ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Introduction: Why Actors?

Programming Reactive Systems

Roland Kuhn

Where Actors came from

A selection of events in the history of Actors:

Carl Hewitt et al, 1973: Actors invented for research on artificial intelligence

Gul Agha, 1986: Actor languages and communication patterns

Ericsson, 1995: first commercial use in Erlang/OTP for telecommunications platform

Philipp Haller, 2006: implementation in Scala standard library

Jonas Bonér, 2009: creation of Akka

Threads

CPUs are not getting faster anymore, they are getting wider:

- multiple execution cores within one chip, sharing memory
- virtual cores sharing a single physical execution core

Threads

CPUs are not getting faster anymore, they are getting wider:

- multiple execution cores within one chip, sharing memory
- virtual cores sharing a single physical execution core

Programs running on the computer must feed these cores:

- running multiple programs in parallel (multi-tasking)
- running parts of the same program in parallel (multi-threading)

Example: Bank Account

```
class BankAccount {

  private var balance = 0

  def deposit(amount: Int): Unit =
    if (amount > 0) balance = balance + amount

  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error("insufficient funds")
}
```

Example: Bank Account

```
def withdraw(amount: Int): Int = {
  val b = balance
  if (0 < amount && amount <= b) {
    val newBalance = b - amount
    balance = newBalance
    newBalance
  } else {
    throw new Error("insufficient funds")
  }
}
```

Executing this twice in parallel can violate the invariant and lose updates.

Synchronization

Multiple threads stepping on each others' toes:

- demarcate regions of code with "don't disturb" semantics
- make sure that all access to shared state is protected

Synchronization

Multiple threads stepping on each others' toes:

- demarcate regions of code with "don't disturb" semantics
- make sure that all access to shared state is protected

Primary tools: lock, mutex, semaphore

Synchronization

Multiple threads stepping on each others' toes:

- demarcate regions of code with "don't disturb" semantics
- make sure that all access to shared state is protected

Primary tools: lock, mutex, semaphore

In Scala every object has a lock: `obj.synchronized { ... }`

Bank Account with Synchronization

```
class BankAccount {

  private var balance = 0

  def deposit(amount: Int): Unit = this.synchronized {
    if (amount > 0) balance = balance + amount
  }

  def withdraw(amount: Int): Int = this.synchronized {
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error(”insufficient funds”)
  }
}
```

Composition of Synchronized Objects

```
def transfer(from: BankAccount, to: BankAccount, amount: Int): Unit = {
  from.synchronized {
    to.synchronized {
      from.withdraw(amount)
      to.deposit(amount)
    }
  }
}
```

Composition of Synchronized Objects

```
def transfer(from: BankAccount, to: BankAccount, amount: Int): Unit = {
  from.synchronized {
    to.synchronized {
      from.withdraw(amount)
      to.deposit(amount)
    }
  }
}
```

Introduces Dead-Lock:

- `transfer(accountA, accountB, x)` in one thread
- `transfer(accountB, accountA, y)` in another thread
- one lock taken by each, nobody can progress

We want Non-Blocking Objects

- blocking synchronization introduces dead-locks
- blocking is bad for CPU utilization
- synchronous communication couples sender and receiver

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

The Actor Model

Programming Reactive Systems

Roland Kuhn

What is an Actor?

The Actor Model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

What is an Actor?

The Actor Model represents objects and their interactions, resembling human organizations and built upon the laws of physics.

An Actor[1]

- is an object with identity
- has a behavior
- only interacts using *asynchronous* message passing

[1] Hewitt, Bishop, Steiger: *A Universal Modular Actor Formalism for Artificial Intelligence, IJCAI 1973*

The Actor Trait

```
type Receive = PartialFunction[Any, Unit]

trait Actor {
  def receive: Receive
  ...
}
```

The `Actor` type describes the behavior of an Actor, its response to messages.

A Simple Actor

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case ”incr” => count += 1
  }
}
```

This object does not exhibit stateful behavior.

Making it Stateful

Actors can send messages to addresses (`ActorRef`) they know:

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
    case ("get", customer: ActorRef) => customer ! count
  }
}
```

How Messages are Sent

```
trait Actor {
  implicit val self: ActorRef
  def sender: ActorRef
  ...
}

abstract class ActorRef {
  def !(msg: Any)(implicit sender: ActorRef = Actor.noSender): Unit
  def tell(msg: Any, sender: ActorRef) = this.!(msg)(sender)
  ...
}
```

Sending a message from one actor to the other picks up the sender's address implicitly.

Using the Sender

```
class Counter extends Actor {
  var count = 0
  def receive = {
    case "incr" => count += 1
    case "get"  => sender ! count
  }
}
```

The Actor's Context

The `Actor` type describes the behavior, the execution is done by its `ActorContext`.

```
trait ActorContext {
  def become(behavior: Receive, discardOld: Boolean = true): Unit
  def unbecome(): Unit
  ...
}

trait Actor {
  implicit val context: ActorContext
  ...
}
```

Changing an Actor's Behavior

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case ”incr” => context.become(counter(n + 1))
    case ”get”  => sender ! n
  }
  def receive = counter(0)
}
```

Changing an Actor's Behavior

```
class Counter extends Actor {
  def counter(n: Int): Receive = {
    case "incr" => context.become(counter(n + 1))
    case "get"  => sender ! n
  }
  def receive = counter(0)
}
```

Functionally equivalent to previous version, with advantages

- state change is explicit
- state is scoped to current behavior

Similar to "asynchronous tail-recursion".

Creating and Stopping Actors

```
trait ActorContext {
  def actorOf(p: Props, name: String): ActorRef
  def stop(a: ActorRef): Unit
  ...
}
```

Actors are created by actors.

“stop” is often applied to “self”.

An Actor Application

```
class Main extends Actor {
  val counter = context.actorOf(Props[Counter], ”counter”)

  counter ! ”incr”
  counter ! ”incr”
  counter ! ”incr”
  counter ! ”get”

  def receive = {
    case count: Int =>
      println(s”count was $count”)
      context.stop(self)
  }
}
```

The Actor Model of Computation

Upon reception of a message the actor can do any combination of the following:

- send messages
- create actors
- designate the behavior for the next message