Fundusze Europejskie Wiedza Edukacja Rozwój
Politechnika Wrocławska
Unia Europejska Europejski Fundusz Społeczny

„ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej”

Programming Paradigms Tutorials
Akka Actors

Actors

The Actor Model provides a higher level of abstraction for writing concurrent and distributed systems. It alleviates the developer from having to deal with explicit locking and thread management, making it easier to write correct concurrent and parallel systems. An actor represents an independent computation unit. Some important characteristics are:

- an actor encapsulates its state and part of the application logic
- actors interact only through asynchronous messages and never through direct method calls
- each actor has a unique address and a mailbox in which other actors can deliver messages
- the actor will process all the messages in the mailbox in sequential order (the default implementation of the mailbox being a FIFO queue)
- the actor system is organized in a tree-like hierarchy
- an actor can create other actors, can send messages to any other actor and stop itself or any actor is has created

Actor classes are implemented by extending the Actor class and implementing the receive method. The receive method should define a series of case statements that defines which messages specific Actor can handle, using standard Scala pattern matching. Example:

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)

  def receive = {
    case "msg" => log.info("received msg")
    case _      => log.info("received unknown message")
  }
}

object Main extends App {
  val system = ActorSystem("MySystem")
  val myActor = system.actorOf(Props[MyActor], name =
"myactor")
…
}
```

The actor lifecycle

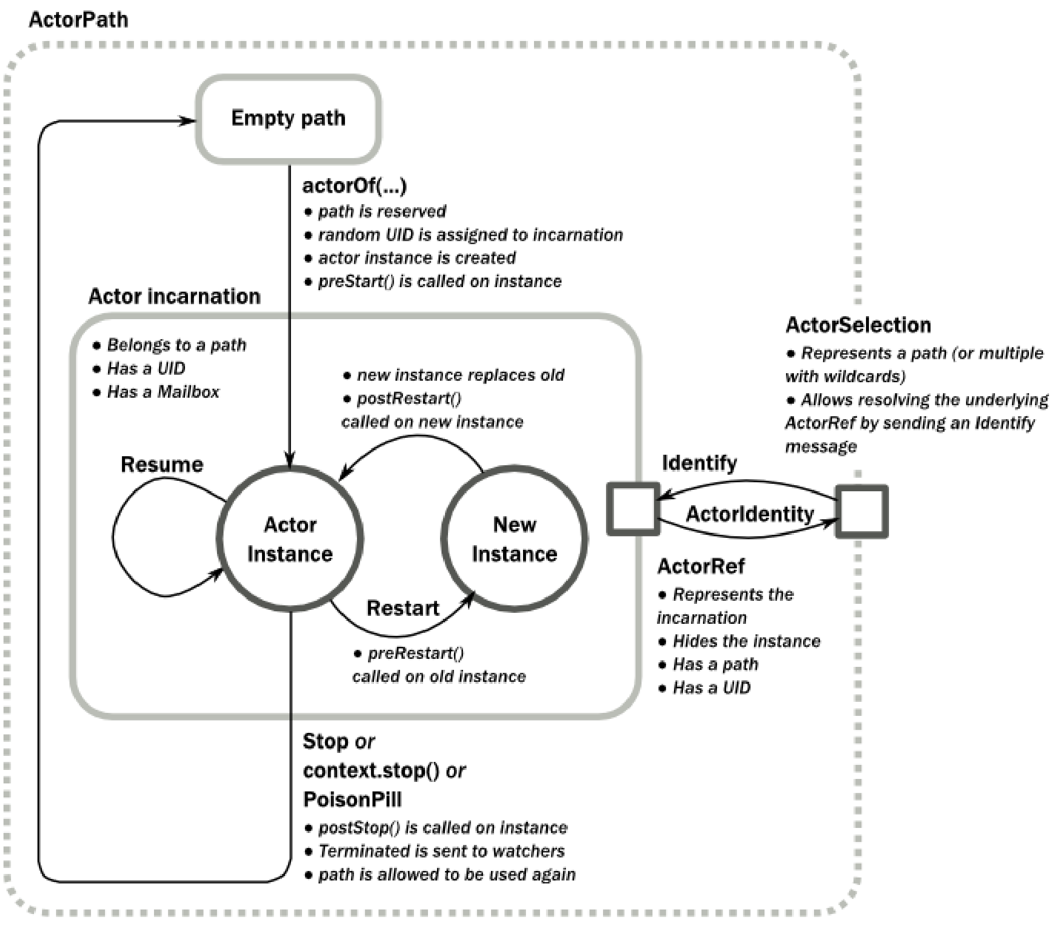
ActorPath
Empty path
actorOf(...)
• path is reserved
• random UID is assigned to incarnation
• actor instance is created
• preStart() is called on instance
Actor incarnation
• Belongs to a path
• Has a UID
• Has a Mailbox
• new instance replaces old
• postRestart() called on new instance
Resume
Actor Instance
New Instance
Restart
• preRestart() called on old instance
ActorSelection
• Represents a path (or multiple with wildcards)
• Allows resolving the underlying ActorRef by sending an Identify message
Identify
ActorIdentity
ActorRef
• Represents the incarnation
• Hides the instance
• Has a path
• Has a UID
Stop or context.stop() or PoisonPill
• postStop() is called on instance
• Terminated is sent to watchers
• path is allowed to be used again

Fig. Actor lifecycle.

The lifecycle of an actor begins when the actorOf( ) method is called either by the system or by the context. First actor needs to be created by ActorSystem and then it can be a parent for other actors. The preRestart() and postRestart() methods help handle what should happen in case when an actor fails and it is restarted for some reason.
When an actor is crashing, it's preRestart() method is called. This is the place where you'd release all the acquired resources and perform any other cleanup operations you need to perform. The default implementation terminates all the children and calls the postStop() method. The last method in the lifecycle of an actor is the postStop( ). Once this method is called, the actor will stop receiving any new messages and the current messages will be redirected to dead letter mailbox.

Exercises

1. Write actor which welcome 3 known users, represented by message with name in specific way for example msg(“Anna”) -> “Hi Anna, msg(“Tom”) -> “Hello Tom”) and show some generic message for unknown name.

2. Write actors which simulate ping-pong game. They can hit or miss a ball (write some random function). Players are represented by same class of actor and ball

is a message with current match result. The game is finished when one of players get 10 points.

3. Write a game between 3 actors: they send ball (as a message) to each other (randomly). The ball is represented as:
   ```
   case class Ball(count: Int)
   ```
   where count is throwing counter which is increment by each actor who has a ball.