Concepts of Concurrent Computation

Spring 2015

Lecture 6: SCOOP

Sebastian Nanz
Chris Poskitt

What is SCOOP?

- An object-oriented programming model for concurrency: Simple Concurrent Object-Oriented Programming
- Goal: "reasonability" – the programmer's ability to reason about the execution of programs based only on their text
  - As in sequential O-O programming, with contracts
  - Guarantee: freedom from data races
- Programmers don't have to explicitly manage locking: the runtime of SCOOP handles it, according to their specifications

SCOOP design and development

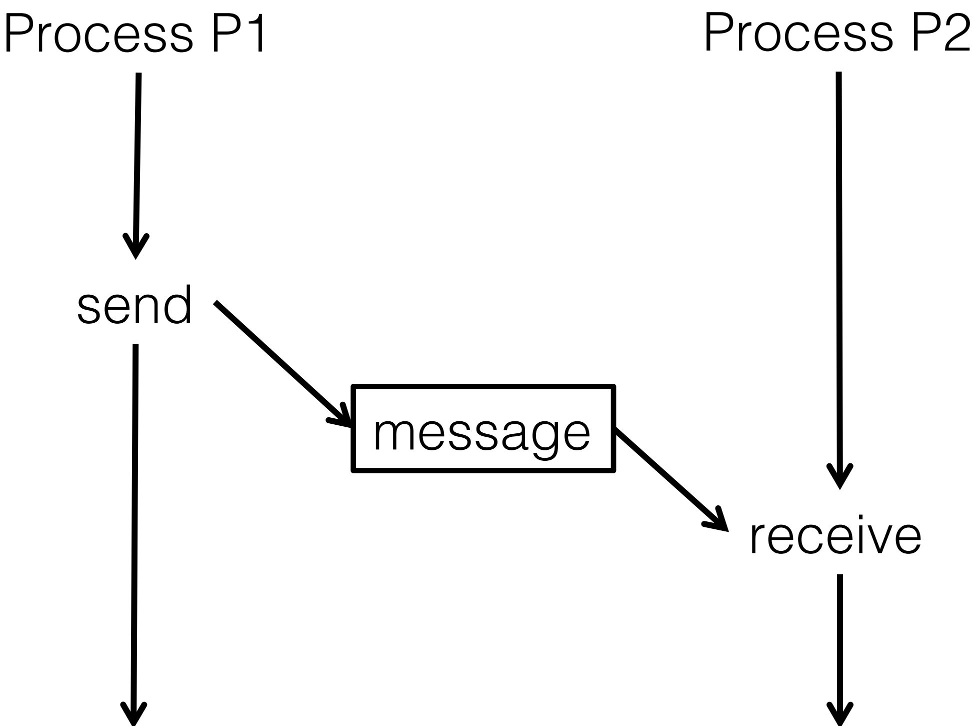
- First version described by Bertrand Meyer in a CACM article (1993) and in chapter 32 of Object-Oriented Software Construction, 2nd edition, 1997
- Prototype implementation at ETH Zurich (2005–2008)
- Improvements and extensions of the model at ETH Zurich (2009–today), in particular new runtime
- Further development supported by an Grant from the European Research Council (2012–2017)
  - CME Project, more information at cme.ethz.ch
- Production implementation at Eiffel Software, part of EiffelStudio

Message-passing communication

- The communication between processes in SCOOP is based on message-passing communication
- Communication is achieved by sending messages between processes, in contrast to shared-memory communication
- Models of message-passing communication include the Actor model and the π-calculus (see later lectures)
- The main distinction is between synchronous and asynchronous message passing (see following slides)

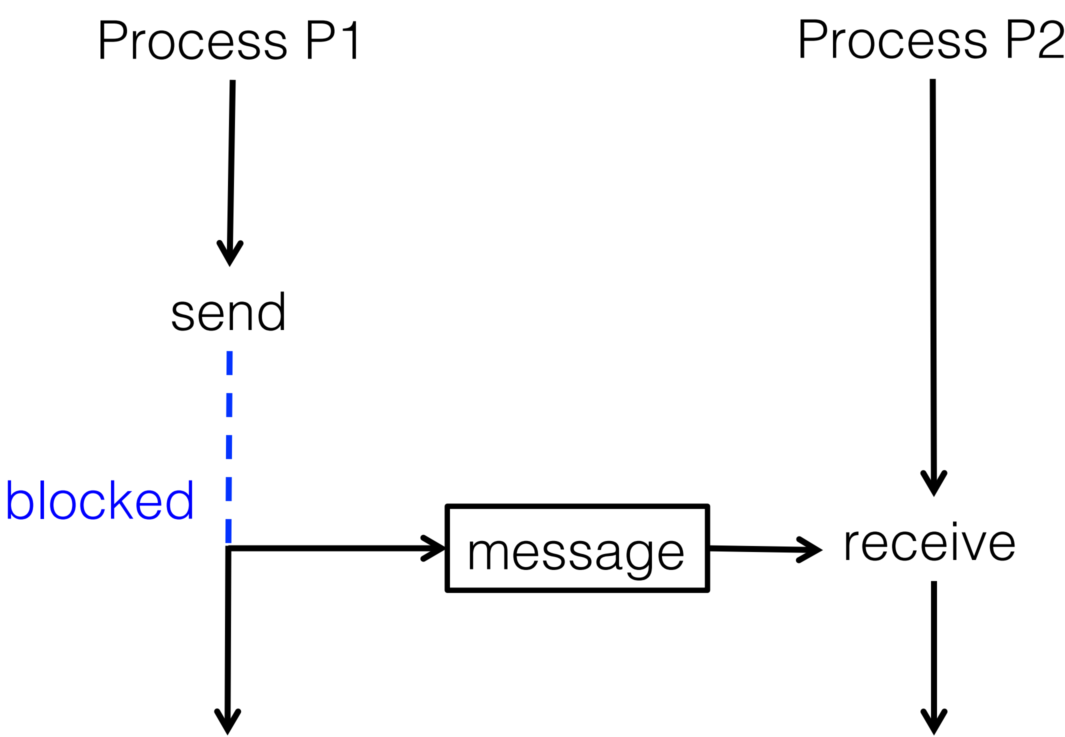
Asynchronous message passing

- Asynchronous: the sender sends a message and continues, regardless of whether the message has been received
- Requires buffer space
- **Analogy:** Email



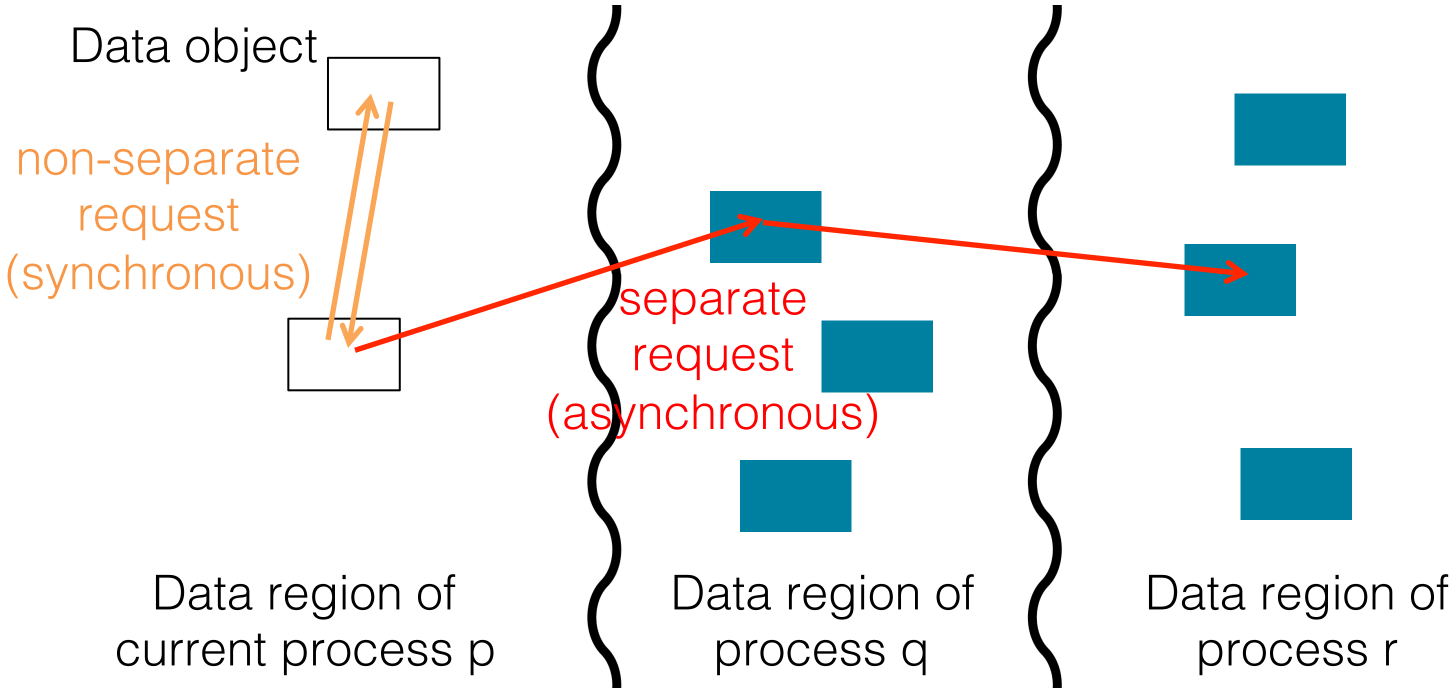
Synchronous message passing

- Synchronous: the sender blocks until the receiver is ready to receive the message
- **Analogy:** Phone call



SCOOP: A message-passing model

- Intuition
  - Every process has its own data it handles exclusively; other processes may request to execute operations on this data on their behalf – this ensures freedom from data races
  - Requests are synchronous on **non-separate** (local) data, and asynchronous on **separate** (remote) data



Processors and regions

- Terminology: a process handling sequential execution on the objects it owns is called a processor
- All calls targeting a given object will be executed by a single processor, its handler
- This partitions the set of objects into regions
- Objects located on different regions from the point of view of the current processor are called separate
- The object locality is captured by the type system

```
                      -- object located on:
  ▪ x: T              --   the current processor
  ▪ x: separate T     --   a potentially different processor
```

Object and processor creation

- Creation of separate objects
  - Create a new processor
  - Place the object on the new processor
- Example

```
x: separate X
y: Y

create x  -- create new processor and place x on it
create y  -- place y on the current processor
```

Concurrent execution

- Fundamental semantic rule: a call `x.r (a)` is
  - synchronous for non-separate `x`
  - potentially asynchronous for separate `x`
- Why only potentially asynchronous?
  - Dynamic target type not separate
    - A **`separate`** declaration does not specify the processor: only states that the object might be handled by a different processor
    - E.g., in some execution, the value of `x.y` might be a reference to an object in the current region (including **`Current`**)
  - Resynchronization (wait-by-necessity) – discussed later
  - Lock passing – discussed later

Trusting what you read ("reasonability")

- Potential interference of other threads makes it difficult to interpret concurrent programs

  Assume we define `b : separate STACK [INTEGER]`

  and then have this code:

```
b.push (10)
-- instructions that are
-- not affecting the buffer
x := b.top
-- x = ?
```

Interfering instructions from other threads

- SCOOP addresses this problem by defining pieces of code (the routine bodies) that can be reasoned about sequentially

Exclusive access

- **Exclusive access guarantee:** A routine call guarantees exclusive access to the handlers of **all** separate arguments

```
compute (b: separate STACK [INTEGER])
  do
    b.push (10)
    -- instructions not affecting the buffer...
    x := b.top
    -- x = 10
  end
```

Exclusive access to b

- **Separate argument rule:** SCOOP requires the target of a separate call to be a formal argument of the enclosing routine
- If the rule is obeyed, we call the target controlled
- Guarantee + rule ensure sequential reasoning within routine bodies

A downside: Wrappers

- The separate argument rule makes it necessary to wrap also single calls on separate targets

  Instead of `b.push (10)` you have to define

  ```
  wrap_push (b: separate STACK [INTEGER]; i: INTEGER)
    do
      b.push (i)
    end
  ```

  and then call `wrap_push (b, 10)`

- There are suggestions for alternative syntax which make this wrapping unnecessary, but they are not implemented yet ☹

Example: Dining Philosophers (1)

```
class PHILOSOPHER
feature
  make (left, right: separate FORK)
    do
      left_fork := left
      right_fork := right
    end
  eat (left, right: separate FORK) do ... end
    -- Exclusive access to both the left and the right forks is
    -- secured before eating (multi-reservation of exclusive access)
  think do ... end
  live
    do
      from until false loop
        think
        eat (left_fork, right_fork)
      end
    end

feature {NONE}
  left_fork: separate FORK
  right_fork: separate FORK
end
```

Example: Dining Philosophers (2)

```
class DINING_PHILOSOPHERS
feature
  make
    local
      left_fork: separate FORK
      right_fork: separate FORK
      philosopher: separate PHILOSOPHER
    do
      -- create n philosophers and launch them
      ...
      create philosopher.make (left_fork, right_fork)
      launch_philosopher (philosopher)
      ...
    end

feature {NONE}
  launch_philosopher (a_philosopher: separate PHILOSOPHER)
    do
      a_philosopher.live
    end
end
```

Resynchronization: Wait-by-necessity

- Resynchronization after a separate call uses the wait-by-necessity mechanism
- The client will wait for the result of a query

```
x.f
x.g (a)
y.f
...
value := x.some_query -- Wait until all calls on x have
                      -- finished and the value is returned
```

- Synchrony vs. asynchrony revisited:

  for a separate target `x`

  - `x.command (...)` is potentially asynchronous
  - `v := x.query (...)` is always synchronous

Lock passing

- Motivation
  - A processor p requests that a processor q executes a feature, but q needs access to objects to which p has currently exclusive access
  - This leads to a deadlock

```
f (x: separate X; y: separate Y)
  do
    x.g (y)  -- x waits for y to become available,
             -- to which Current has exclusive access
  end
```

- To avoid this situation, lock passing is used
  - If the client has exclusive access to arguments of a separate call, the client transfers the exclusive access to the supplier until the call returns
  - The client has to wait until it regains exclusive access

Condition synchronization

- How to express condition synchronization?
  - Use contracts: preconditions become wait conditions
  - Elegant: conditions are explicit (as boolean expressions)

```
put (b: separate BOUNDED_BUFFER [INTEGER]; i: INTEGER)
    -- Store i into buffer b.
  require
    not b.is_full  -- wait condition
  do
    b.append (i)
  end
```

- Semantics: A call with separate arguments and wait condition waits until
  - all corresponding objects are available and
  - the wait condition is fulfilled

Example: Producer-Consumer

```
class PRODUCER
feature
  put (b: separate BOUNDED_BUFFER [INTEGER]; i: INTEGER)
    require
      not b.is_full
    do
      b.append (i)
    end
  ...
end



class CONSUMER
feature
  take (b: separate BOUNDED_BUFFER [INTEGER]): INTEGER
    require
      not b.is_empty
    do
      ... := b.remove
    end
  ...
end
```

Wait conditions vs. correctness conditions

- The following example mixes a wait condition and a correctness condition

```
put (b: separate BOUNDED_BUFFER [INTEGER]; i: INTEGER)
    -- Store i into buffer b.
  require
    not b.is_full   -- wait condition
    i > 0           -- correctness condition
  do
    b.append (i)
  end
```

- Two different semantics
  - Separate: wait condition (wait until fulfilled)
  - Non-separate: correctness condition (fail if not fulfilled)

Type System

Type system: Intuition

- The semantics of a call changes depending on whether a call is executed on a separate or non-separate target
  - non-separate: synchronous
  - separate: potentially asynchronous
- The type system ensures that the semantics is preserved

  Assume we have

```
nonsep: T
sep: separate T
```

  then

```
sep := nonsep -- ✅ allowed: calls on sep will be always
              -- synchronous, but separate guarantees
              -- only potentially asynchronous
```

```
nonsep := sep -- ❌ not allowed: calls on nonsep will be
              -- potentially asynchronous, but non-separate
              -- is supposed to guarantee synchronous
```

Type system: Subtyping rules

- Programming languages with subtyping
  - Subtyping relation $D \subseteq C$, meaning that D is a subtype of C
  - $D \subseteq C$ expresses that an entity of type D can be safely used in a context where C is expected
- Separate types are pairs (α, C)
  - Separateness $\alpha \in \{$non-separate, separate$\}$
  - Regular type C
- Given the subtyping relation of Eiffel, we can derive the subtyping relation for separate types
  1. For all C, D, α: $D \subseteq_{Eiffel} C \Leftrightarrow (\alpha, D) \subseteq_{SCOOP} (\alpha, C)$
  2. For all C, α: $(\text{non-separate}, C) \subseteq_{SCOOP} (\text{separate}, C)$

Example: Subtyping

```
-- B inherits from A
a: separate A
b: B
c: separate B
f (x: separate A; y: B) do ... end
```

- `f (a, b)` ✅
- `f (a, c)` ❌
- `f (b, b)` ✅
- `a := b` ✅
- `a := c` ✅
- `b := c` ❌
- `c := b` ✅

Example: Passing a string

```
-- Client
s: STRING

f (b: separate B)
  do
    b.g (s)
  end
```

```
-- Supplier
class B
feature
  g (s1: separate STRING)
    do
      create s.make_from_separate (s1)
    end

  s: STRING
end
```

- In this example, a string needs to be passed
  - An assignment `s := s1` in feature `g` would not be valid
  - Therefore `make_from_separate` is used
- Also, this is another example for lock passing
  - The client has exclusive access to `s`, as it is non-separate
  - In the call `b.g (s)`, lock passing occurs

Type combinators

- Result type T_{query_result} of a query call `x.f (...)`?

  $T_{query_result} = (\alpha_x * \alpha_f, C_{f_result})$ where $\alpha_x * \alpha_f$ is defined as:

α_x \ α_f	non-separate	separate
non-separate	non-separate	separate
separate	separate	separate

- Expected actual argument type T_{actual} in `x . f (a)`?

  $T_{actual} = (\alpha_x * \alpha_{formal}, C_{formal})$ where $\alpha_x * \alpha_{formal}$ is defined as:

α_x \ α_{formal}	non-separate	separate
non-separate	non-separate	separate
separate	$\bot$	separate

Not possible: actual argument must be non-separate from the target, not the client

Example: Result type combinator

- Is the following example accepted by the compiler?

```
-- Client
a: A

r (x: separate T)
  do
    a := x.b
  end
```

```
-- Supplier
class T
feature
  b: A
end
```

Example: Argument type combinator

- Is the following example accepted by the compiler?

```
-- Client
b: A

r (x: separate Z)
  do
    x.f (b)
  end
```

```
-- Supplier
class Z
feature
  f (a: A)
     do
       a.f
     end
end
```

Expanded classes

- Eiffel has so-called expanded classes (e.g. `INTEGER`) where values are actual values instead of references
- Expanded objects can be passed to separate calls even if the supplier expects a non-separate argument
- Example: when replacing reference type A in the previous example by expanded type `INTEGER`, the example is valid

```
-- Client
b: INTEGER

r (x: separate Z)
  do
    x.f (b)
  end
```

```
-- Supplier
class Z
feature
  f (a: INTEGER)
     do
       a.f
     end
end
```

Dynamic type of a separate object

- We can use Eiffel's object tests to determine the dynamic type of a separate object
- An object test succeeds if the run-time type of its source conforms in all of
  - Detachability (see Eiffel's void-safety mechanism)
  - Locality
  - Class type to the type of its target
- We can downcast a separate entity to a non-separate one if the entity represents a non-separate object at runtime

```
meet_friend (p: separate PERSON)
    do
      if attached {PERSON} p.friend as ap then
        visit (ap)
      end
    end
```

Genericity

- Entities of generic types may be separate

```
list: LIST [BOOK]
list: separate LIST [BOOK]
```

- Actual generic parameters may be separate

```
list: LIST [separate BOOK]
list: separate LIST [separate BOOK]
```

- All combinations are meaningful and can be useful

- Separateness is relative to the object of the generic class

  e.g. elements of `list: separate LIST [BOOK]` are non-separate with respect to `list` but separate with respect to **`Current`**

Implementation

Reasoning guarantees

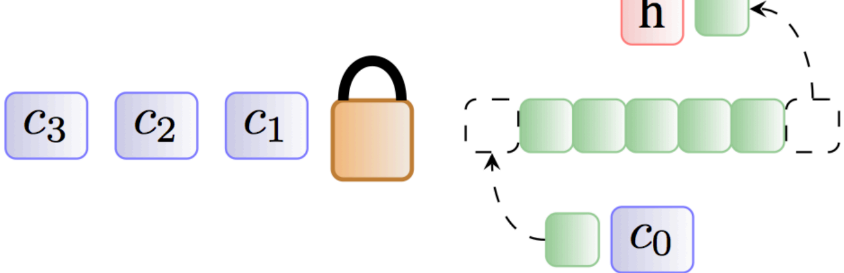
- There are two key reasoning guarantees that an implementation of SCOOP must provide
  1. Non-separate calls execute immediately and are synchronous
  2. Calls to another handler to which exclusive access is granted will be executed in the order they are logged, and there will be no intervening calls logged from other clients
- The second guarantee provides strong control over the order in which messages are processed

  In other message-passing models (e.g. the Actor model, see later lecture) the sending processes typically do not know the order of processing of their messages

How to achieve exclusive access (1)

- To achieve the reasoning guarantees, a simple lock-based scheme can be used

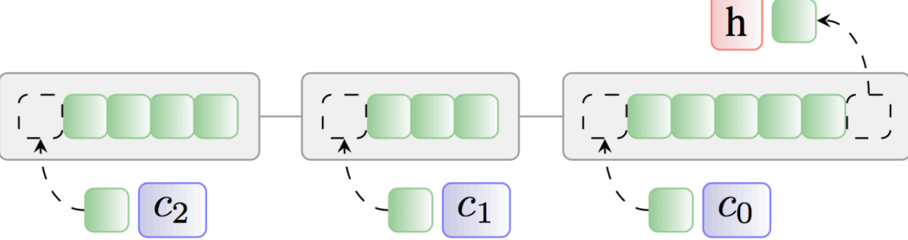
  Client c_0 places calls in a queue for handler h to dequeue and process; other clients c_1, …, c_3 must wait until the current client is finished



- Because of the contention for the handler lock, this solution is inefficient

How to achieve exclusive access (2)

- Instead of the naïve implementation, we can use multiple queues that can be enqueued into by multiple clients
  - Each handler maintains a queue-of-queues (gray boxes)
  - Each client has their own private queue (green boxes) it enqueues into, finishing with an end token
  - A handler processes only one private queue at a time and moves on to the next only when the end token is reached, maintaining the reasoning guarantees



- Handler contention is no longer a performance bottleneck, as clients can enqueue requests at any time

Conclusions

Trade-offs

- Advantages
  - Safety: no data races
  - “Reasonability”: processing order of messages as expected
  - Lock management taken care of by the runtime
  - Elegant integration of condition synchronization
  - Reservation of multiple objects at once
- Disadvantages
  - Need to “wrap” feature calls – fixable by syntax, to be released
  - Restrictions imposed on the programmer, reduced flexibility
  - Very different from models with threads and locks, learning curve
  - Performance: current implementation is among the most efficient of safe concurrent languages, but in general safety is not without cost
  - Currently, no multiple readers – fixed by an extension of the model (passive processors), to be released