Operating Systems

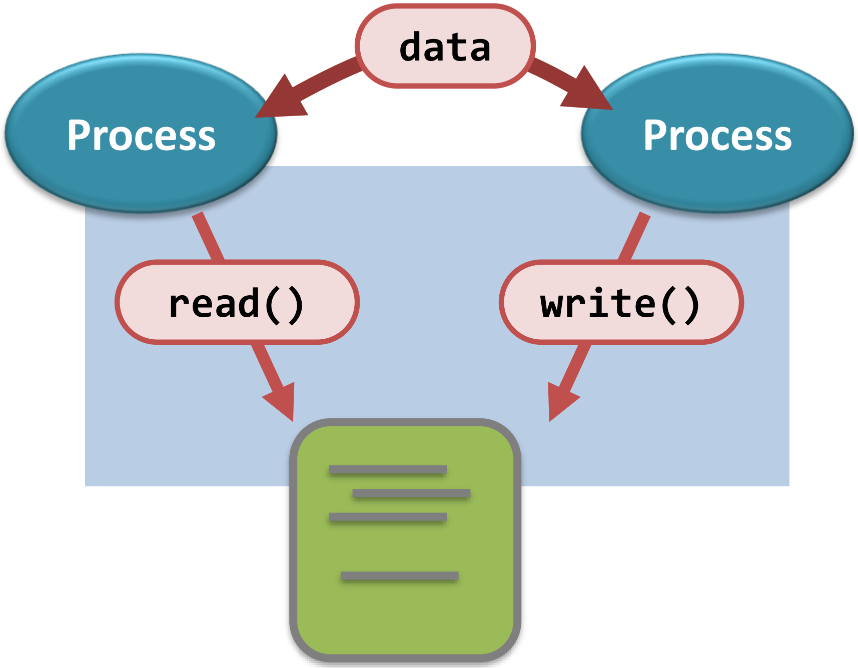
Associate Prof. Yongkun Li
中科大-计算机学院 副教授
http://staff.ustc.edu.cn/~ykli

Ch5
Process Communication & Synchronization
-Part 3

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**



Shared objects

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

Topics

Deadlock

- Concept
- Necessary conditions
- Characterization
- Solutions

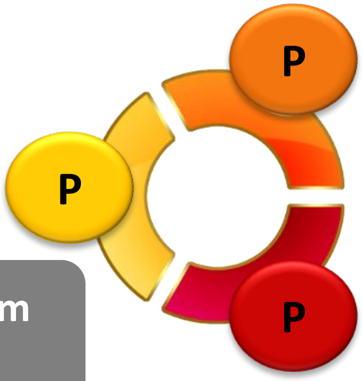
Classic problems

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

The Deadlock Problem

Classic IPC problems

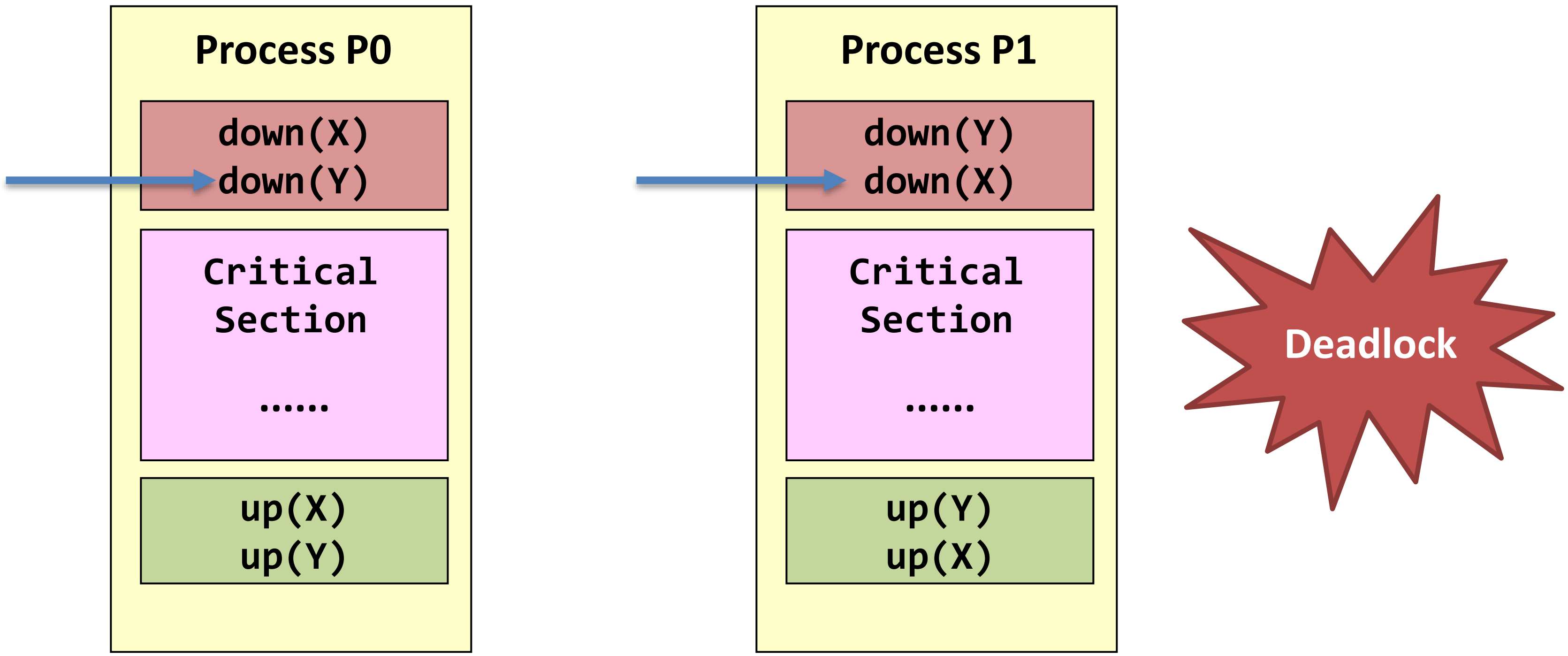
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem



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

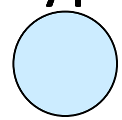
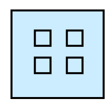
- **Requirement #1: Mutual Exclusion**.
  - Only one process at a time can use a resource
- **Requirement #2. Hold and wait.**
  - A process must be holding at least one resource and waiting to acquire additional resources held by other processes

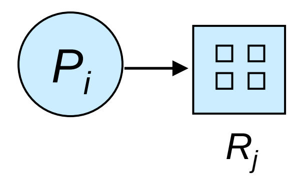
Deadlock Requirements

- **Requirement #3: No preemption.**
  - A resource can be released only voluntarily by the process holding it after that process has completed its task

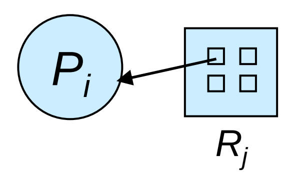
- **Requirement #4. Circular wait.**
  - There exists a set {P_0, P_1, …, P_n} of waiting processes such that P_0 waits for P_1, P_1 waits for P_2, …, P_{n-1} waits for P_n , P_n waits for P_0

How to Handle Deadlocks

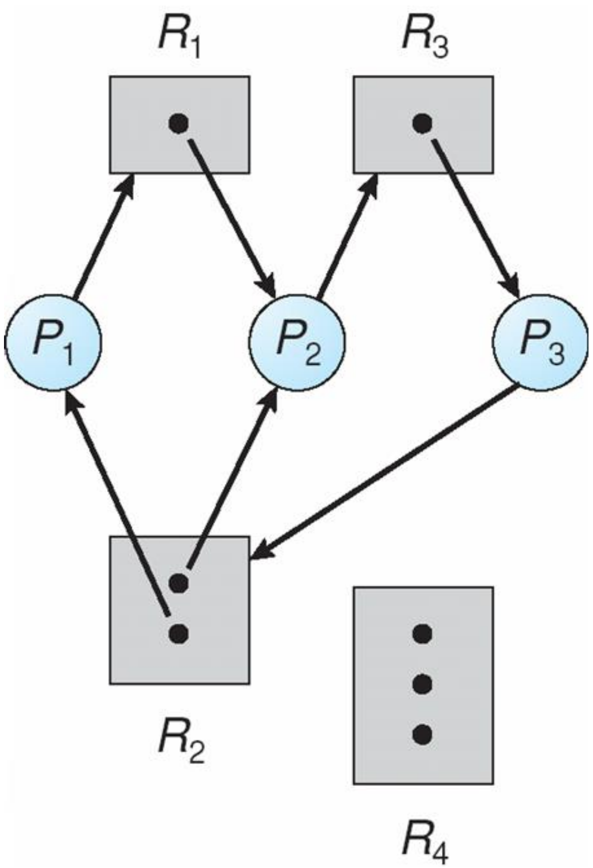
- Deadlock characterization: Deadlocks can be described using **resource-allocation graph**
  - Set V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$: processes 
    - $R = \{R_1, R_2, \ldots, R_m\}$: all resource types (each type may have multiple instances) 
  - Set E
    - **request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

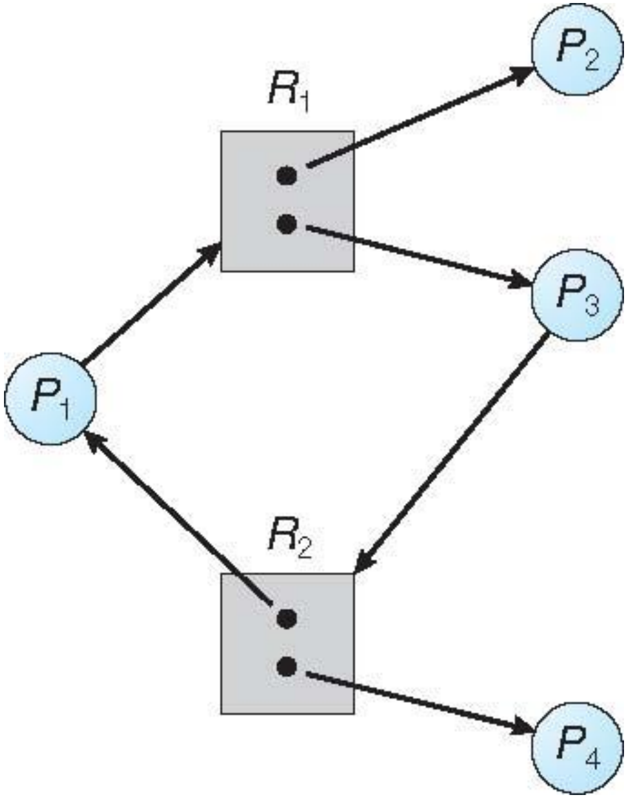


    - **assignment edge** – directed edge $R_j \rightarrow P_i$



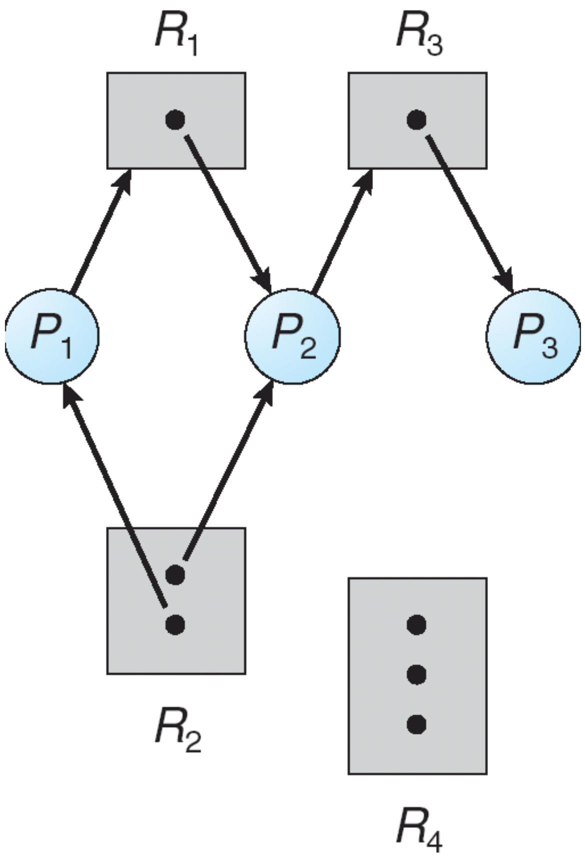
Examples





How to Handle Deadlocks

- **Detect** deadlock and recover
  - Case 1: Each resource has one instance
    - Resource-allocation graph: detect the existence of a cycle



No cycles

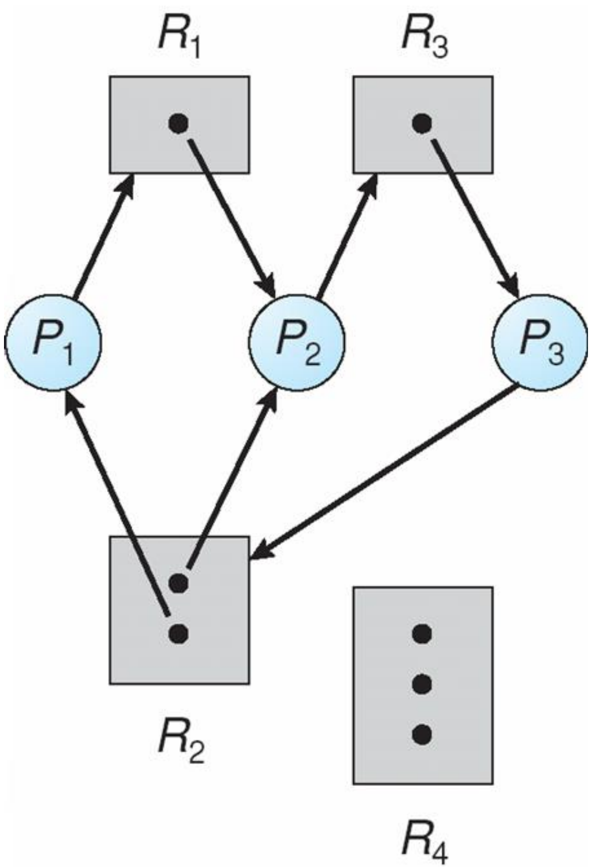
No deadlock

Contains a cycle

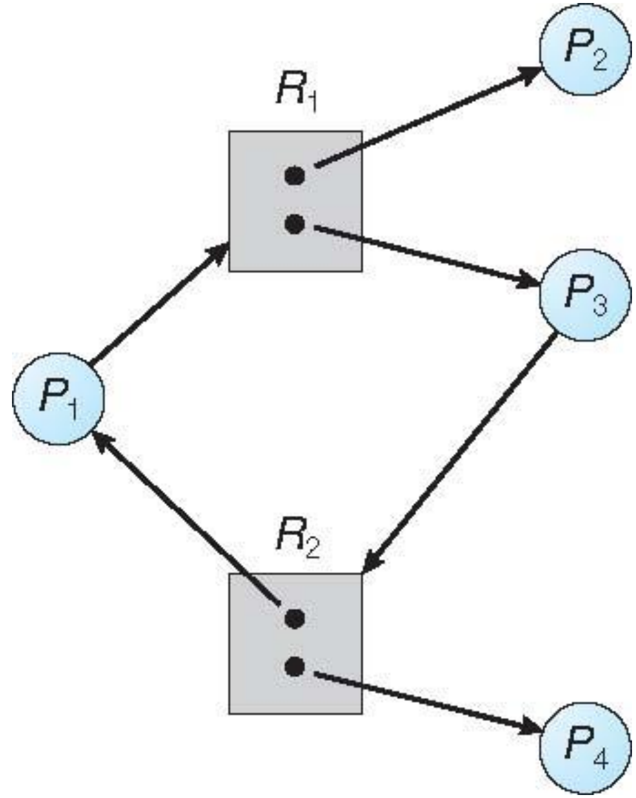
Case 1: only one instance per resource

Case 2: several instances per resource type: possible deadlock

Examples



Deadlock



No deadlock

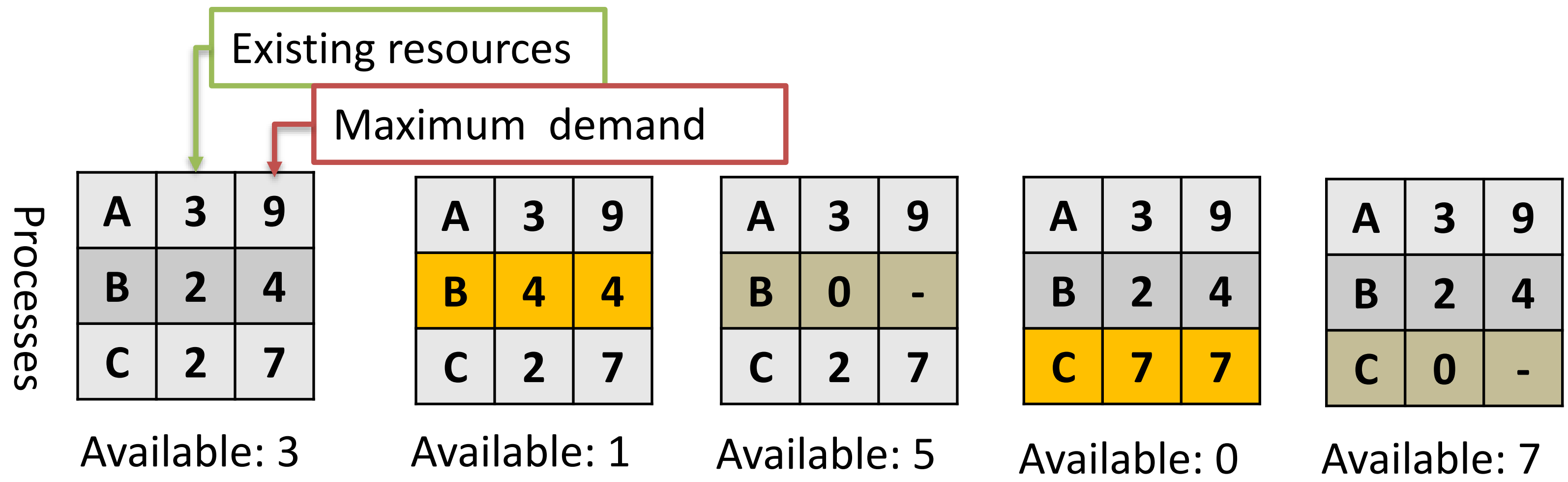
How to Handle Deadlocks

- **Detect** deadlock and recover
  - Case 2: Each resource has multiple instances
    - Matrix method: four data structures
      - Existing (total) resources (m types): $(E_1, E_2, \ldots, E_m)$
      - Available resources: $(A_1, A_2, \ldots, A_m)$
      - 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)

Existing resources

Maximum demand

Processes

A	1	6
B	0	5
C	2	4
D	4	7

Available: 3

Initial state

B requests one resource

Accept

A	1	6
B	1	5
C	2	4
D	4	7

Available: 2

Safe state

Running order: C D B A

B requests one resource

reject

A	1	6
B	2	5
C	2	4
D	4	7

Available: 1

Unsafe state

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

How to Handle Deadlocks

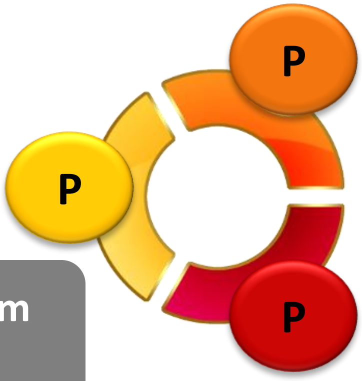
- **Ignore** the problem and pretend that deadlocks never occur (stop functioning and restart manually)
  - 鸵鸟算法（假装没发生）
  - Used by most operating systems, including UNIX and windows
  - Deadlocks occur infrequently, avoiding/detecting it is expensive

- 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?

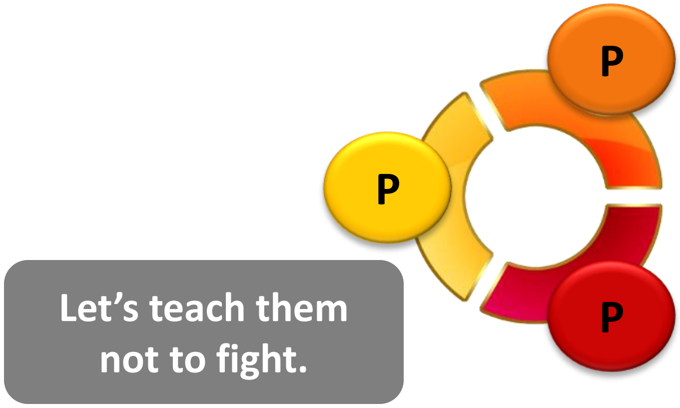
- All the IPC classical problems use **semaphores** to fulfill the synchronization requirements.

	Properties	Examples
Producer-Consumer Problem	Two classes of processes: **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.

The Deadlock Problem

Classic IPC problems

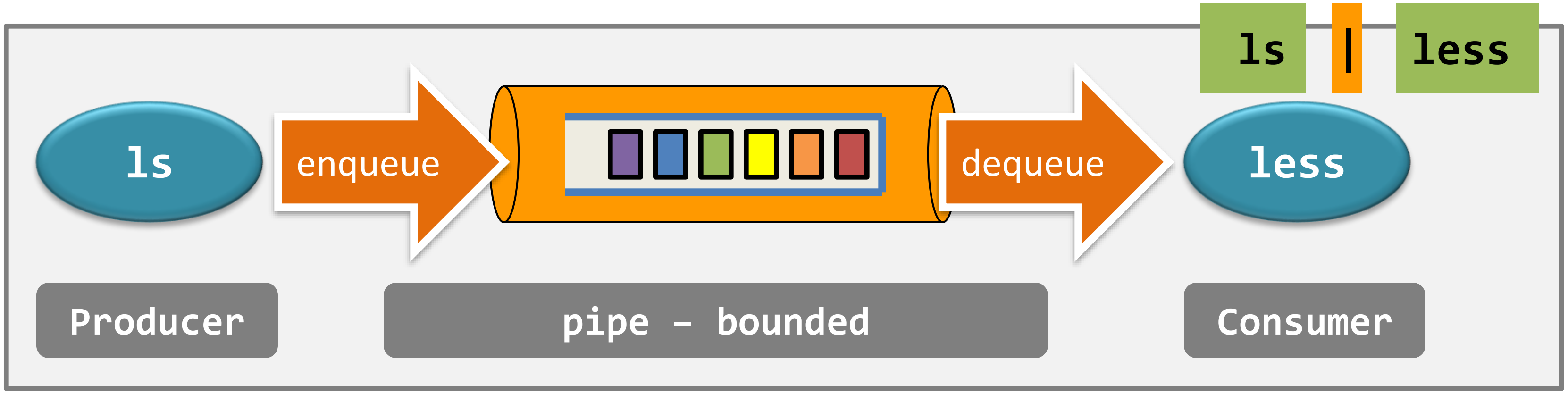
- **Producer-consumer problem**
- Dining philosopher problem
- Reader-writer problem



Producer-consumer problem – recall

- 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 – 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

- **<u>Pipe is working fine. Is it enough?</u>**
  - 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 **<u>the use of shared objects and semaphores</u>**, 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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        insert_item(item);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        item = remove_item();
        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();


        insert_item(item);


    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {


        item = remove_item();


        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();


        insert_item(item);


    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {


        item = remove_item();


        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();

        down(&mutex);
        insert_item(item);
        up(&mutex);

    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {

        down(&mutex);
        item = remove_item();
        up(&mutex);

        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();

        down(&mutex);
        insert_item(item);
        up(&mutex);

    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

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

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

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;
```

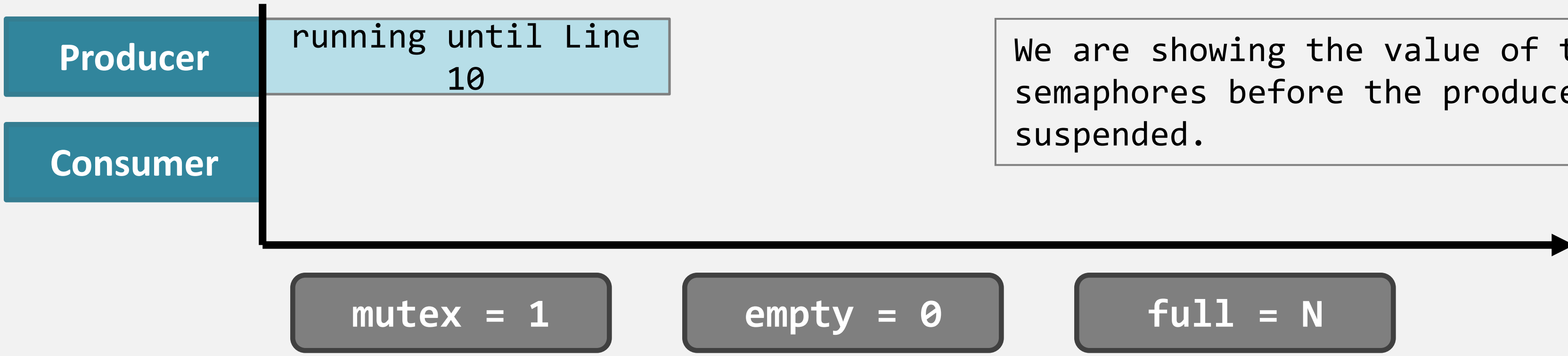
Producer function

```
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

```
void consumer(void) {
    int item;

    while(TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

Producer-consumer problem – question



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

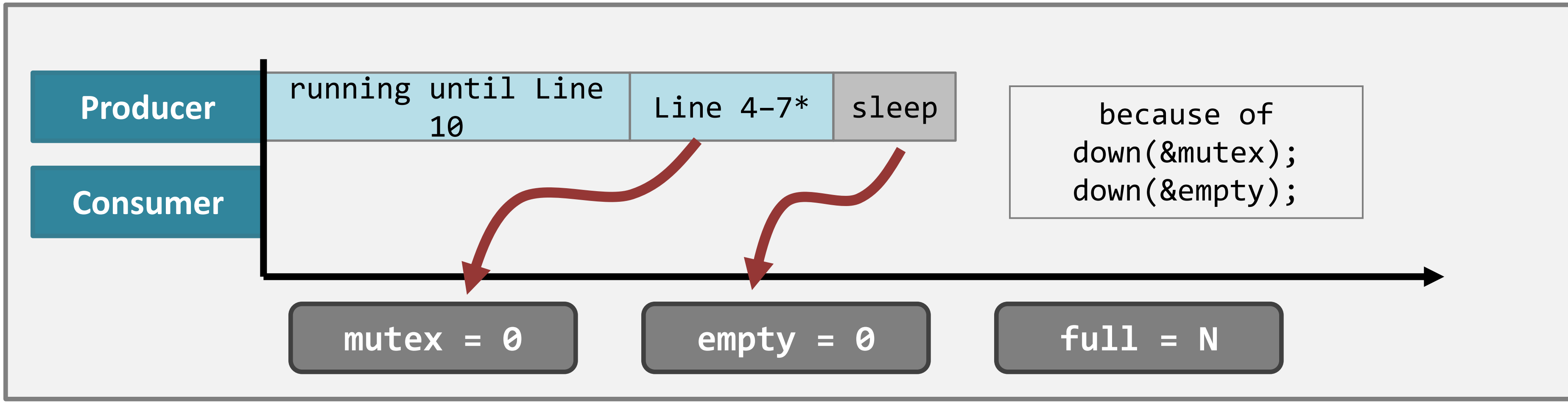
Producer function

```
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



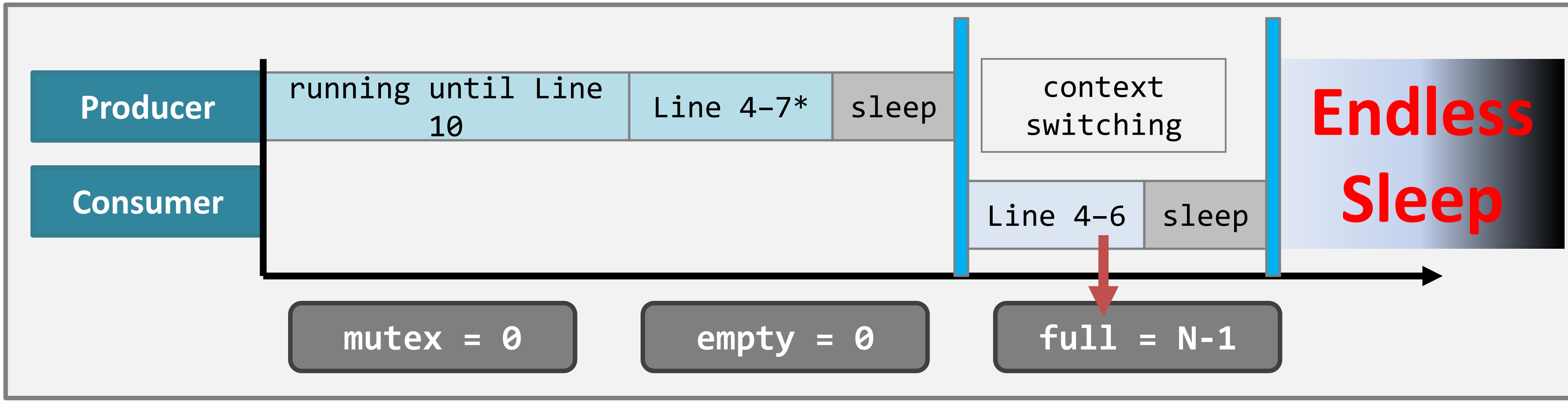
Producer function

```
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

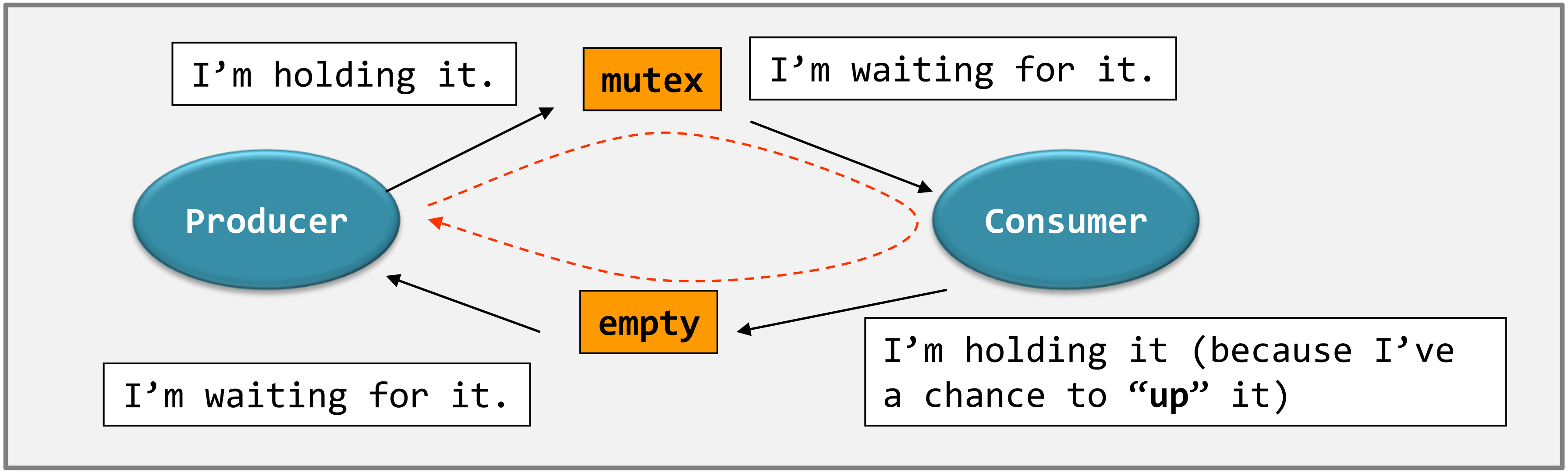
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          down(&mutex);
7*          down(&empty);
8           insert_item(item);
9           up(&mutex);
10          up(&full);
11      }
12  }
```

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.



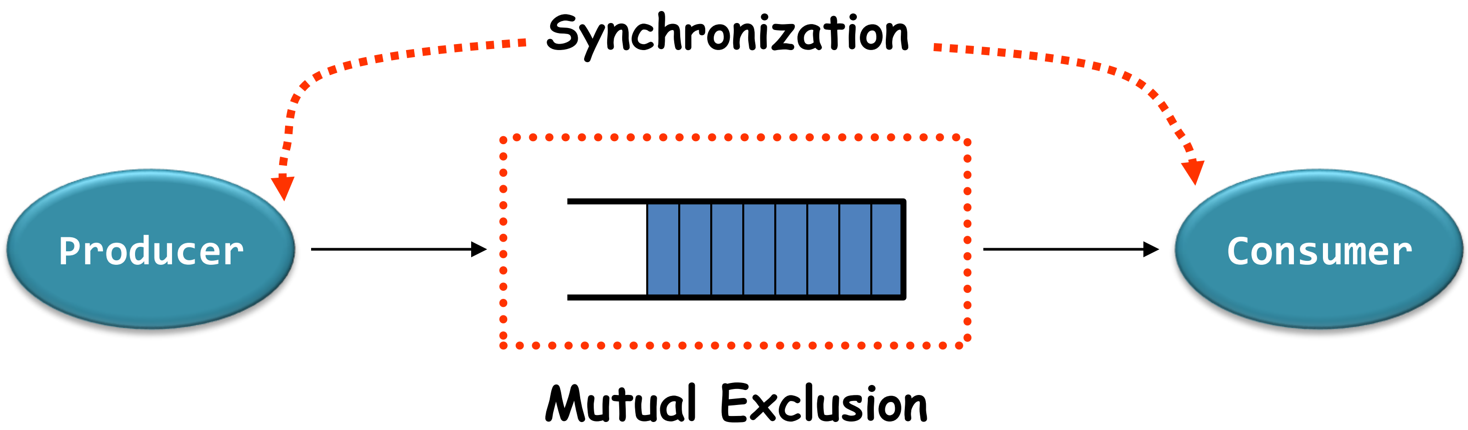
Producer-consumer problem

- **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.



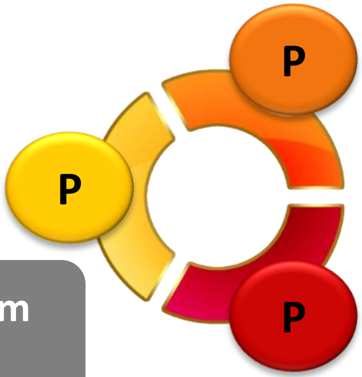
Summary on producer-consumer problem

- 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

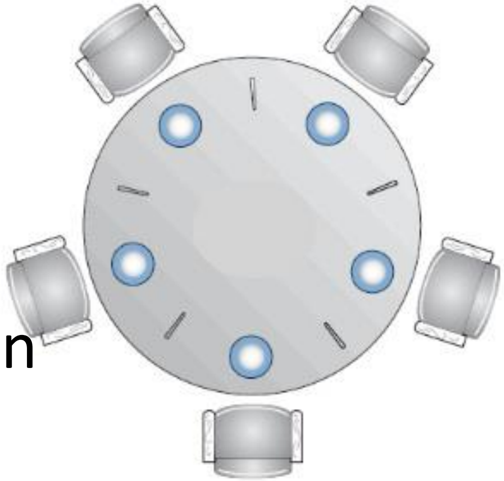
- Producer-consumer problem
- **Dining philosopher problem**
- Reader-writer problem



Let’s teach them not to fight.

Dining philosopher – introduction

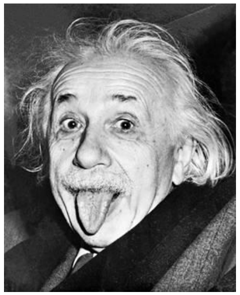
- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks.



- 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

Process

Chopsticks

Shared Object

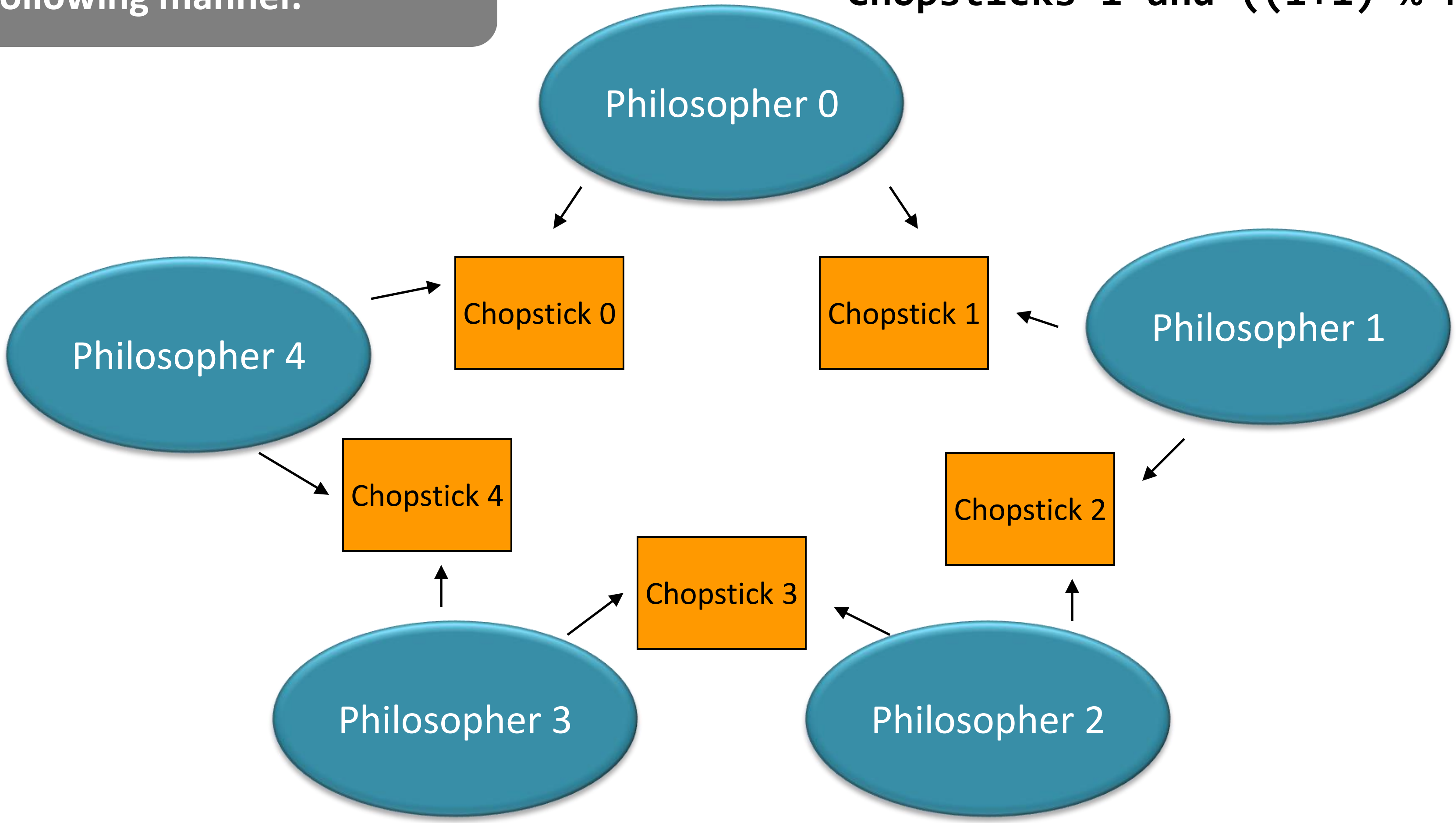
Spaghetti

Consider to have infinite supply.

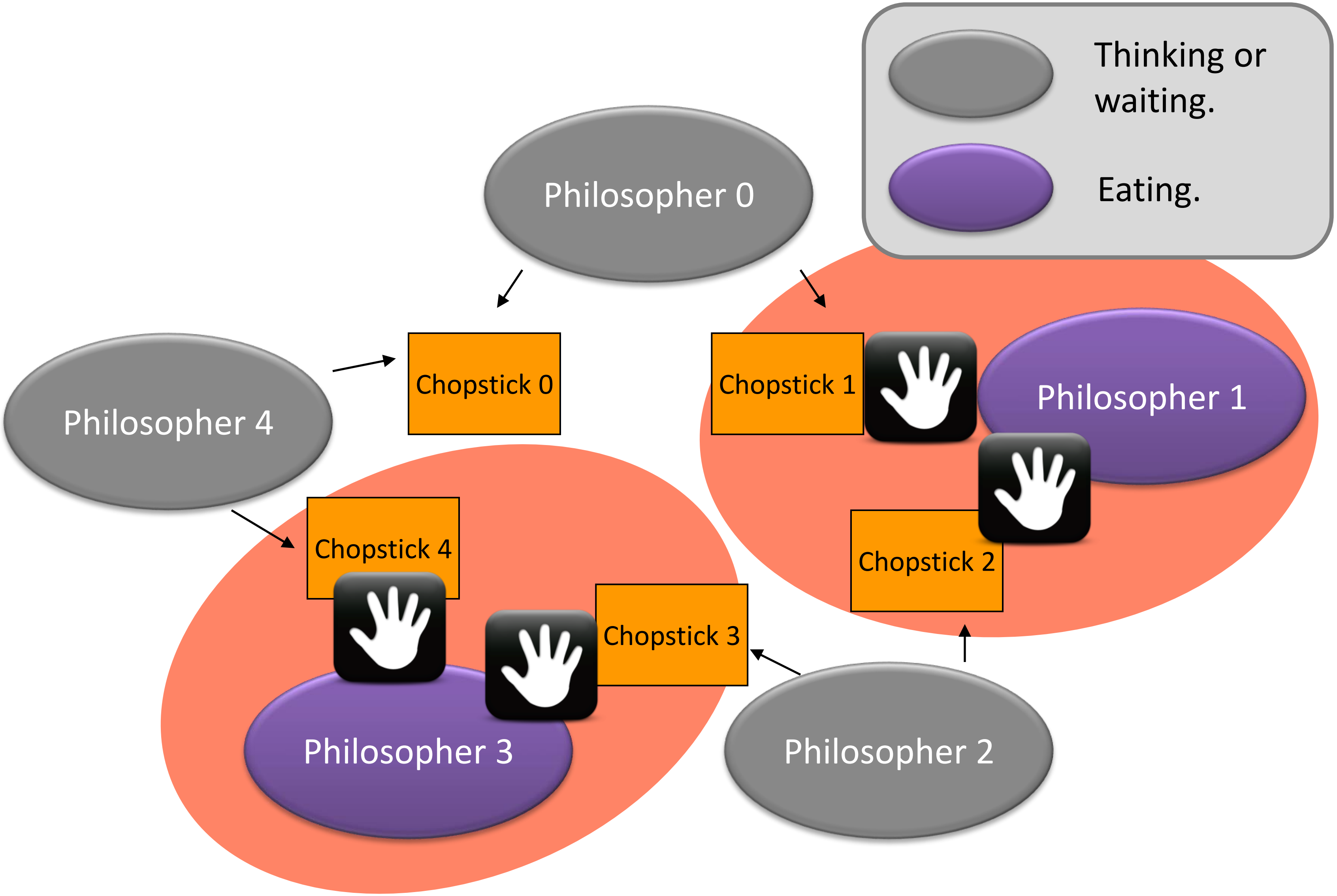
Dining philosopher – introduction

The chopsticks are arranged in the following manner.

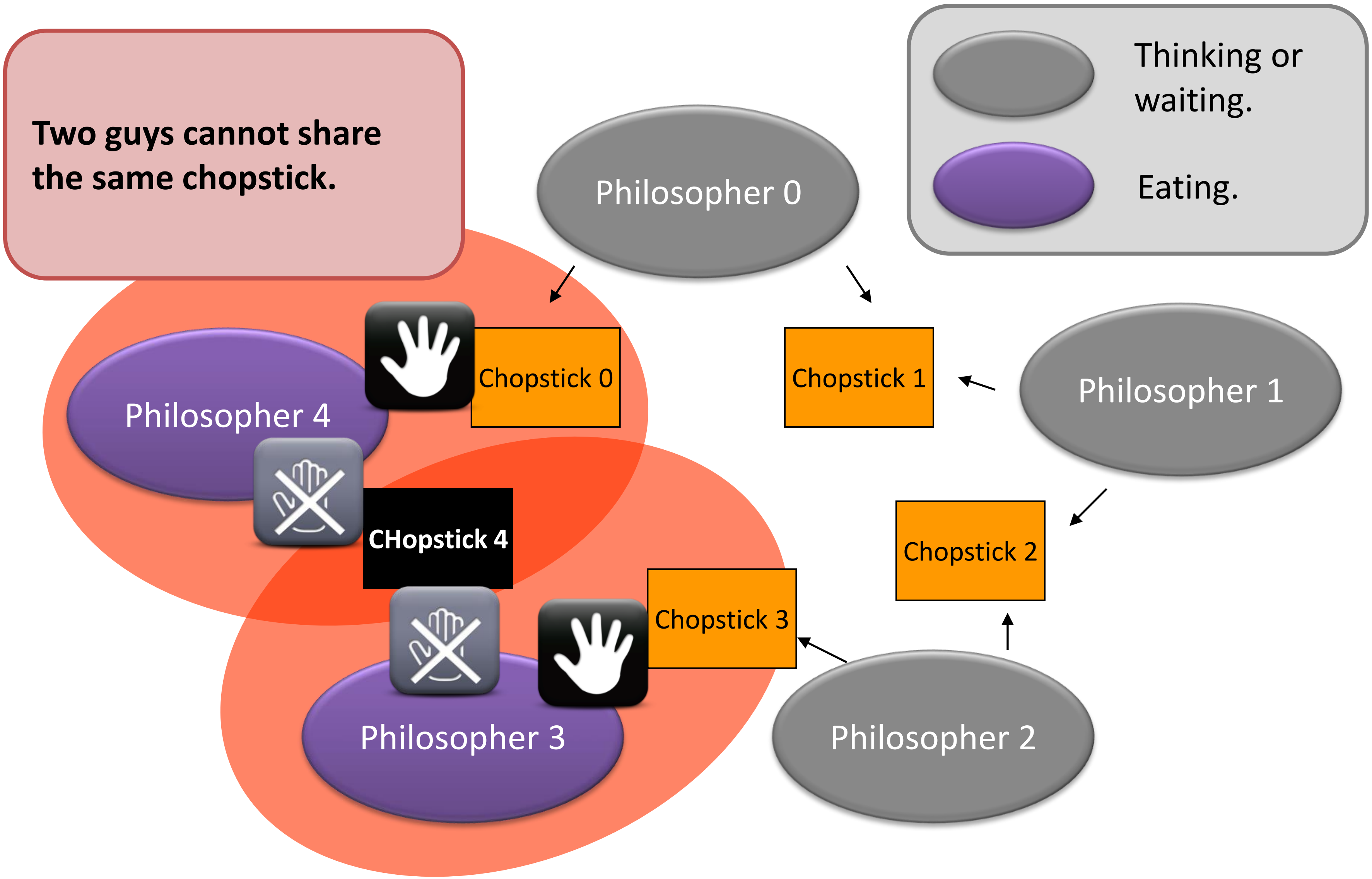
```
Philosopher i needs
Chopsticks i and ((i+1) % N);
```



Dining philosopher – introduction



Dining philosopher – introduction



Dining philosopher – requirement #1

- **<u>Mutual exclusion</u>**
  - 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 proposal 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]);
}
```

Section Entry

Critical Section

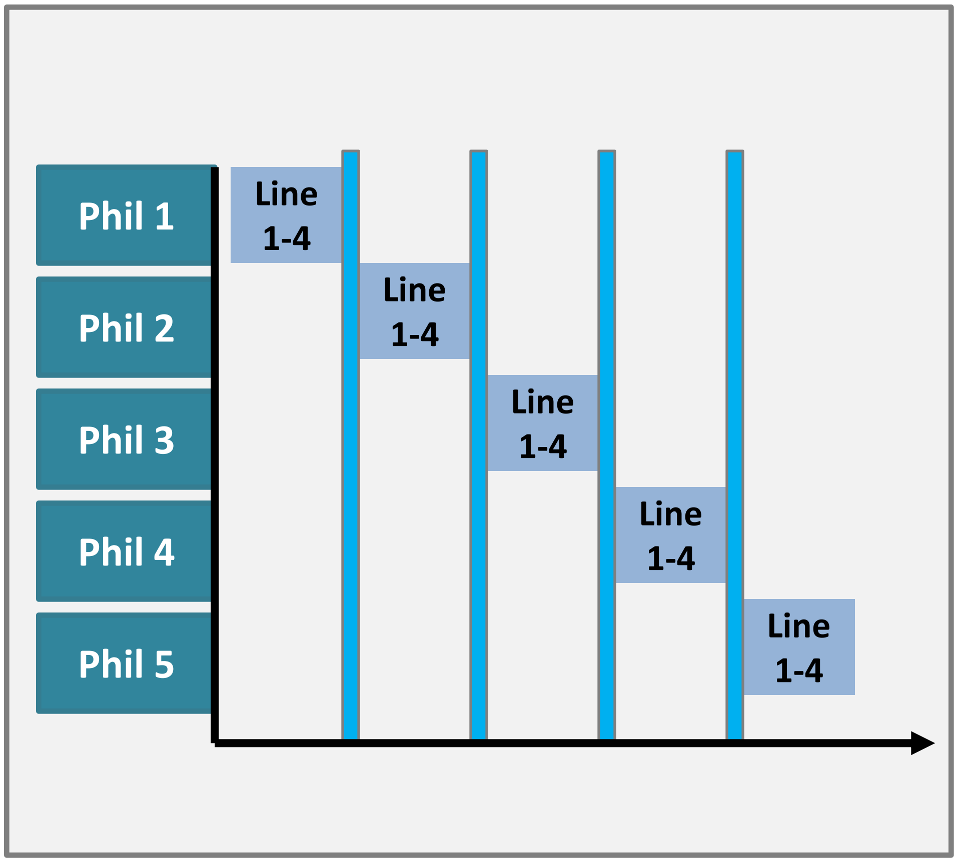
Section Exit

Main Function

```
void philosopher(int i) {
    while (TRUE) {
        think();

        take(i);
        take((i+1) % N);

        eat();

        put(i);
        put((i+1) % N);
    }
}
```

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?

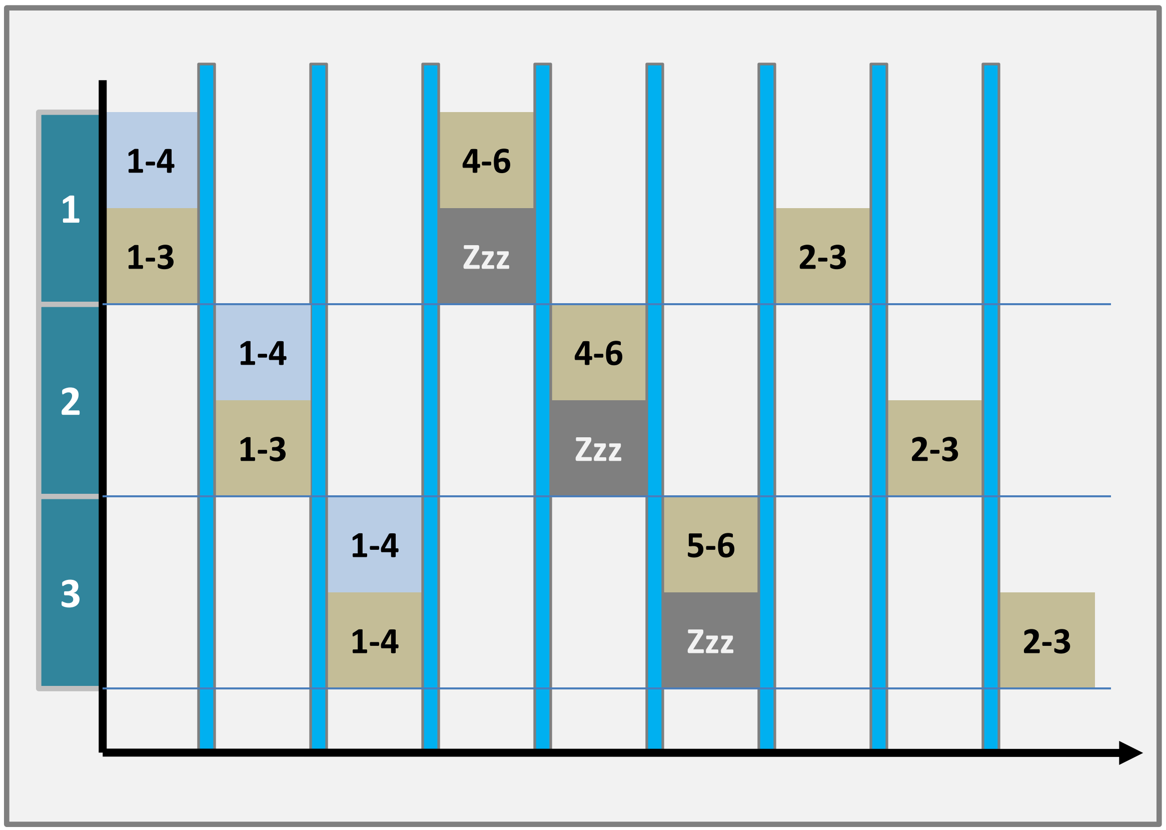
The code: meeting requirement #2?

```
void philosopher(int i) {
    while (TRUE) {
      think();
      take(i);
      eat();
      up(&chop[i]);
      up(&chop[(i+1)%N]);
    }
}
```

```
void take(int i) {
  while(TRUE) {
    down(&chop[i]);
    if (isUsed((i+1)%N)) {
      up(&chop[i]);
      sleep(1);
    }
    else {
      down(&chop[(i+1)%N]);
      break;
    }
  }
}
```

Dining philosopher – meeting requirement #2?

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



Assume N = 3 (because the space is limited)

```
void take(int i) {
  while(TRUE) {
    down(&chop[i]);
    if (isUsed((i+1)%N)) {
      up(&chop[i]);
      sleep(1);
    }
    else {
      down(&chop[(i+1)%N]);
      break;
    }
  }
}
```

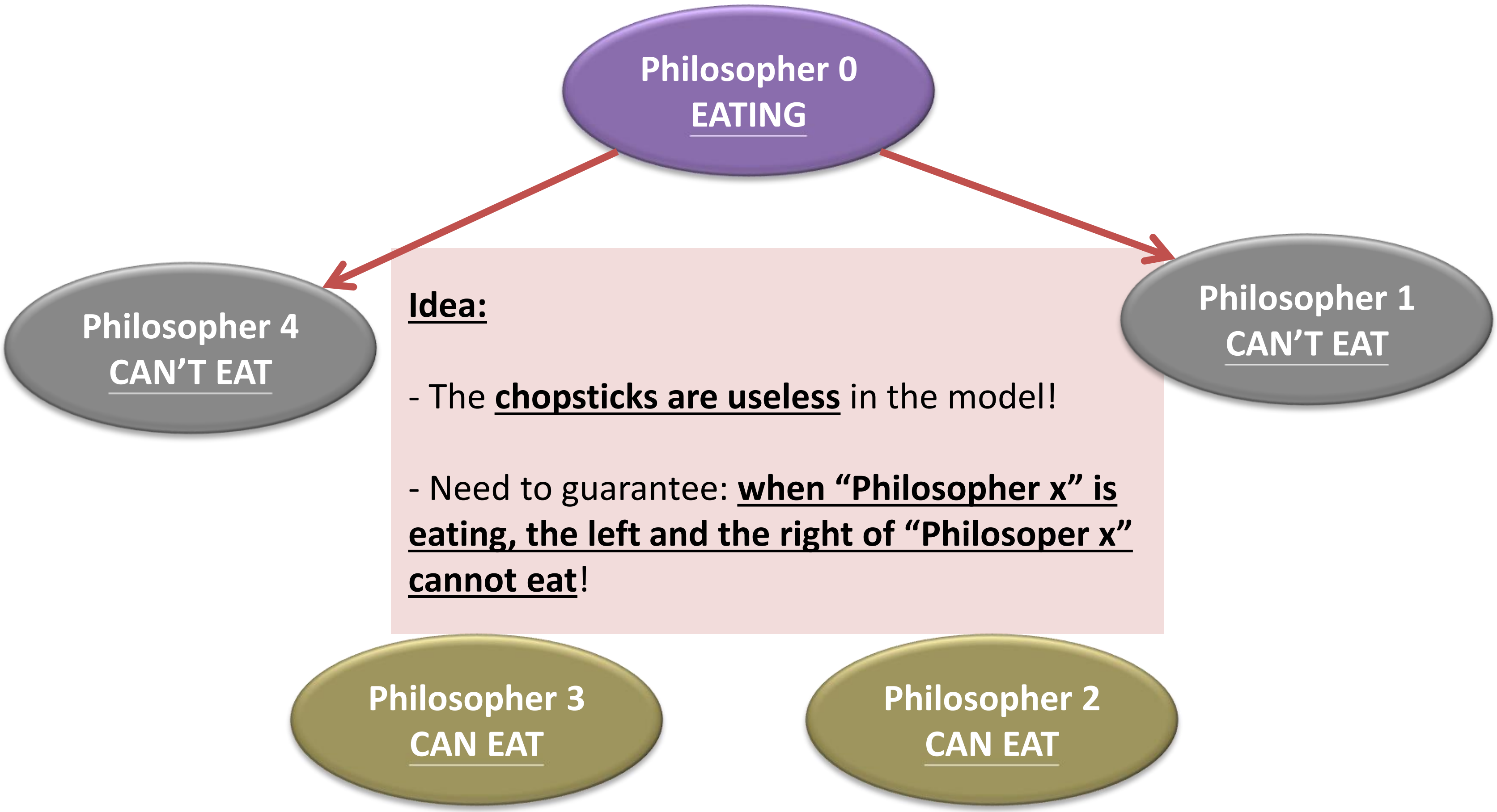
```
void philosopher(int i) {
    while (TRUE) {
      think();
      take(i);
      eat();
      up(&chop[i]);
      up(&chop[(i+1)%N]);
    }
}
```

Dining philosopher – before the final solution.

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

Problems
<u>Model a chopstick as a semaphore is intuitive, but is not working.</u> The problem is that we are afraid to "**`down()`**", as that may lead to a deadlock.
<u>Using `sleep()` to avoid deadlock is effective, yet bringing another problem.</u> We can always create an execution order that keeps all the philosophers busy, but without useful output.

Dining philosopher – before the final solution.



Idea:

- The **chopsticks are useless** in the model!

- Need to guarantee: **when "Philosopher x" is eating, the left and the right of "Philosoper x" cannot eat**!

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

```
void philosopher(int i) {
    think();
    take(i);
    eat();
    put(i);
}
```

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

<u>I will explain the code later.</u>

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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.

<u>Guess:</u>

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

To fulfill the synchronization requirement.

<u>Question.</u> 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

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Question. What are they doing?

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

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

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

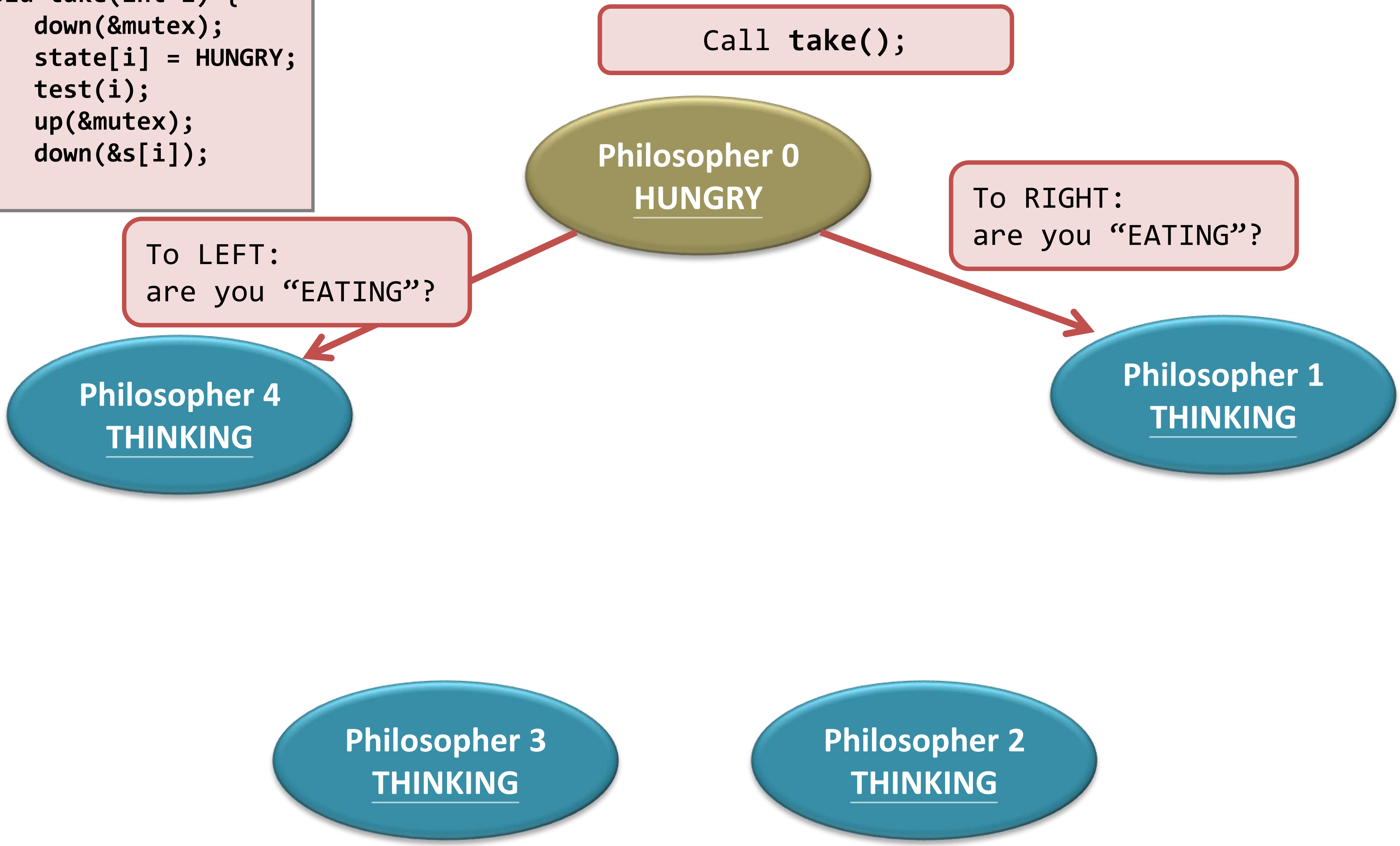
Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

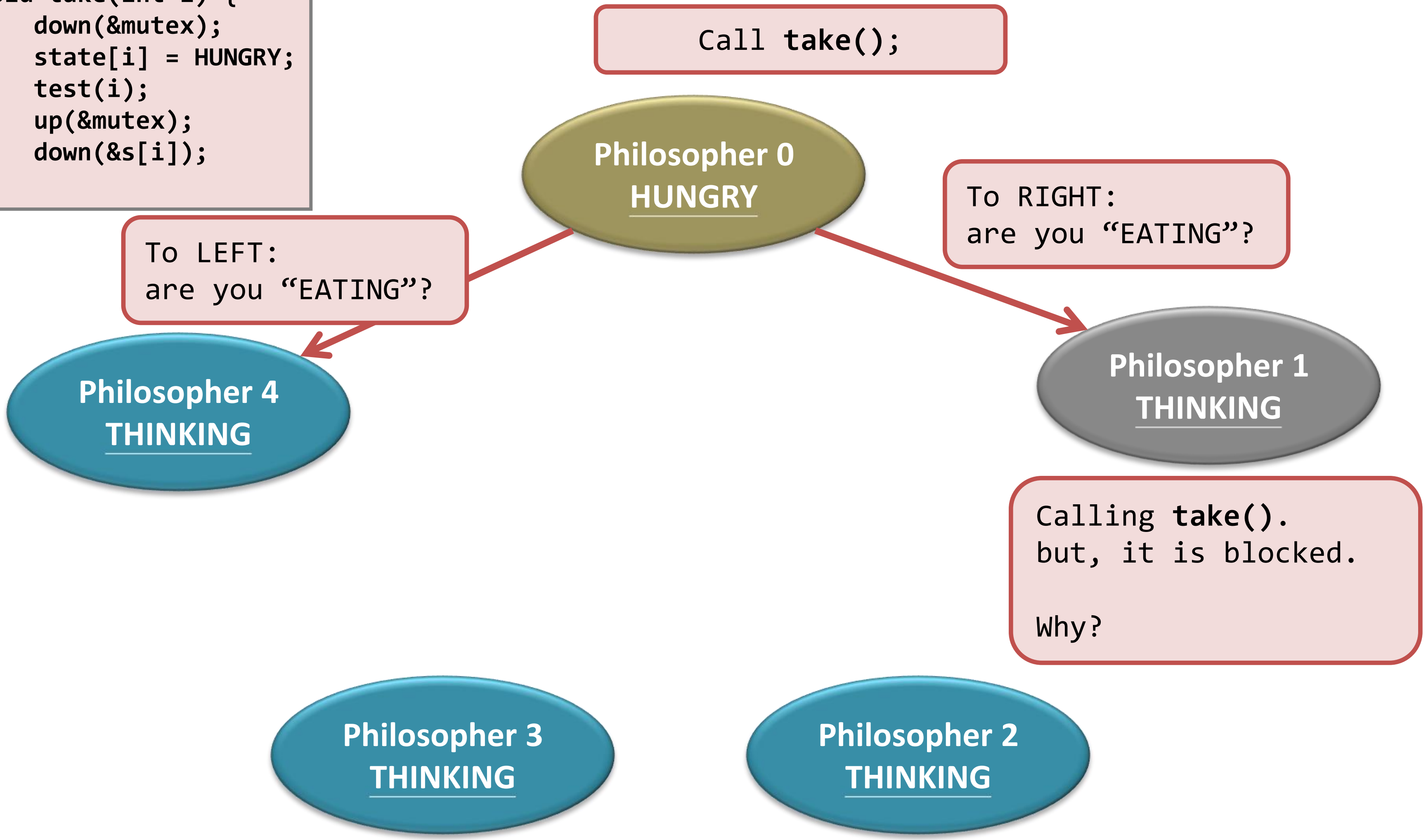
```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

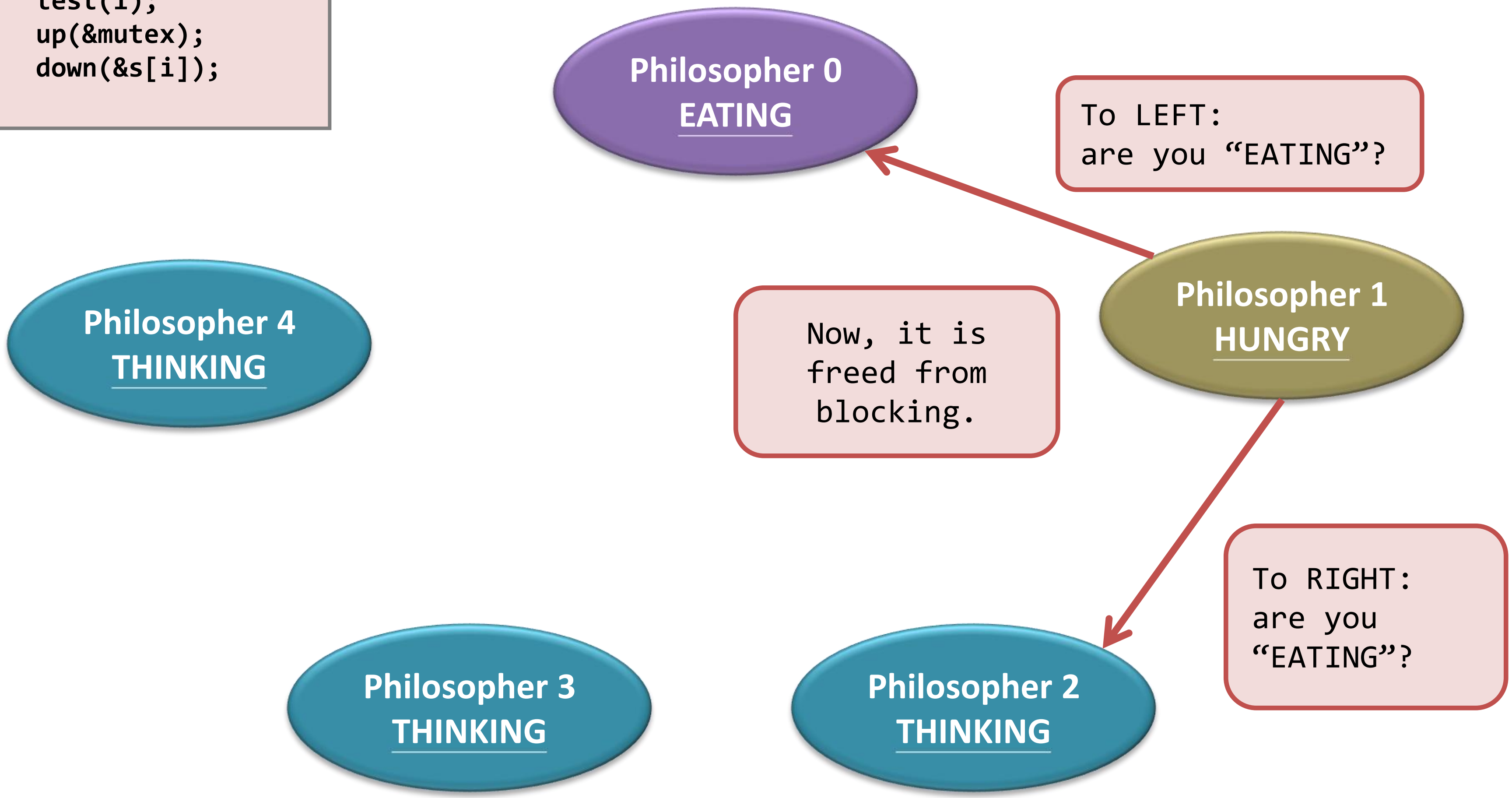
```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

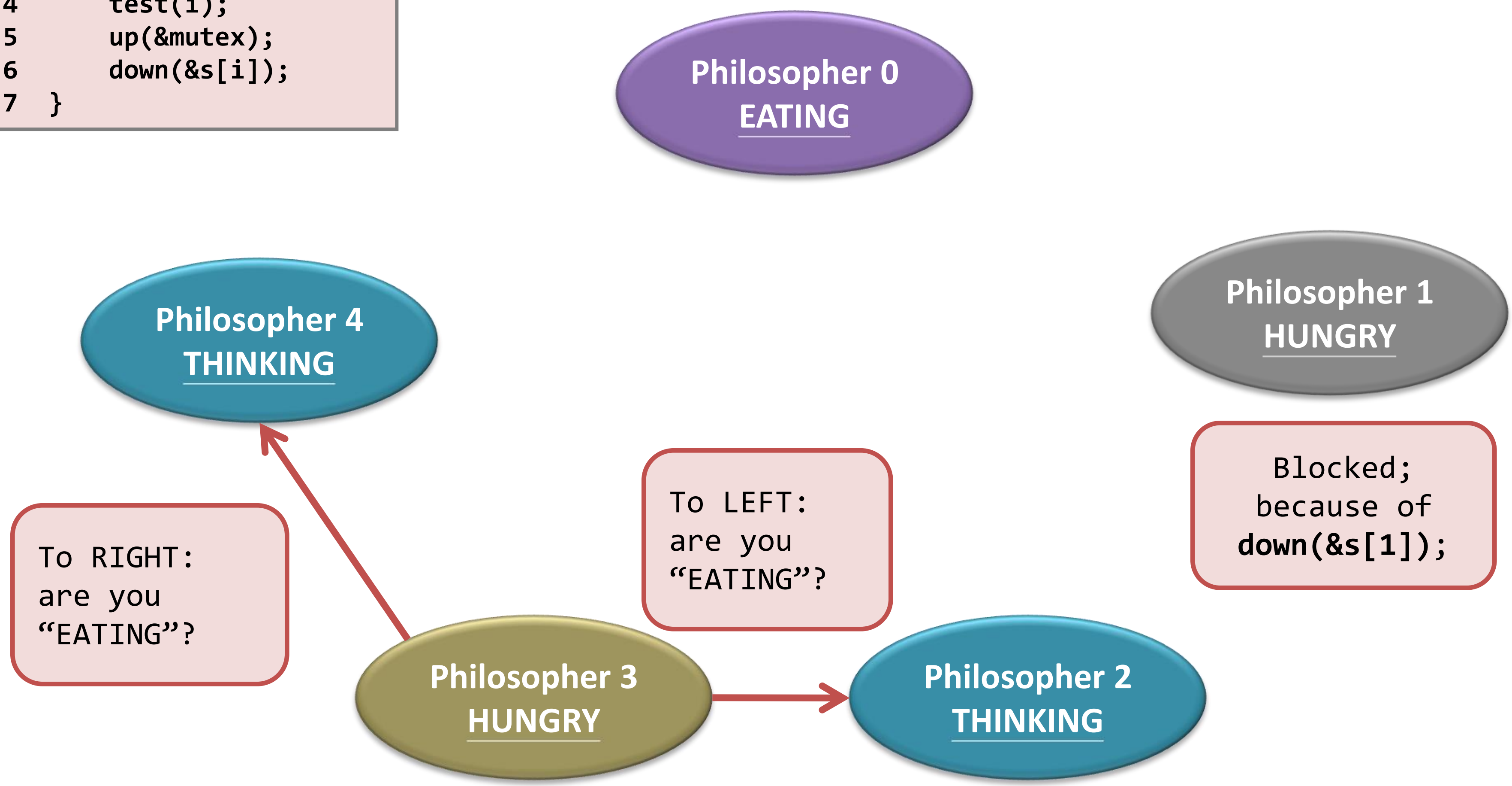
```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

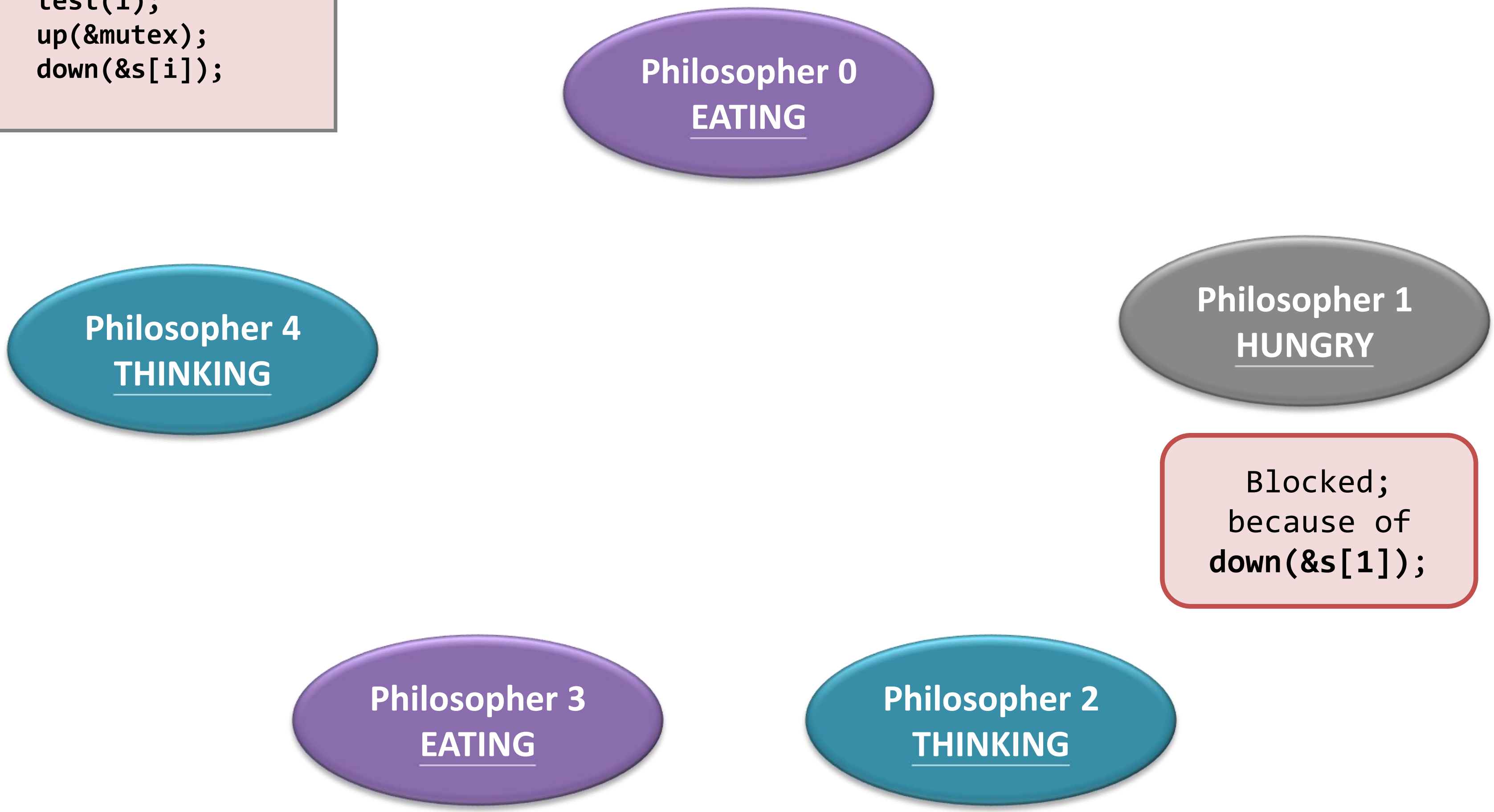
```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```



Dining philosopher – the final solution.

Section entry

```
void take(int i) {
     down(&mutex);
     state[i] = HUNGRY;
     test(i);
     up(&mutex);
     down(&s[i]);
}
```

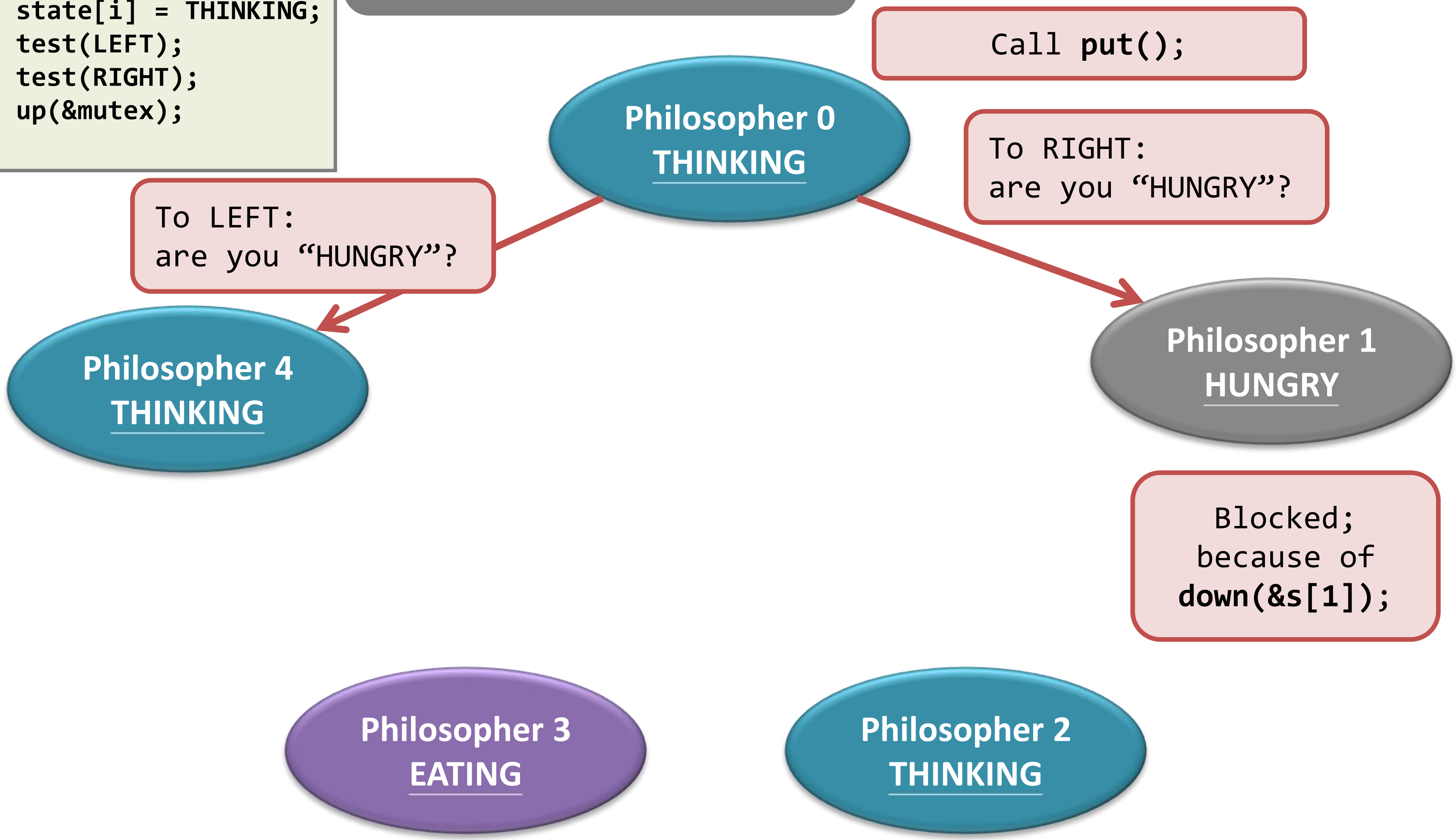


Dining philosopher – the final solution.

Section exit

```
void put(int i) {
     down(&mutex);
     state[i] = THINKING;
     test(LEFT);
     test(RIGHT);
     up(&mutex);
}
```

An illustration: How can Philosopher 1 start eating?

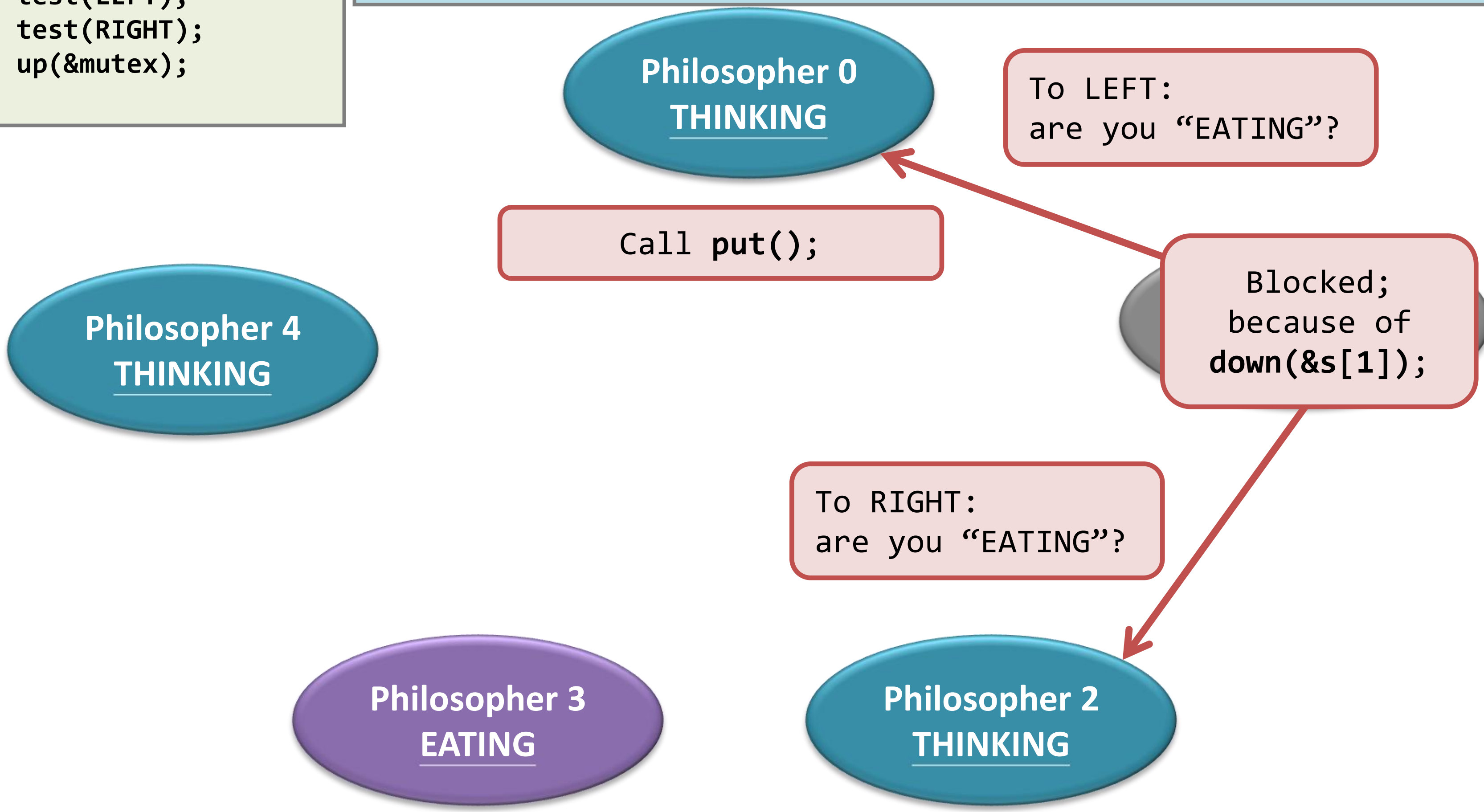


Dining philosopher – the final solution.

Section exit

```
void put(int i) {
      down(&mutex);
      state[i] = THINKING;
      test(LEFT);
      test(RIGHT);
      up(&mutex);
}
```

```
void test(int i) {
     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```



Dining philosopher – the final solution.

Section exit

```
void put(int i) {
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}
```

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Philosopher 0
THINKING

Call **put();**

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

Philosopher 4
THINKING

Remove your blocked state by calling **up(&s[1]);**

Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

共享

与i在sharing里才改变

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

`void take(int i) {`
`    down(&mutex);`
`    state[i] = HUNGRY;`
`    test(i);`
`    up(&mutex);`
`    down(&s[i]);`
`}`
	`void put(int i) {`
	`    down(&mutex);`
	`    state[i] = THINKING;`
	`    test(LEFT);`
	`    test(RIGHT);`
	`    up(&mutex);`
	`}`

Dining philosopher - summary

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

Section entry

```
void take(int i) {
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}
```

Extremely important helper function

```
void test(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Dining philosopher - summary

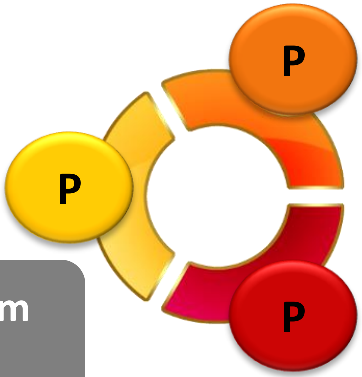
- Solution to IPC problem can be difficult to comprehend.
  - Usually, intuitive methods failed.
  - Depending on time, e.g., sleep(1), does not guarantee a useful solution.

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

The Deadlock Problem

Classic IPC problems

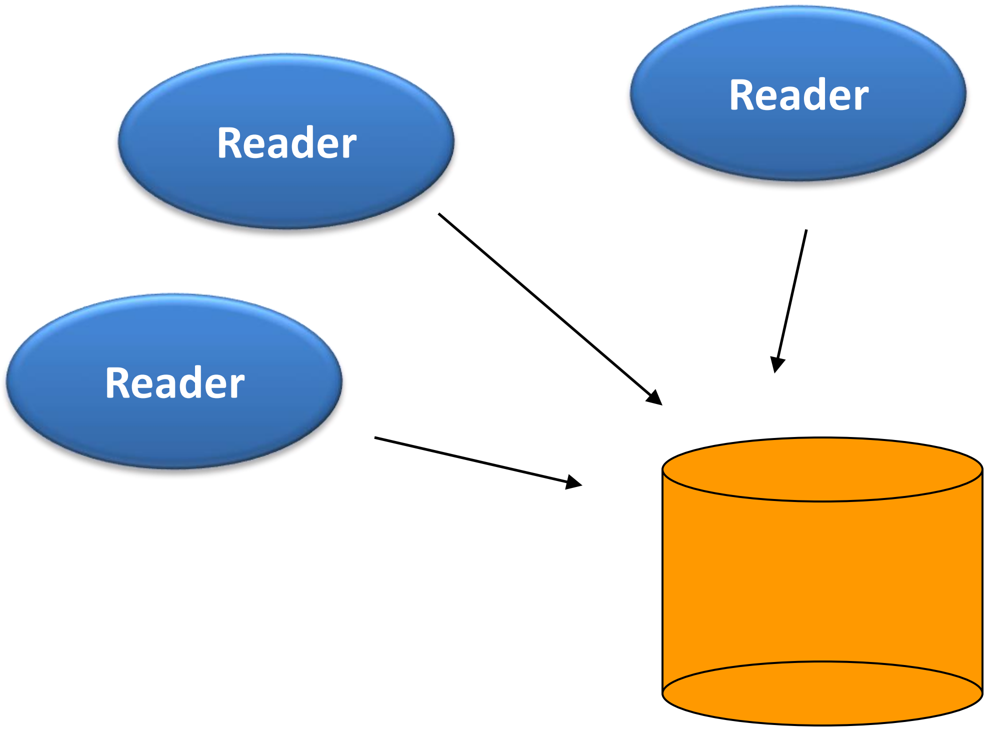
- Producer-consumer problem
- Dining philosopher problem
- **Reader-writer problem**



Let's teach them not to fight.

Reader-writer problem – introduction

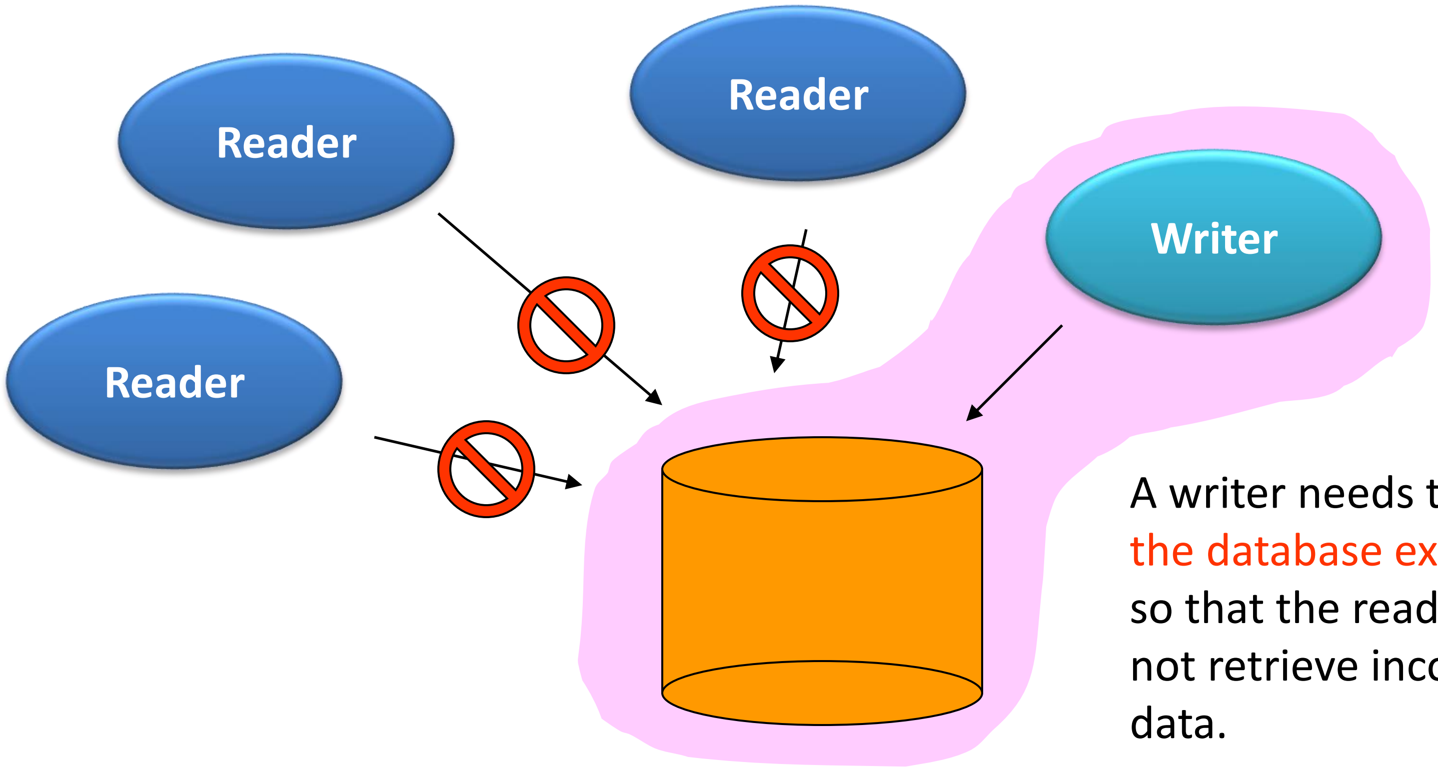
- It is a concurrent database problem.



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

Reader-writer problem – introduction

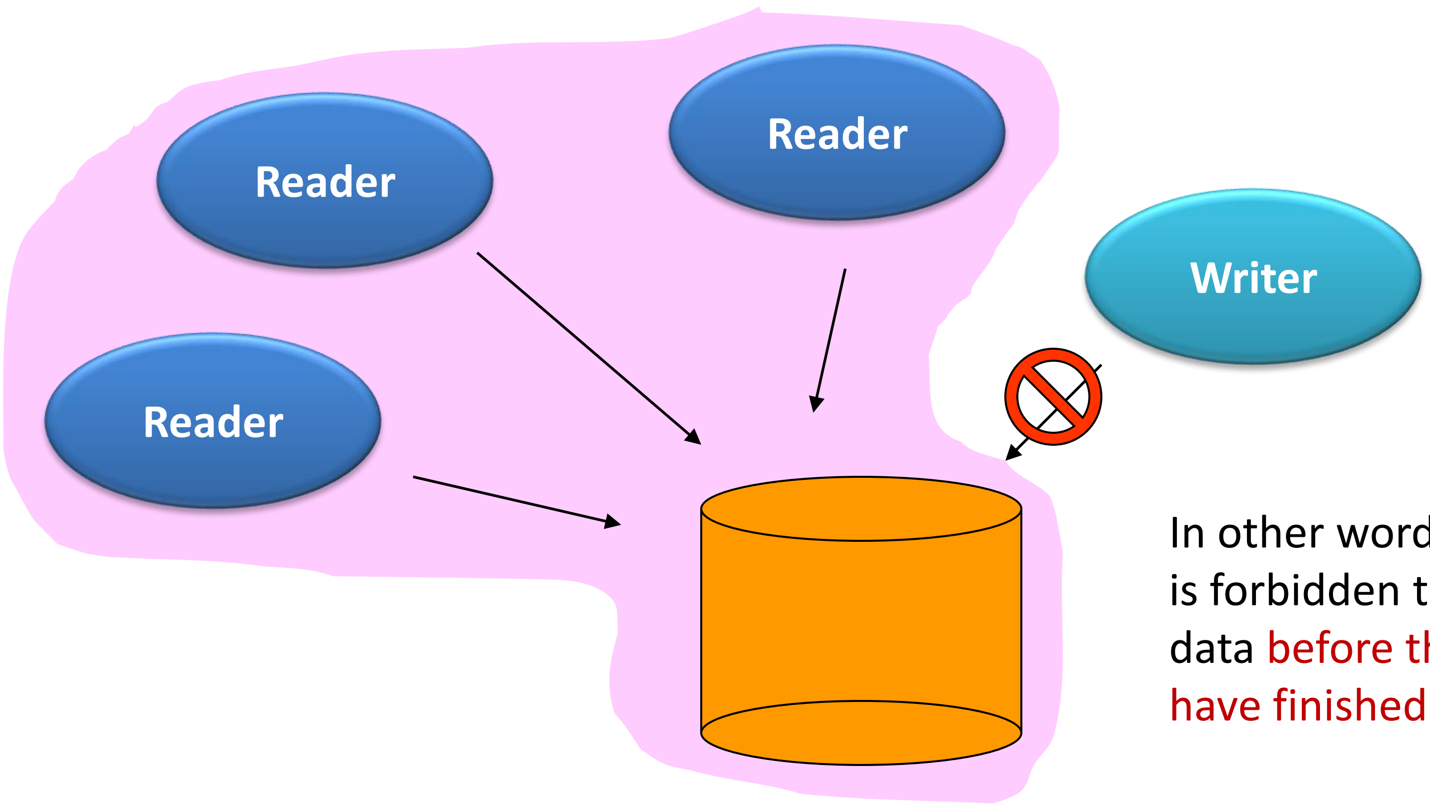
- It is a concurrent database problem.



A writer needs to lock the database exclusively so that the readers would not retrieve inconsistent data.

Reader-writer problem – introduction

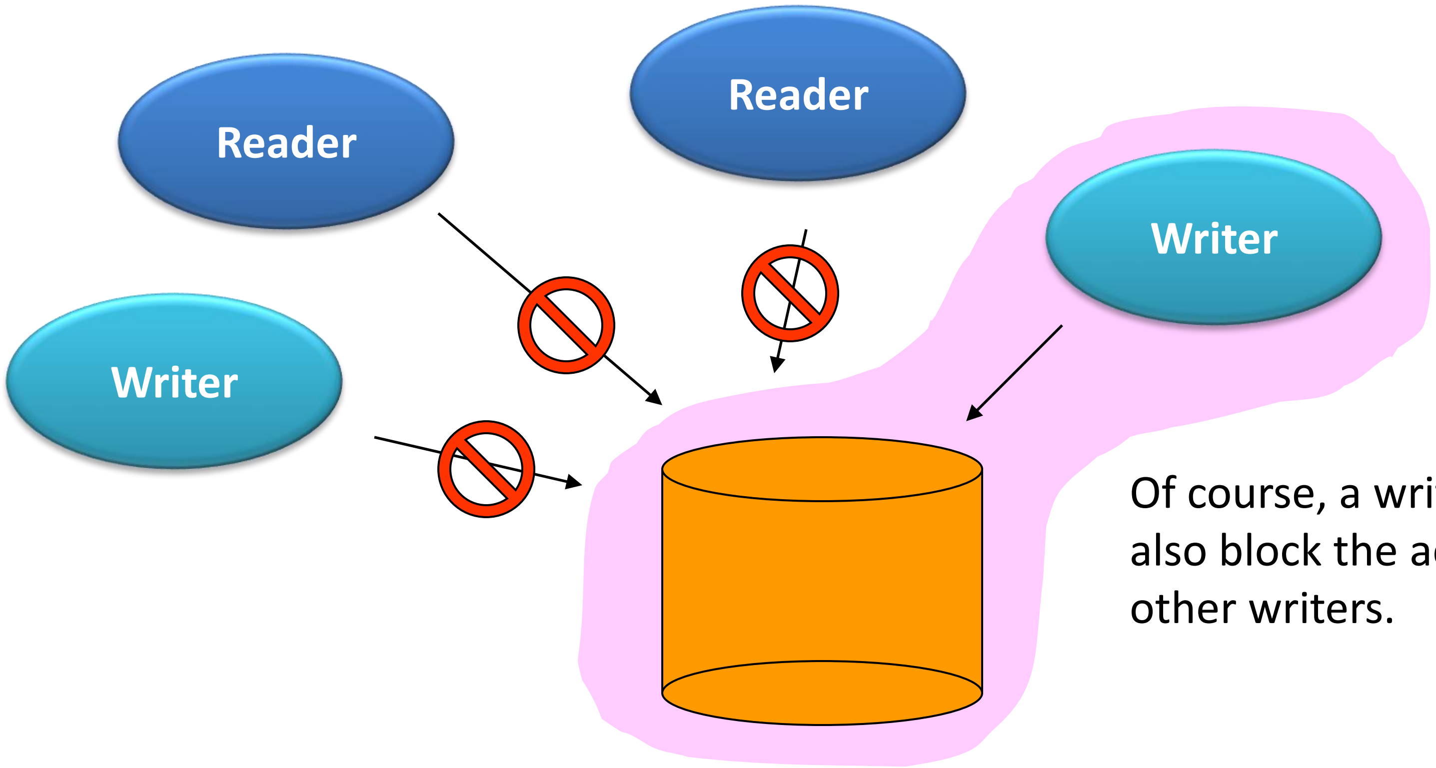
- It is a concurrent database problem.



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.



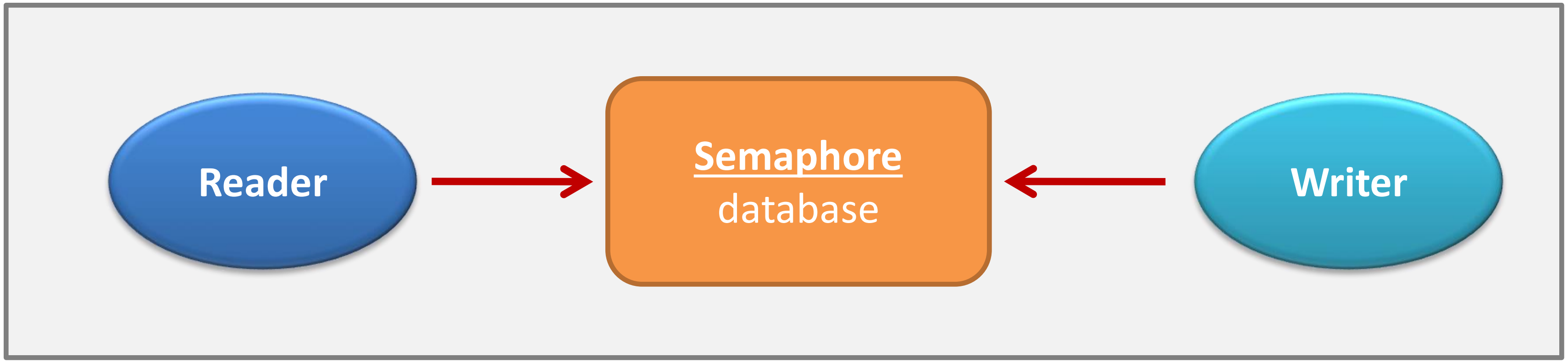
Of course, a writer will also block the access from other writers.

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.

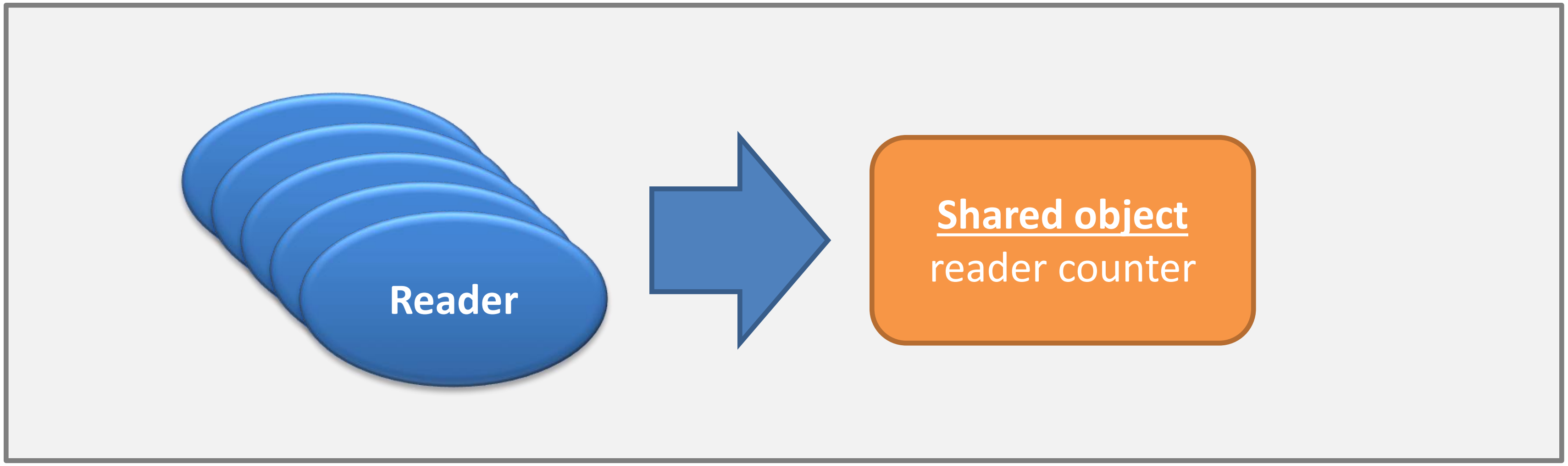
Reader-writer problem – solution outline

- **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

```
void writer(void) {
    while(TRUE) {
        prepare_write();
        down(&db);
        write_database();
        up(&db);
    }
}
```

Section Entry: `prepare_write(); down(&db);`

Critical Section: `write_database();`

Section Exit: `up(&db);`

Reader Function

```
void reader(void) {
    while(TRUE) {
        down(&mutex);
        read_count++;
        if(read_count == 1)
            down(&db);
        up(&mutex);
        read_database();
        down(&mutex);
        read_count--;
        if(read_count == 0)
            up(&db);
        up(&mutex);
        process_data();
    }
}
```

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);`

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

```
void writer(void) {
    while(TRUE) {
        prepare_write();
        down(&db);

        write_database();

        up(&db);
    }
}
```

Section Entry

Critical Section

Section Exit

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

```
void reader(void) {
    while(TRUE) {
        down(&mutex);
        read_count++;
        if(read_count == 1)
            down(&db);
        up(&mutex);

        read_database();

        down(&mutex);
        read_count--;
        if(read_count == 0)
            up(&db);
        up(&mutex);
        process_data();
    }
}
```

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

Cooperating Processes

Processes Communication

→ **IPC methods**: Shared memory, Pipes, Sockets

Race Condition

Mutual Exclusion

→ **How to realize**: Define **critical section**

→ **How to implement**:
- 4 requirements & 5 schemes
- **Semaphore**

Process Synchronization

Deadlock

→ **Classic problems**:
- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem