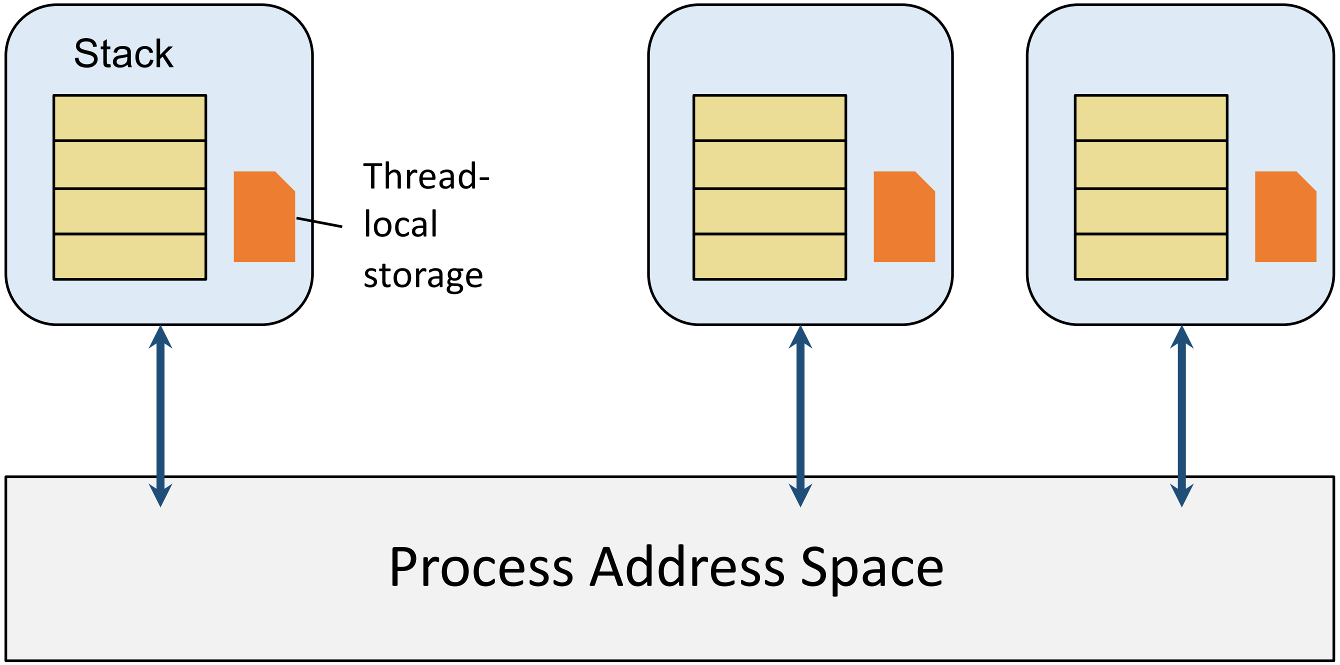
Introducing Actors with Akka

Traditional Threading Model

- Multiple threads sharing a single address space

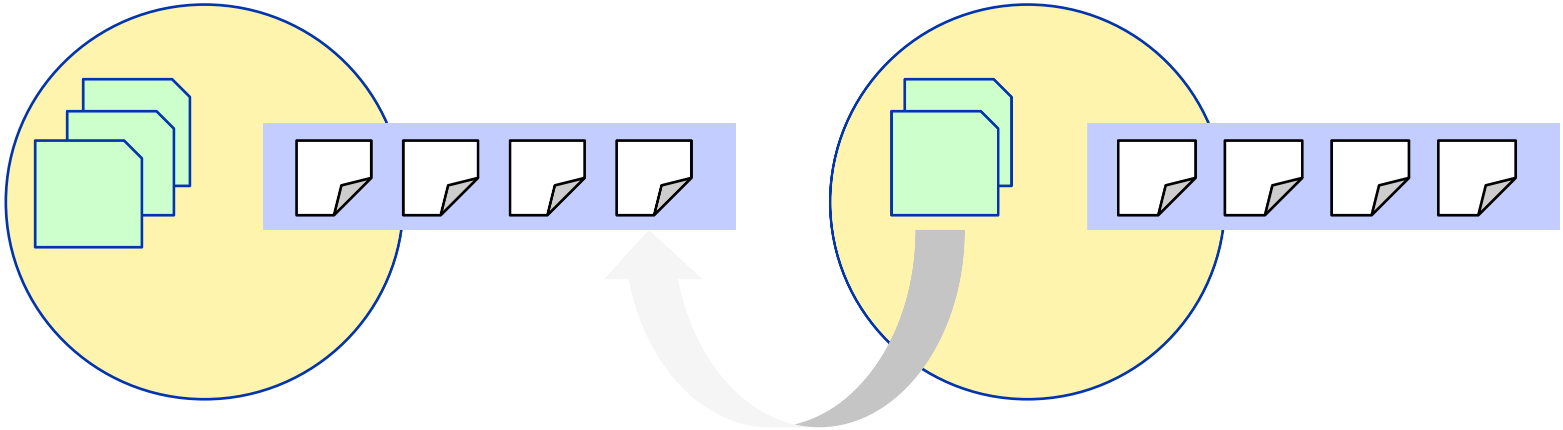


Issues With The Traditional Model

- Threads no longer viewed as lightweight
  - stack size 512K to 2MB
  - limits number of threads that can be created
- Protection of shared mutable state is hard
  - locking very difficult to get right
  - based on notion of blocking and context switching
  - many problems are timing related
- Much boiler plate needed
  - low level constructs need management

Actors

- An alternative approach to concurrency and distribution
- Actor is a small, self-contained processing unit
  - contains state, behaviour and mailbox
- Actors communicate by sending messages
  - asynchronously



Actors

- Should not share any mutable state
    - can have mutable state internally but nothing exposed
- Should communicate using immutable messages
- Should communicate asynchronously
- Behave reactively
    - Only perform calculations in response to messages
- Can exist within one process or across processes
    - also across machines
- Should provide a safe model for handling failures

A Simple Example

- Two Actors implementing "TickTock" example

- Message types
  - usually defined as Algebraic Data Type

```
import akka.actor._

sealed abstract class Message

case class StartTicking ( tocker: ActorRef ) extends Message
case object TickMessage extends Message
case object TockMessage extends Message
```

A Simple Example

- The Actors

```
import akka.actor._
class TickActor extends Actor with ActorLogging {
  log.info("Creating Tick Actor")
  override def receive = {
    case StartTicking(tocker) => log.info("Starting... Tick");
                                 tocker ! TockMessage
    case TickMessage => log.info("Tick");
                        Thread.sleep(500); sender ! TockMessage
  }
}

class TockActor extends Actor with ActorLogging {
  log.info("Creating Tock Actor")
  override def receive = {
    case TockMessage => log.info("Tock");
                        Thread.sleep(500); sender ! TickMessage
  }
}
```

A Simple Example

- The driver application

```
object ActorApp extends App {

  val ttSystem = ActorSystem("TickTock")

  val ticker = ttSystem.actorOf( Props[TickActor] )
  val tocker = ttSystem.actorOf( Props[TockActor] )

  ticker ! StartTicking(tocker)

  Thread.sleep(5000)
  ttSystem.shutdown

}
```

Create and initialise the actors

Send start message

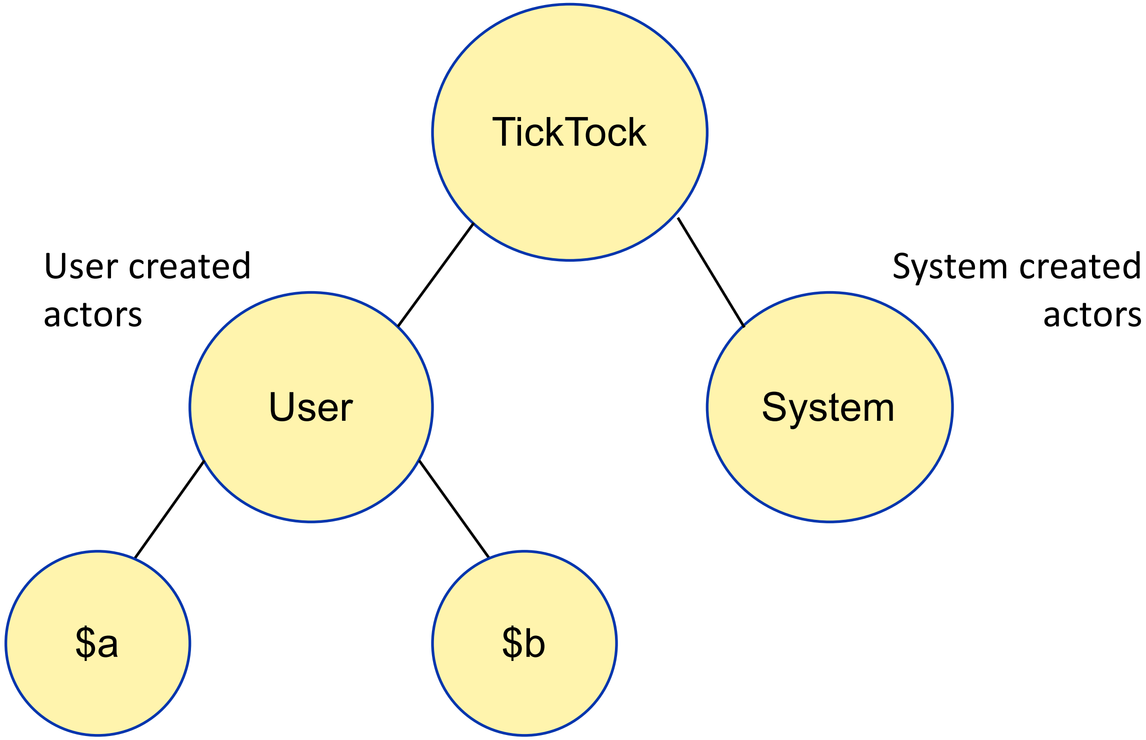
Wait 5 seconds then shut down

```
[INFO] [06/25/2013 18:18:48.893] … [akka://TickTock/user/$a] Creating Tick Actor
[INFO] [06/25/2013 18:18:48.897] … [akka://TickTock/user/$b] Creating Tock Actor
[INFO] [06/25/2013 18:18:48.898] … [akka://TickTock/user/$a] Starting... Tick
[INFO] [06/25/2013 18:18:48.898] … [akka://TickTock/user/$b] Tock
[INFO] [06/25/2013 18:18:49.397] … [akka://TickTock/user/$a] Tick
…
```

Actor Application Structure and Naming

- Actors exist in a hierarchy
  - Important for error handling and recovery

- Pathname identifies individual actors



Request/Response Operation

- Actor communication encouraged to be asynchronous
  - "fire and forget"
  - no implicit reply

- Request/response communications possible
  - use `ask` method rather than `tell` method
  - `?` rather than `!`

- Leverages Futures for handling replies

Request/Response Example

- Actor generates and sends a random `Int` value between 0 and 100

```
import akka.actor._

case object GetRandomInt

class RandomNumActor extends Actor with ActorLogging {

  log.info("Creating the Random Number Generator Actor")
  val rGen = new scala.util.Random

  override def receive = {
    case GetRandomInt => sender ! Math.abs(rGen.nextInt) % 100
  }
}
```

Message

Request/Response Example

- Send request and handle response as `Future[Int]`

```
import akka.actor._
import akka.pattern.ask
import scala.concurrent.duration._
import scala.concurrent.ExecutionContext.Implicits.global

object RNActorApp extends App {

  val rnSystem = ActorSystem("RandomNumbers")
  val rand = rnSystem.actorOf(Props[RandomNumActor], "RandomNumGen")

  implicit val timeout = Timeout(1 seconds)
  val rNumFuture = (rand ? GetRandomInt).mapTo[Int]

  rNumFuture onSuccess {
      case i => println(s"=> $i")
  }
  rnSystem.shutdown
}
```

Request/Response Example

- Demonstrating async nature of calls

```
// Setup as before …
  1 to 5 foreach { n =>
         (rand ? GetRandomInt).mapTo[Int].onSuccess {
             case i => println(s"$n => $i")
           }
  }
…
```

```
[INFO] [06/25/2013 19:18:49.923] … [akka://RandomNumbers/user/RandomNumGen]
                                              Creating the Random Number Generator Actor
2 => 0
5 => 78
1 => 38
3 => 26
4 => 58
```

Request/Response Example

- Blocking on each request until response arrives

```
// Setup as before …
  1 to 5 foreach { n =>
      val rn: Int = Await.result(
                        (rand ? GetRandomInt).mapTo[Int], 1 second)
    println(s"$n => $rn")
  }
…
```

```
[INFO] [06/25/2013 19:22:45.109] … [akka://RandomNumbers/user/RandomNumGen]
                                    Creating the Random Number Generator Actor
1 => 11
2 => 5
3 => 51
4 => 86
5 => 22
```

Additional Akka Features

- Java API
    - completely interoperable with Scala API
- "Let it crash" failure management
    - based on hierarchical actor structure
    - highly flexible recovery
- Dynamic reconfiguration of actors
    - changing behaviour while application is running
- Flexible dispatching of requests to actors
    - "routers"
- Clustering support
    - from 2.2