ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Introduction: Why Actors?

Principles of Reactive Programming

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

```scala
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

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

Composition of Synchronized Objects

```scala
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