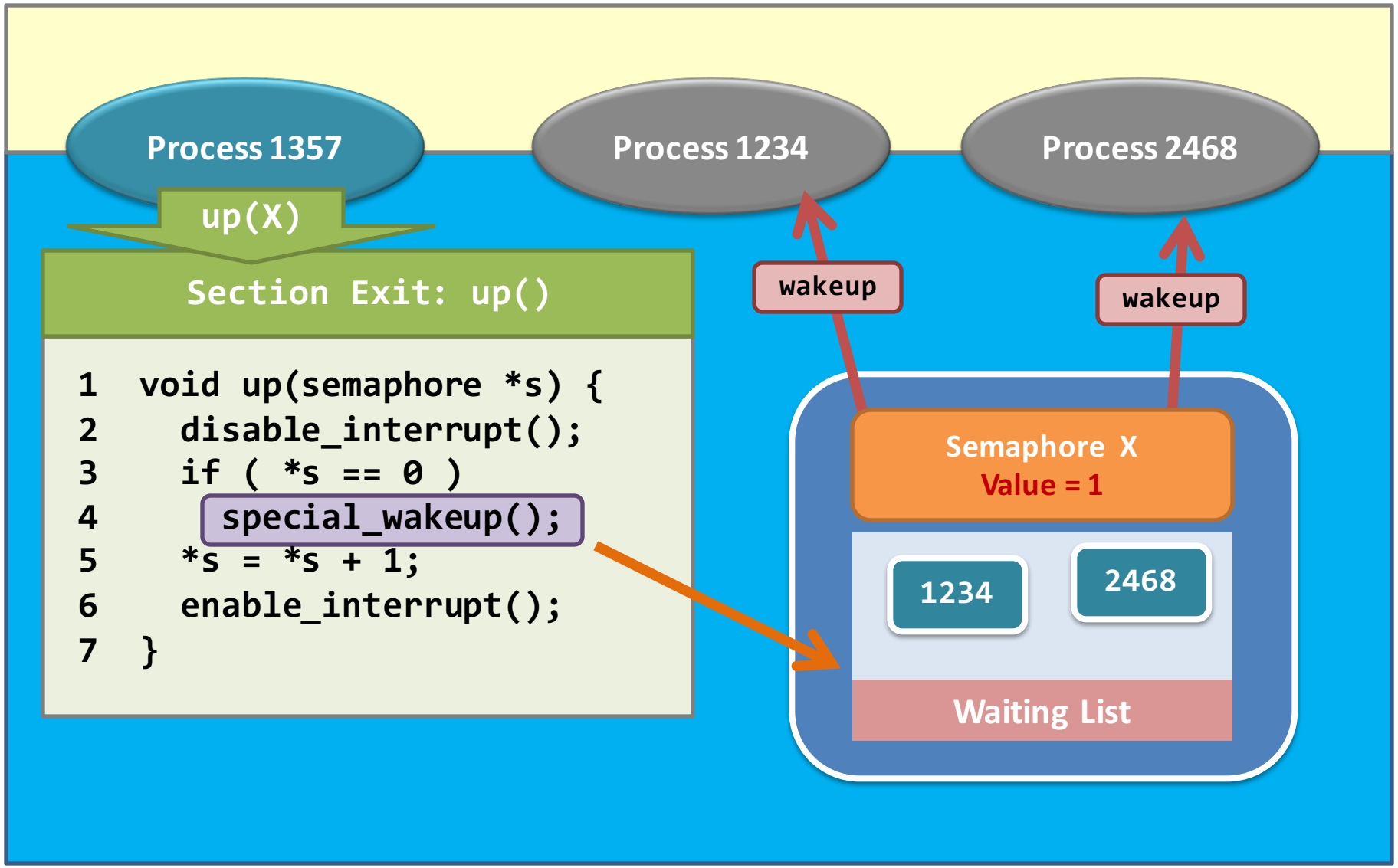
down(X)

Section Entry: down()

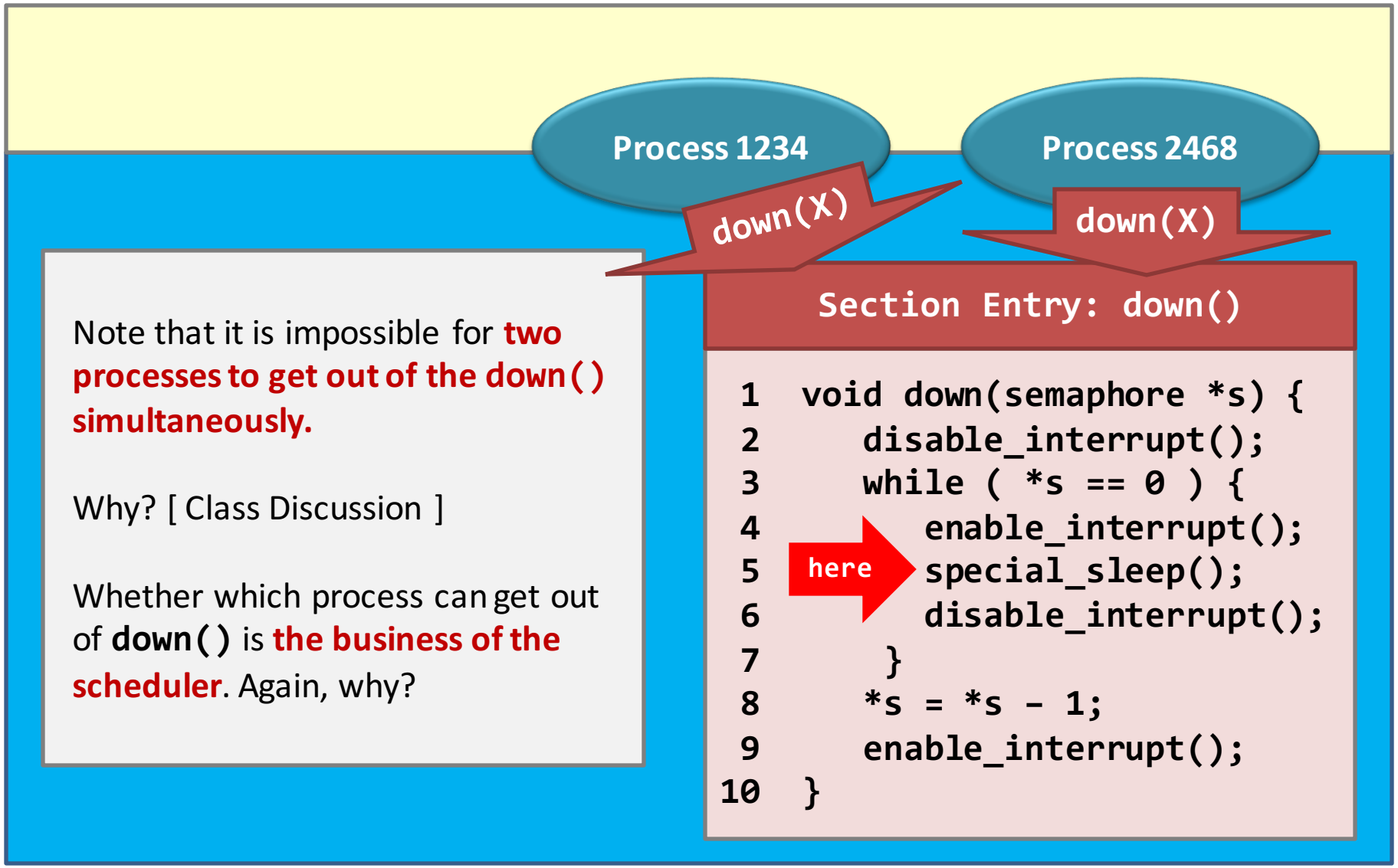
```
 1  void down(semaphore *s) {
 2     disable_interrupt();
 3     while ( *s == 0 ) {
 4        enable_interrupt();
 5        special_sleep();
 6        disable_interrupt();
 7      }
 8     *s = *s - 1;
 9     enable_interrupt();
10  }
```



Final proposal: Semaphore – details



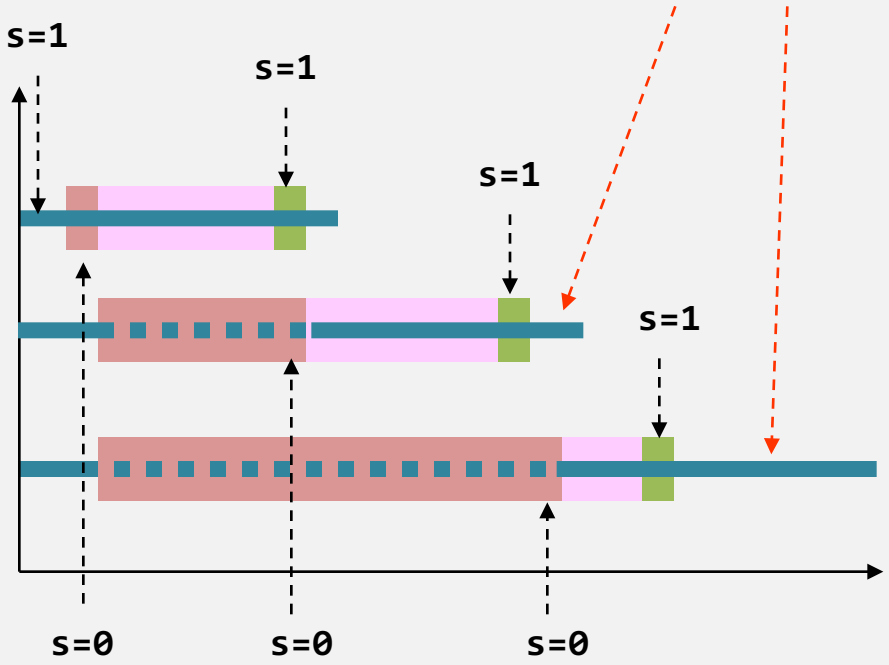
Final proposal: Semaphore – details



Final proposal: 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;  /* from kernel */
*s = 1;        /* initial value */
```

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

entry

exit

Summary…on section entry and section exit

- Remember, what happened is just the implementation of mutual exclusion.

	Working?	Comments
Disabling interrupts (on user-space)	Yes	Too dangerous, should not be used by normal programs.
Lock variable	No	Not worth mentioning
Strict alternation	Yes and No	Not a good one, violating one mutual exclusion requirement.
Peterson's solution	Yes	Busy waiting has a potential "*priority inversion problem*".
Semaphore	Yes	**BEST CHOICE**.

Summary…on semaphore

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

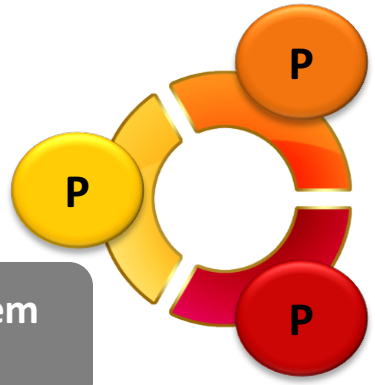
Definition of atomic operation

- 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.

Inter-process communication (IPC)

- What, why, and how?
- The problem: race condition.
- Mutual exclusion:
  - how to achieve?
  - how to implement?
- **Classic IPC problems.**



Let's teach them not to fight.

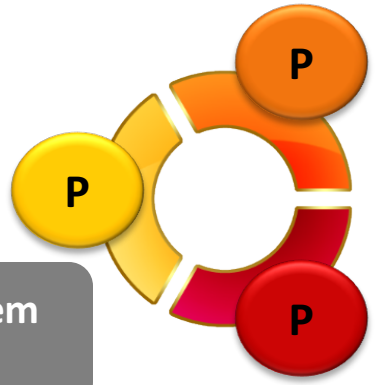
What are the problems?

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

	Properties	Examples
Producer-Consumer Problem	Two classes of processes: **producer** and **consumer**; 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: **reader** and **writer**. No limit on the number of the processes of each class.	Database.

Inter-process communication (IPC)

- Classic IPC problems.
  - Producer-consumer problem.

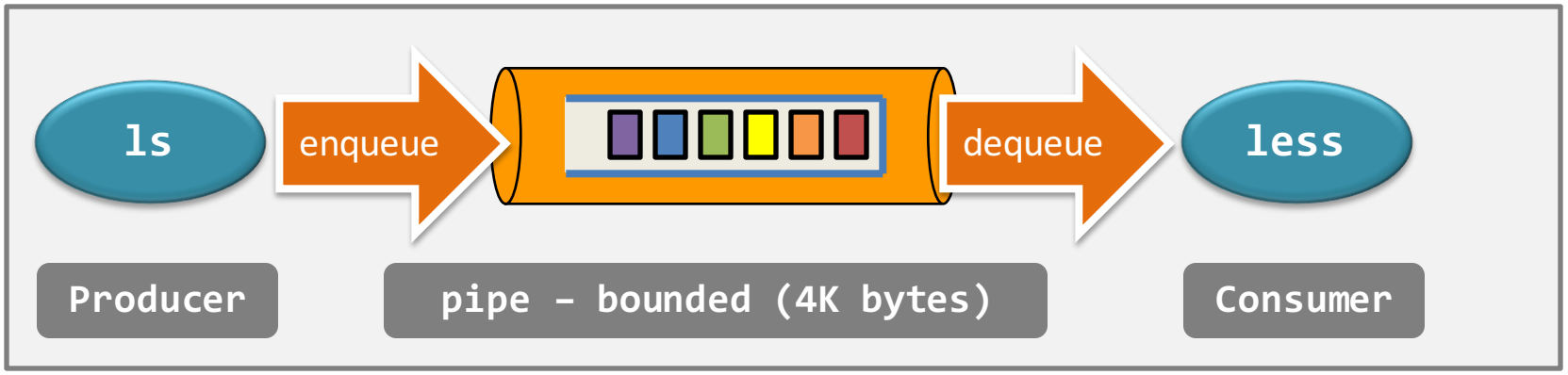


Let's teach them not to fight.

Producer-consumer problem – introduction

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

A bounded buffer	-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).
A producer process	-It produces a unit of data, and -writes that a piece of data to the tail of the buffer at one time.
A consumer process	-It removes a unit of data from the head of the bounded buffer at one time.



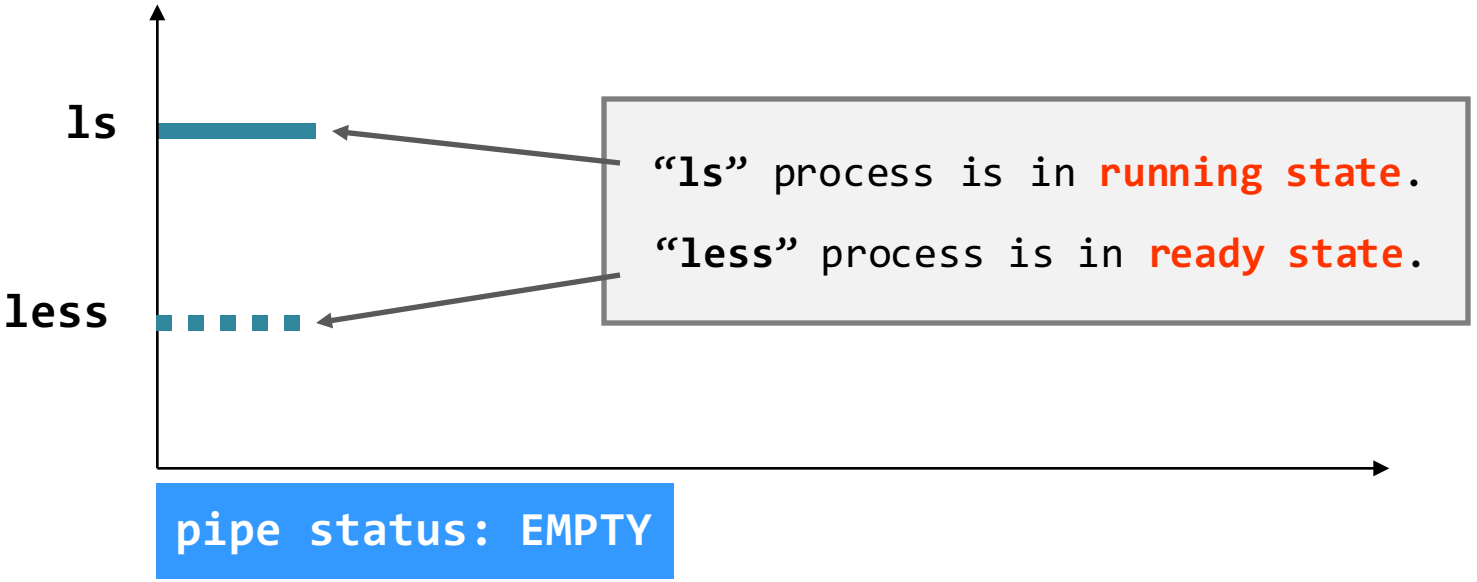
Producer-consumer problem – introduction

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 – how pipe works?

Assumptions

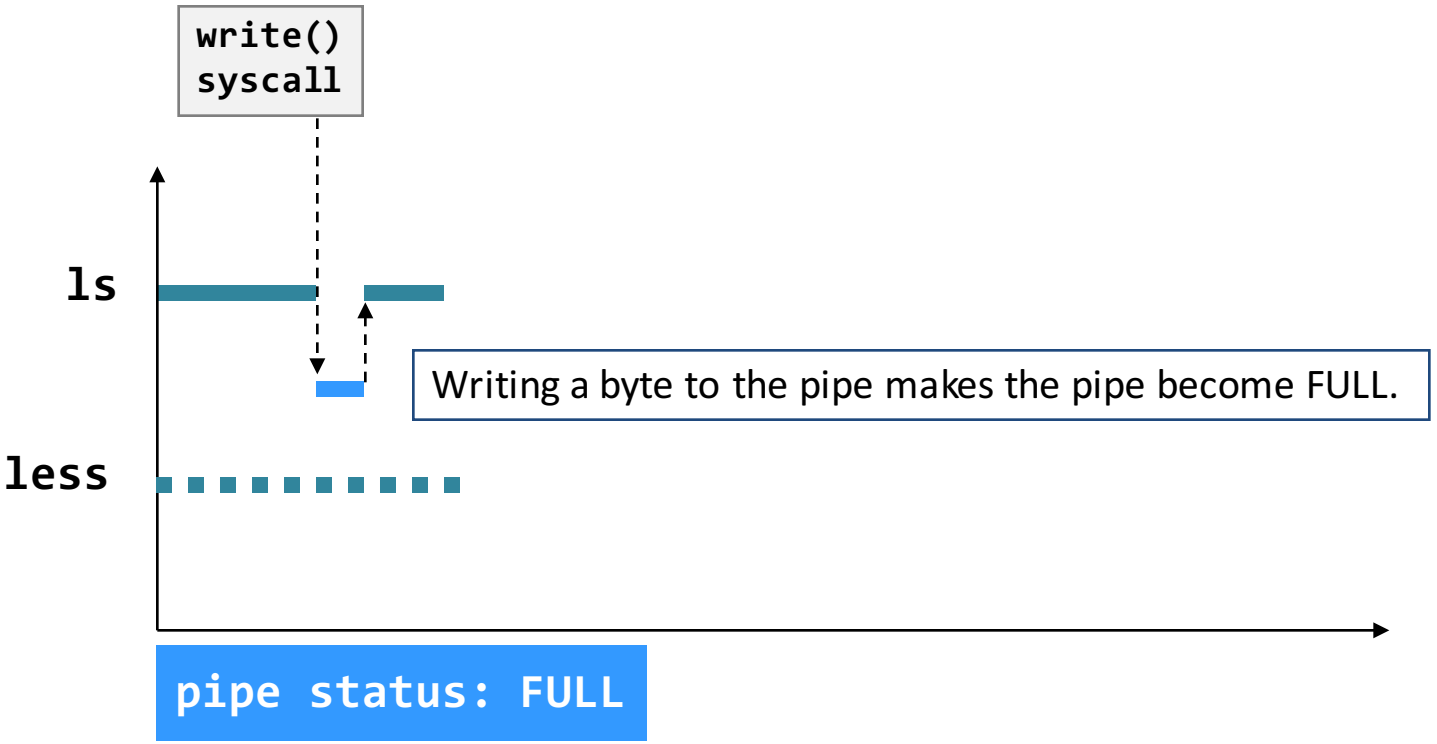
- The pipe is a queue of **1 byte** only!
- Each **write()** system call write **1 byte** to the pipe.
- Each **read()** system call read **1 byte** from the pipe.



Producer-consumer problem – how pipe works?

Assumptions

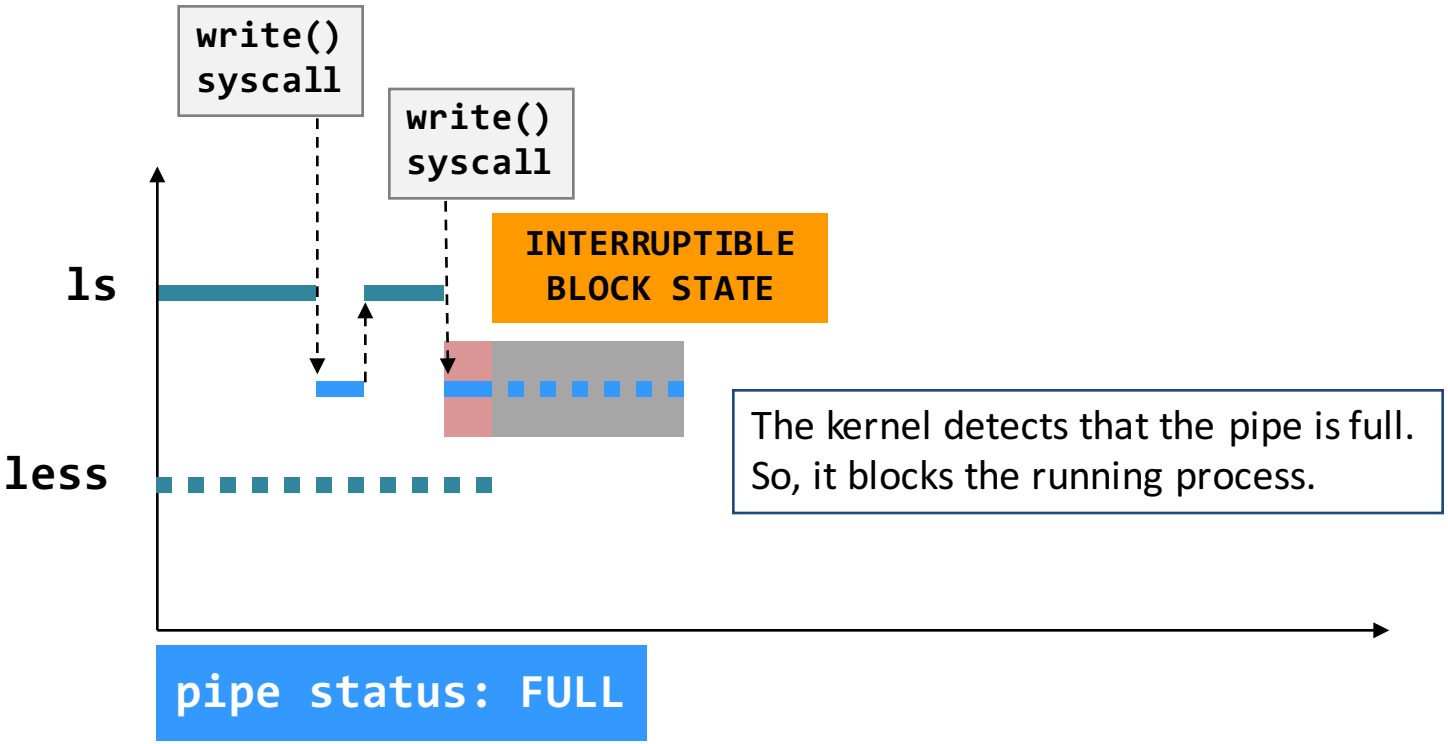
- The pipe is a queue of **1 byte** only!
- Each **write()** system call write **1 byte** to the pipe.
- Each **read()** system call read **1 byte** from the pipe.



Producer-consumer problem – how pipe works?

Assumptions

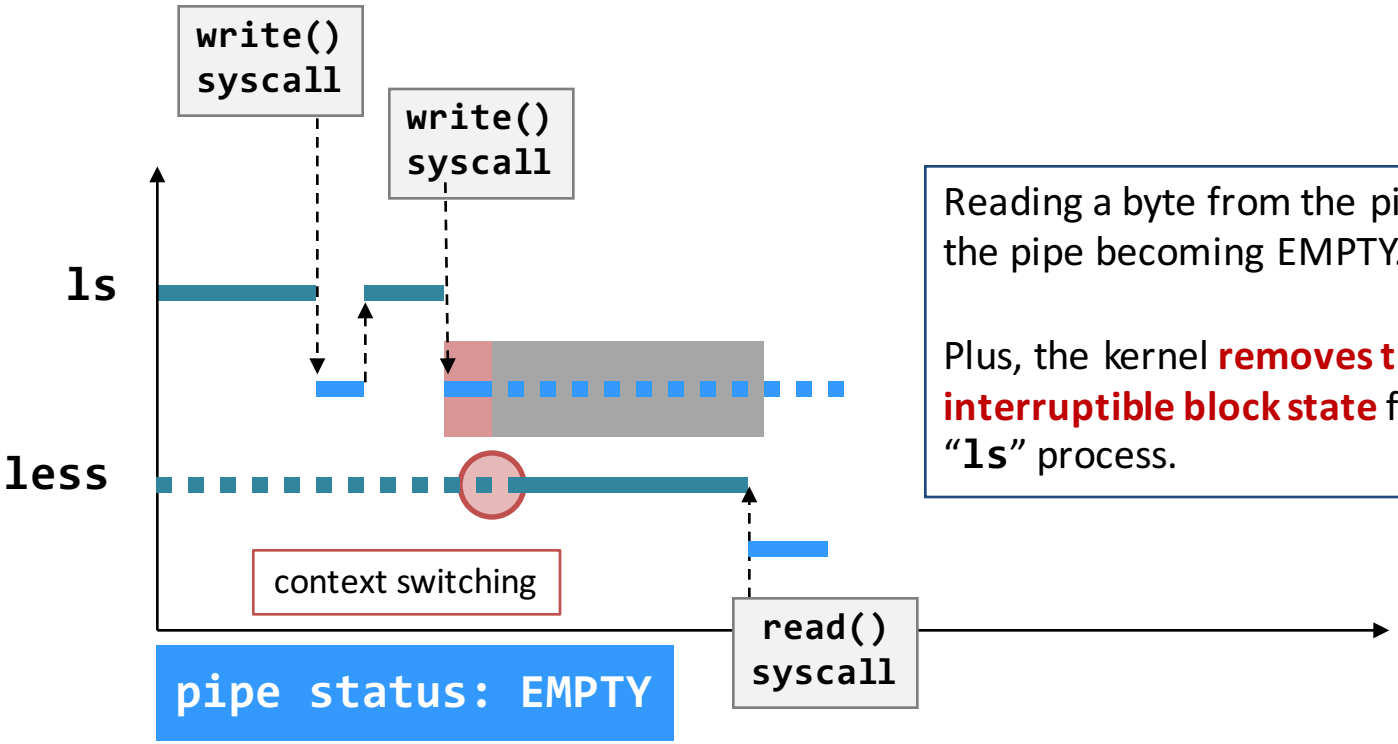
- The pipe is a queue of **1 byte** only!
- Each **write()** system call write **1 byte** to the pipe.
- Each **read()** system call read **1 byte** from the pipe.



Producer-consumer problem – how pipe works?

Assumptions

- The pipe is a queue of **1 byte** only!
- Each **write()** system call write **1 byte** to the pipe.
- Each **read()** system call read **1 byte** from the pipe.



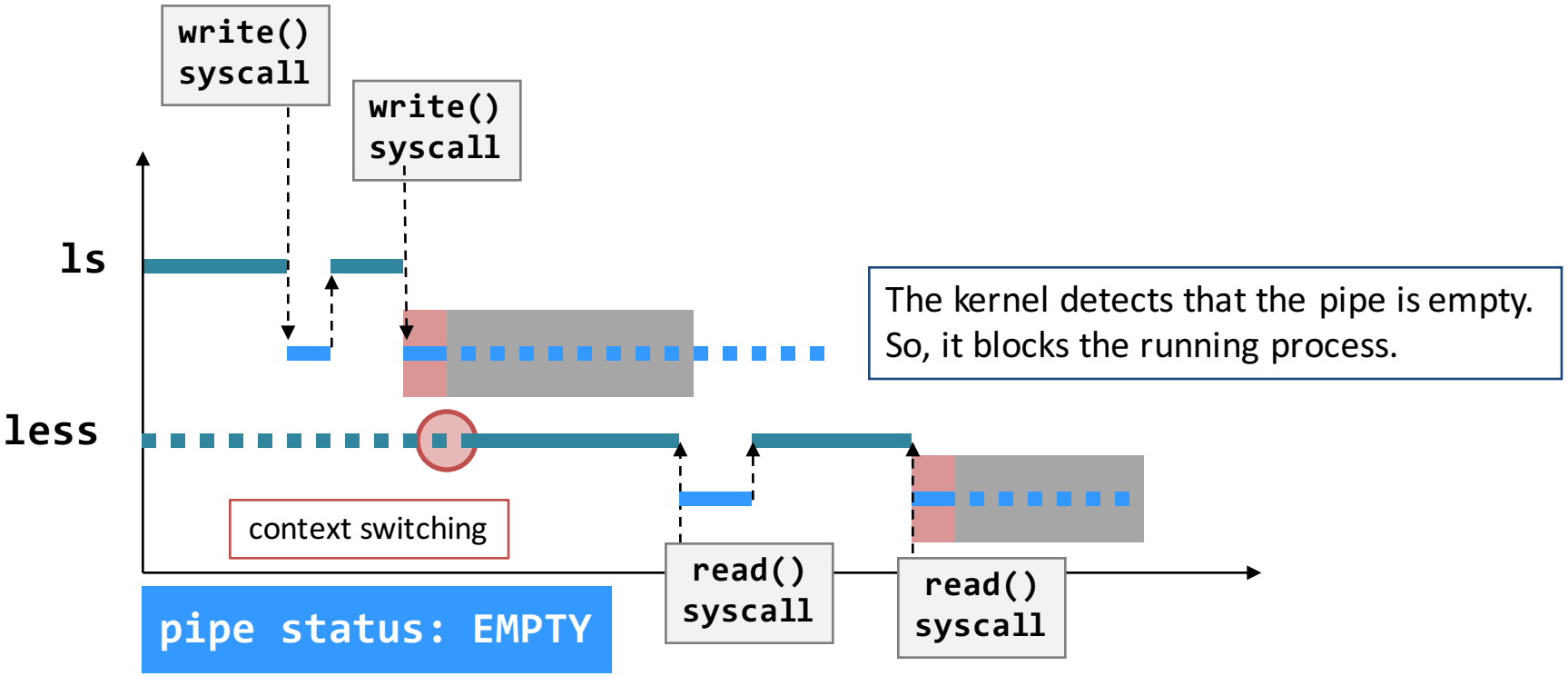
Reading a byte from the pipe makes the pipe becoming EMPTY.

Plus, the kernel **removes the interruptible block state** from the "**ls**" process.

Producer-consumer problem – how pipe works?

Assumptions

- The pipe is a queue of **1 byte** only!
- Each **write()** system call write **1 byte** to the pipe.
- Each **read()** system call read **1 byte** from the pipe.

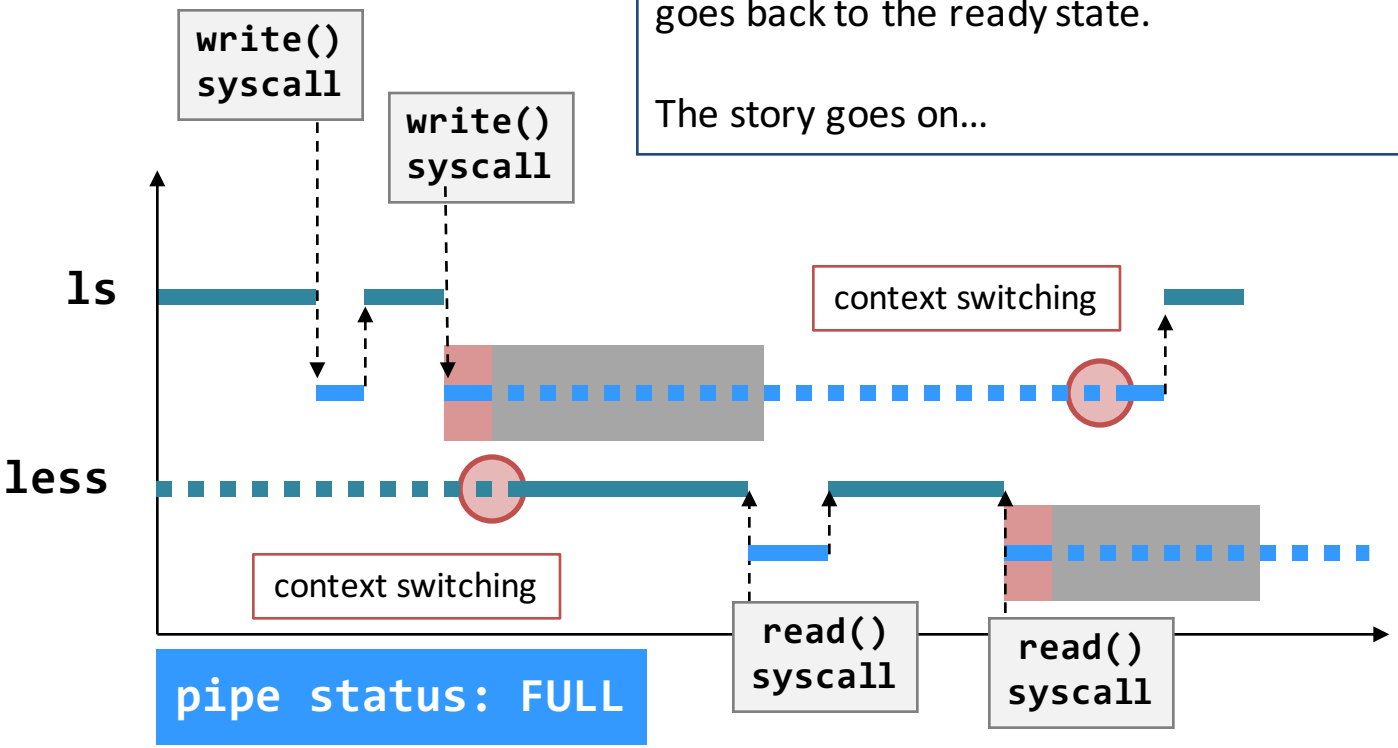


Producer-consumer problem – how pipe works?

When the "**`ls`**" process goes back to the running state, it **immediately writes a byte to the pipe so as to complete the execution of the `write()` system call**.

Then, the pipe is no longer empty, and the "**`less`**" process goes back to the ready state.

The story goes on…



Producer-consumer problem – introduction

- "**Hey, pipe is working fine. What are you going to teach?**" you asked.
  - 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 this case.

- In the following, we revisit the producer-consumer problem with **the use of shared objects and semaphores**, instead of pipe.

Producer-consumer problem – solution

Note

The functions "**insert_item()**" and "**remove_item()**" are accessing the bounded buffer.

The size of the bounded buffer is "**N**".

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 – questions…

Question #1.
Can we remove any one of the semaphores, "empty", "full", or "mutex"?

Question #2.
Can we swap Lines 6 & 7 of the producer?

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 – question #1

Question #1.
Can we remove any one of the semaphores, "empty", "full", or "mutex"?

Hint #1: ask yourself, **"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          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 #1

Question #1.
Can we remove any one of the semaphores, "empty", "full", or "mutex"?

Hint #1: ask yourself, **"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          down(&empty);
 7          down(&mutex);
 8          insert_item(item);
 9          up(&mutex);
10          up(&full);
11      }
12  }
```

The "**mutex**" stands for mutual exclusion.

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

Class discussion: can the initial value of "**mutex**" be 2? If yes, what is the meaning?

Producer-consumer problem – question #1

Question #1.
Can we remove any one of the semaphores, "empty", "full", or "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 – question #1

- The two variables are not for mutual exclusion, but a task called **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 – question #1

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?

See? 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 – question #1

- See? 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 #2

Question #2.
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;
```

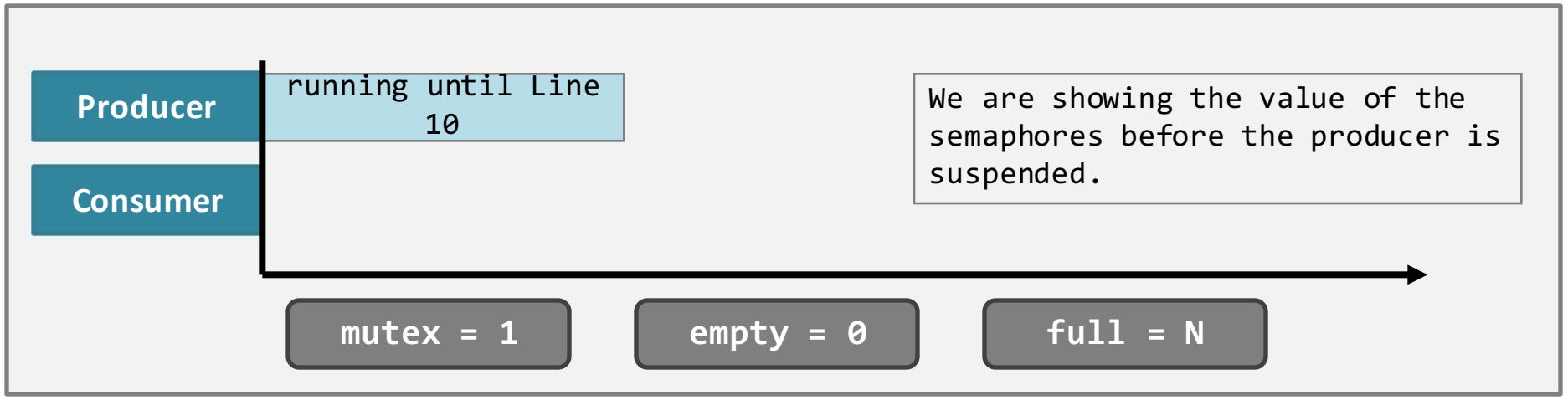
Producer function

```
 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 #2



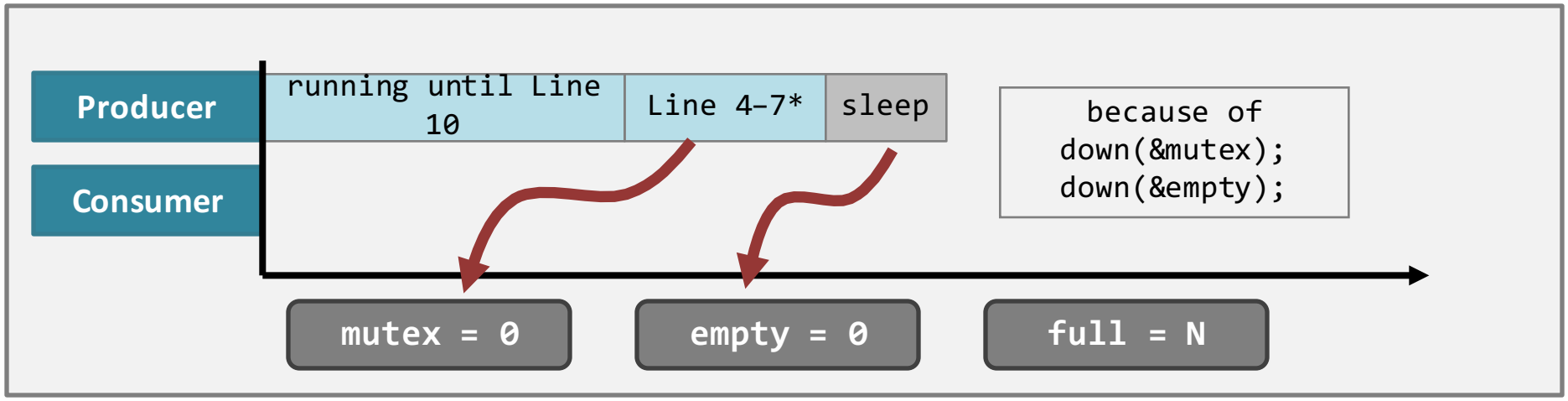
Producer function

```
 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 #2



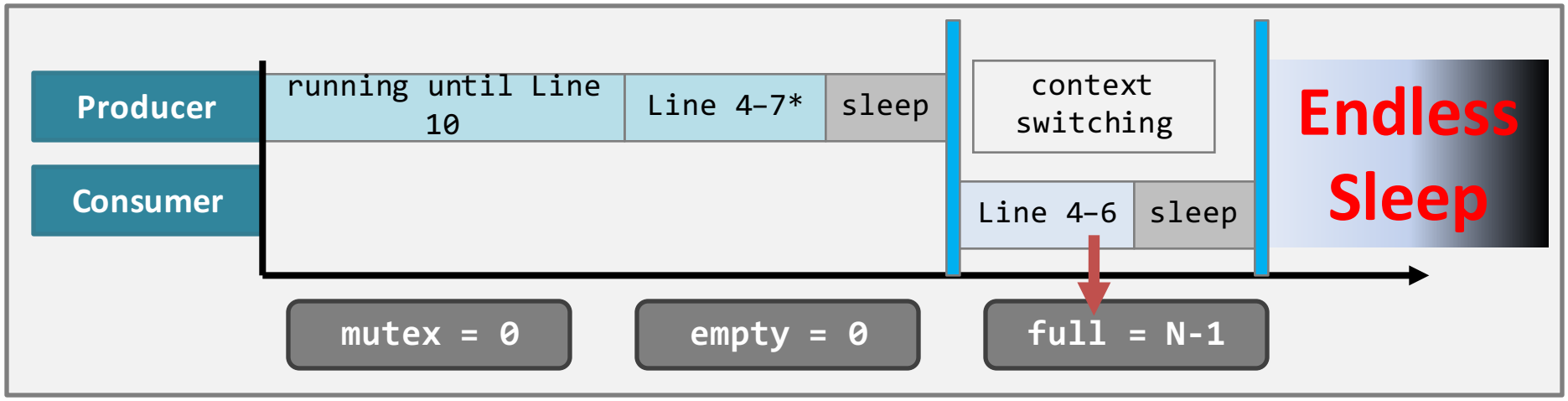
Producer function

```
 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 #2



Producer function

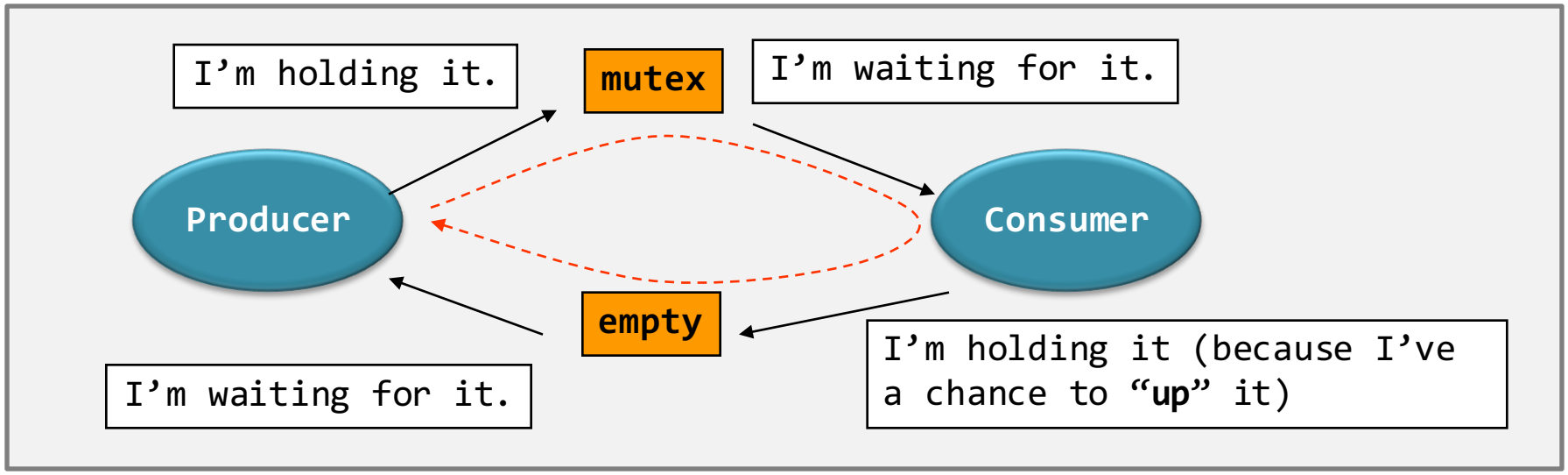
```
 1  void producer(void) {
 2      int item;
 3
 4      while(TRUE) {
 5          item = produce_item();
 6*         down(&mutex);
 7*         down(&empty);
 8          insert_item(item);
 9          up(&mutex);
10          up(&full);
11      }
12  }
```

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 #2

- This scenario is called a **deadlock**, and it 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.

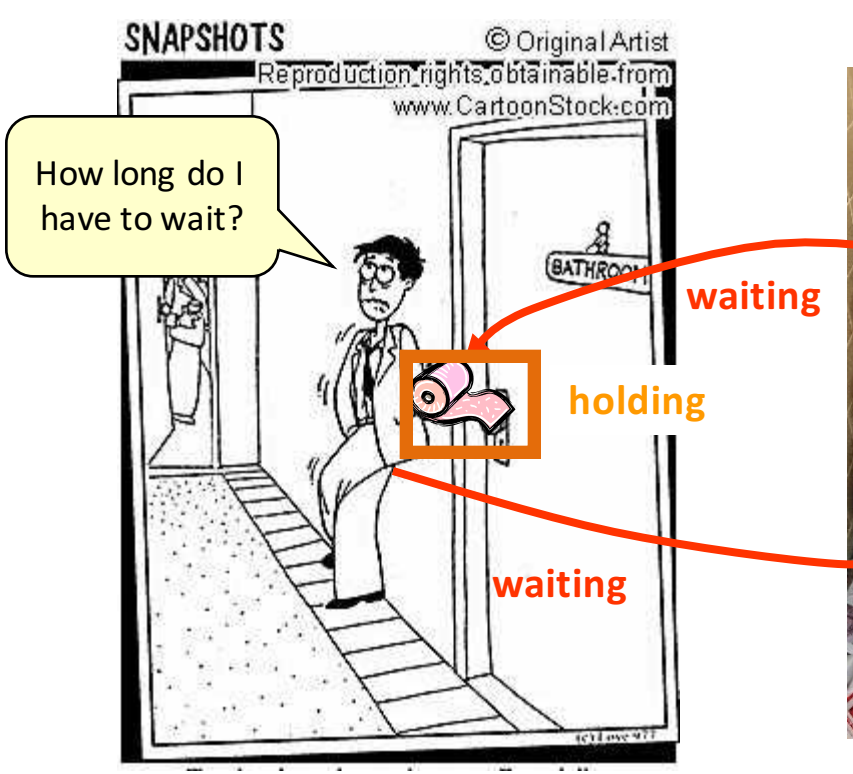


Producer-consumer problem – question #2

- This scenario is called a **deadlock**, and it 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.

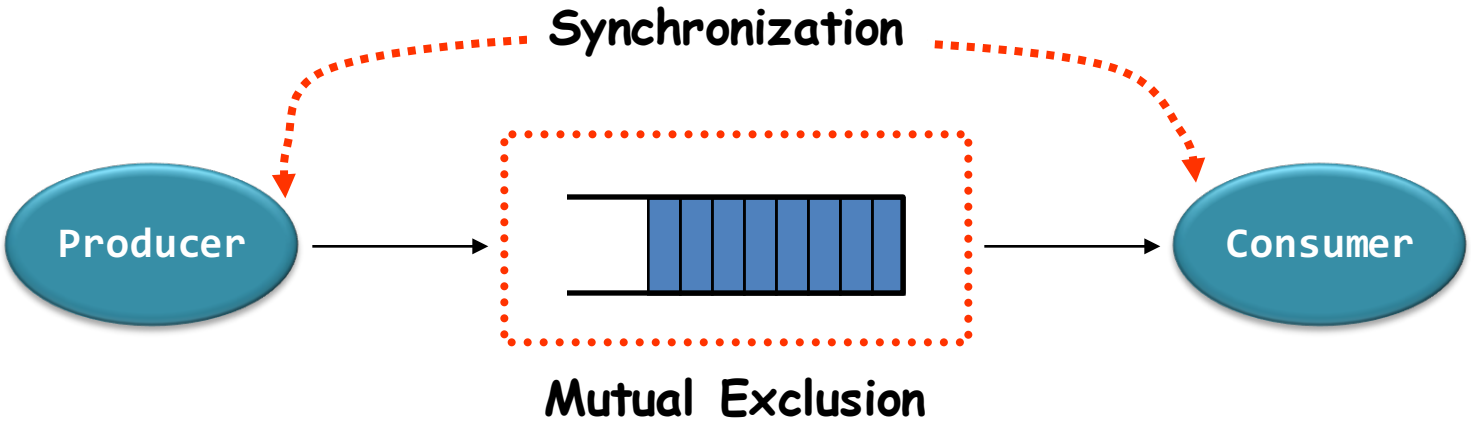
- **No progress could be made by all processes + All processes are blocked.**
  - **Implication**: careless implementation of the producer-consumer solution can be disastrous.

Pseudo-real-life example of deadlock



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.
    - **Class discussion.** What if the buffer's size is unlimited?



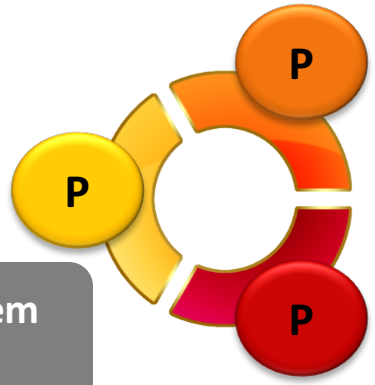
Summary on producer-consumer problem

- How to guarantee mutual exclusion?
  - A semaphore is used as the entry and the exit of the critical sections.
  - We also call that semaphore a **binary semaphore**.

- 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.

Inter-process communication (IPC)

- Classic IPC problems.
  - Producer-consumer problem.
  - Dining philosopher problem.

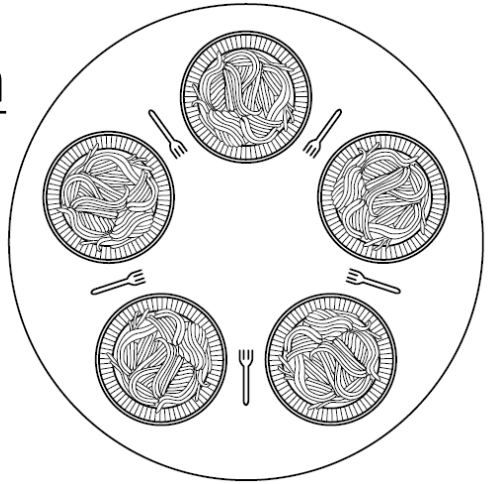


Let's teach them not to fight.

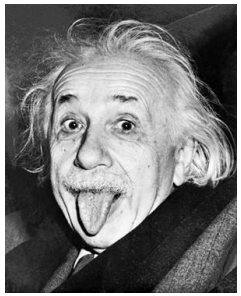
Dining philosopher – introduction

- 5 philosophers, 5 plates of spaghetti, and 5 forks.
- The jobs of each philosopher are to sleep and to eat (and that's why I want to be a philosopher!).
- They are so old that they **need exactly two forks** in order to eat the spaghetti.

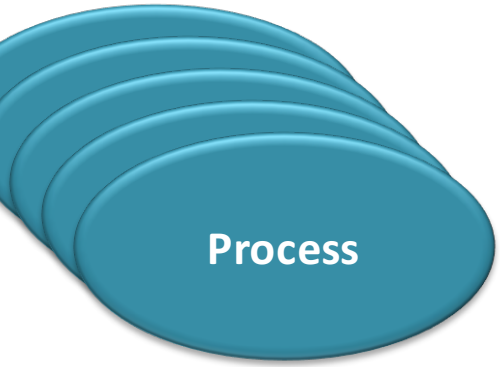
- Question: how to construct a synchronization protocol such that
  - they will not be **starved to death**, and
  - will not result in any **deadlocking scenarios**?



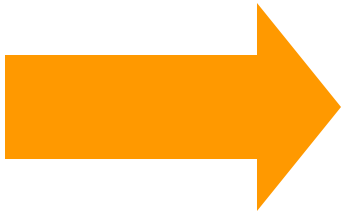
Dining philosopher – introduction

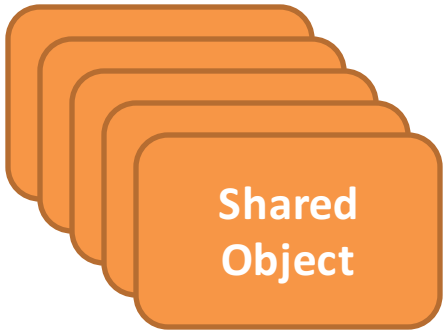


Philosophers

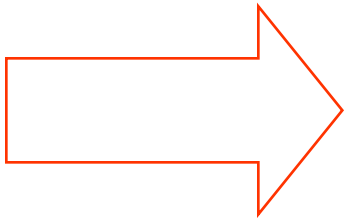


Forks





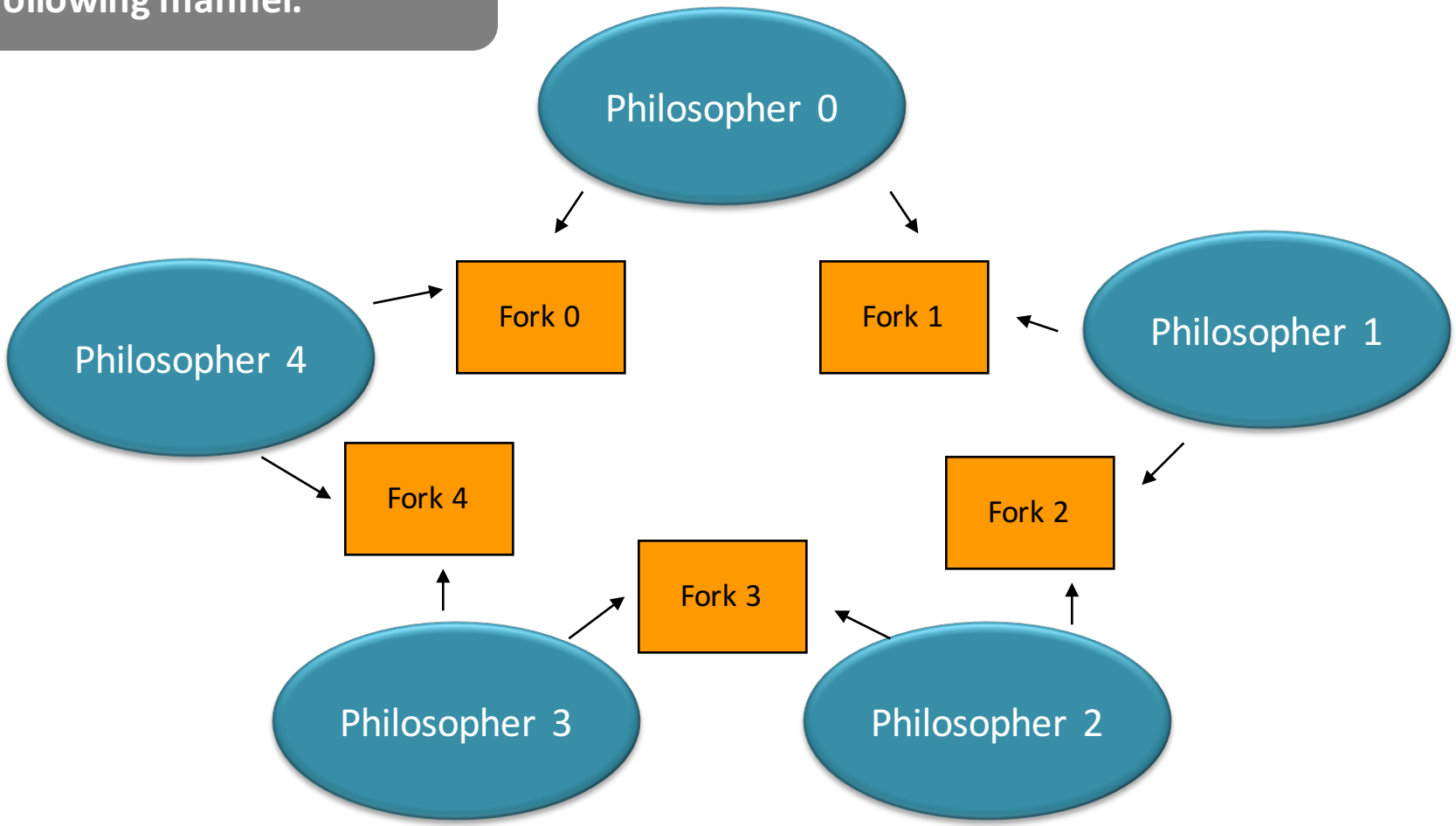
Spaghetti



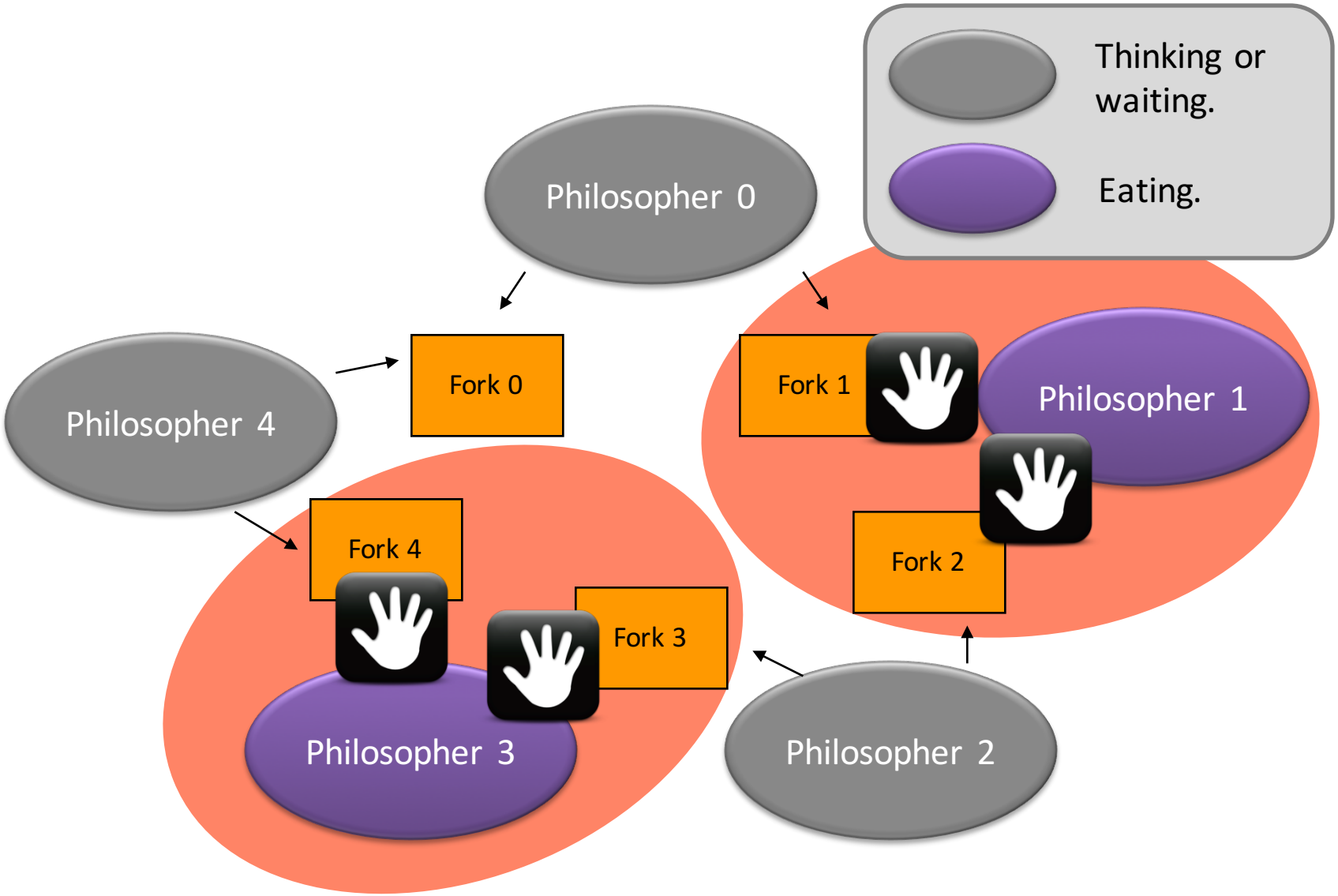
Consider to have infinite supply.

Dining philosopher – introduction

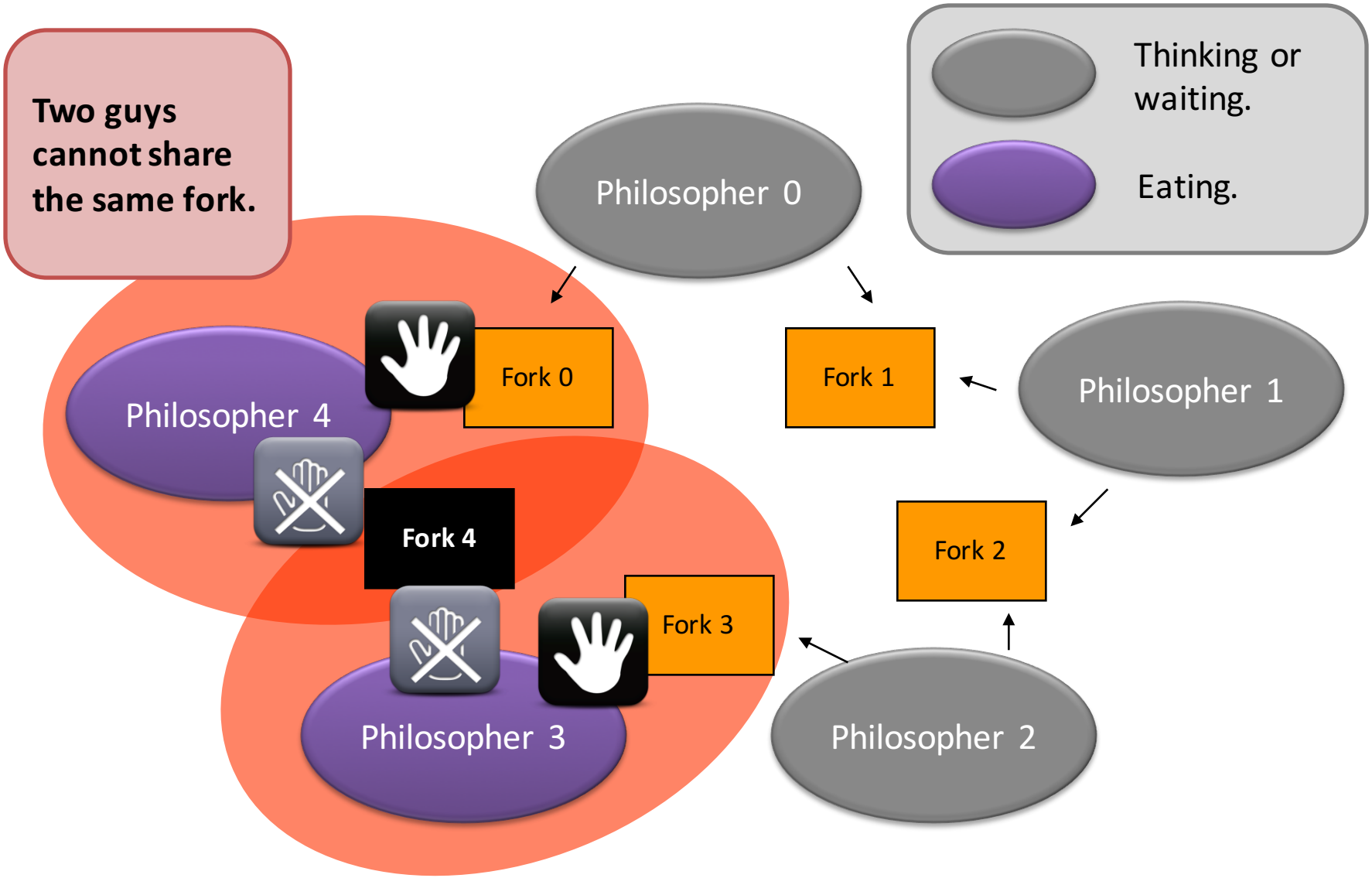
The forks must be arranged in the following manner.



Dining philosopher – introduction



Dining philosopher – introduction



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 forks**!

- Let's proposal the following solution:
  - When you are hungry, you have to check if anyone is using the forks that you need.
  - If yes, you have to wait.
  - If no, **seize both forks**.
  - After eating, put down all your forks.

Dining philosopher – meeting requirement #1?

A quick question: what should be initial values?

Shared object

```
#define  N  5
semaphore fork[N];
```

Helper Functions

```
void take_fork(int i) {
    down(&fork[i]);
}
```

```
void put_fork(int i) {
    up(&fork[i]);
}
```

Section Entry

Critical Section

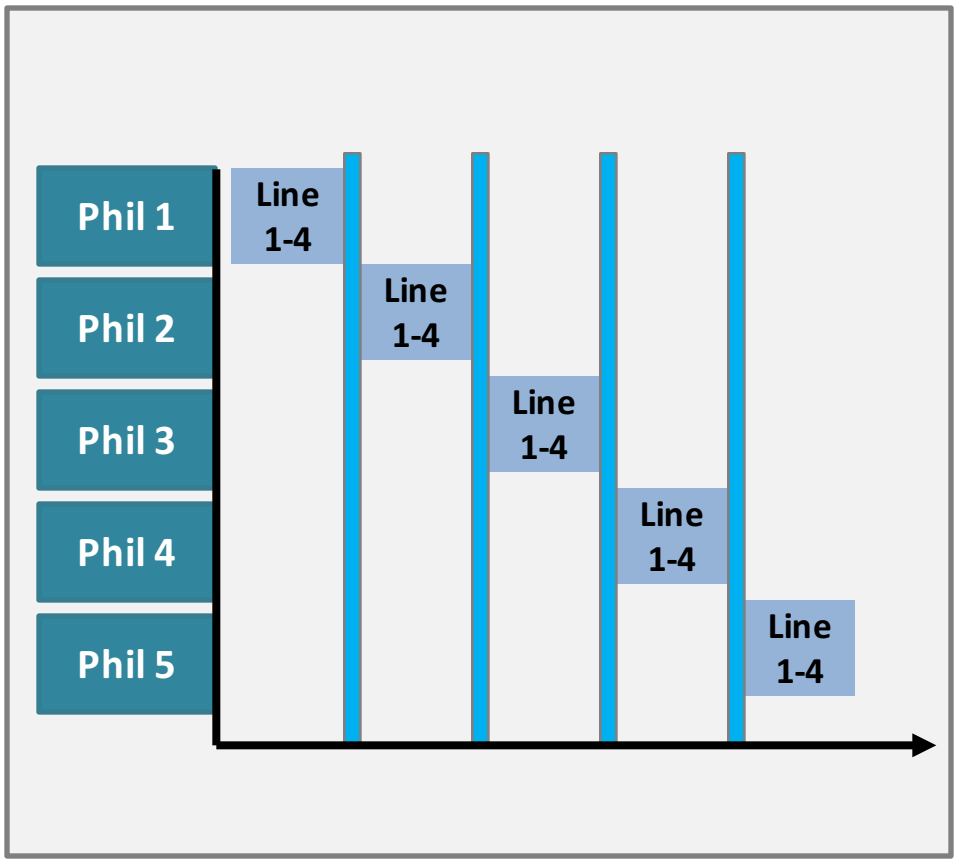
Section Exit

Main Function

```
 1 void philosopher(int i) {
 2     while (TRUE) {
 3         think();
 4         take_fork(i);
 5         take_fork((i+1) % N);
 6         eat();
 7         put_fork(i);
 8         put_for((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_fork(i);
5         take_fork((i+1) % N);

6         eat();

7         put_fork(i);
8         put_for((i+1) % N);
9     }
10 }
```

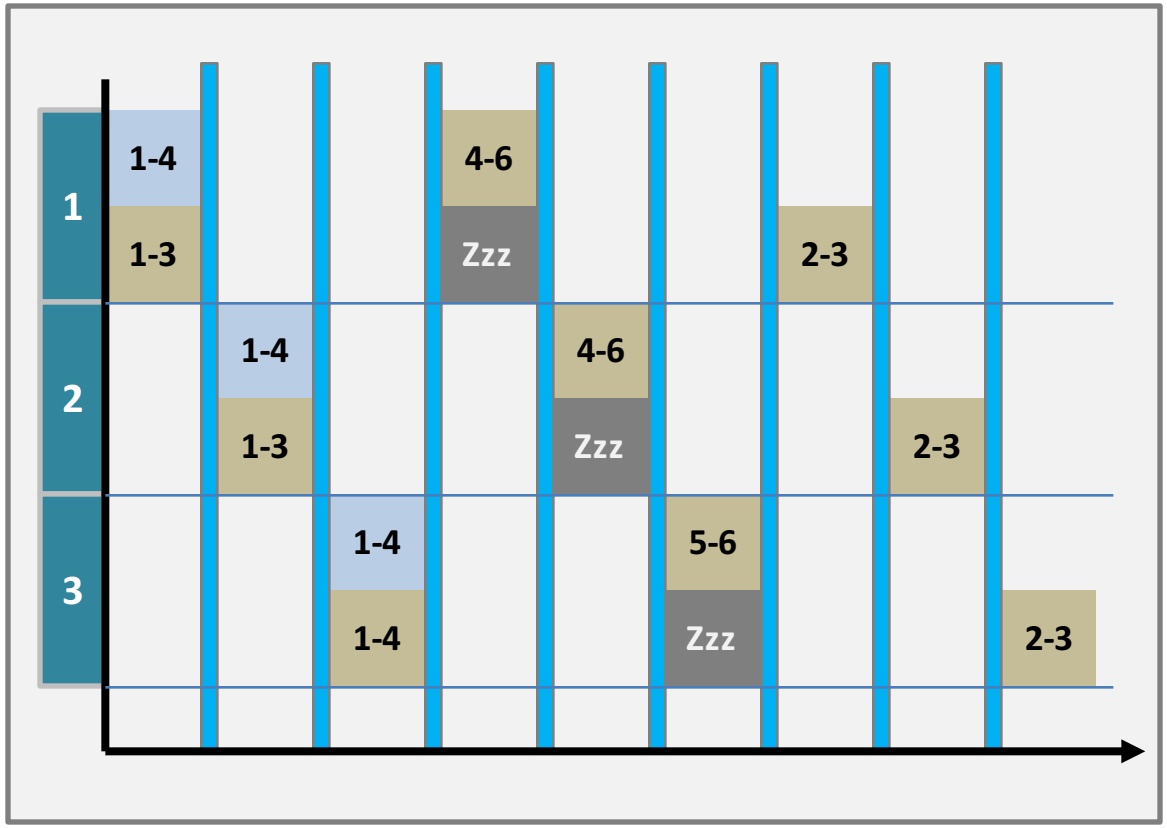
Dining philosopher – requirement #2

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

- How about the following suggestions:
  - First, a philosopher **takes a fork**.
  - If a philosopher finds that he cannot take the second fork, then he should **put down the first fork**.
  - Then, the philosopher **goes to sleep** for a while.
  - Again, the philosopher tries to get both forks until both forks are seized.

Dining philosopher – meeting requirement #2?

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



Assume N = 3 (because the space in PPT is limited)

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

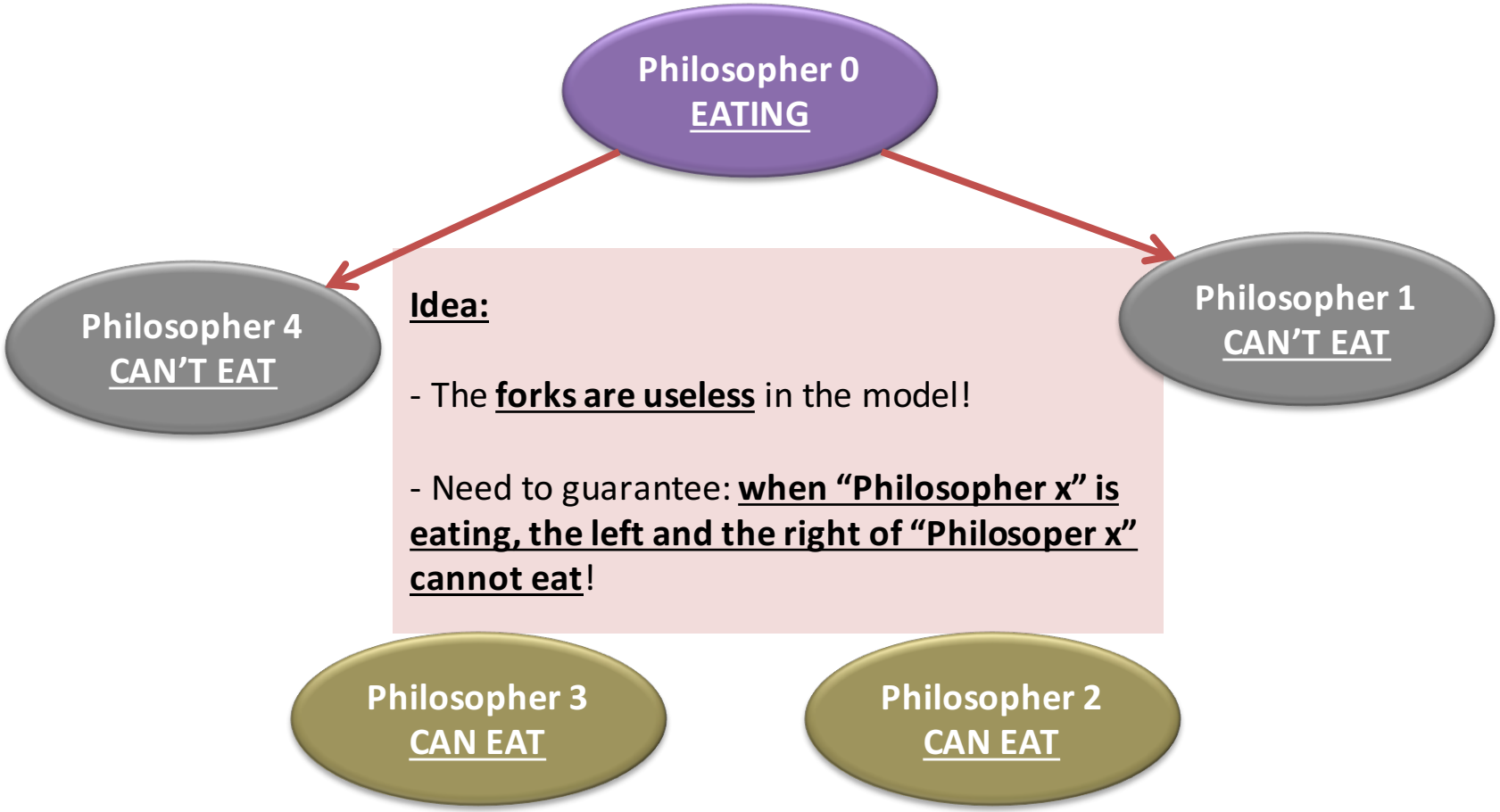
```
 1 void philosopher(int i) {
 2     while (TRUE) {
 3       think();
 4       take_forks(i);
 5       eat();
 6       up(&fork[i]);
 7       up(&fork[(i+1)%N)]);
 8     }
 9 }
```

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 fork 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_forks(i);
4      eat();
5      put_forks(i);
6  }
```

Don't panic.

I'll explain the codes piece by piece.

Section entry

```
1  void take_forks(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_forks(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 }
```

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.

To fulfill the synchronization requirement.

Question. Tell me (later), what are the initial values of the "**s[N]**" array?

Dining philosopher – the final solution.

Section entry

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

Extremely important helper function

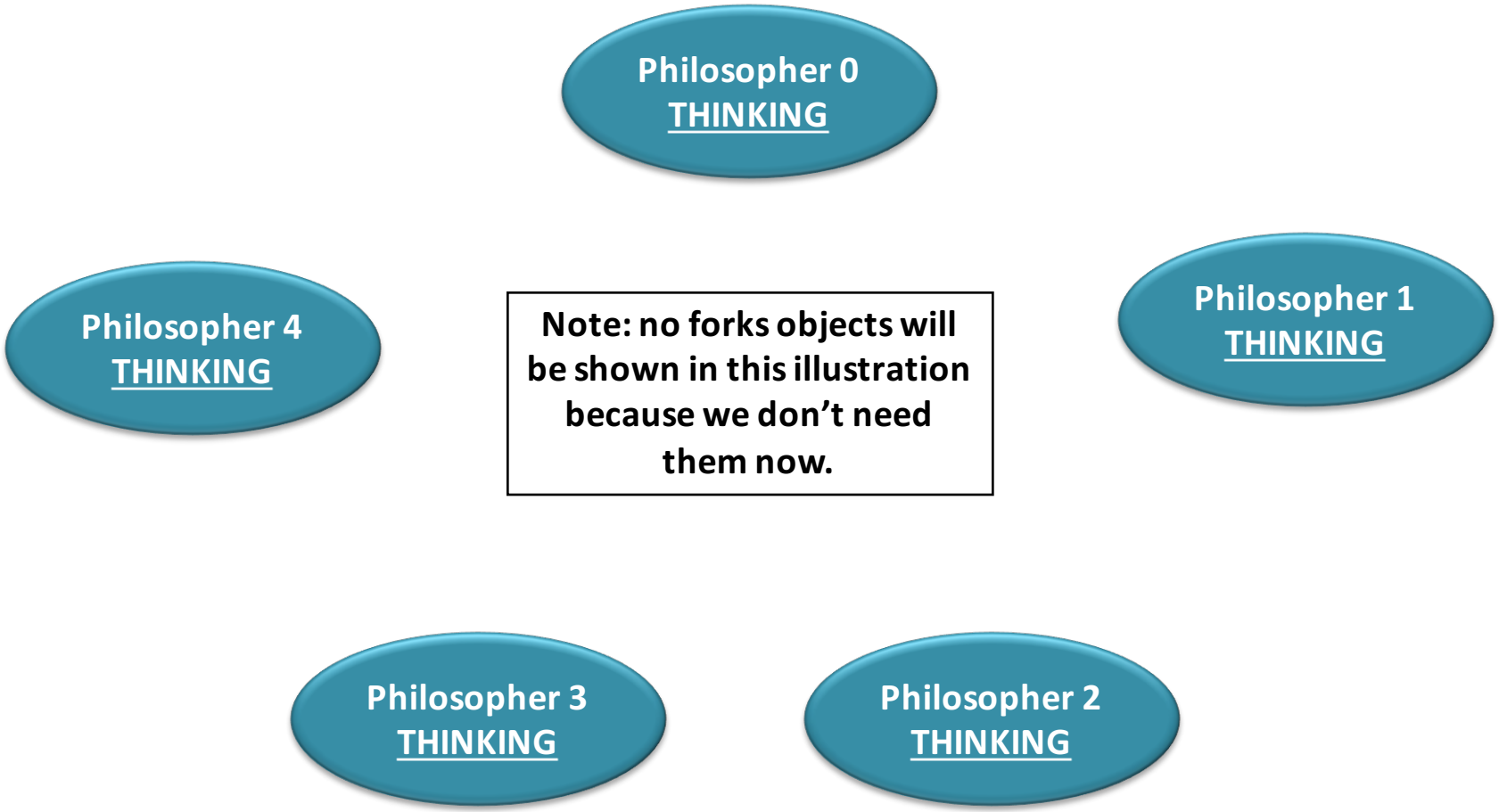
```
1 void test(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         up(&s[i]);
5     }
6 }
```

Wake up the one who can eat!

Dining philosopher – the final solution.

Don't print

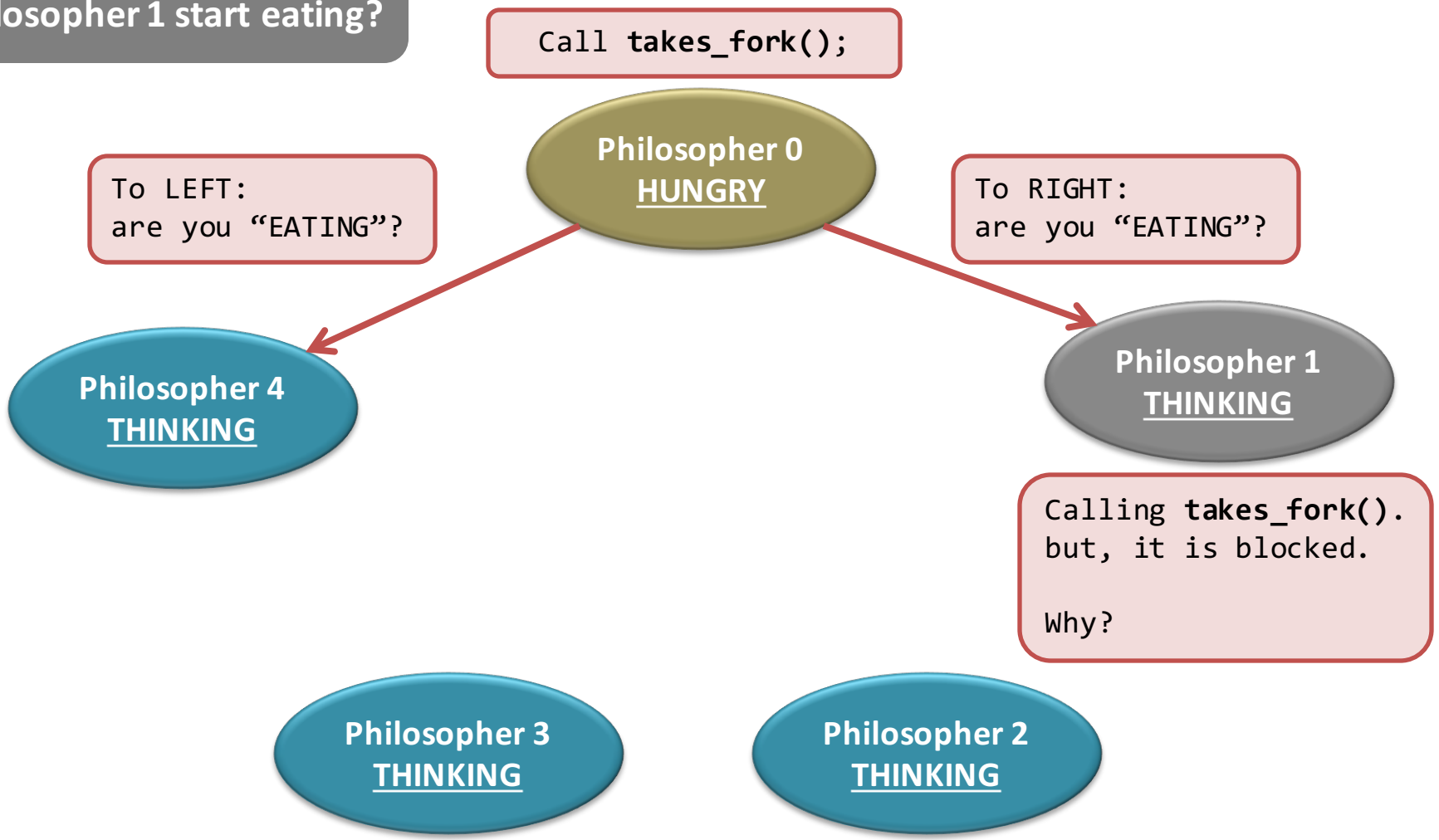
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

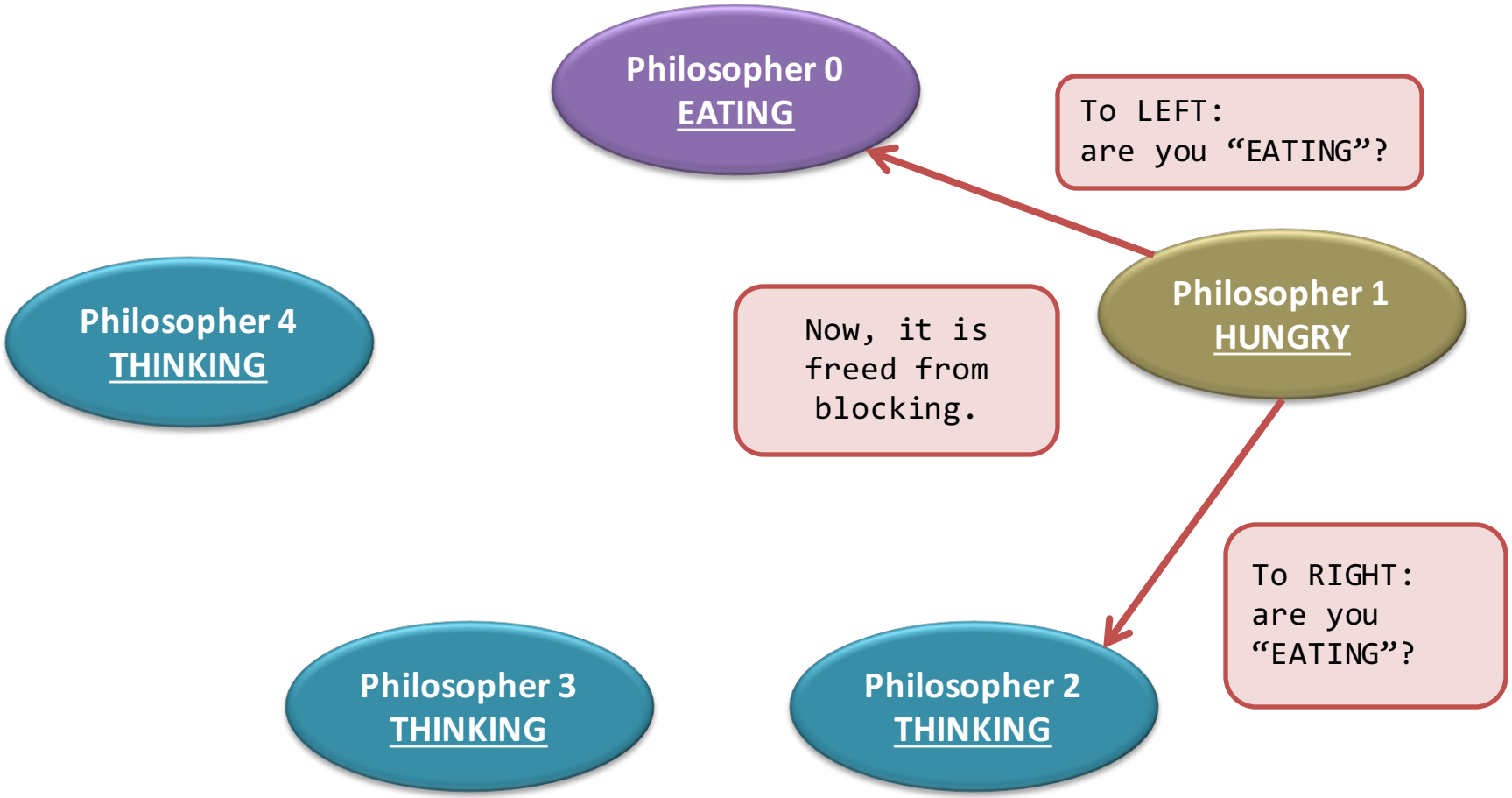
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

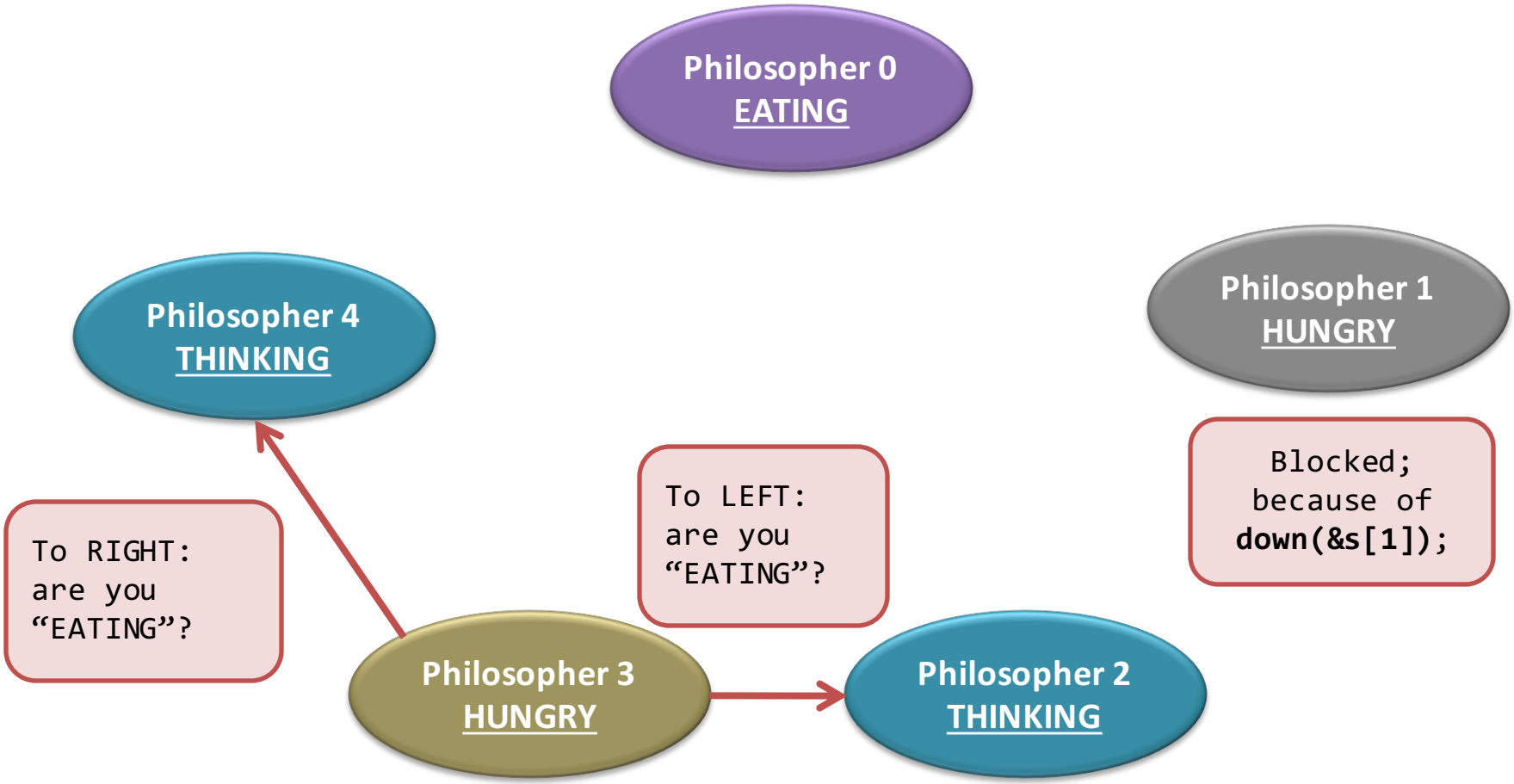
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

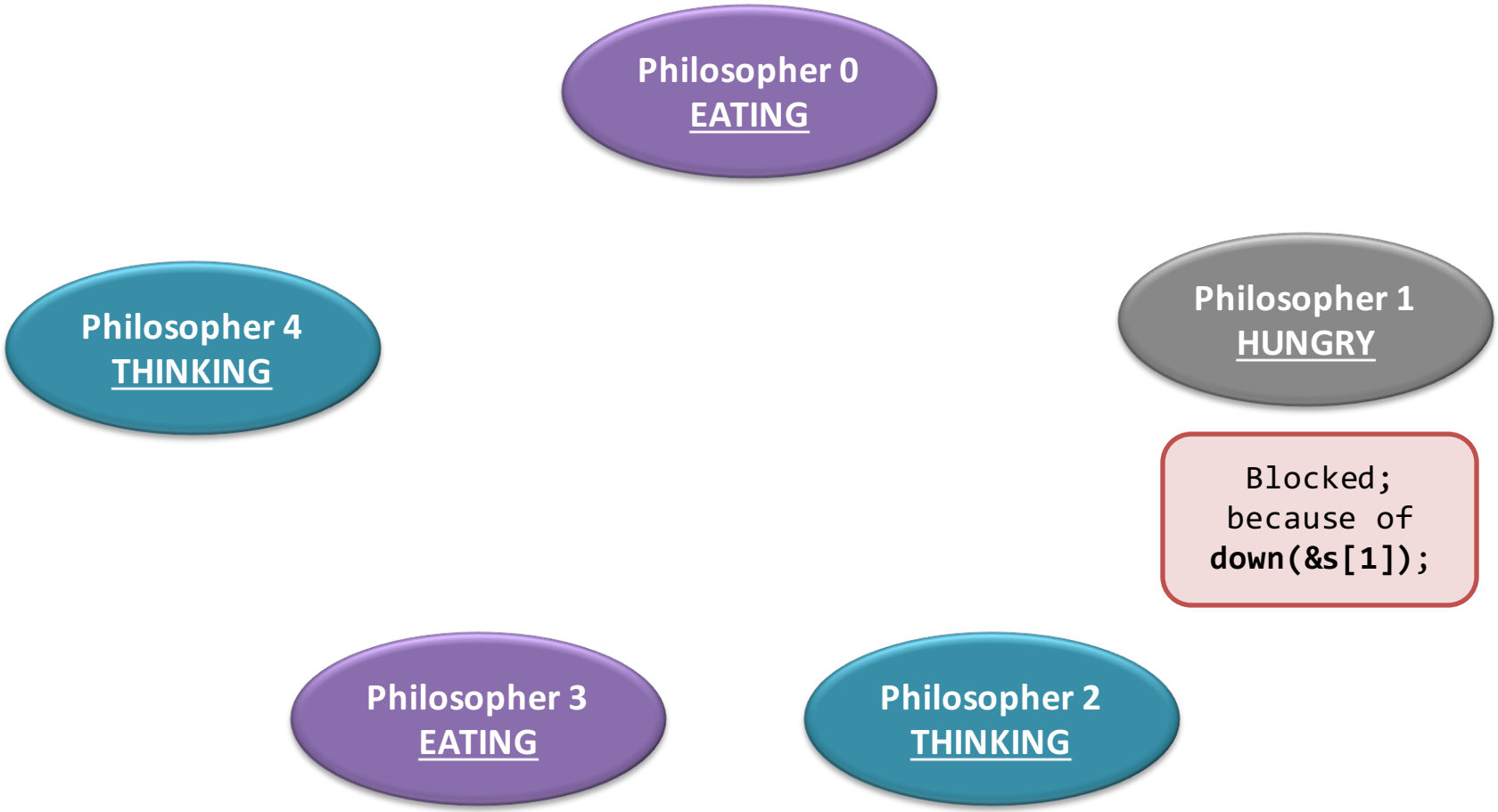
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

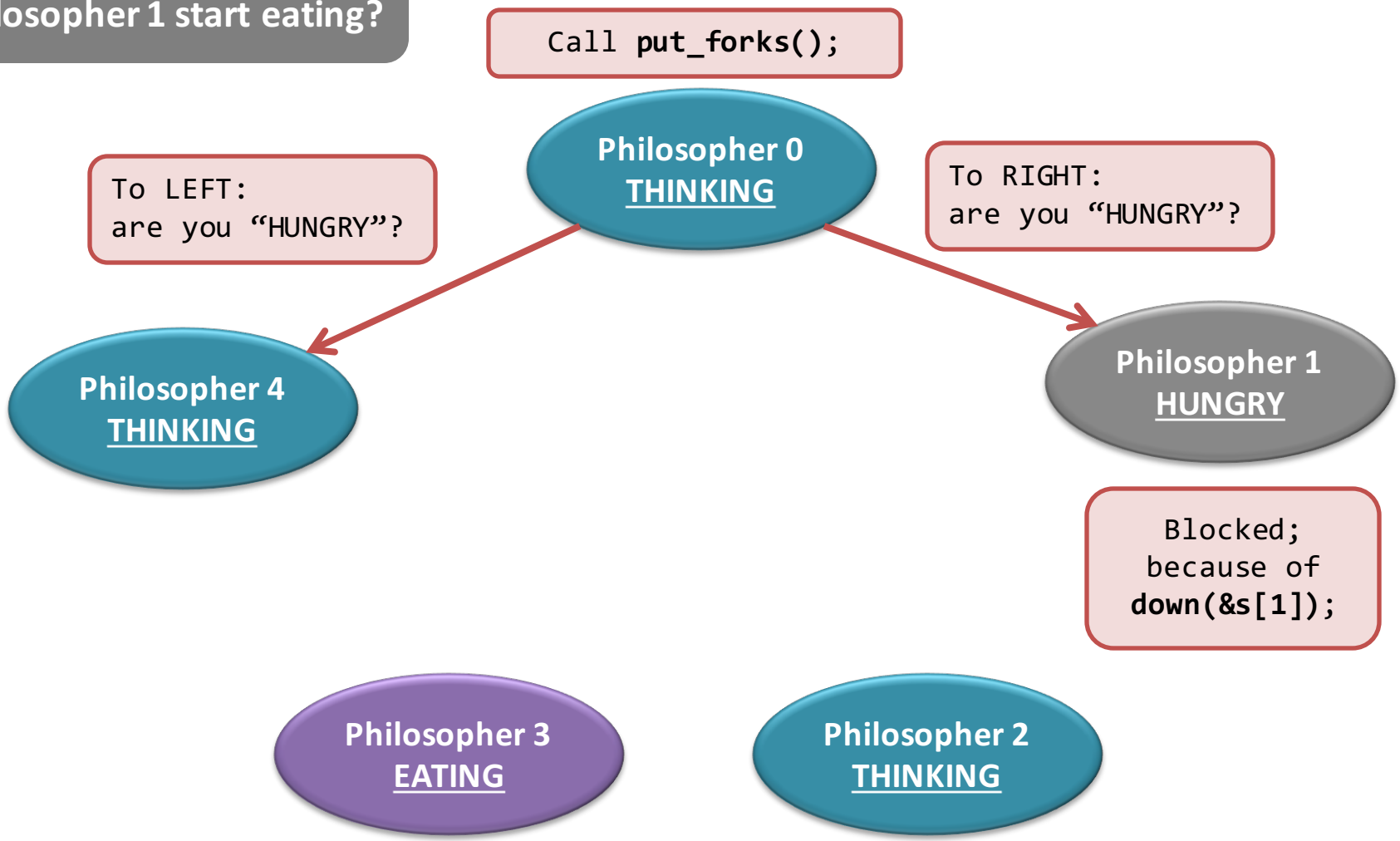
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

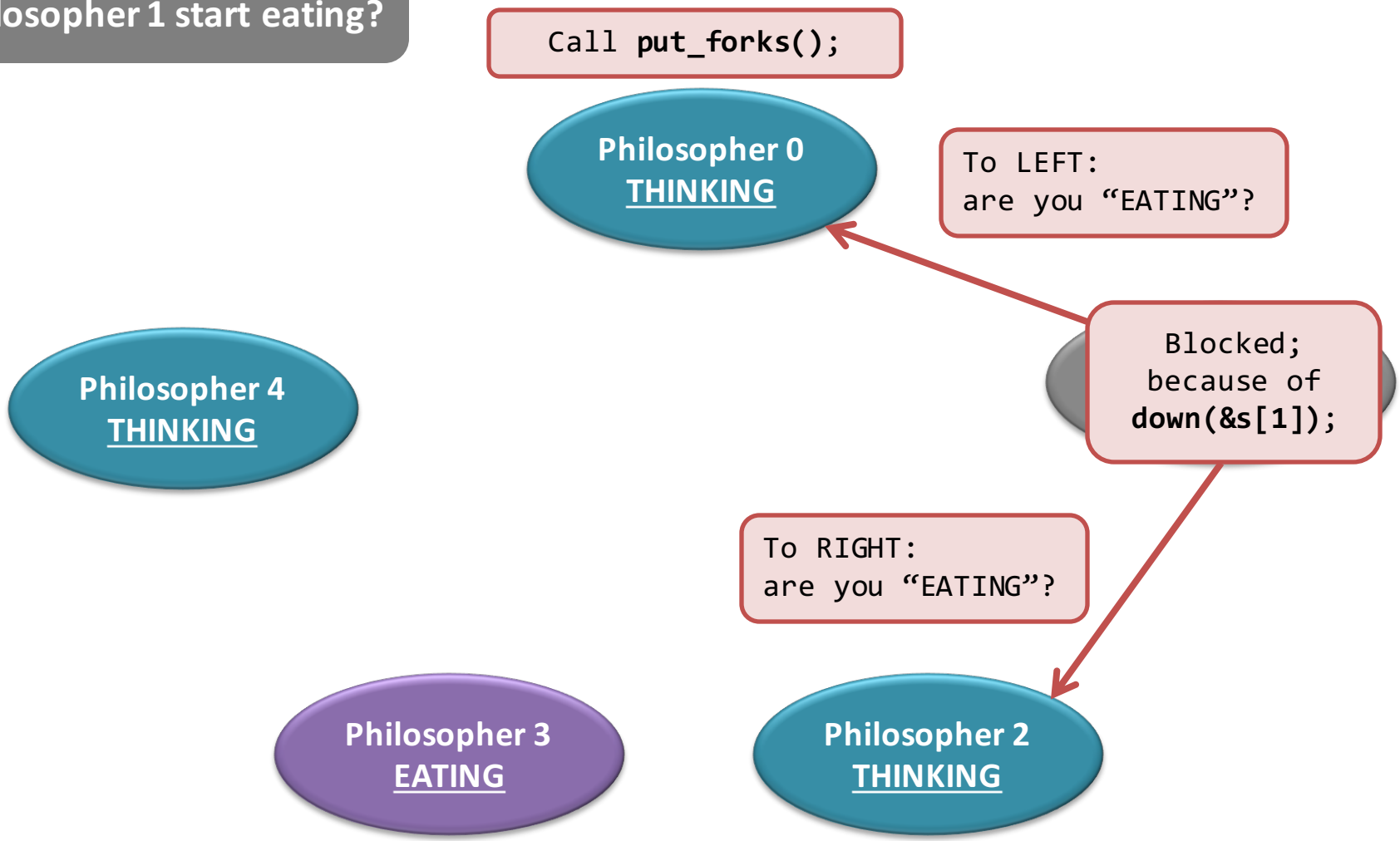
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

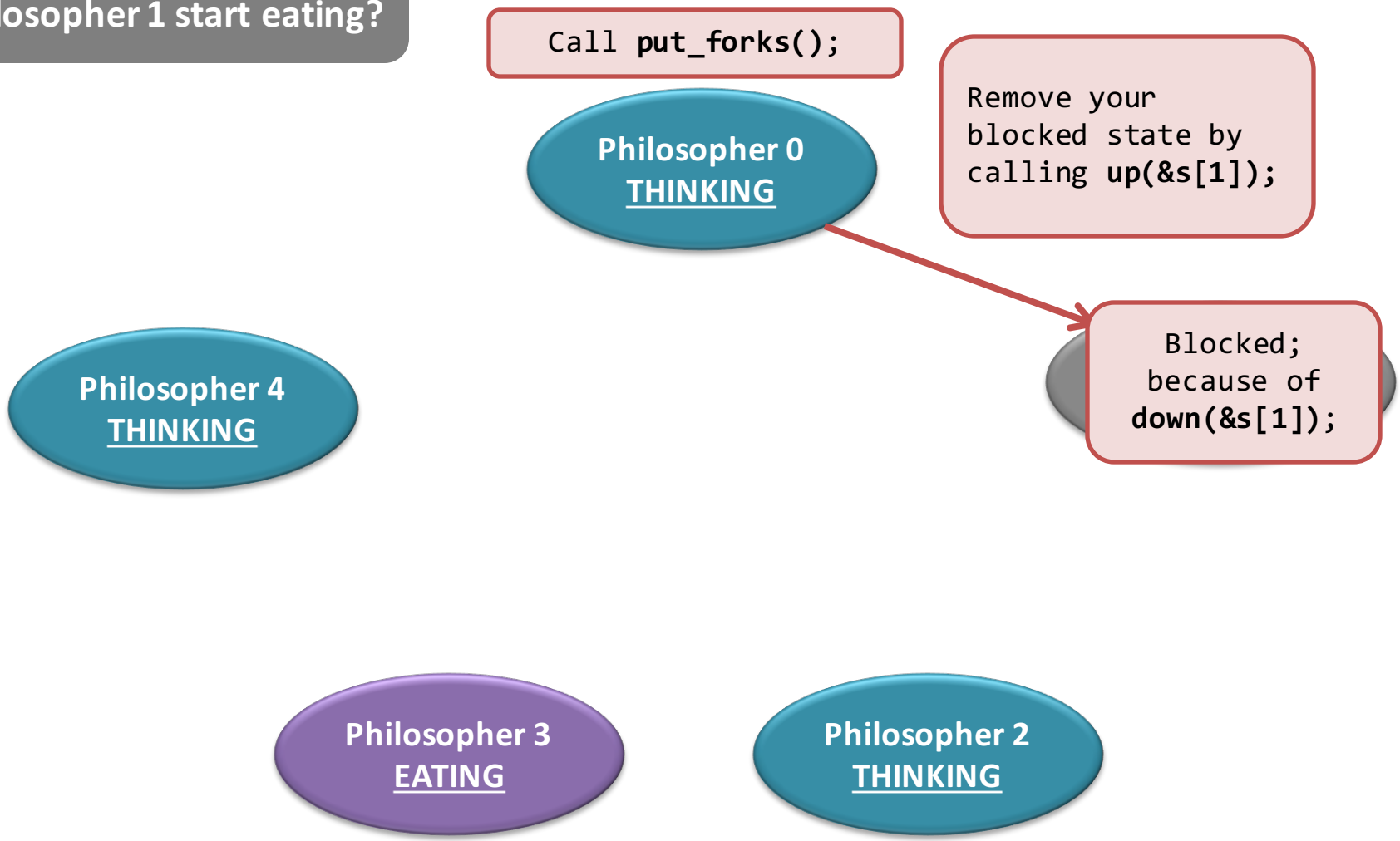
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

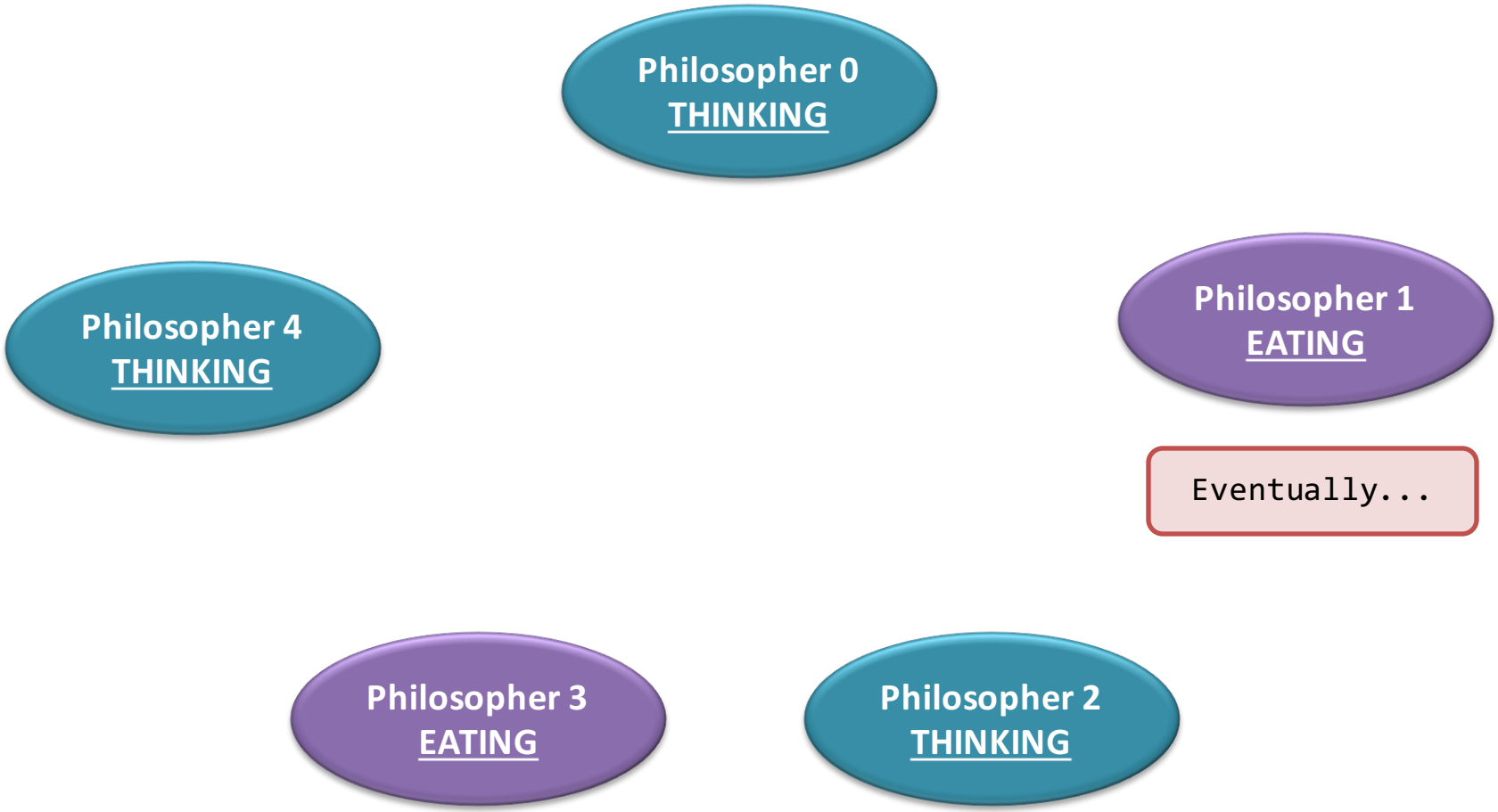
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

An illustration: How can Philosopher 1 start eating?



Dining philosopher - summary

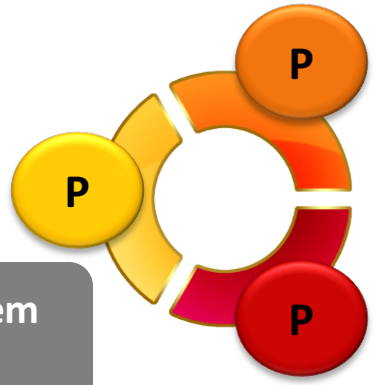
- Solution to IPC problem can be difficult to comprehend.
  - Usually, intuitive methods failed.
  - Depending on time, e.g., sleep(1), does not guarantee a useful solution.

- As a matter of fact, dining philosopher **is not restricted to 5 philosophers**.

- The midterm examination in 2009 has shown one of the usage of this IPC problem: **traffic control**!

Inter-process communication (IPC)

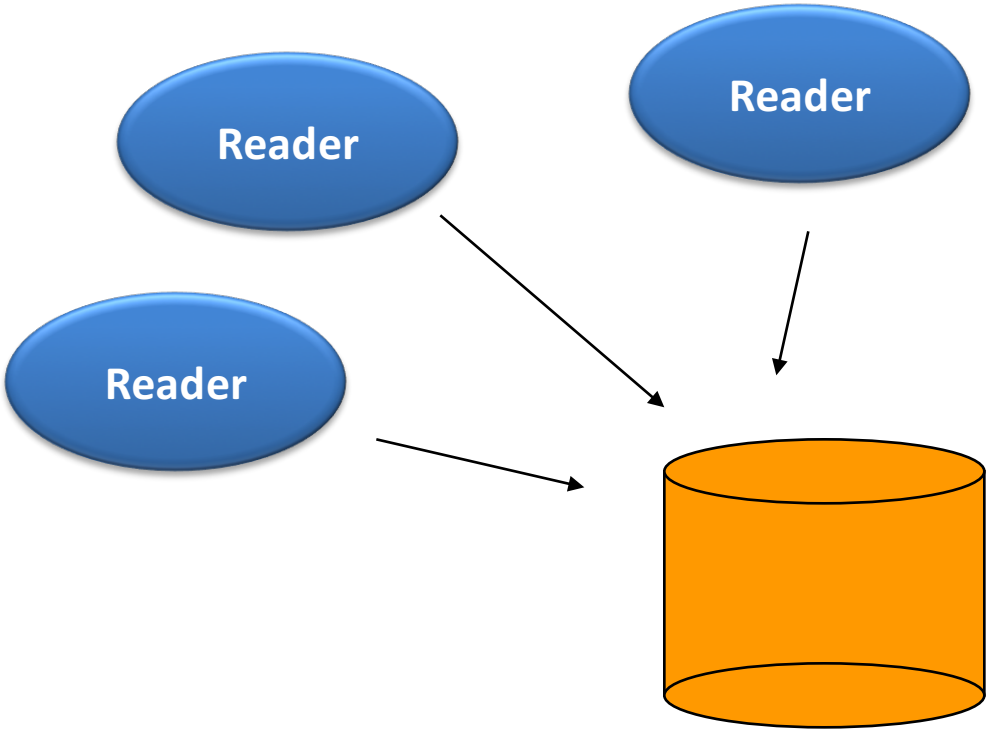
- Classic IPC problems.
  - Producer-consumer problem.
  - Dining philosopher problem.
  - Reader-writer problem.



Let's teach them not to fight.

Reader-writer problem – introduction

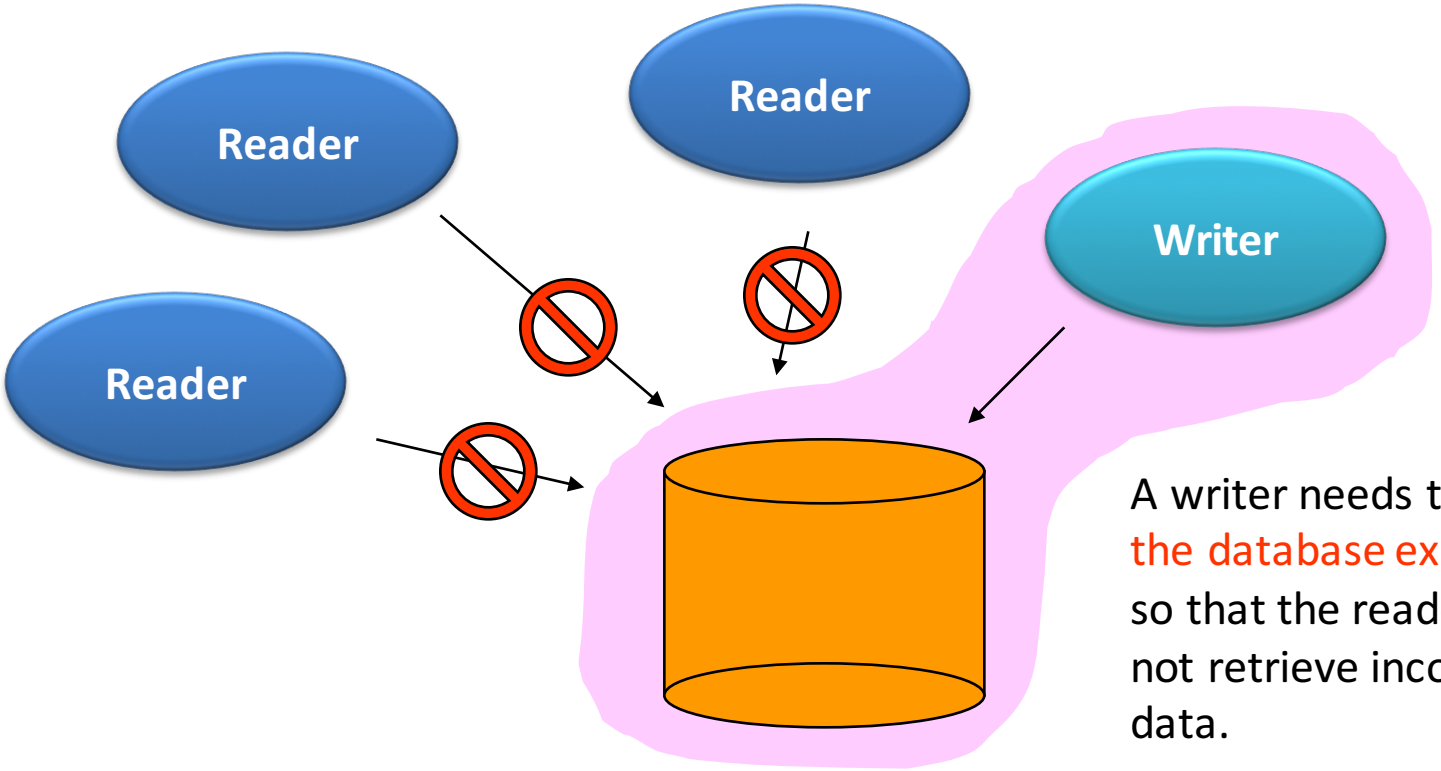
- It is a concurrent database problem.



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

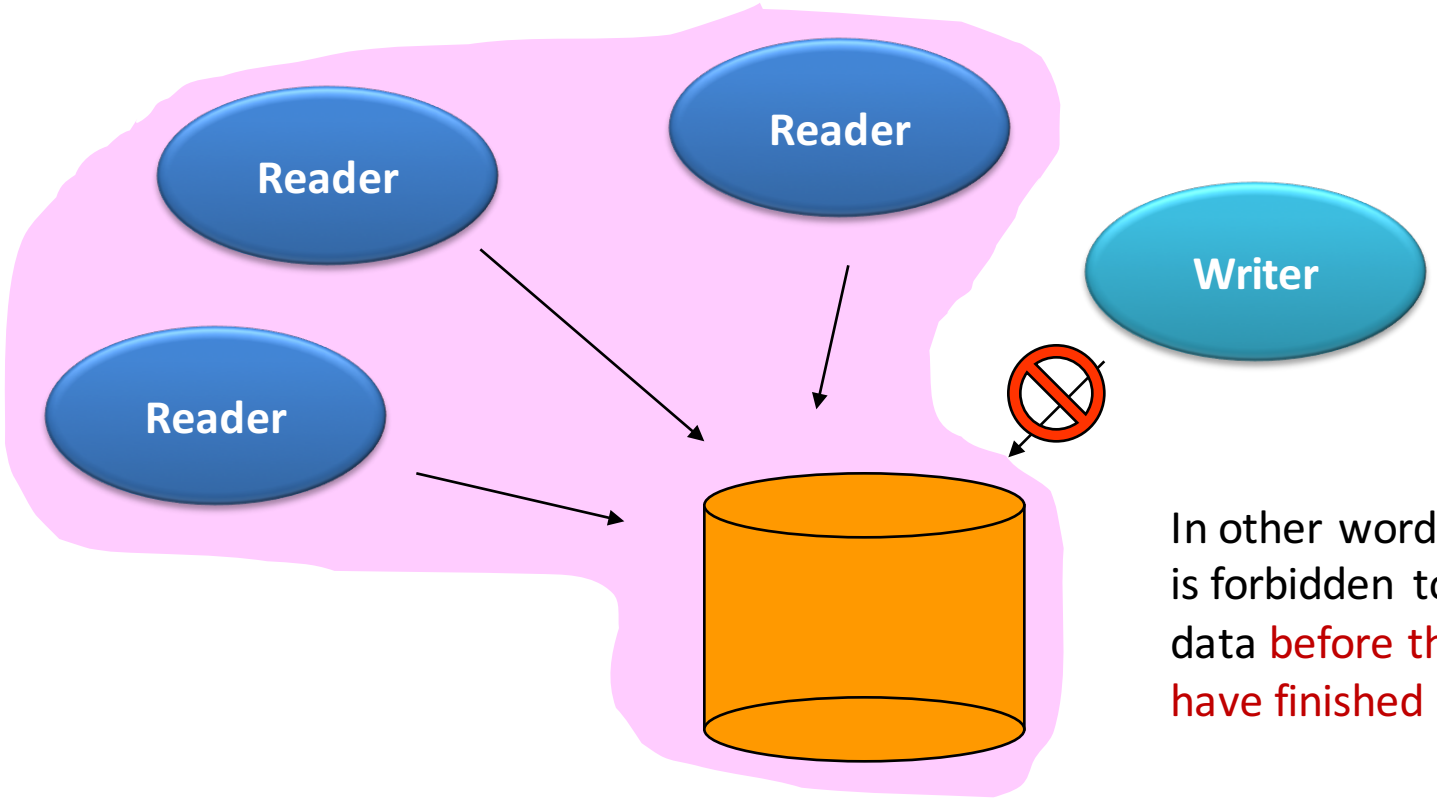
Reader-writer problem – introduction

- It is a concurrent database problem.



Reader-writer problem – introduction

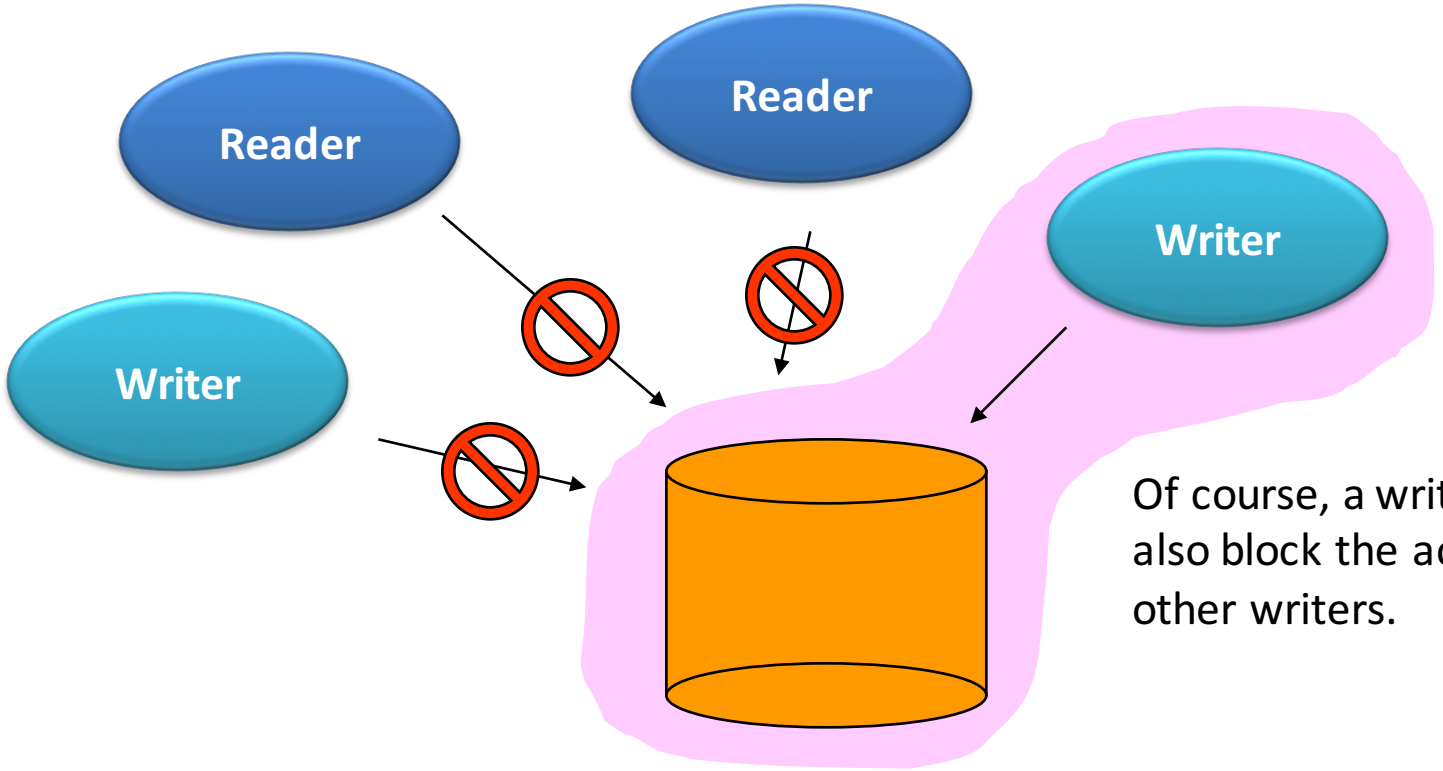
- It is a concurrent database problem.



In other words, a writer is forbidden to write any data before the readers have finished reading.

Reader-writer problem – introduction

- It is a concurrent database problem.



Reader-writer problem – introduction

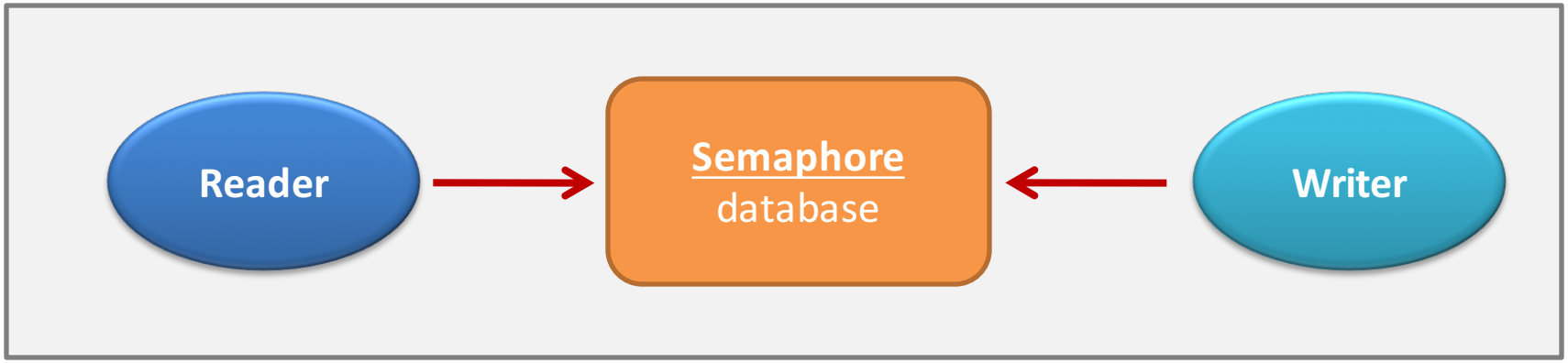
- A mutual exclusion problem.
  - The database is a shared object.

- 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.

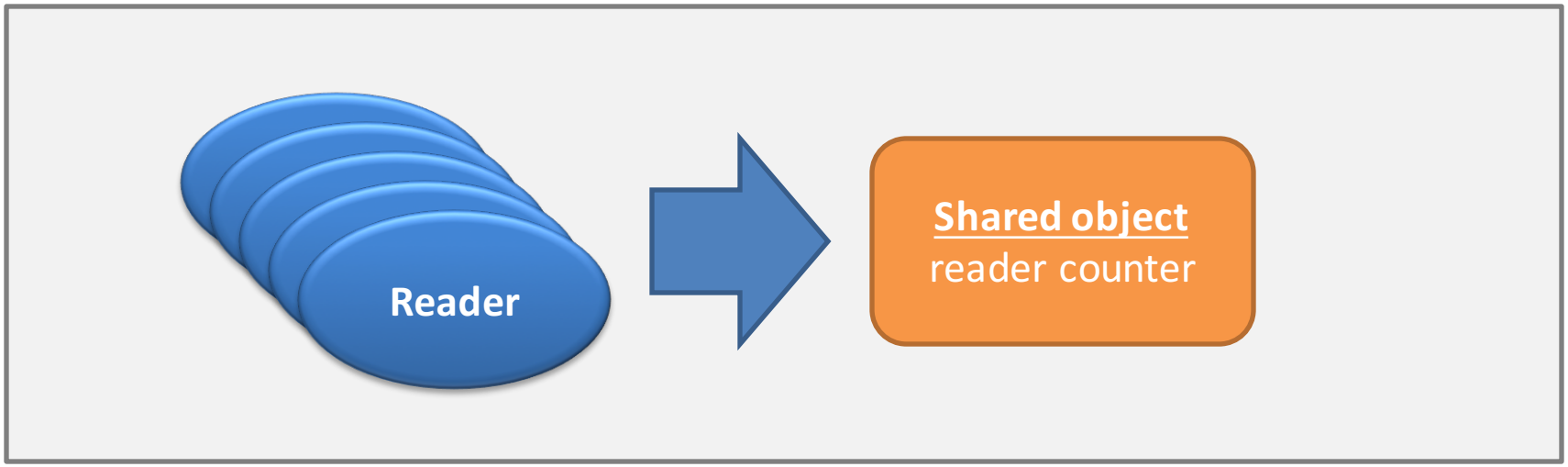
Reader-writer problem – solution outline

- **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) {
       prepare_write();
       down(&db);
       write_database();
       up(&db);
7      }
8  }
```

Section Entry

Critical Section

Section Exit

Reader Function

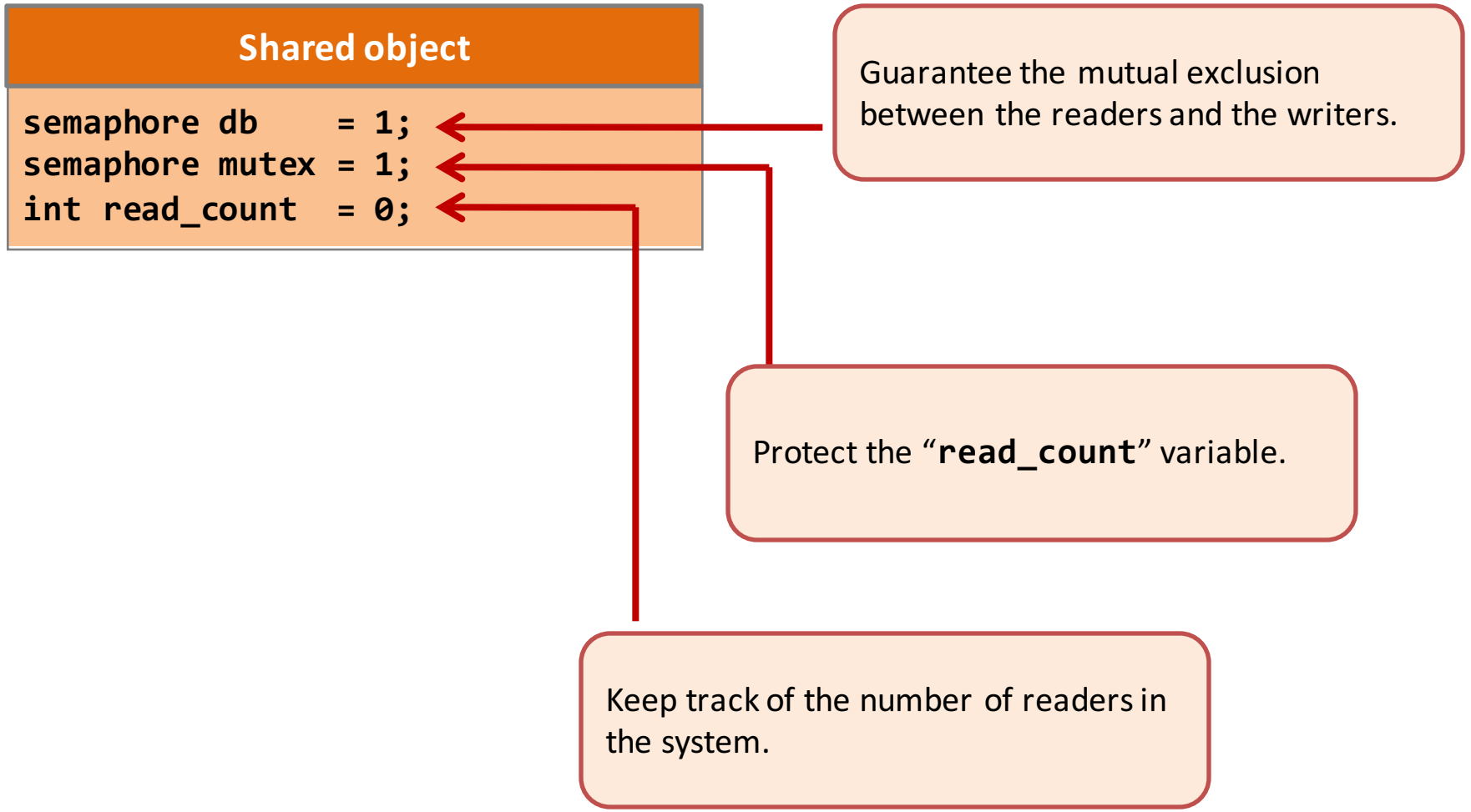
```
1  void reader(void) {
2      while(TRUE) {
       down(&mutex);
       read_count++;
5      if(read_count == 1)
6          down(&db);
7      up(&mutex);
       read_database();
       down(&mutex);
       read_count--;
11     if(read_count == 0)
12         up(&db);
13     up(&mutex);
14     process_data();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

Section Entry

Critical Section

Section Exit

Reader-writer problem – final solution



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 theirs critical section.

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 number of processes;
  - Processes have to be synchronized in order to generate useful output;
  - Each resource may be shared as well as limited, and there may be more than one shared processes.

- The synchronization algorithms have the following requirements in common:
  - Guarantee mutual exclusion;
  - Uphold the correct synchronization among processes; and
  - (must be) Deadlock-free.