



Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski



SCOOP: lectures

2

- Lecture 7 (16 May): computational model, synchronous and asynchronous feature calls, synchronisation.
- Lecture 8 (23 May): traitors, validity rules, type system for SCOOP.
- Lecture 9 (30 May): Design by Contract in concurrent context, inheritance, inheritance anomalies.
- Lecture 10 (6 June): advanced O-O techniques: genericity, agents, attached types.
- Lecture 11 (13 June): deadlocks and how to prevent them, future developments in SCOOP.



SCOOP: exercises

3

- Exercise 8 (23 May): SCOOP tools and examples.
Announcement of final project.
- Exercise 9 (30 May): SCOOP: examples, debugging.
- Exercise 10 (6 June): SCOOP: inheritance, genericity.
- Exercise 11 (13 June): SCOOP: barrier, lock passing.
- Exercise 12 (20 June): SCOOP: project Q&A.
- Exercise 13 (27 June): SCOOP: project Q&A.
- Deadline for projects: 4 July



Lecture 7: Computational model of SCOOP



Outline

5

- Processors
- Separate objects, separate entities
- Synchronous and asynchronous call semantics
- Mutual exclusion
- Condition synchronisation
- Wait-by-necessity

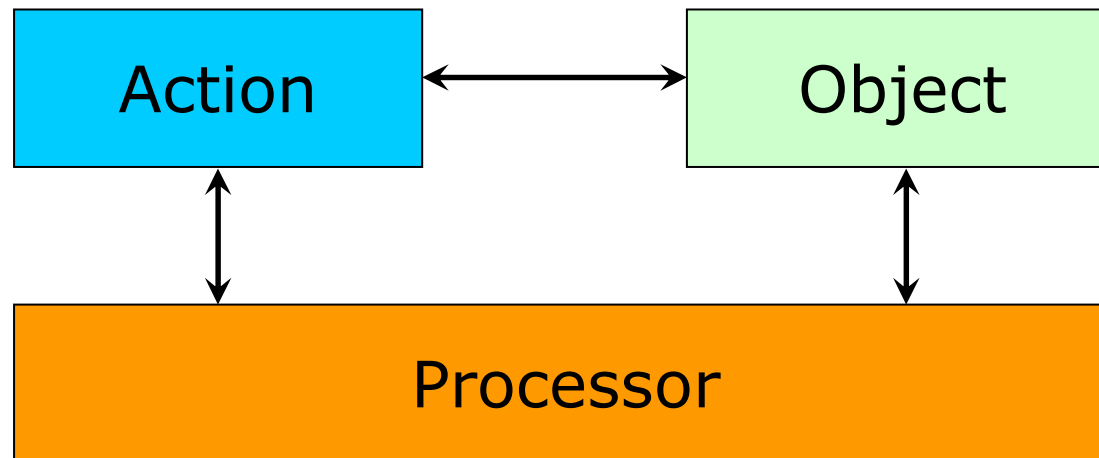


Basic idea of OO computation

6

To perform a computation is

- To apply certain **actions**
- To certain **objects**
- Using certain **processors**



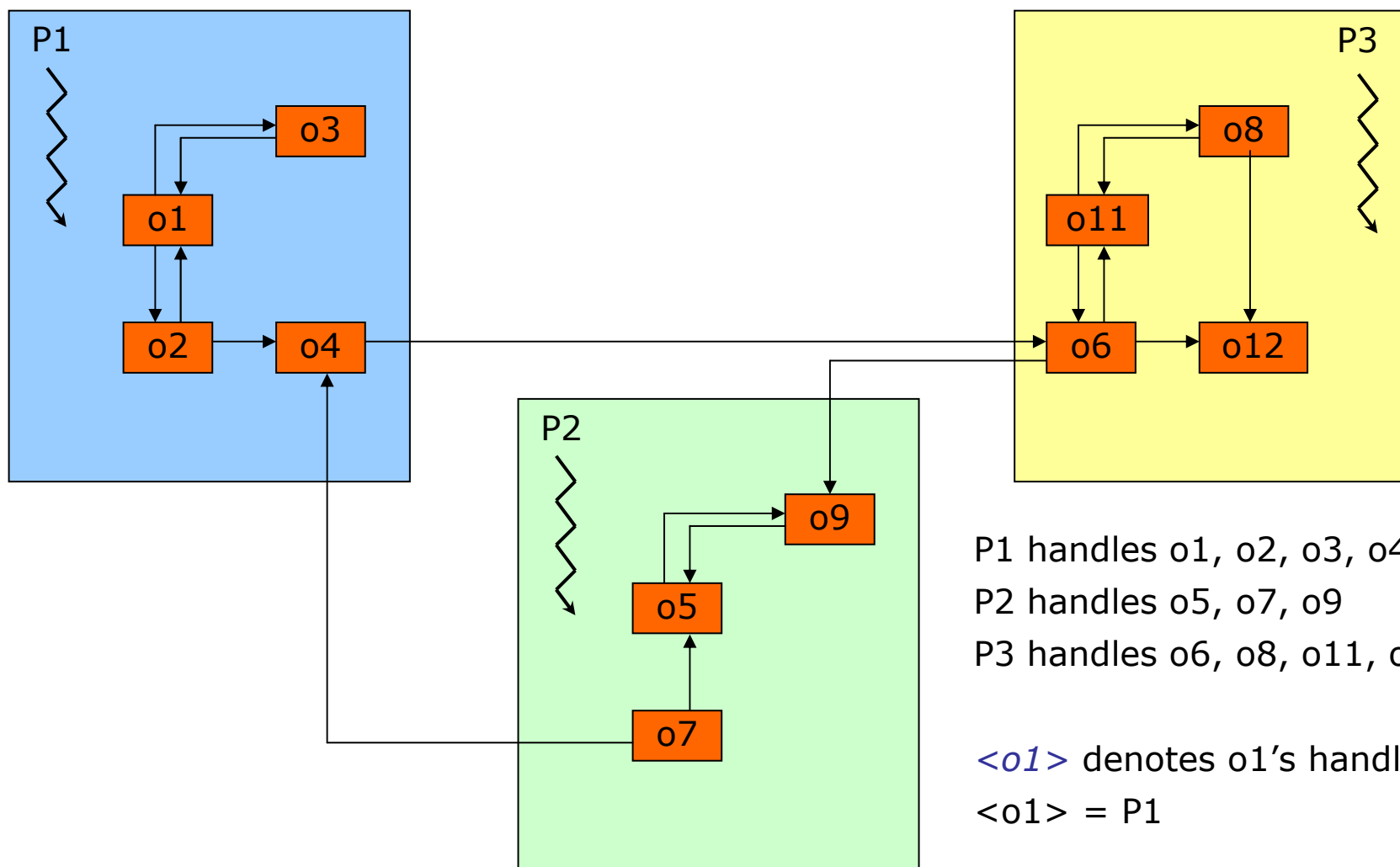


Processors

- Processor: a **thread of control** supporting sequential execution of instructions on one or several objects.
- All actions on a given object are executed by its **handling processor**. **No shared access to objects!**
- We say that the object is **handled** by its processor.
 - This relationship is **fixed**, i.e. we do not consider migration of objects between processors.
- Each processor, together with all object it owns, can be seen as a sequential subsystem.
- A (concurrent) **software system** is composed of such subsystems.



Software system





Processors (cont.)

9

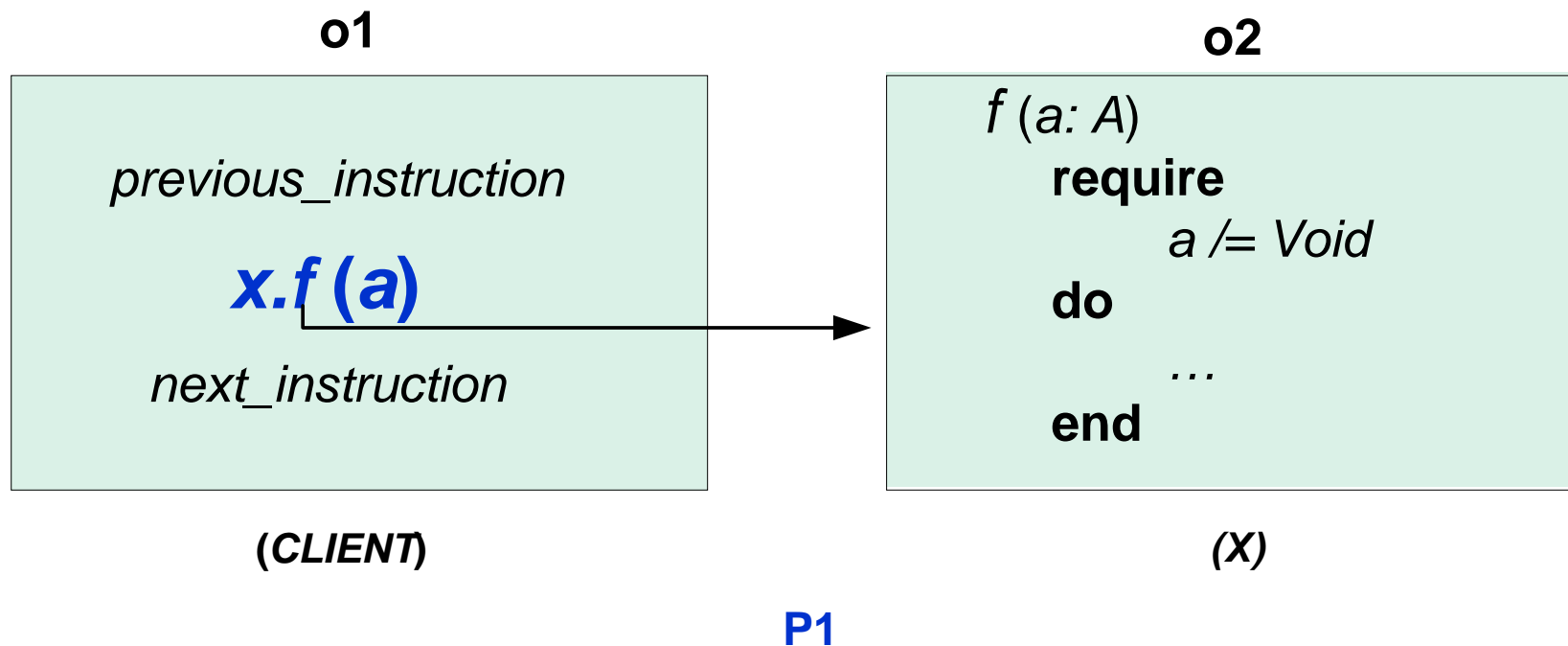
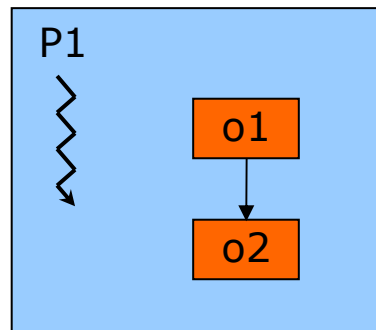
- Processor is an abstract concept
- Do not confuse it with a CPU!

- A processor can be implemented as:
 - Process
 - Thread
 - Web service
 - .NET AppDomain
 - ???



Feature call - synchronous

$x: X$
...
 $x.f(a)$



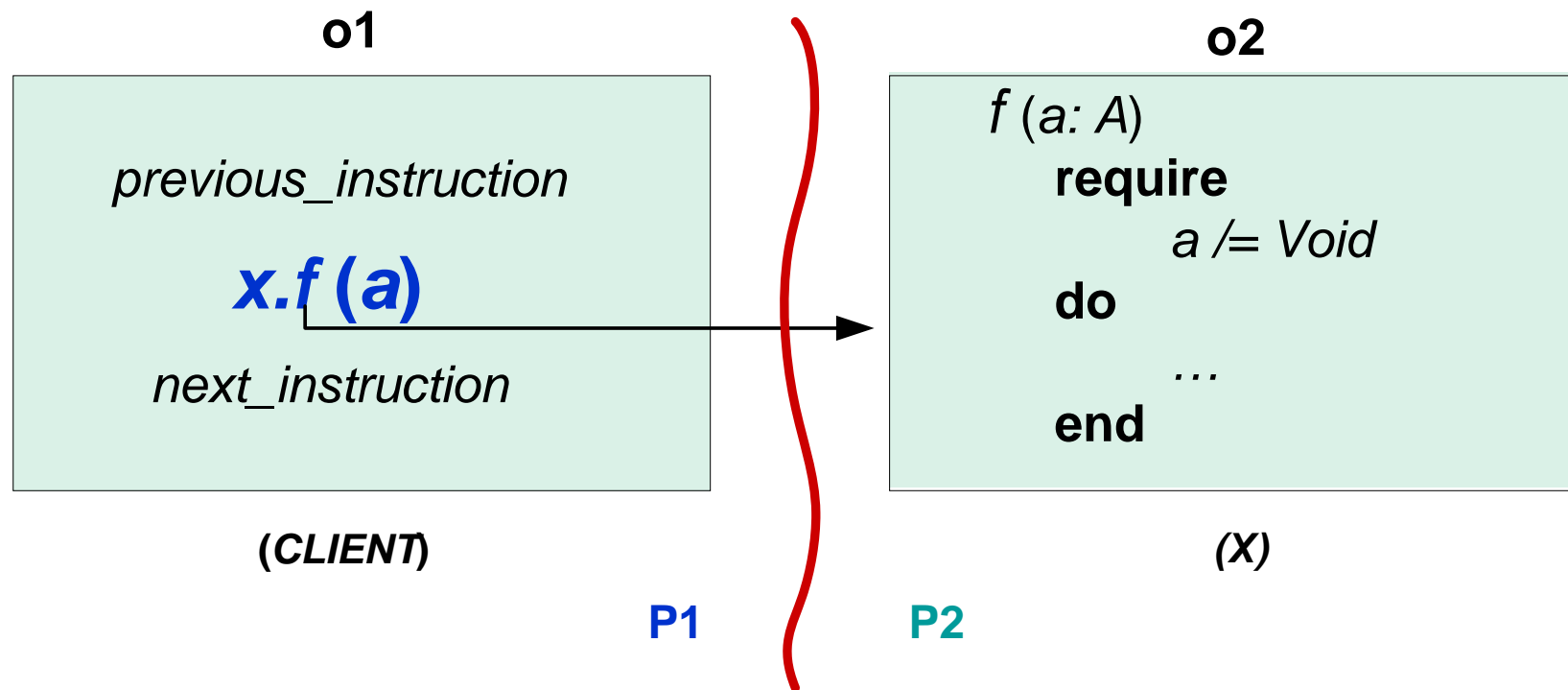
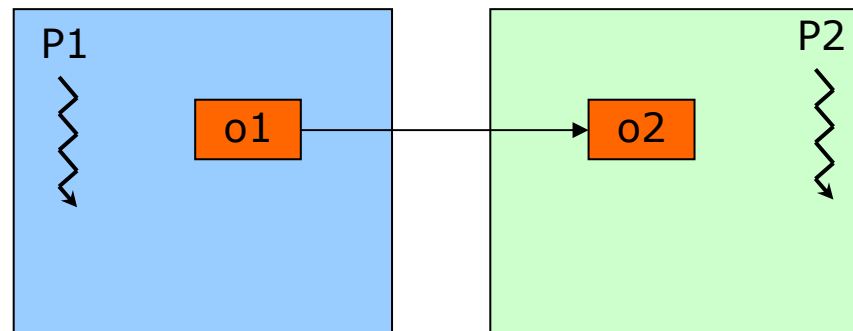


Feature call - asynchronous

x: separate X

...

x.f(a)

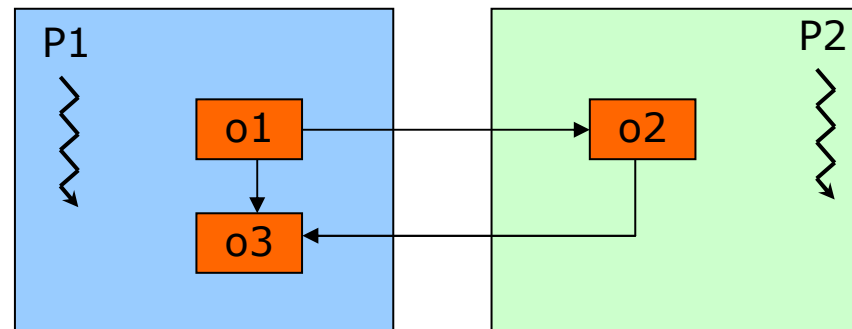




Separate objects

12

- Calls to non-separate objects are synchronous
- Calls to separate objects are asynchronous



- QUIZ: Which objects are separate here?



Separate entities

13

- Separate entities are declared with **separate** keyword
 x : **separate** X

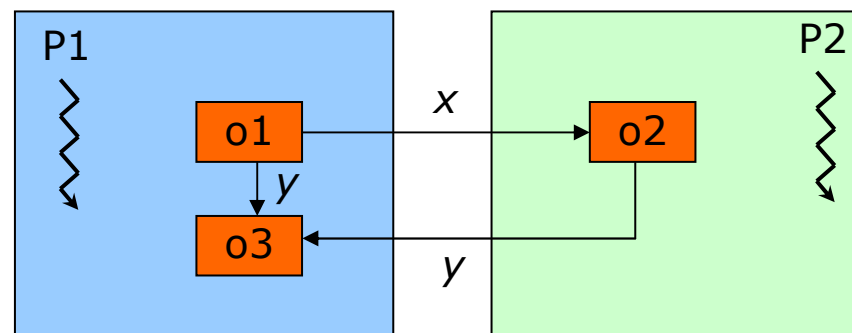
- Does a separate entity always denote a separate object?
 x, y : **separate** X

...

$y := x.y$

-- Is y a separate entity?

-- Does it denote a separate object?



- Separate entities denote **potentially separate** objects



Synchronisation

14

- Processors are *sequential*
- Concurrency is achieved by *interplay* of several processors
- Processors need to *synchronise*
- Three forms of synchronisation in SCOOP
 - mutual exclusion
 - condition synchronisation
 - wait-by-necessity



If no mutual exclusion

15

- Programmer writes:

```
my_stack: separate STACK [INTEGER]
```

...

```
{ my_stack.push (5)
  y := my_stack.top  -- Are we sure that y = 5 ?
```

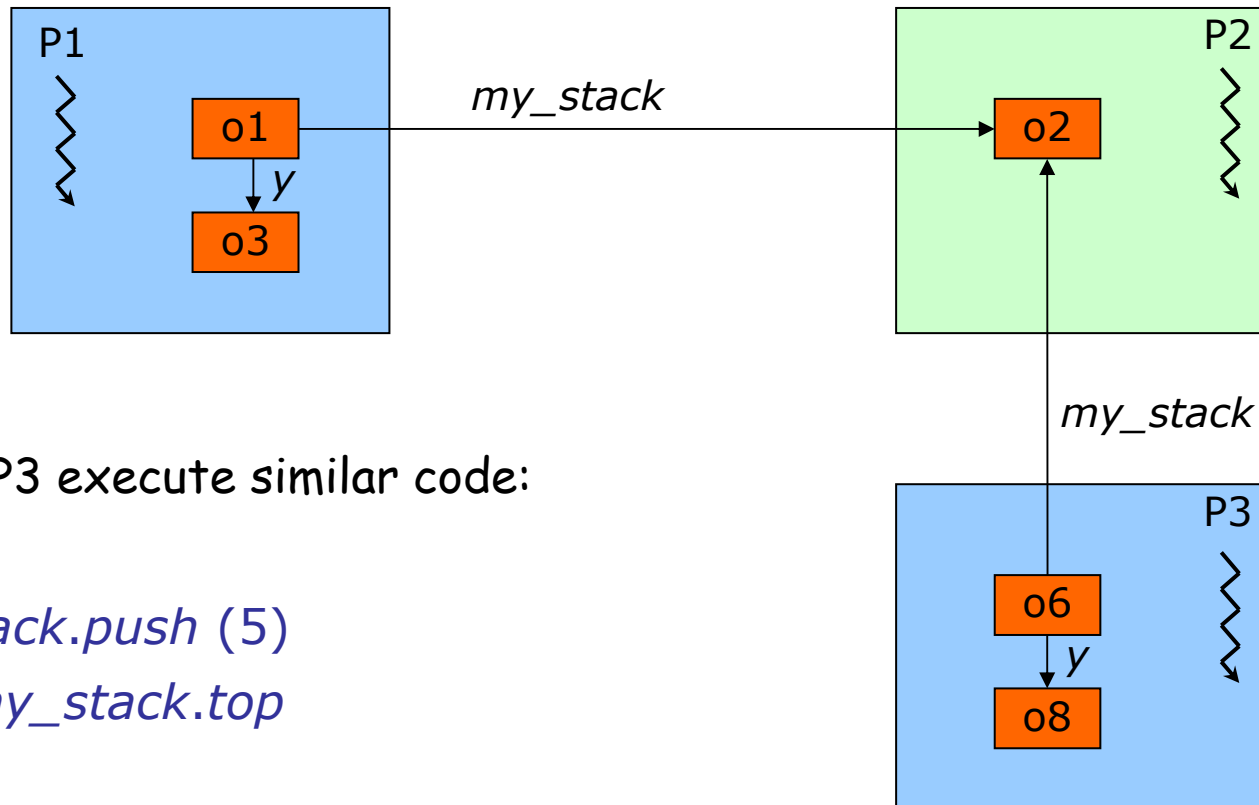
What could have
happened here?

We need a **critical section** to avoid atomicity violations.



Problematic scenario

16



P1 and P3 execute similar code:

-- P1

my_stack.push (5)

y := my_stack.top

-- P3

my_stack.push (100)

y := my_stack.top



Mutual exclusion in SCOOP

17

- Require target of separate call to be formal argument of enclosing routine:

```
push_and_retrieve (s: separate STACK [INTEGER];  
                    value: INTEGER)  
    -- Push `value' on top of `s' then retrieve top of `s'  
    -- and assign it to `y'.  
  
    { do  
      s.push (value)  
      y := s.top ← No other processor can  
                  access s in the meantime  
    } end
```

```
my_stack: separate STACK [INTEGER]
```

```
...
```

```
push_and_retrieve (my_stack, 5) -- Now we are we sure that y=5
```

- Body (do ... end) of enclosing routine is a **critical section** with respect to its separate formal arguments.



Separate argument rule

18

The target of a separate call must be a formal argument of the enclosing routine

Separate call: $a.f(\dots)$ where a is a separate entity



Wait rule

A routine call with separate arguments will execute when all corresponding objects are available

and hold them exclusively for the duration of the routine



Condition synchronisation

20

- Very often client only wants to execute certain feature if some condition (guard) is true:

```
store (buffer: separate BOUNDED_BUFFER [INTEGER];  
      value: INTEGER) is
```

```
    -- Store `value' into `buffer'.
```

```
require
```

```
    buffer_not_full: not buffer.is_full
```

```
do
```

```
    buffer.put (value)
```

```
end
```

Hey, it's a precondition,
not a guard!

How should it work?

```
my_buffer: separate BOUNDED_BUFFER [INTEGER]
```

```
...
```

```
store (my_buffer, 5)
```



Back to the basics: contracts in Eiffel

21

```
store (buffer: BUFFER [INTEGER]; value: INTEGER)
```

```
is
```

```
-- Store `value' into `buffer'.
```

```
require
```

```
buffer_not_full: not buffer.is_full
```

```
value > 0
```

```
do
```

```
buffer.put (value)
```

```
ensure
```

```
buffer_not_empty: not buffer.is_empty
```

```
end
```

```
...
```

```
store (my_buffer, 10)
```

Precondition

Postcondition



From preconditions to wait-conditions²²

```
store (buffer: separate BUFFER [INTEGER]; value: INTEGER)
```

```
is
```

```
-- Store `value' into `buffer'.
```

```
require
```

```
buffer_not_full: not buffer.is_full
```

```
value > 0
```

```
do
```

```
buffer.put (value)
```

```
ensure
```

```
buffer_not_empty: not buffer.is_empty
```

```
end
```

```
...
```

```
store (my_buffer, 10)
```

Preconditions have
wait semantics



Why new semantics?

23

- Preconditions are obligations that client has to satisfy before the call

$\{Pre_r\}$ **call** r $\{Post_r\}$

- Easy peasy:

```
if not my_buffer.is_full and then 5 then  
    store (my_buffer, 5)  
end
```

I know that precondition holds before the call!



Wait rule revisited

24

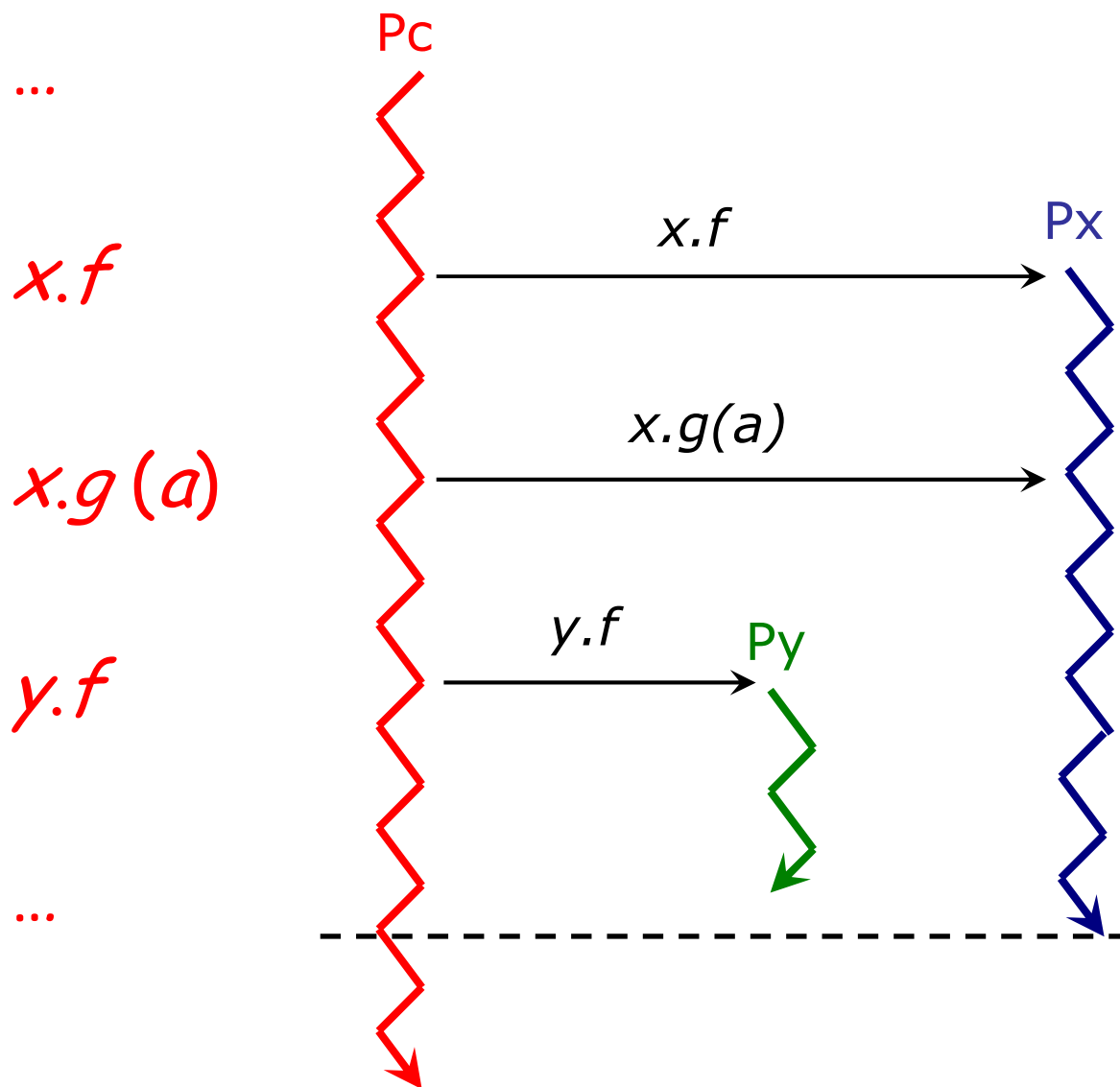
A routine call with separate arguments
will execute when all corresponding
objects are available

and preconditions are satisfied

and hold the objects exclusively for the
duration of the routine



Resynchronising clients and suppliers





Wait by necessity

26

- No explicit mechanism for resynchronisation after separate call.
- Client will only wait when it needs to:

x.f

x.g (a)

y.f

...

value := x.some_query

Wait here!

- This is called **wait-by-necessity**



Do we *really* need to wait?

27

- Can we do better than that?

x.f

x.g (a)

y.f

...

value := x.some_query

x.f

y.f

z := value

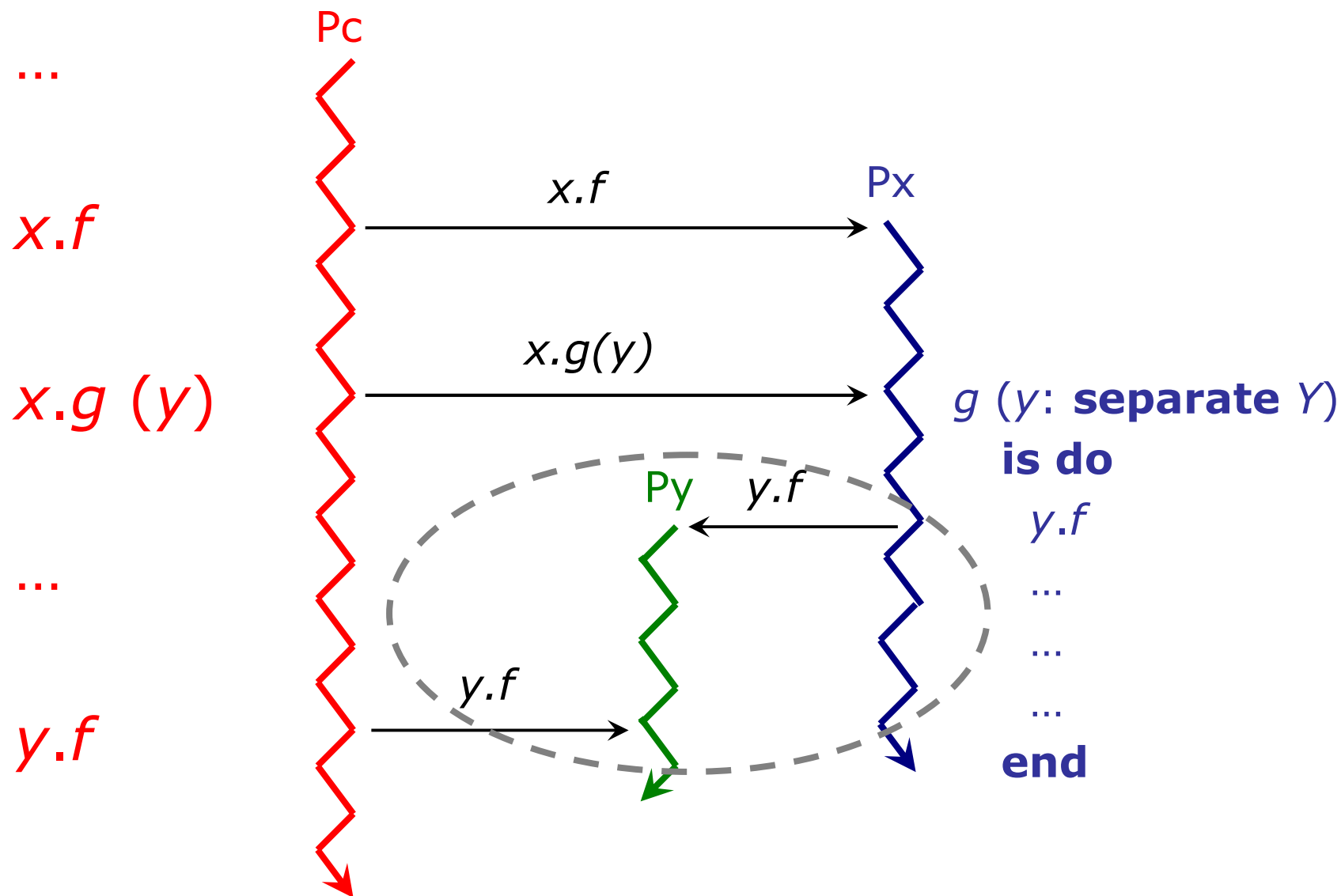
value := value + 1

We only need
to wait here!

- Does not change the basic SCOOP model
- Consider it to be an optimisation



A problem





Lock passing

29

- Original SCOOP approach:
 - Make x wait until y becomes available
 - "Business Card principle" for dealing with tricky cases
 - Not flexible
- Lock passing approach:
 - Let x get exclusive access on y immediately
 - "Pass the lock on y "
 - But: client that passes the lock has to wait
 - In fact, client can pass all the locks
 - You can still implement previous scenario



Lock passing

r (*x*: separate *X*; *y*: separate *Y*) is

do

x.f

x.g (*y*) -- Pass your locks to *x* and wait for *x* to finish.

y.f

...

value := *x.some_query*

end

Both calls are synchronous!



Summary: computational model

31

- Software system is composed of several **processors**
- Processors are **sequential**; concurrency is achieved through their interplay
- Separate entity denotes a *potentially* **separate object**
- Calls to non-separate objects are **synchronous**
- Calls to separate objects are **asynchronous**



Summary: computational model

32

- Mutual exclusion
 - Locking through **argument passing**
 - Routine **body** is **critical section**
- Condition synchronisation
 - **wait-semantics for preconditions**
- Re-synchronisation of client and supplier:
 - **wait-by-necessity**
- **Lock passing** through argument passing



Summary: wait rule

33

A routine call with separate arguments
will execute when all corresponding
objects are available

and wait-conditions are satisfied

and hold the objects exclusively for the
duration of the routine