Operating Systems

Eric Lo

15 – Synchronization II

In Synchronization I

Achieving Mutual Exclusion – Locking approach

- Locking by Disabling Interrupt
  - This facility is not available to user level program though
  - Doable for kernel level but
    - **Multi-core complication**
      - It is possible that another core modifying the shared object in the memory
      - Multi-core CPU bundles with **atomic instructions** to make sure one core's modification is atomic
  - All subsequent discussions assume all instructions are atomic
    - E.g., x++ is atomic
    - Details of atomic instruction will be covered in topic "Synchronization II"

40

Atomic Instructions

- E.g., `test_and_set()` & `compare_and_swap()`
- E.g., Two `test_and_set(x)`s are concurrently executed by **different CPUs/cores** on the same memory location x
  - x will get updated by one by one
  - Can be used through newer versions of C/C++
    - E.g., C11 or above

p.s. C+11 has a Thread library, even easier to write multi-threading; that library is written using Pthread; programs using that are less portable than using Pthread directly

Achieving Mutual Exclusion - Revisit

- Lock-based
  - Spin-based lock
    - E.g., `pThread_spin_lock`
      - What is inside?
        - » Spin on 1 shared variable + **atomic instruction** to update it
        - » Spin on 2 shared variables + **atomic instruction** to update them (~Peterson)
        - » Spin on shared variables + **atomic instruction** to update them in an even smarter way
          - (=Ticket, MCS algorithm, etc.)
  - Sleep-based lock
    - E.g., POSIX `semaphore, pThread_mutex_lock`
      - What is inside?
        - » yield() + **atomic instructions** to update the extra shared variables
        - » Need to work with kernel's scheduler -> context switch to kernel space -> expensive ☹
        - » No waste the CPU to spin but let others use it (if any) ☺
- Lock-free
  - No such a call like `pthread_lock_free`
  - Let the threads update the shared variables freely
  - Right before leaving the CS
    - use atomic-instruction to <u>test</u> if there is any race condition
      - No? <u>set</u> a new value for the shared variable
      - Yes? Restart and Retry

https://stackoverflow.com/questions/5869825/when-should-one-use-a-spinlock-instead-of-mutex

Basic Spinlock implementation using `test_and_set()`

```
Initialize _Atomic lock = 0;

while (TestAndSet(&lock) == 1)
        ; //do nothing, spinning; this loop quits when 0==1
```

//spin_lock

when reaching this point, =you get the lock → lock = 1

But others should be spinning

```
<critical
..............................
                          section>
lock = 0; //release the lock
```

//spin_unlock

```
<remainder

section>
```

```
//pseudocode (indeed a single atomic instruction)
boolean test_and_set(boolean *lock){
    boolean initial = *lock; //original value
    *lock = 1; //set
    return initial; //return the original value!
}
```

compare_and_swap()

- Maurice Herlihy was awarded the Edsger W. Dijkstra Prize for his seminal 1991 paper "Wait-Free Synchronization"
  - the incapability of TAS when synchronizing >2 threads

```
//pseudocode (indeed a single atomic instruction)
boolean cas(int *value, int expected, int new) {
	if (*value != expected) {
		return false;
	}
	value = new;
	return true;
}
```

Basic Spinlock implementation using CAS

- `lock` – a shared variable

This can be wrapped as 'spin-lock()'

```
Initialize _Atomic lock = 0;
thread-function() {
while (!compare_and_swap(&lock, 0, 1))
        { ; }//                         //spin lock
<critical
        ........................
section>
lock = 0; //release the lock            //unlock
<remainder

section>
}
```

```
//pseudocode (indeed a single atomic instruction)
boolean cas(int *value, int expected, int new) {
        if (*value != expected) {
                return false;
        }
        *value = new;
        return true;
}
```

Check your knowledge

- Do you see pthread_mutex_lock?
- Is it lock-free?

```
Initialize _Atomic lock = 0;
thread-function() {
while (!compare_and_swap(&lock, 0, 1))
        { ; }//                        //spin lock
<critical
        ........................
section>
lock = 0; //release the lock
<remainder

section>
}
```

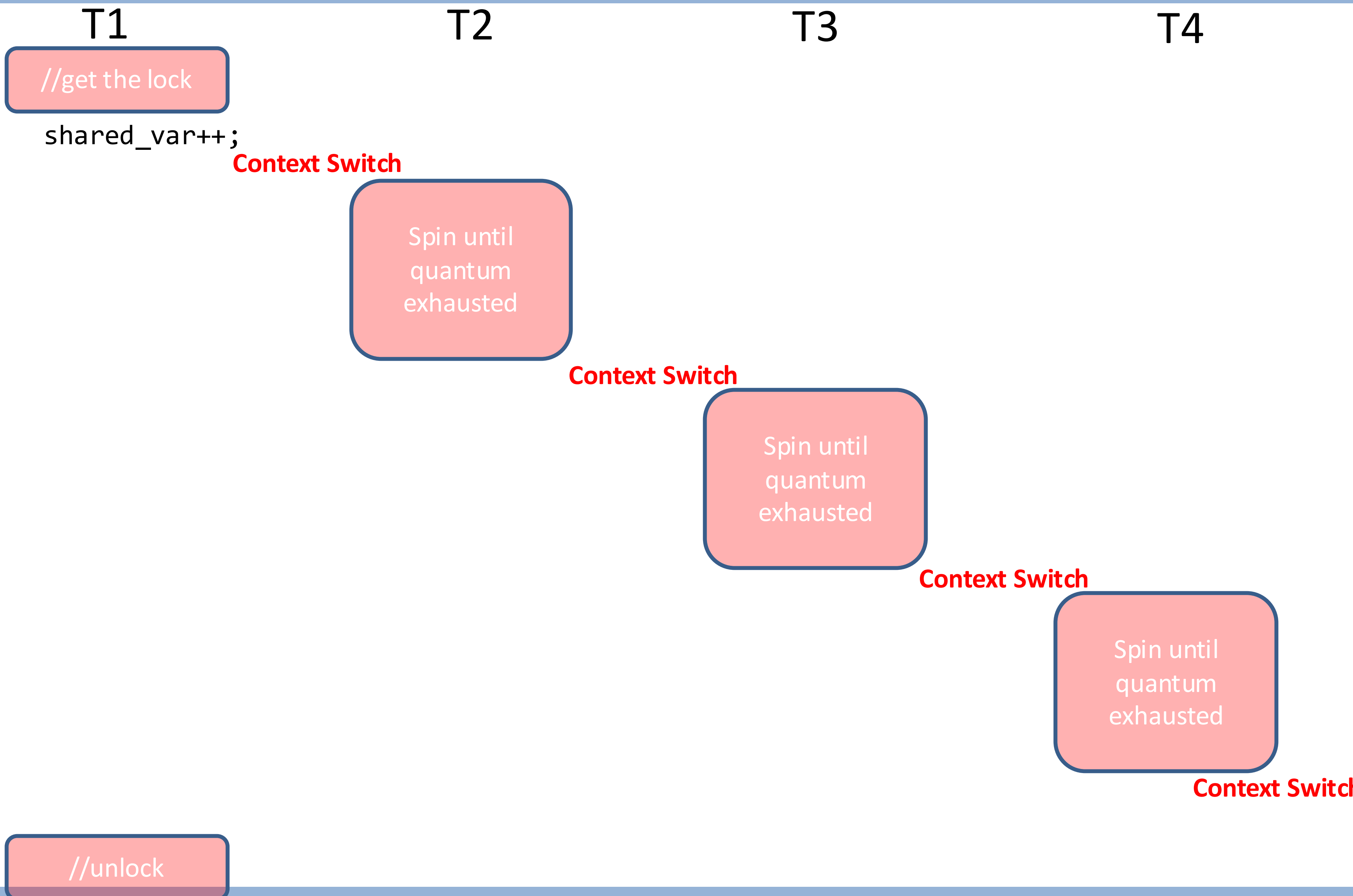
Counting: Lock-based vs. Lock-free

```
Initialize _Atomic lock = 0;
thread-function() {
while (!cas(&lock, 0, 1))
        { ; }//                    //spin lock
<critical
        shared_var++;
section>
lock = 0; //release the lock
<remainder


section>
}
```

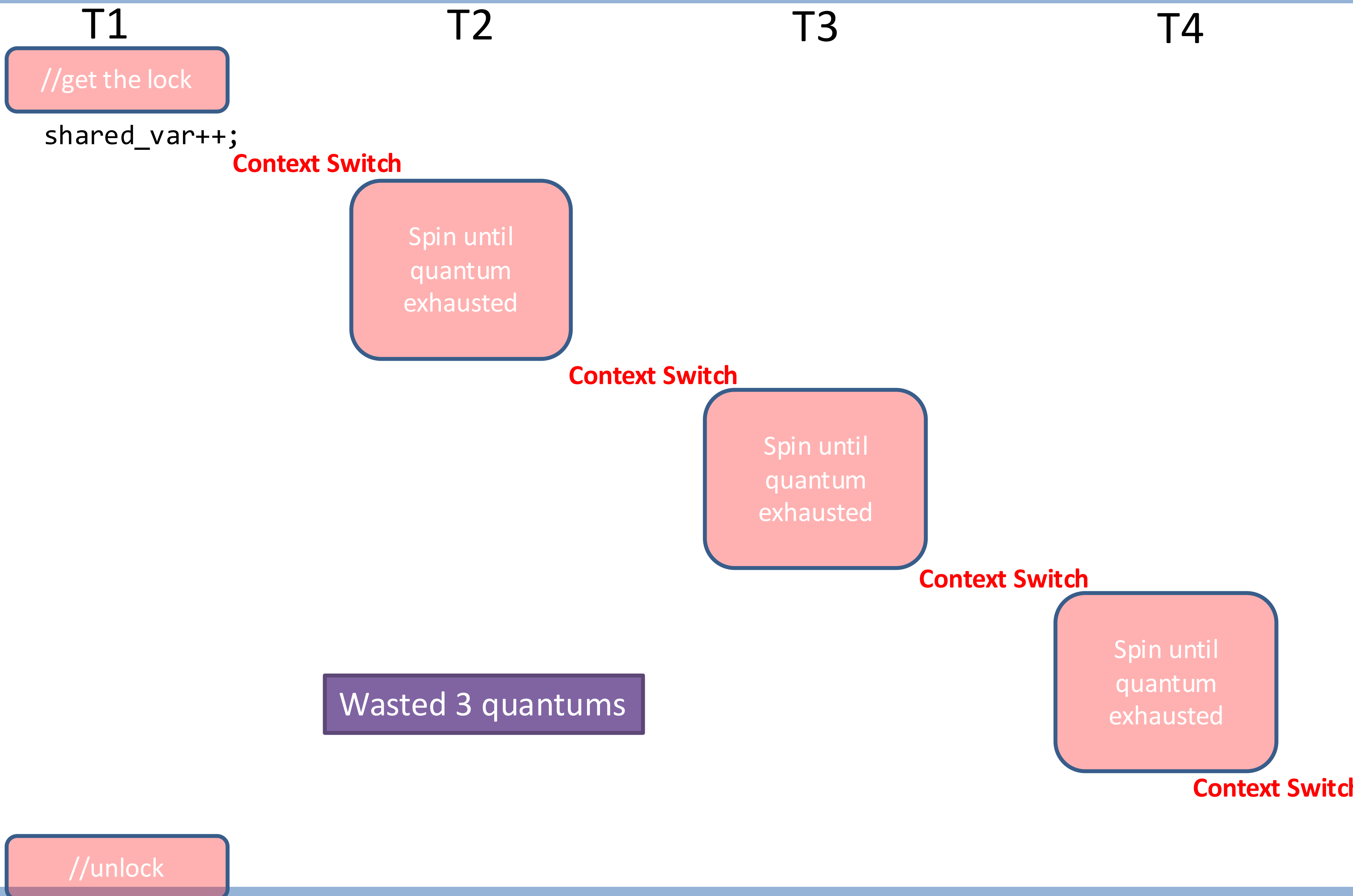
```
Initialize _Atomic lock = 0;
thread-function() {
do {
  <critical
  pure = shared_var;                    //optimistic
                                        //do the CS first
  section>
while (!cas(&shared_var, pure, pure++))
<remainder                              //redo if contaminated


section>
}
```

```
boolean cas(int *value, int expected, int new) {
        if (*value != expected) {
                return false;
        }
        value = new;
        return true;
}
```

Counting: Lock-based (4 threads; 1 core)



Counting: Lock-based (4 threads; 1 core)



Counting: Lock-free (4 threads; 1 core)

T1	T2	T3	T4
pure=10			
Context Switch			
	pure=10 shared_var=11		
	Context Switch		
		pure=11 shared_var=12	
		Context Switch	
			pure=12 shared_var=13
			Context Switch
cas(13, 10, pure++); Contaminated! cas failed RE-DO!			

```
do {
  &lt;critical
  pure = shared_var;          //optimistic
                              //redo if contaminated
  section&gt;
while (!cas(&shared_var, pure, pure++))
```

Counting: Lock-free (4 threads; 1 core)

T1 T2 T3 T4

```
pure=10
```

Context Switch

```
pure=10
shared_var=11
```

Context Switch

```
pure=11
shared_var=12
```

Context Switch

```
pure=12
shared_var=13
```

Context Switch

```
cas(13, 10, pure++);
Contaminated! cas failed

RE-DO!
```

Wasted 1 quantum

```
do {
  <critical
  pure = shared_var;          //optimistic
                              //redo if contaminated
  section>
while (!cas(&shared_var, pure, pure++))
```

That’s too good to be true…

- It could also be like this:

Counting: Lock-free (4 threads; 1 core)

T1	T2	T3	T4
pure=10			
Context Switch			
	pure=10 shared_var=11 (success!)		
	Context Switch		
		pure=11	
		Context Switch	
			pure=11
			Context Switch
cas(11, 10, pure++); Contaminated! cas failed			
RE-DO! Wasted 1 quantum			
shared_var=12 (success!)			
		cas(12, 11, pure++); Contaminated! cas failed	
		RE-DO! Wasted 1 quantum	
			cas(12, 11, pure++); Contaminated! cas failed
			RE-DO! Wasted 1 quantum

Lock-free vs Lock-based

- Lock-based incurs problems of:
  - Lock-holder sleeping
  - Lock-holder dies
    - System dies with it
  - Deadlock
    - System needs to detect and resolve it
- Lock-free is free of the above
  - But lock-free also has its problems, e.g., ABA
  - Very difficult to code
    - Usually the code added to deal with ABA problem introduces another instance of ABA problem …

Lock-free needs to take care of the ABA problem

Time

```
Thread 1:
Pop(*lfstack){                          //Originally, lfstack->head=0x01
      orig = atomic_load(lfstack);      //orig.head=0x01
      next.head = orig.head->next;      //next.head=0x02
```

Context switch

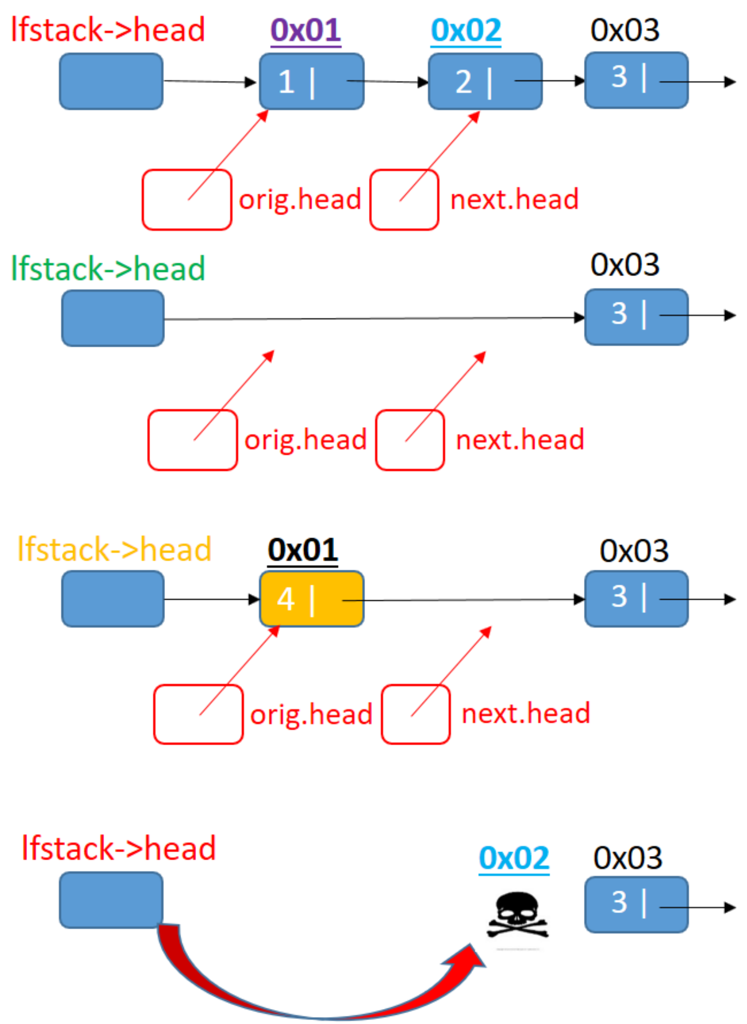
```
Thread 2:
Pop(*lfstack) //free 0x01
Pop(*lfstack) //free 0x02
              //now, lfstack->head=0x03
```

Context switch

```
Thread 3:
Push(*lfstack, 4){
      Node *node = malloc(sizeof(Node)); //allocator reuses the same memory
                                         // block 0x01, so node=0x01
      ...                 //when thread 3 pushes node into the stack, it let
                          //lfstack->head point  to node, then, lfstack->head=0x01;
}
```

Context switch

```
Thread 1:
      CAS(lfstack, &orig, next); //here, it compares the content points to by
//lfstack with the content of orig, that is, lfstack->head and orig.head, for equality.
//For lfstack->head, which is 0x01, while for orig.head, which is also 0x01, they are
//the same, then CAS succeeds! So it sets lfstack->head=next.head=0x02! Then,
//lfstack->head will point to a freed memory! Error!
}
```



One solution to ABA problem

– Add a ‘counter’ to indicate how many people have modified it before
– So, even if the content is the same, we can still detect inconsistency

Lock-free vs Wait-free

Lock-free	Wait-free

• Guarantee progress for **some threads,**
  • Regardless the state/speed of other threads
  • Guarantee **system-wise** progress
	• Guarantee progress for **every thread,**
	  • Regardless the state/speed of other threads
	  • Guarantee **per-thread** progress
	• Sometimes impossible to achieve
	• Thread1.operation-X() must finish in a finite number of steps
	  • Regardless of the state/speed of the other thread

E.g., lock-free stack, lock-free hashtable	E.g., The lockless page cache patches to the Linux kernel are an example of a wait-free system.

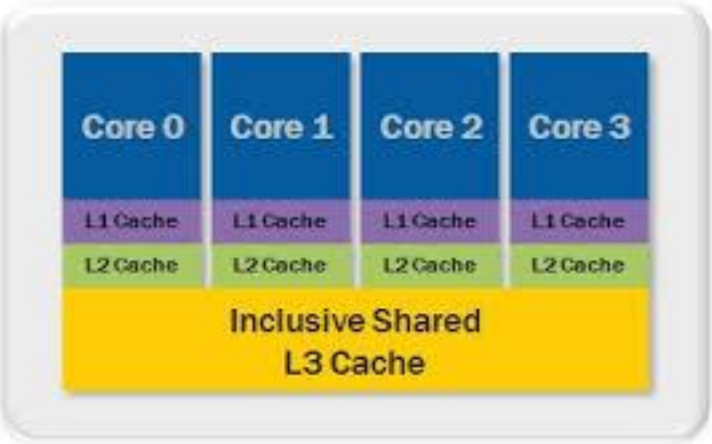
Which one is easier to achieve?

CAS in C11 (programming)

- C11's CAS is not restricted to `int` type only.
  - The more usual one is **pointer-based**. Check out the lab!
  - Lecture: use the int version
- C11's CAS has weak and strong version
  - `_Bool atmoic_compare_exchange_weak(volatile A *object, C *expected, C desired)`
    - Check out the lab for the meaning of
      - Volatile
      - A: atomic
      - C: non-atomic
    - For our course using x86, no difference

Memory Consistency Model

- Weak and strong refer to memory consistency model
  - At the beginning, Core 1:
    - LOAD memory address 0x888888 (content="old")
    - STORE *0x888888 = "new"
    - Update stills in Core 1's local L1
    - (not yet flush to memory)
  - Then, Core 2:
    - Read memory address 0x888888
  - Whether core 2 reads "old" or "new"
  - Strong consistency memory model (we assume this in the course)
    - x86 (i.e., Intel and AMD)➔ **cache coherence** ➔core 2 reads "new"
  - Weak consistency memory model
    - ARM (Advanced RISC Machine) ➔ almost all mobile devices ➔ no cache coherence ➔ core 2 reads "old"
      - To avoid that happens, programmers have to explicit add **memory fence** themselves



Multi-object Synchronization

Inter-process communication (IPC)

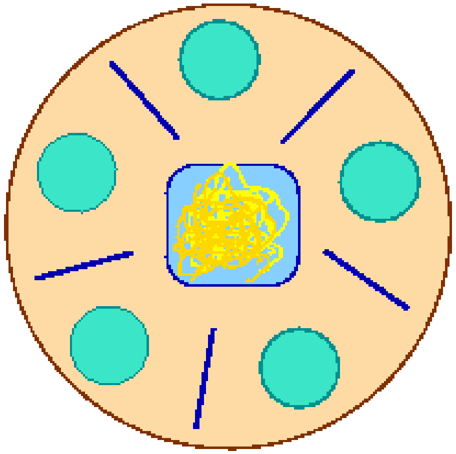
- Classic IPC problems.

- Producer-consumer problem (single object synchronization)

- Dining philosopher problem (multi-object synchronization)

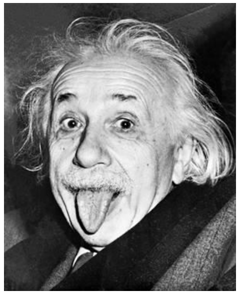
Dining philosopher – introduction

- 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 **starve to death**, and
  - will not result in any **deadlock scenarios**?
    - A waits for B's chopstick
    - B waits for C's chopstick
    - C waits for A's chopstick ....

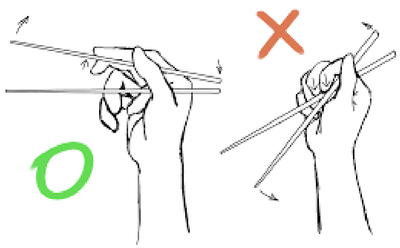


It's a multi-object synchronization problem

Dining philosopher – introduction



Philosophers



Chopsticks

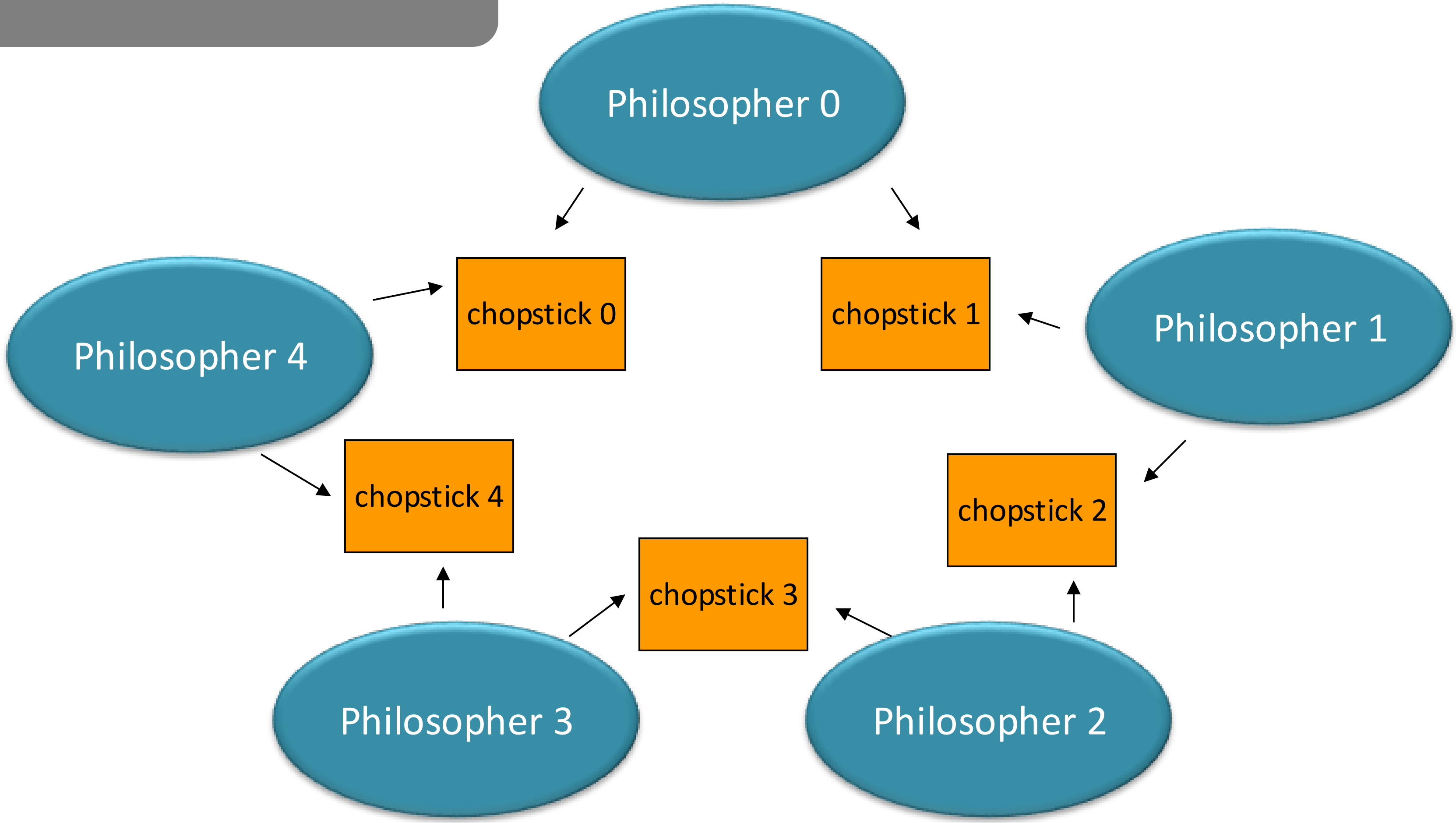
Process

Shared Object

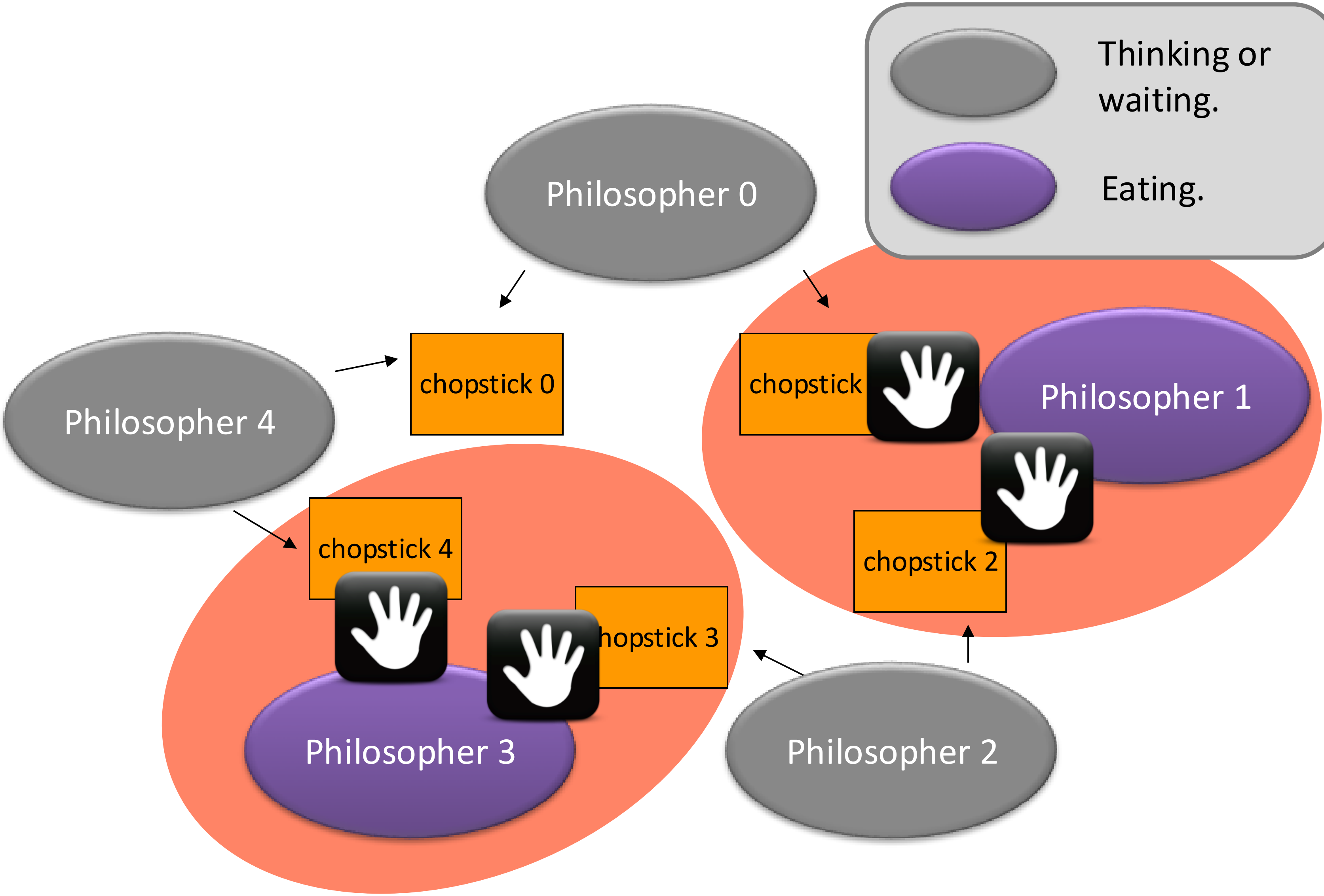
A process needs two shared resources in order to do some work

Dining philosopher – introduction

Imagine the problem as



Dining philosopher – introduction



Dining philosopher – requirement #1

- **<u>Mutual exclusion</u>**
  - While you are eating, people can't steal your chopstick
    - Two persons can't hold the same chopstick
- Let's propose the following solution:
  - When you are hungry, you have to check if anyone is using the chopsticks that you need.
  - If yes, you wait.
  - If no, **seize chopstick-left, chopstick-right**.
  - After eating, put down all your chopsticks.

Dining philosopher – meeting requirement #1?

Shared object

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

Five binary semaphores

Helper Functions

```
void take_chopstick(int i)
{
    swait(&chopstick[i]);
}
```

```
void put_chopstick(int i) {
    spost(&chopstick[i]);
}
```

Main Function

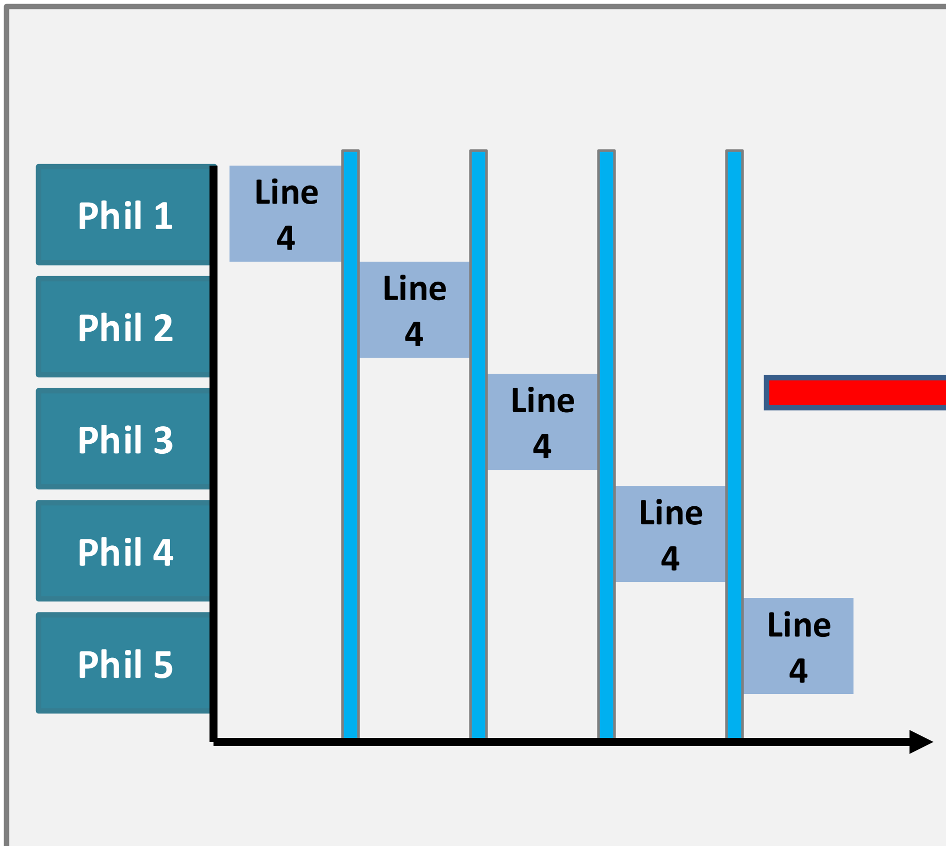
```
void philosopher(int i) {
    while (TRUE) {
        think();

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

        eat();

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

Section Entry: `take_chopstick(i); take_chopstick((i+1) % N);`

Critical Section: `eat();`

Section Exit: `put_chopstick(i); put_chopstick((i+1) % N);`

Dining philosopher – deadlock



Main Function

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

4           take_chopstick(i);
5           take_chopstick((i+1) % N);

6           eat();

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

Dining philosopher – requirement #2

- **Synchronization**
  - **Deadlock free**

- How about the following suggestions:
  - First, a philosopher **takes a chopstick**.
  - If a philosopher finds that she cannot take the second chopstick, then she should **put it down**.
  - Then, the philosopher **goes to sleep** for a while.
  - When wake up, she retries
  - Loop until both chopsticks are seized.

Dining philosopher – meeting requirement #2?

<u>Potential Problem</u>: Philosophers are all busy (no deadlock), but no progress (<u>starvation</u>)

```
Imagine:
- all pick up their left chopsticks,
 - seeing their right chopsticks unavailable (because P1's right chopstick is taken
by P2 as her left chopstick) and then putting down their left chopsticks,
  - all sleep for a while
    - all pick up their left chopsticks, ....
```

What's the problem?

- Requirement:
  - Atomic Transaction on **Multiple** Objects
  - All (2 chopsticks) or nothing
- Yet:
  - We took two chopsticks non-atomically

Dining philosopher – one solution idea

Philosopher 0
EATING

Philosopher 4
CAN'T EAT

Philosopher 1
CAN'T EAT

1:

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

- **A Philosopher waits for a Philosopher instead of wait for a chopstick**

Philosopher 3
CAN EAT

Philosopher 2
CAN EAT

2: Add a captain (coordinator) for help

Dining philosopher – a solution.

Shared object

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

int state[N];
semaphore mutex = 1;
semaphore p[N] = 0;
```

Identifying i's "left" and "right"

The states of the philosophers, including "**EATING**", "**THINKING**", and "**HUNGRY**".

Remember, this is a shared array.

To guarantee mutual exclusive access to the "**state[N]**" array.

To model a philosopher (instead of a chopstick)

Dining philosopher – a solution.

Shared object

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

int state[N];
semaphore mutex = 1;
semaphore p[N] = 0;
```

Main function

```
void philosopher(int i) {
    think();
    take_chopsticks(i);
    eat();
    put_chopsticks(i);
}
```

```
void swait(semaphore *s) {
   disable_interrupt();
       *s = *s - 1;
   if ( *s < 0 ) {
      enable_interrupt();
      sleep();
      disable_interrupt();
    }
   enable_interrupt();
}
```

Section entry

```
void take_chopsticks(int i) {
    swait(&mutex);
     state[i] = HUNGRY;
     captain(i);
    spost(&mutex);
    swait(&p[i]);
}
```

Section exit

```
void put_chopsticks(int i) {
    swait(&mutex);
     state[i] = THINKING;
     captain(LEFT);
     captain(RIGHT);
    spost(&mutex);
}
```

```
void spost(semaphore *s) {
  disable_interrupt();
  *s = *s + 1;
  if ( *s <= 0 )
    wakeup();
  enable_interrupt();
}
```

Extremely important helper function

```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        spost(&p[i]);
    }
}
```

Dining philosopher – **Hungry**

Tell the captain that you are hungry

If one of your neighbors is eating, the captain just does nothing for you and returns

Then, you wait for your chopsticks (later, the captain will notify you when chopsticks are available)

Critical Section

Section entry

```
1  void take_chopsticks(int i) {
2      swait(&mutex);
3       state[i] = HUNGRY;
4       captain(i);
5      spost(&mutex);
6      swait(&p[i]);
7  }
```

Extremely important helper function

```
void captain(int i) { //atomic captain
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        spost(&p[i]); //s.t. p[i] don't need to wait in line 6 of take_chopsticks
    }
}
```

Dining philosopher – Finish eating

Tell the captain:
Try to let your **left neighbor** to eat.

Tell the captain:
Try to let your **right neighbor** to eat.

Section exit

```
void put_chopsticks(int i)
{
    swait(&mutex);
     state[i] = THINKING;
     captain(LEFT);
     captain(RIGHT);
    spost(&mutex);
}
```

Extremely important helper function

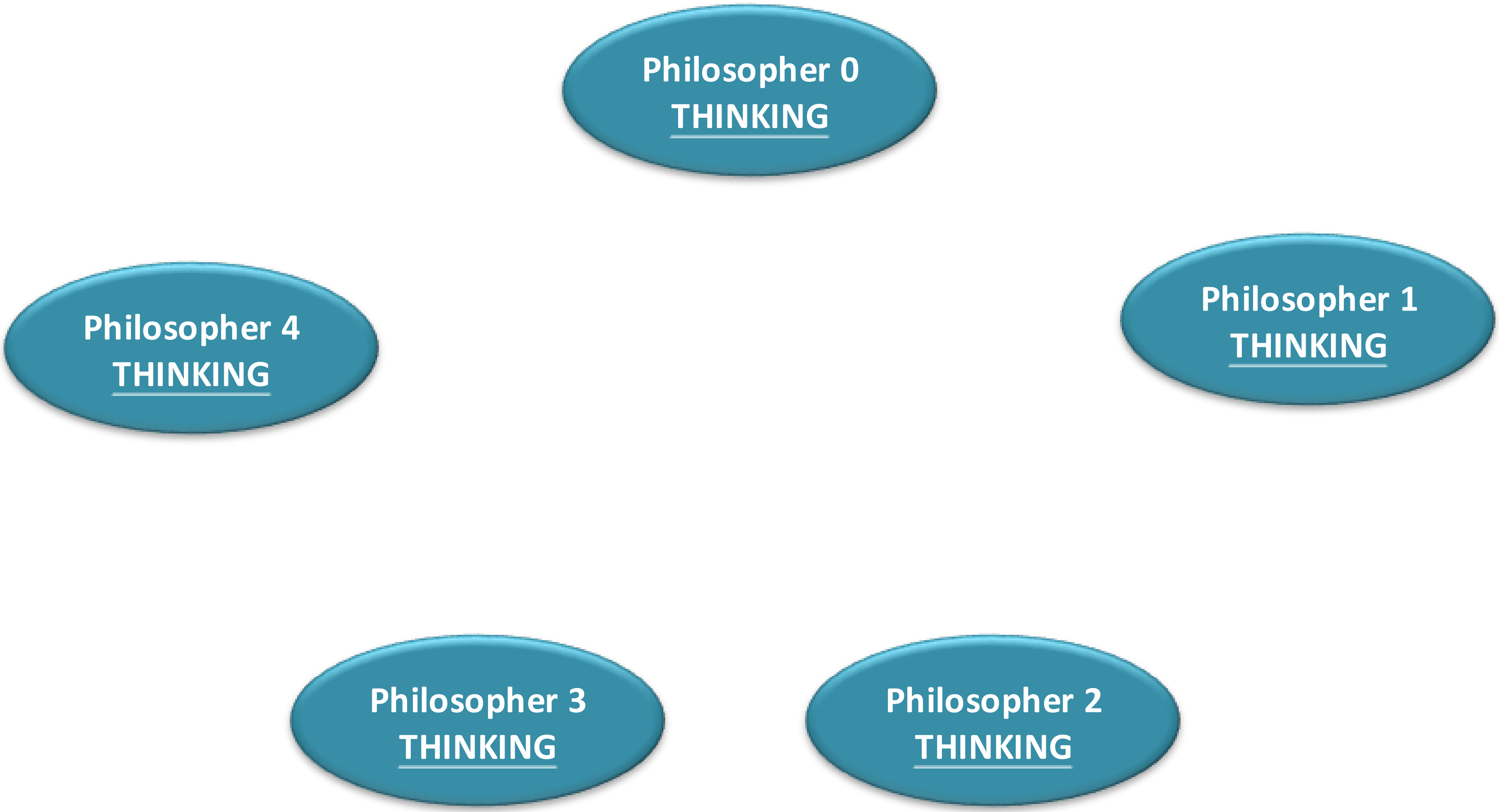
```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        spost(&p[i]);
    }
}
```

Wake up the one who is waiting

Dining philosopher – the final solution.

Don't print

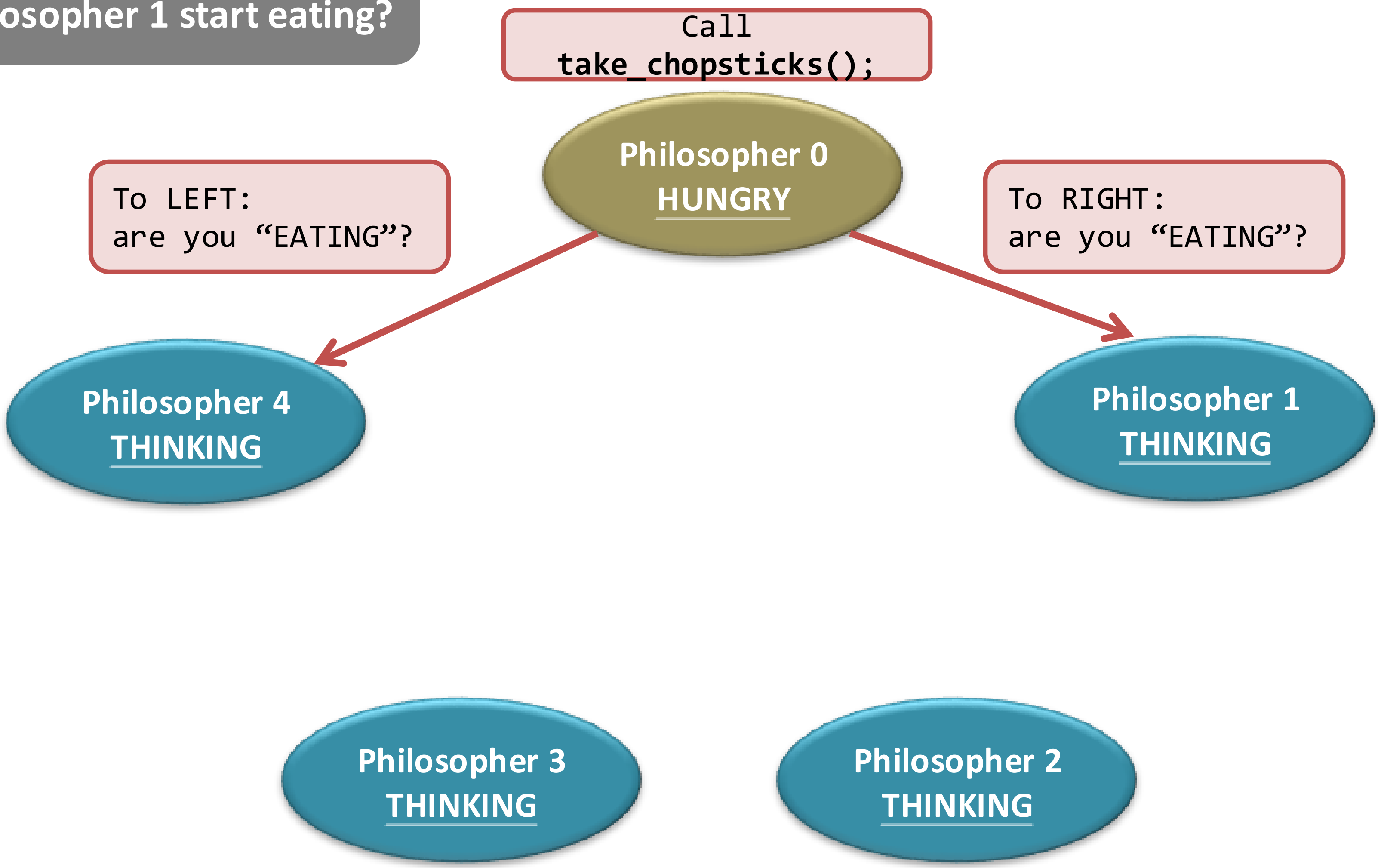
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don't print

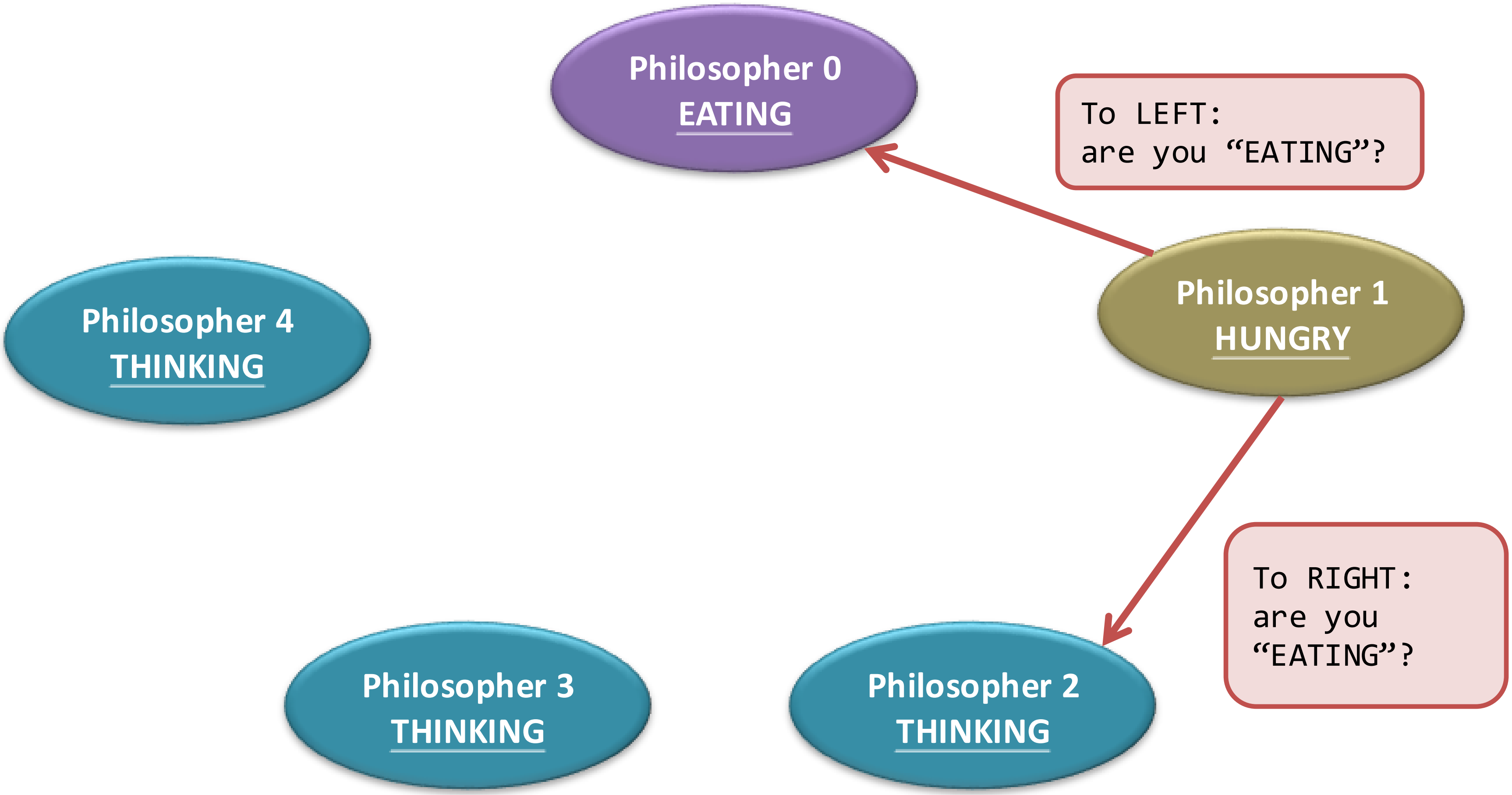
An illustration: How can Philosopher 1 start eating?



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?

Section entry

```
void take_chopsticks(int i) {
     swait(&mutex);
     state[i] = HUNGRY;
     captain(i);
     spost(&mutex);
     swait(&p[i]);
}
```

```
//as P0 is eating, captain(i) returns
      w/o doing anything;
         swait(&p[1]);
```

Philosopher 0
EATING

Philosopher 1
HUNGRY

Philosopher 4
THINKING

To RIGHT:
are you
"EATING"?

To LEFT:
are you
"EATING"?

Philosopher 3
HUNGRY

Philosopher 2
THINKING

Dining philosopher – the final solution.

Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
void take_chopsticks(int i) {
    swait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    spost(&mutex);
    swait(&p[i]);
}
```

Philosopher 0
EATING

Philosopher 4
THINKING

Philosopher 1
HUNGRY

Blocked; because of `swait(&p[1]);`

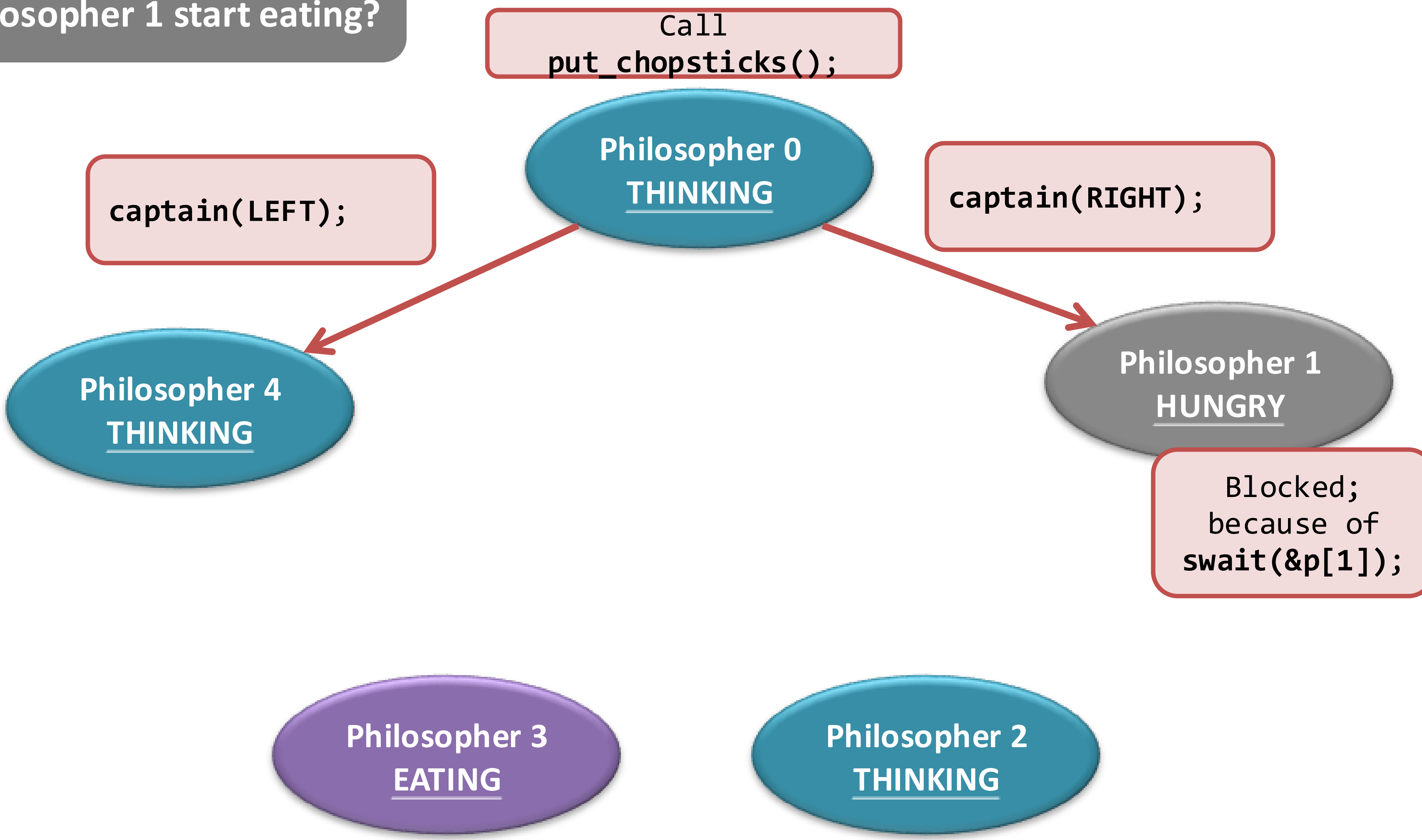
Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the final solution.

Don't print

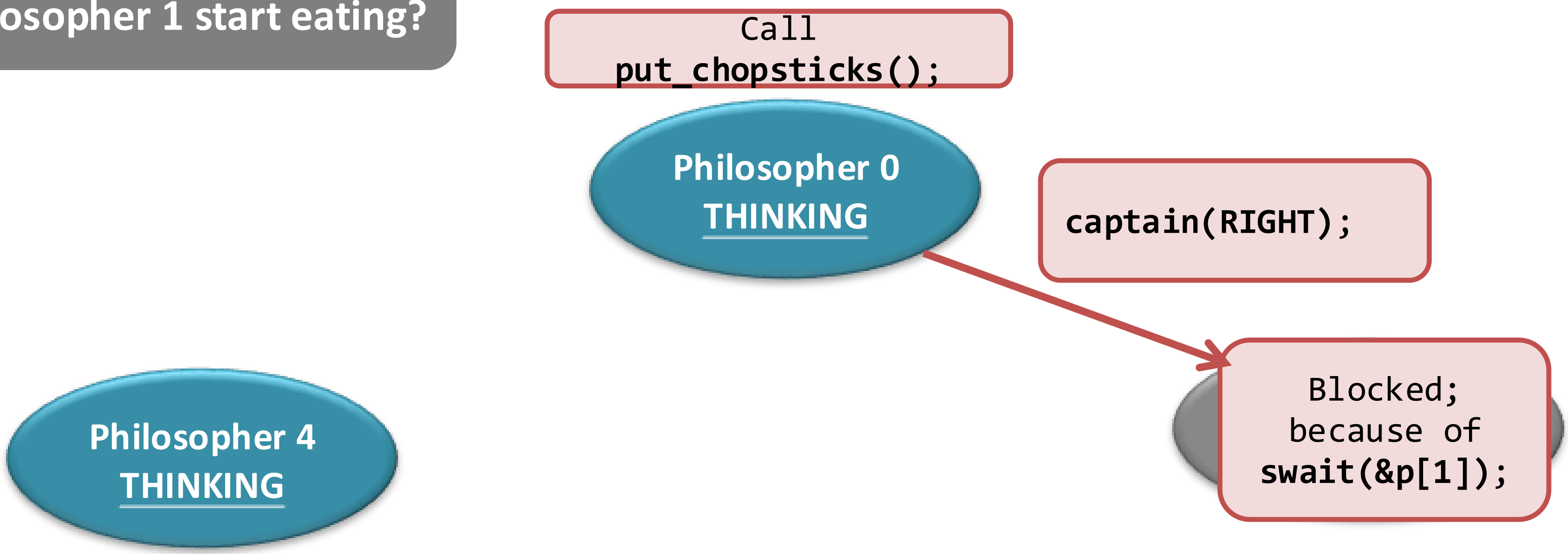
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final solution.

Don’t print

An illustration: How can Philosopher 1 start eating?




```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        spost(&p[i]);
    }
}
```

Wake up !

Dining philosopher – the final solution.

Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
void take_chopsticks(int i) {
    swait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    spost(&mutex);
    swait(&p[i]);
}
```

Wake up

Philosopher 0
THINKING

Philosopher 1
EATING

Philosopher 4
THINKING

Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the core

5 philosophers → ideally how many chopsticks?

how many chopsticks do we have now?

Very common in today's cloud computing multi-tenancy model

Dining philosopher

- Atomic Transaction on **Multiple** Objects
- It is a fundamental problem in database area
  - They have better solutions for it
    - E.g., OCC