Concurrent Object-Oriented Programming

Bertrand Meyer, Piotr Nienaltowski
Lectures Overview

- Lecture 1: Introduction and Motivation for Concurrent Programming
- Lecture 2: Classic Approaches to Concurrent Programming
- Lecture 3: Objects and Concurrency
- Lecture 4: From Concurrent to Distributed Programming
Lecture 1: Motivation and Introduction
Definition

Concurrent programming

- = parallel programming on a single processor?
- = is about handling I/O on a host?
Observations

- Computers have evolved
  - From Turing machines
  - Over von Neumann’s
  - ...

- Typically
  - Parallel architectures for efficiency
  - Distributed architectures for
    - Reliability, serviceability
    - Necessity
      - Laptops, handhelds, mobile phone, ...
“Single Program”

- Only one physical node
- A single running program
- Input \( \rightarrow \) Output
How about I/O?

- If program runs continuously
  - Input, e.g., keyboard, must be handled
  - Input/output is simultaneous, i.e., **concurrent**

- **Interruptions** can be used
  - Jumps given by **interruption vector**

- Parts of program to run without interruption
  - **Mask** interruptions

- Can introduce different levels of priorities
“Multiple Programs”

- Only one physical node
- For the time being: a running program is called a task (also process)
- Tasks are totally independent from one another
- This defines the notion of “parallelism” (but not of parallel programming)
Tasks Overlapping “in Time”

- Tasks compete for processor
  - Priorities can help
- Scheduling
  - Defines the allocation of processor(s) to tasks
- Batch processing is simplest case
  - No I/O
- Preemption
  - The processor can be revoked from an uncompleted task
  - Exploit processor capacity
But there is an **operating system**, typically for I/O

- The operating system (kernel) occupies part of the memory

- Tasks can execute OS code (through **system calls** or interruptions)

- The OS code may modify the OS memory

- Tasks can thus **interfere** with each other
Interference in Concurrency

```c
int counter := 0;
increase_counter() {
    counter := counter + 1;
}
```

- Consider concurrent programming with preemptive scheduling, 
  **reentrant procedure** `increase_counter()`
- Operations of concurrent invocations will be interleaved
- Result if t1 and t2 execute `increase_counter()` concurrently?
Tasks Overlapping “in Space”

- Resources and data shared, e.g., screen
- Preemption
  - Inconsistencies can occur if possible at any point
  - Critical resources must be handled with care
    - E.g., mutual exclusion needed in critical sections
- Masking interruptions (which might trigger preemption)?
  - Incur losses, e.g., I/O
  - Prohibit I/O in critical sections, yet I/O represent critical resources in first place
“Multi-Threaded Programs”

- Several tasks (threads) created by the same program
- Primary objective: reduce the time needed for context-switching
- “Light” tasks that share variables
Tasks vs Threads

- Discrepancies?
  - Code? Can be the same also with tasks, cf. `fork`
  - Data? Can be the same (besides OS part), cf. `shared memory`

- Size does matter!
  - Context switching is expensive
    - Page table
    - File descriptors
    - Processor state
    - Resource tables
    - ...
    - ...

Chair of Software Engineering

Concurrent Object-Oriented Programming
“Parallel Programs”

- At any moment, \( n \) tasks are running \((n > 1)\)
- The memory will usually provide atomic r/w operations
- There can be a global clock ("tightly coupled" systems)
- Different architectures (MIMD, SIMD) and memory consistency models as opposed to SISD
- Different granularities (instruction, function, program)
Parallel computing is inherently concurrent

- Focus on large scientific calculations

Issues

1. Identify tasks, and parallelizable parts (programmer work)
2. Schedule the tasks to minimize idle time

Approaches

- Provide abstractions which make some of 1 transparent to programmer
- “Architectures” for increased performance, reduced data access time in response to 2
- Tasks can interfere through the network
- Transmitted data is copied to/from the OS memory
- No global clock
- “Loosely coupled” systems
- Very different networks can be used
Parallel computing can be done on distributed system
- “Emulate” parallel hardware
- Special case of distributed computing with assumptions

Is distributed computing vs concurrent computing just a matter of granularity?
- Cf. threads vs tasks?
Interference in Distribution

- As long as no failures are considered
  - No additional ones

- But nothing is perfect
  - Failures can occur
    - Hosts
    - Tasks: usually unit of failure, but 1 per host
    - Communication
  - FLP-Impossibility result [Fischer, Lynch, Patterson’85]
    - A failed process can not be distinguished from a very slow one
Concurrency: Why?

- Different reasons
  - Efficiency
    - Time (load distribution)
    - Cost (resource sharing)
  - Reliability
    - Redundancy (fault tolerance)
  - Serviceability
    - Availability (multiple access)

- Likely a mixture
But mostly: Necessity

- Why computers in the first place?
  - Make everyday (work) life easier
    - e.g. book flights
  - Computer systems used to “model” life
    - Explicit in workflow
    - Object-oriented programming
      - e.g. plane, ticket, ...
- This world is concurrent!
  - e.g. limited number of seats on the same plane
  - e.g. several booking agents active at the same time
Attempt of Redefinition

- Real-time: concurrency with timing constraints
- Parallel: explicit, heavy computations, possibly specific hardware
- Distributed: physically disparate hosts
  - Note: parallel can be distributed
- Peer-to-peer: distributed, decentralized, scalability-centric
- Ubiquitous, pervasive: peer-to-peer, resource constraints
- Ad-hoc mobile: ubiquitous, devoid of fixed communication backbone
But be Careful...

- Many PCs with dual processors
- Support for sourcing out threads
- Shared memory model sometimes assumed for distributed systems
- ...

Chair of Software Engineering
Concurrent Object-Oriented Programming
Computation consists of
- *Tasks*
- which use *synchronization* mechanisms to
- ensure *consistency* of data handled concurrently by tasks
In an Object World

- Slowly try to map
  - *Tasks* execute *actions* on behalf of *objects*
  - *Objects* have *consistency* requirements depending on their semantics
  - *Actions* to be performed must be *synchronized*
Ideally

- Actions are feature calls
  - Metaphor of objects as computational entities interacting by feature calls remains

- Synchronization is expressed by the way these calls are made
  - Might lead to restrictions
  - Some synchronization might be derived implicitly
Object-Oriented Programming

- **Object-based** programming provides
  - encapsulation of data (information hiding)
  - well defined interface for operations
  - identity

- **Class-based** programming provides
  - abstraction and classification mechanism (ADT)
  - code reuse through composition and inheritance
They seem very different!
(even dealing with orthogonal concerns)

but

Robin Milner said [Matsuoka 1993]:
"I can't understand why objects [of O-O languages] are not concurrent in the first place".
Why did Robin Milner say that?

- **Identifying concepts:**
  - **Object with task, as**
    - both (appear to) encapsulate data
    - both have an autonomous behavior
    - both are computational structures created at run-time
  - **Routine invocation with message passing**
But...

- With an after-look, this comparison seems rather deceptive, and overly simplifying
  - Variable sharing versus encapsulation?
  - What about inheritance and composition?
  - What about garbage collection?
  - What about remote interaction?
  - What about failures?

- Most of the O-O language mechanisms serve purposes that do concern neither nor
Illustration of Issues in Current Object-Oriented Approaches to $\bigcirc$ and $\bigtriangleup$
Language Approaches to  and  

- Possibilities [Briot et al.'98]
  - The integration approach
  - The library approach
  - The reflection approach

- These approaches can be combined
The Integration Approach

- Identify concepts found in the language with external ones
- Introduce new (syntactic) constructs
  - It is the simplest approach
    - Difficult to modify a language (compilers, etc.)
- It leads to cleaner/leaner code
- But it can’t address everything!
  - Little flexibility
The Library Approach

- The most common way
- Provides an API to the programmer
- Wraps “native” code (e.g., OS calls)
- Use through inheritance or composition
- Approach of choice for middleware
The Reflection Approach

- Reflection
  - Enables to alter program “interpretation”
  - Jumps to Meta-Level and back through **Meta-Object Protocol (MOP)**
- Certain languages (Scheme, Smalltalk) provide such capability
- E.g., use reflection to intercept method calls (reification)
- Often combined with the library approach
- The code is often elegant
- The execution is often inefficient
class Counter {
    int value := 0;

    public void increase() {
        value := value + 1;
    }
}

Counter c := …;
c.increase();
Concurrent Object-Oriented Programming

- Possibility to create (concurrent) threads and to synchronize
- Each object has an exclusive locking facility
- Creation of a thread by inheriting from `Thread`
- `wait()`, `notify()`, `notifyAll()` are methods encapsulating native code
- `synchronized` keyword
class Counter {
    int value := 0;

    public synchronized void increase() {
        value := value + 1;
    }
}

Counter c := ...;

c.increase();
class Barrier {
    int num_waiting = 1
    ...
    public synchronized void join() {
        if (num_waiting < 3) {
            num_waiting += 1; wait();
        }
        notifyAll();
    }
}

class Client extends Thread {
    Barrier b = ...;
    ...
    public void run () {
        ... //joining the barrier
        b.join ();
        ... // all clients have joined
    }
}
Execution

- Each thread has its own stack of calls
- Objects do not belong to threads!
- Which thread is awoken by a `notifyAll()` is not specified
- Limited to 🌐 (one CPU)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>c3.run</td>
<td></td>
<td></td>
<td>b.join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.join</td>
<td></td>
<td></td>
<td>b.join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.wait</td>
<td></td>
<td></td>
<td>b.wait</td>
</tr>
<tr>
<td>Thread 3</td>
<td></td>
<td>c2.run</td>
<td></td>
<td></td>
<td>b.join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.join</td>
<td></td>
<td></td>
<td>b.join</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.wait</td>
<td></td>
<td></td>
<td>b.notifyall</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b.wait</td>
<td></td>
<td></td>
<td>b.notifyall</td>
</tr>
</tbody>
</table>
Evaluation

- What kind of approach?
  - Library, integration, reflection?

- Limitations?
  - Thread ordering?
  - Locking granularity?
- What if an object is on a remote host?
- Threading primitives are not enough
- Example solutions relying on other libraries:
  - Networking sockets
  - JavaRMI, CORBA, Jini
  - Java Message Service
  - JavaSpaces
  - ...
- Provides a communication (RPC) layer
- Compatible with CORBA (IIOP in javax.rmi)
- Its interface
  - a stub/skeleton generator (rmic)
  - a naming service (object registry)
Principle

- Invocations
  - Transformed to messages, and sent to the « other side » (*marshaling*) by stub

- The « other side »: *skeleton*
  - Server-side counterpart to the stub
  - Extracts request arguments from message (*unmarshaling*) and invokes the server object
  - Marshals return value and sends it to the invoker side, where stub unmarshals it and returns the result to invoker
Interaction Scheme

Caller

Proxy / Stub

I00II...I0I

Skeleton

Callee

Concurrent Object-Oriented Programming
JavaRMI Step 1: Write an Interface

```java
import java.rmi.*;

public interface HelloInterface extends Remote {

    /* return the message of the remote object, such as "Hello, world!". 
    exception RemoteException if the remote invocation fails. */

    public String say() throws RemoteException;
}
```

- String is **serializable** (it can be marshaled)
import java.rmi.*;
import java.rmi.server.

class Hello extends UnicastRemoteObject implements HelloInterface {
    private String message;

    public Hello (String msg) throws RemoteException {
        message = msg;
    }

    public String say() throws RemoteException {
        return message;
    }
}

- Inherits from UnicastRemoteObject
- rmic Hello will generate stub and skeleton
- in main() method to register:
    Naming.rebind ("Hello", new Hello ("Hello, world!"));
JavaRMI Step 3: Write a Client

```java
import java.rmi.*;

public static void main (String[] argv) {
    try {
        HelloInterface hello = (HelloInterface) Naming.lookup("//se.inf.ethz/Hello");
        System.out.println(hello.say());
    } catch (Exception e) {
        System.out.println("HelloClient exception: " + e);
    }
}
```

- Uses the `lookup` function of the naming service
- The remote object is accessed via a `proxy` (a.k.a. object handle, surrogate)
- `rmiregistry` starts naming service
Evaluation

- What kind of approach?
  - Library, integration, reflection?

- Limitations (of Proxies [Liebermann’86])?
  - Network latency?
  - Failures?
  - Consistency/synchronization?
Conclusions

- Concurrent OO programming is not simply about deploying an OO program on several tasks
  - Consistency requirements guide synchronization scheme
  - Have to think about concurrency from start

- Distributed OO programming is not simply about deploying a COO program on several hosts
  - Remote nature of things changes semantics
  - Have to think about distribution from start