Concurrent Object-Oriented Programming

Arnaud Bailly, Bertrand Meyer and Volkan Arslan
Lecture 2: Task Creation and Communication Primitives
What is a Task?

- It is an entity with resources:
  - memory (so-called address space), and
  - processing time.
- A task may have a **priority**.
- A task may **execute** an **activity**.
- A task may be **idle** (waiting).
- A task may **terminate**.
- The **degree of parallelism** of a system at a given time is the number of **active** tasks executing on that system.
Special case: Synchronous Tasks

- **Synchronous tasks:**
  - The same program is executed on all CPUs
  - Progression is in **lock-step**.
  - Tasks are **always** active.
  - Typical of SIMD for Data-Parallelism (issues)

```
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
y := 3;
if (x = y)
x := 1;
```
We shall only consider the **asynchronous** task model!
Tasks execute actions **on behalf** of objects.

- Assume a program counter $PC \in \mathbb{N}$
- Define a partial function $\varepsilon : task \times \mathbb{N} \mapsto (object \times action)$
- $\varepsilon$ is total on $\mathbb{N}$ and can be:
  - surjective ($n \geq 1$ tasks for 1 object) **intra-object parallelism**
  - bijective (1 task for 1 object)
  - injective ($\exists$ passive objects) **not surjective:** inter-object parallelism
An object can be **passive:**
- either not supported by any task, or
- supported by a task but no autonomous behavior.
- **Data structure** or **service repository.**
- It is **re-active,** acting only when called.

An object can be **active:**
- Has a task of its own.
- Method for the object’s **body** (**live** routine).
- **live** terminates: task stops (passivation?).
- Active, it may or may not provide services.
- Many **mixed** approaches (CEiffel, ProActive,...).
The nature of (Inter)actions

- Message passing:
  - explicit send/receive primitives.
- Remote routine invocation:
  - transparent remote invocation,
  - no receive statement necessary.
- Synchronous communication:
  - the client is blocked until the supplier finishes.
- Asynchronous communication:
  - a call never blocks the client.
Relevant issues

How to create tasks?

How to make tasks active?

How to coordinate tasks?
Essentially

Routine invocations execute in client’s task.

Thread 1  Thread 2  c1  c2  c3  b

Thread 3

c1.run  c3.run  c2.run
b.join  b.join  b.join
b.wait  b.wait  b.notifyall
b.join  b.wait  b.notifyall

CPU Time
Passive Objects without Task (1)

- Allow intra-object concurrency.
- Use synchronous interaction.
- Clients can communicate through side-effects.
- Concerns orthogonal to task creation.
- Objects can be garbage-collected.
- Exception: implicit mobility like in Oz.
Passive objects with a Task

- Essentially
- One or many tasks.
- Accessed through routine invocation.
- Routines execute in the supplier’s task(s).
- Can easily simulate active objects.
- Examples: SCOOP, Actors...
- Garbage collection is problematic (shared task).
Active objects

- Upon creation, a task is idle!
- Creating an active object is:
  - create a passive object and a task,
  - “start” the object (autonomous routine).
- in Java: `t = new Thread(); t.start;`
- Starting is automated: live routine (POOL).
- An active objects raises the degree of parallelism.
- Use message passing or method invocation.
Example **live** routine in POOL

```plaintext
CLASS Queue
...
Method enq (item : T)
BEGIN  cell ! put (rear, item);
         rear := (rear+1) MOD size
END enq

METHOD deq (): T
BEGIN RESULT cell ! get (front) ; %% early return
         front := (front + 1) MOD size
END deq

BODY DO IF empty THEN ANSWER (enq)
ELSIF full THEN ANSWER (deq)
ELSE       ANSWER ANY FI OD
YDOB
```
Other Possibility in CEiffel

- Communication based on method invocation.
- One can place **autonomy annotations**.
- Annotated method **step** is repeated forever.

```ceiffel
class MOVING
creation Init
...
feature {}
...
step is -->--
do
  position.set(...)
end
...
end -- MOVING
```
Coordination

- Objects have a **state**.
- The set of available routines may depend on the current state.
- The **Queue** exhibits such a **non-uniform** service.
- Activity-centered coordination:
  - clients coordinate to prevent indelicate actions.
  - Invocations are never delayed (synchrony).
- Boundary coordination:
  - the supplier may delay the acceptance of a call.
- Introduction of **synchronization code**.
Issues in Communication

- active objects have different address spaces!
- Call-by-value or call-by-name?
  - Call-by-value: sending values away.
  - Call-by-name: sending references away.
- For objects:
  - weak versus strong mobility,
  - mobility of active objects?
- Issue with meta-classes (Smalltalk)
  - an object needs a copy of its code to execute.
  - replicating stateful classes is expensive.
  - keep (meta-)classes stateless (distr. Smalltalk).
Efficiency (1)

- Using synchronous interaction

\[
x := s.f(3)
\]

\[
x := s.f(3)
\]

\[
x := s.g(4)
\]
Efficiency (2)

- Using asynchronous interaction
Synchrony vs Asynchrony

- Synchrony coded with asynchrony:
  - send and return message for every interaction.
  - Can be quite heavy!
- Asynchrony coded with synchrony:
  - intermediate passive buffer between objects.
- Synchrony tends to be too strong (inefficient).
- Asynchrony can be cumbersome.
- Allow the two of them?
- Multicast and broadcast allow high fan-out (GARF).
Using synchronous interaction

- Task 1
  - c.run
  - $x := s.f(3)$
  - $x := s.f(3)$
  - $x := s.g(4)$

- Task 2
  - $s.f(3)$
  - $s.f(3)$
Other Improvements

- Serve many invocations at a time
  - using synchronization primitives,
  - allow pure methods to execute freely, or
  - tag concurrent methods (CEiffel).
- create one task for each invocation (Diadem).
- Use a task pool to delegate calls (ACT++).
- wait-by-necessity.
Next time

- SCOOP Primer
- Coordinating objects, first act.