ACTOR-BASED MODEL FOR CONCURRENT PROGRAMMING

Sander Sõnajalg
Contents

- Introduction to concurrent programming
- Shared-memory model vs. actor model
- Main principles of the actor model
- Actors for light-weight processes
- Actors for distributed programming
- Importance & perspectives of actor model
Concurrent programming

- = when programs are designed as collection of interacting computational processes that may be executed in parallel (Wikipedia)

- Inter-dependant processes
  - executing simultaneously
  - affecting each-other's work
  - must exchange information to do so
Concurrent programming: motivation

- Concurrency in local computer
  - Let part of a local program do something useful in the background while it is waiting for some input
  - Clients need to be served in parallel (web server)
- Parallelizing things for distributing
  - Distributing for inclusion of new resource
  - Distributing because functional requirements demand it
Implementing Concurrent Programming

• **GOAL**: Implement a library which would provide necessary abstractions on top of operating system resources to simplify concurrent programming

• **What we need?**
  - Units that could be executed in parallel (concurrency primitives)
  - Means of communication between these units
The Traditional Way: Threads & Shared Memory

- Smallest executable unit is **thread**
  - Usually, operating system threads are used
  - Virtual threading is possible, but less common ("green threads")
- Communication is implemented by sharing variables
Threads & Shared Memory

Thread 1
- Acquire lock for slot 5, write value of x
- Fail; go back to sleep

Thread 2
- Go to sleep, wait for x from T1
- Check if T1 has written x
- Fail; go back to sleep
- Read value of x

Shared memory
The Traditional Way: Threads & Shared Memory

• Pros:
  • Fast

• Cons:
  • Complicated & error-prone client code
  • Not extendable to distributed programming
  • Threads are heavy-weight – not too scalable

• Examples:
  • Standard concurrency libraries in Java, C#, etc.
The OTHER Way: 
The Actor Model

- Smallest executable unit is **an actor**
  - An actor is a concurrency primitive that does not share any resources with other actors
- Communication is implemented by actors sending each-other messages
Actors: formal definition

- Actors are computational agents which map each incoming communication to a triple consisting of:
  - A finite set of communications sent to other actors
  - A new behavior (which will govern the response to the next communication processed)
  - A finite set of new actors created

(Gul A. Agha, 1985)
Workflow: event-driven + asynchronous

A1

create

message x

message z

A2

create

message y

A3

c1
c2
c3

time

initial system start

x

y

z
Actors and event-driven programming

• Rather than explicitly “sleeping” or “waking up”, actors “react” to the “events”
  • “events” = interactions with other actors = messages

• Actors get passive when they've finished their previous tasks and nobody has sent them new messages of interest

• Passive actors activate immediately when somebody has sent them an interesting message
Asynchronous message passing

- Message sending is asynchronous
- Each actor has a “mailbox” for storing messages that are not consumed immediately.
- If message arrives when actor is busy working on a previous message, it gets stored in it's mailbox.
- If actor arrives at a point where it waits new messages to continue, it first looks through it's mailbox.
  - Messages of unsuitable type are ignored
  - First suitable message allows actor to continue
Asynchronous message passing: Example of message receiving

```
receive {
    case Approve() => ..
    case Cancel() => ..
}
..  
```

**CASE A**

*contents of mailbox*

<table>
<thead>
<tr>
<th>Answer(7)</th>
<th>Cancel()</th>
<th>Approve()</th>
</tr>
</thead>
</table>

**CASE B**

*contents of mailbox*

<table>
<thead>
<tr>
<th>Blaa()</th>
<th>Answer(7)</th>
<th>Blaa()</th>
</tr>
</thead>
</table>
class Counter extends Actor {
    override def run: unit =
    loop(0)

    def loop(value: int): unit = {
        Console.println("Value: " + value)
        receive {
            case Incr() =>
                loop(value + 1)
            case Value(p) =>
                p ! value
                loop(value)
            case _ =>
                loop(value)
        }
    }
}
Actors for light-weight processes

- Thread-based concurrency usually directly mapped to operating system threads
  - 1 code-level thread = 1 operating system thread
- Operating system threads are expensive!
- Most actor-based concurrency libraries don't map actors directly to operating system threads (“light-weight actors”)
  - 1 operating system thread = 1 active actor + unlimited amount of inactive actors
Actors in distributed programming

- The actor model naturally extends from concurrency inside one computer to concurrency in a distributed network of computers
- Local and remote actors “look the same”
- The concurrency framework will deal with the technical details:
  - message-passing over the network
  - message serialization
  - actor identifiers management
  - network connection management
The Actor model

• Pros:
  • Event-driven approach is more intuitive
  • Directly and transparently applicable for distributed programming
  • Implements light-weight processes => better scaling

• Cons:
  • Message-passing is slower
  • Libraries and solutions not yet as “mature” (e.g. no good frameworks for Java before 2008)
Actor model implementations

- Separate concurrency languages
  - Erlang
  - SALSA
- Frameworks
  - Scala Actor Framework for Scala
  - Kilim, FunctionalJava for Java
  - Kamaelia, Eventlet, PARLEY for Python
  - Dramatis, Revactor for Ruby
Current importance & perspectives

- Cuncurrency in today's middleware almost always thread-based
- Hardware evolves towards more parallel architectures
- IT-industry evolves towards workforce being more costly than hardware
- In 2008, actor frameworks exist for all popular programming languages
- Cloud computing (thin clients, etc.)