Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 4:

Introduction to concurrency & SCOOP
The SCOOP programming model

Basic operation of OO programming: \( x.f(\ldots) \)

Can be a command or a query:

\[
\begin{align*}
& \quad r(c: \text{separate CONFERENCE} ; p: \text{PAPER}) \\
& \quad \text{require} \\
& \quad \quad c.\text{submission\_open} \\
& \quad \text{do} \\
& \quad \quad c.\text{submit}(p) \\
& \quad \quad \ldots \\
& \quad \quad \text{if } c.\text{accepted}(p) \text{ then } \text{rejoice} \text{ end} \\
& \quad \text{end} \\
& \quad r(\text{icse} , \text{latest}) \\
& \quad \text{-- Exclusive access when needed}
\end{align*}
\]
Three risks

Data race
- Incorrect concurrent access to shared data

Deadlock
- Computation cannot progress because of circular waiting

Starvation
- Execution favors certain processes over others, which never get executed
Thank you for calling Ecstatic Opera Company. How can I help you?

(Joan) I need a single seat for next Tuesday’s performance of Pique Dame.

Let me check... You’re in luck! Just one left. Eighty dollars.

Great. I’ll go for it.

Just a moment while I book it.

Thanks.

Sorry, there are no more seats available for Tuesday.
## Data race: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
<th>Available seats</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Theatre</td>
<td><em>Available seats?</em></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Jane</td>
<td><em>Seats left?</em></td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Joan</td>
<td><em>Seats left?</em></td>
<td>Yes</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Joan (fast to react)</td>
<td><em>Please book!</em></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Jane (slow to react)</td>
<td><em>Please book!</em></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Jane’s agent (fast to act)</td>
<td><em>Try to book</em></td>
<td><strong>Success</strong></td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Joan’s agent (slow to act)</td>
<td><em>Try to book</em></td>
<td><strong>Failure</strong></td>
<td>0</td>
</tr>
</tbody>
</table>
I’d like to change my Tuesday evening seat for the matinee performance.

Both shows are sold out, but I heard there was a customer who wanted to change the other way around. Matinee booking is handled by a different office, so let me call them and make the change.

Thanks.

(Ten minutes later.) “The number is still busy.”
# Deadlock: scenario

<table>
<thead>
<tr>
<th>Time step</th>
<th>Active participant</th>
<th>Request or action</th>
<th>Answer or result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Agent 1</td>
<td>Matinee available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Agent 2</td>
<td>Evening available for exchange?</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Agent 1</td>
<td>Start dialing call to agent 2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Agent 2</td>
<td>Start dialing call to agent 1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Agent 1</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 2 is trying to call</td>
</tr>
<tr>
<td>6</td>
<td>Agent 2</td>
<td>Finish dialing</td>
<td>Busy signal, because agent 1 is trying to call</td>
</tr>
<tr>
<td>7</td>
<td>Agent 1 &amp; Agent 2</td>
<td>Repeat steps 3 to 6 forever as the result remains the same: busy signals</td>
<td></td>
</tr>
</tbody>
</table>
Starvation

Jane keeps calling, but agents always pick up someone else's call
### Execution sequences

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x := 0</strong></td>
<td></td>
</tr>
<tr>
<td><strong>P1</strong></td>
<td><strong>P2</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>x := 0</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>x := x + 1</strong></td>
</tr>
</tbody>
</table>

- **Execution can give rise to this execution sequence:**

  Instruction executed with Thread ID and line number

  Variable values after execution of the code on the line
Execution sequences

Possible execution sequences considering all interleavings:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1</td>
<td>x := 2</td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td>x := 0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>x := 0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>x := 2</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>x := 0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>x := x + 1</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>x := 2</td>
</tr>
</tbody>
</table>
Data races (race conditions)

If processes (OS processes, threads) are completely independent, concurrency is easy

Usually, however, threads *interfere* with each other by accessing and modifying common resources, such as variables and objects

- Unwanted dependency of the computation’s result on nondeterministic interleaving is a *race condition* or *data race*
- Such errors can stay hidden for a long time and are difficult to find by testing
Dining philosophers
The dining philosophers problem

$n$ philosophers are seated around a table; between each pair there is a single fork.
Each philosopher only thinks and eats.
To eat, a philosopher needs both left and right forks (so two adjacent philosophers cannot eat at the same time).

The problem: devise an algorithm enabling philosophers to follow this scheme, without deadlock.
Dining philosophers: solution attempt 1

Each philosopher first picks up the right fork, then the left fork, and then starts eating; after having eaten, the philosopher puts down the left fork, then the right one.

- The philosophers can deadlock!
Dining philosophers: solution attempt 2

Each philosopher successively:

- Picks up right fork and the left fork *at the same time*
- Starts eating
- After having eaten, puts them both back down

A philosopher could *starve!*
SCOOP background

Simple Concurrent Object-Oriented Programming

First version described in CACM article (1993) and chapter 32 of Object-Oriented Software Construction, 2nd edition, 1997

Prototype implementation at ETH (2005-2010)
Recent production implementation at Eiffel Software, part of EiffelStudio

Recent descriptions: Piotr Nienaltowski’s 2007 ETH PhD; Morandi, Nanz, Meyer (2011)
Example 1: bank transfer, from sequential to concurrent

```plaintext
transfer (source, target: ACCOUNT; amount: INTEGER)

-- Transfer amount, if available, from source to target.
do
  if source.balance >= amount then
    source.withdraw (amount)
    target.deposit   (amount)
  end
end
```

```
transfer (Jane, Jill, 100)
Jane  Jill  Joan
100   0    0
transfer (Jane, Joan, 100)
-100  0    100
```
Bank transfer (better version)

transfer (source, target: ACCOUNT; amount: INTEGER)

-- Transfer amount from source to target.

require
  source.balance >= amount

do
  source.withdraw (amount)
  target.deposit (amount)

ensure
  source.balance = old source.balance - amount
  target.balance = old target.balance + amount

end
Example 2: hexapod robot

Hind legs have force sensors on feet and retraction limit switches

Ganesh Ramanathan, Benjamin Morandi, IROS 2011
Hexapod locomotion

Alternating protraction and retraction of tripod pairs

- Begin protraction only if partner legs are down
- Depress legs only if partner legs have retracted
- Begin retraction when partner legs are up
Hexapod coordination rules

**R1**: Protraction can start only if partner group on ground

**R2.1**: Protraction starts on completion of retraction

**R2.2**: Retraction starts on completion of protraction

**R3**: Retraction can start only when partner group raised

**R4**: Protraction can end only when partner group retracted

Sequential implementation

TripodLeg lead = tripodA;
TripodLeg lag = tripodB;

while (true)
{
    lead.Raise();
    lag.Retракt();
    lead.Swing();
    lead.Drop();

    TripodLeg temp = lead;
    lead = lag;
    lag = temp;
}
Multi-threaded implementation

```csharp
private object m_protractionLock = new object();

private void ThreadProcWalk(object obj)
{
    TripodLeg leg = obj as TripodLeg;
    while (Thread.CurrentThread.ThreadState != ThreadState.AbortRequested)
    {
        // Waiting for protraction lock
        lock (m_protractionLock)
        {
            // Waiting for partner leg drop
            leg.Partner.DroppedEvent.WaitOne();
            leg.Raise();
        }
        leg.Swing();

        // Waiting for partner retraction
        leg.Partner.RetractedEvent.WaitOne();
        leg.Drop();

        // Waiting for partner raise
        leg.Partner.RaisedEvent.WaitOne();
        leg.Retract();
    }
}
```
begin_protracion(partner, me:separate LEG_GROUP_SIGNALER)

require
  my_legs_retracted : me.legs_retracted
  partner_down : partner.legs_down
  partner_not_protracting : not partner.protracion_pending

do
  io.put_string (group_name)
  io.put_string (" : begin_protracion ")
  io.put_new_line

  tripod.lift

  me.set_protracion_pending(true)

end
Hexapod coordination rules

R1: Protraction can start only if partner group on ground
R2.1: Protraction starts on completion of retraction
R2.2: Retraction starts on completion of protraction
R3: Retraction can start only when partner group raised
R4: Protraction can end only when partner group retracted

Example 3: dining philosophers

Listing 4.33: Variables for Tanenbaum’s solution

```python
1 state = ['thinking'] * 5
2 sem = [Semaphore(0) for i in range(5)]
3 mutex = Semaphore(1)
```

The initial value of `state` is a list of 5 copies of `‘thinking’`. `sem` is a list of 5 semaphores with the initial value 0. Here is the code:

Listing 4.34: Tanenbaum’s solution

```python
1 def get_fork(i):
2     mutex.wait()
3     state[i] = 'hungry'
4     test(i)
5     mutex.signal()
6     sem[i].wait()
7
8 def put_fork(i):
9     mutex.wait()
10    state[i] = 'thinking'
11    test(right(i))
12    test(left(i))
13    mutex.signal()
14
15 def test(i):
16    if state[i] == 'hungry' and
17    state(left(i)) != 'eating' and
18    state(right(i)) != 'eating':
19        state[i] = 'eating'
20        sem[i].signal()
```
Dining philosophers in SCOOP

class PHILOSOPHER feature
  live
    do
      from getup until over loop
        think ;
        eat (left, right)
      end
    end
  end

  eat (l, r: separate FORK )
    -- Eat, having grabbed l and r.
    do ... end

  getup do ... end
  over: BOOLEAN
end
The design of SCOOP

SCOOP intends to make concurrent programming as predictable as sequential programming.

A key criterion is “reasonability” (not a real word!): the programmer’s ability to reason about the execution of programs based only on their text.

- As in sequential O-O programming, with contracts etc.

SCOOP is not a complete rework of basic programming schemes, but an incremental addition to the basic O-O scheme: one new keyword

- “Concurrency Made Easy”
Handling concurrency simply

SCOOP narrows down the distinction between sequential & concurrent programming to six properties, studied next:

- **(A)** Single vs multiple “processors”
- **(B)** Regions
- **(C)** Synchronous vs asynchronous calls
- **(D)** Semantics of argument passing
- **(E)** Semantics of resynchronization (lazy wait)
- **(F)** Semantics of preconditions
The starting point (A): processors

To perform a computation is
- To apply certain actions
- To certain objects
- Using certain processors

Sequential: one processor
Concurrent: any number of processors
What makes an application concurrent?

**Processor:**
Thread of control supporting sequential execution of instructions on one or more objects

Can be implemented as:
- Computer CPU
- Process
- Thread
- AppDomain (.NET) ...

The SCOOP model is abstract and does not specify the mapping to such actual computational resources.
Object-oriented programming

The key operation is “feature call”

\[ x \cdot f (\text{args}) \]

where \( x \), the \textbf{target} of the call, denotes an object to which the call will apply the feature \( f \)

Which processor is in charge of executing such a call?
All calls targeting a given object will be executed by a single processor

- The set of objects handled by a given processor is called a **region**
- The processor in charge of an object is its **handler**
SCOOP restriction: one handler per object

- One processor per object: "handler"

- At most one feature (operation) active on an object at any time
The sequential view: O-O feature calls

\[ x.r(a) \]

- **Client**
  - previous
  - \( x.r(a) \)
  - next

- **Supplier**
  - \( r(x: A) \)
  - do
  - ... 
  - end

- **Processor**
(C) The concurrent form of call: asynchronous

Client

\[ x.r(a) \]

\[ \text{next} \]

\[ \text{previous} \]

Client's handler

Supplier

\[ r(x : A) \]

\[ \text{do} \]

\[ \ldots \]

\[ \text{end} \]

Supplier's handler
The two forms of O-O call

To wait or not to wait:

- If same processor, synchronous
- If different processor, asynchronous

Difference must be captured by syntax:

- `x: T`
- `x: separate T` -- Potentially different processor

Fundamental semantic rule: a call `x.r(a)`

- Waits (i.e. is synchronous) for non-separate `x`
- Does not wait (is asynchronous) for separate `x`
Consistency rules: avoiding traitors

\[
\text{nonsep} : T
\]

\[
\text{sep} : \text{separate } T
\]

\[
\text{nonsep} := \text{sep}
\]

\[
\text{nonsep}.p(a)
\]
Since separate calls are asynchronous there is a real danger of confusion
Consider for example

```plaintext
r(remote_stack: separate STACK[T])
   do
      ...
      remote_stack.push(a)
      ...
      Instruction not affecting the stack...
      y := remote_stack.top
   end
```
Access control policy

SCOOP requires the target of a separate call to be a **formal argument** of enclosing routine:

```python
put(s: separate STACK[T]; value: T)
-- Store value into s.

do
  s.put(value)
end
```

To use separate object:

```python
my_stack: separate STACK[INTEGER]
create my_stack
put(my_stack, 10)
```
Separate argument rule

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

Separate call: $x \cdot f(...) \text{ where } x \text{ is separate}$
(D) Wait rule

A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments

\[ a\_routine (\text{nonsep}_a, \text{nonsep}_b, \text{sep}_c, \text{sep}_d, \text{sep}_e) \]

Exclusive access to \textit{sep}_c, \textit{sep}_d, \textit{sep}_e within \textit{a\_routine}
An example: from sequential to concurrent

```plaintext
transfer (source, target: ACCOUNT; amount: INTEGER)
    -- Transfer amount, if available, from source to target.
    do
        if source.balance >= amount then
            source.withdraw (amount)
            target.deposit (amount)
        end
    end
```

**Note**: The `separate` keyword indicates a change from sequential to concurrent execution in the context of a method or function.
class PHILOSOPHER feature
  live
  do
    from getup until over loop
      think; eat (left, right)
    end
  end
end

eat (l, r: separate FORK)
  -- Eat, having grabbed l and r.
  do ... end

getup do ... end
over: BOOLEAN
end
(D) What the wait rule means

Beat enemy number one in concurrent world: atomicity violations

- Data races
- Illegal interleaving of calls

Data races cannot occur in SCOOP
A routine call guarantees exclusive access to the handlers (the processors) of all separate arguments

\( a\_\text{routine}(\text{nonsep}_a, \text{nonsep}_b, \text{sep}_c, \text{sep}_d, \text{sep}_e) \)

Exclusive access to \( \text{sep}_c, \text{sep}_d, \text{sep}_e \) within \( a\_\text{routine} \)
Resynchronization: lazy wait

How do we resynchronize after asynchronous (separate) call? No explicit mechanism!

The client will wait when, and only when, it needs to:

\[ x.f \]
\[ x.g(a) \]
\[ y.f \]
\[
\]
\[ \text{value := } x.\text{some_query} \]

Lazy wait (also known as wait by necessity)
(E) Synchrony vs asynchrony revisited

For a separate target $x$:

- $x \cdot \text{command(...)}$ is asynchronous
- $v := x \cdot \text{query(...)}$ is synchronous
Exercise

If we do want to resynchronize explicitly, what do we do?
What becomes of contracts, in particular preconditions, in a concurrent context?
put \((b: \text{separate QUEUE}[\text{INTEGER}]; v: \text{INTEGER})\)

-- Store \(v\) into \(b\).

require

\[\text{not } b\.\text{is\_full}\]

\[v > 0\]

do

\(b\.\text{put}\,(v)\)

ensure

\[\text{not } b\.\text{is\_empty}\]

end

...

\[\text{put}(\text{my\_buffer}, 10)\]


**Contracts**

```plaintext
put (b: BUFFER [INTEGER] ; i: INTEGER)
    -- Store i into buffer.
    require
        not b.is_full
        i > 0
do
    b.put (i)
ensure
    not b.is_empty
end

... put (my_buffer, 10)
```

**Precondition becomes wait condition**
transfer (source, target: separate ACCOUNT; amount: INTEGER)

-- Transfer amount from source to target.

require
  source.balance >= amount

do
  source.withdraw (amount)
  target.deposit   (amount)

ensure
  source.balance = old source.balance - amount
  target.balance = old target.balance + amount

end
(F) Full synchronization rule

A call with separate arguments waits until:
- The corresponding objects are all available
- Preconditions hold

"Separate call":

\[ x.f(a) \quad -- \text{where } a \text{ is separate} \]
Handling concurrency simply

SCOOP narrows down the distinction between sequential & concurrent programming to six properties, studied next:

- (A) Single vs multiple “processors”
- (B) Regions
- (C) Synchronous vs asynchronous calls
- (D) Semantics of argument passing
- (E) Semantics of resynchronization (lazy wait)
- (F) Semantics of preconditions