CS 3101-2 - Programming Languages: Scala

Lecture 6: Actors and Concurrency

Daniel Bauer (bauer@cs.columbia.edu)

December 3, 2014

1. Actors and Concurrency

Concurrency

- Goal: run several computations at the same time to make better use of resources.
  - CPU and IO
  - Multicore CPUs
  - Cloud instances
- Examples:
  - Perform background computation while waiting for user or network I/O.
  - Speed up computation by splitting it over multiple CPU cores.
- Pandora's Box: Different branches of computation share resources and need to communicate.

Concurrency - multiple processes vs. threading

- Time sharing/scheduling in the operating system
  - Multiple processes.
  - IPC via sockets/pipes/signals.
- Threading
  - Multiple parallel threads within the same process
  - Threads are automatically mapped to different CPUs by OS or VM.
  - IPC via shared memory or messaging.
- Generally, want to allow the developer to write concurrent programs without having to think about the low-level mechanism.

Problems with Concurrency

- Shared resources (Files, IO)
- Read and write access to shared data.
- Different threads communicate with each other. How?

Concurrency in Java

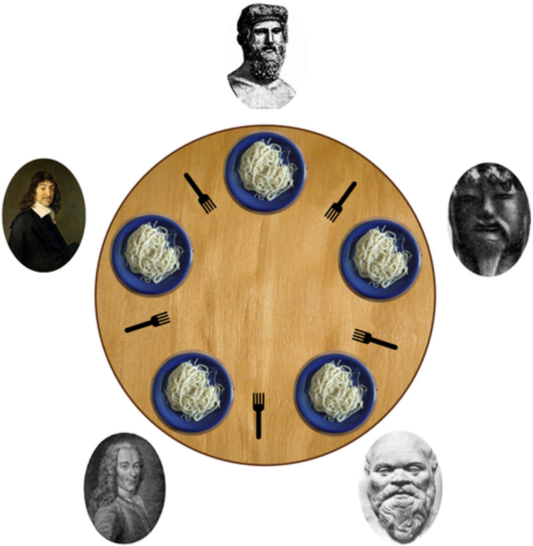
- In Java threads communicate via shared data (multiple threads have read/write access to the same object).
- Threads are special objects (subclass of `Thread` or implementation of `Runnable`).

Java Concurrency Example

- Java uses Monitors:
  - Blocks of code marked as `synchronized`
  - Only one thread can execute every synchronized block at a time.
  - Only one synchronized block can be executed in every object at a time.
  - This establishes a lock on the object's attributes.

```
public synchronized void atomic_add(int y) {
    x = x+y;
}
```

Deadlocks: The Dining Philosophers' Problem



Synchronization in Scala: The Actor Model

- An actor is a self-contained branch of the program.
- Actors share *nothing* with other actors. No locks required.
- Instead actors communicate via message passing.
  - Every actor has a mailbox (infinite message queue).
  - Other actors can send messages (arbitrary Scala objects to the mailbox).
  - Actors consume messages in the order they arrive.
- Scala now uses the actors in **Akka**.

Actors in Akka

- Actors are objects that extend or mix-in the `akka.actors.Actor` trait.
- Actors have a method `receive` that handles incoming messages.
- An ActorSystem is a hierarchical group of actors with a common configuration, used to create new actors.
- Props are configuration objects using when creating an actor.

```
import akka.actor.Actor

class HelloActor extends Actor {
  def receive = {
    case "ping" => println("pong")
    case _      => println("huh?")
  }
}
```

Creating Actors

- An ActorSystem is a hierarchical group of actors with a common configuration, used to create new actors.
- Props are configuration objects using when creating an actor.

```
import akka.actor.{Actor, Props, ActorSystem}

class HelloActor extends Actor {
  def receive = {
    case "ping" => println("pong")
    case _      => println("huh?")
  }
}
```

```
scala> val system = ActorSystem("HelloSystem")
system: akka.actor.ActorSystem = akka://HelloSystem

scala> val props = Props(new HelloActor)
props: akka.actor.Props =
        ...

scala> val actorRef = system.actorOf(props)
actorRef: akka.actor.ActorRef =
    Actor[akka://HelloSystem/user/$b#-1186551897]
```

Passing Messages with '!'

```scala
import akka.actor.{Actor, Props, ActorSystem}

class HelloActor extends Actor {
  def receive = {
    case "ping" => println("pong")
    case _      => println("huh?")
  }
}
```

```
scala> val system = ActorSystem("HelloSystem")

scala> val props = Props(new HelloActor)

scala> val actorRef = system.actorOf(props)
actorRef: akka.actor.ActorRef =
    Actor[akka://HelloSystem/user/$b#-1186551897]

scala> actorRef ! "ping"
pong
```

Multiple Actors

```
import akka.actor.{Actor, ActorSystem, Props}

class HelloActor extends Actor {
  def receive = {
    case "ping" => {println(self.path.name);
                     Thread.sleep(1000);
                     println("pong")}
    case _       => {println(self.path.name);
                     Thread.sleep(1000);
                     println("huh?")}
  }
}

object HelloActor {
  def main(args : Array[String]) {
    val system = ActorSystem("HelloSystem")
    val actor1 = system.actorOf(Props(new HelloActor),
                 name = "actor1")
    val actor2 = system.actorOf(Props(new HelloActor),
                 name = "actor2")
    actor1 ! "ping"
    actor2 ! "test"
    actor1 ! "ping"
  }
}
```

Asynchronous I/O

```scala
import akka.actor.{Actor, ActorSystem, Props}
import scala.io.StdIn.readLine

sealed abstract class Messages
case class Request(val prompt : String) extends Messages
case class Response(val response : String) extends Messages

class InputRequest extends Actor {
    val printer = context.actorOf(Props[PromptDisplay])
    val reader = context.actorOf(Props[InputReader])

    def receive = {
        case Request(prompt) => {
            // Waits for input
            reader ! Request(prompt)
            // Repeatedly prints the prompt
            printer ! Request(prompt)
        }
        case Response(resp) => {
            println("Input was: "+resp)
            context.system.shutdown()
        }
    }
}
```

Asynchronous I/O

```
class InputReader extends Actor {
    def receive = {
        case Request(_) => {
            val input = readLine() // blocks
            sender() ! Response(input)
        }
    }
}

class PromptDisplay extends Actor {
    def receive = {
        case Request(prompt) => {
            println(prompt);
            Thread.sleep(2000);
            self ! Request(prompt);
        }
    }
}
```

Asking actors for values

Retrieve the result of some computation in a non-actor context.

```
import akka.pattern.ask
import akka.util.Timeout
import scala.concurrent.{Await,Future}
import scala.concurrent.duration._

class FancyActor extends Actor {
    def receive = {
        case Compute(input) => {
            val result = ??? // Do some fancy computation
            sender ! result
        }
    }
}

object IoTest {
  def main(args : Array[String]) {
    val system = ActorSystem("System")
    val actor = system.actorOf(Props[FancyActor])

    // needed for ask
    implicit val timeout = akka.util.Timeout(1.second)

    val data = ??? // Some input data
    val msg = Compute(data)
    val future: Future[String] = ask(actor, msg).mapTo[String]

    implicit val ec = system.dispatcher
    val result = Await.result(future, timeout.duration)
    println(result)
  }
}
```

Scala Futures

- We can also use futures without actors!
- Often preferable for parallel computation (if there is no state).

```
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

val future = Future {
  // some Fancy computation here
  27*3
}

println("Okay")
val result = Await.result(future, 1.second)
println(result)
```

Futures for Parallel Computation

```
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

val future = Future {
  // some Fancy computation here
  27*3
}

println("Okay")
val result = Await.result(future, 1.second)
println(result)
```

Approximating Pi

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

```
import scala.concurrent._
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global


def calc(i: Int, nrOfElements: Int): Future[Double] =
  Future {
    val start = i *nrOfElements
    var acc = 0.0
    for (i <- start until (start + nrOfElements))
      acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
    println("branch "+i+" done")
    acc
}

val n = 10
val elements = 1000
val futures = for (i <- 0 until n) yield calc(i, elements)

val result = Future.fold(futures)(0.0)(_+_)
result.onSuccess {
    case pi => println(pi)
}
```