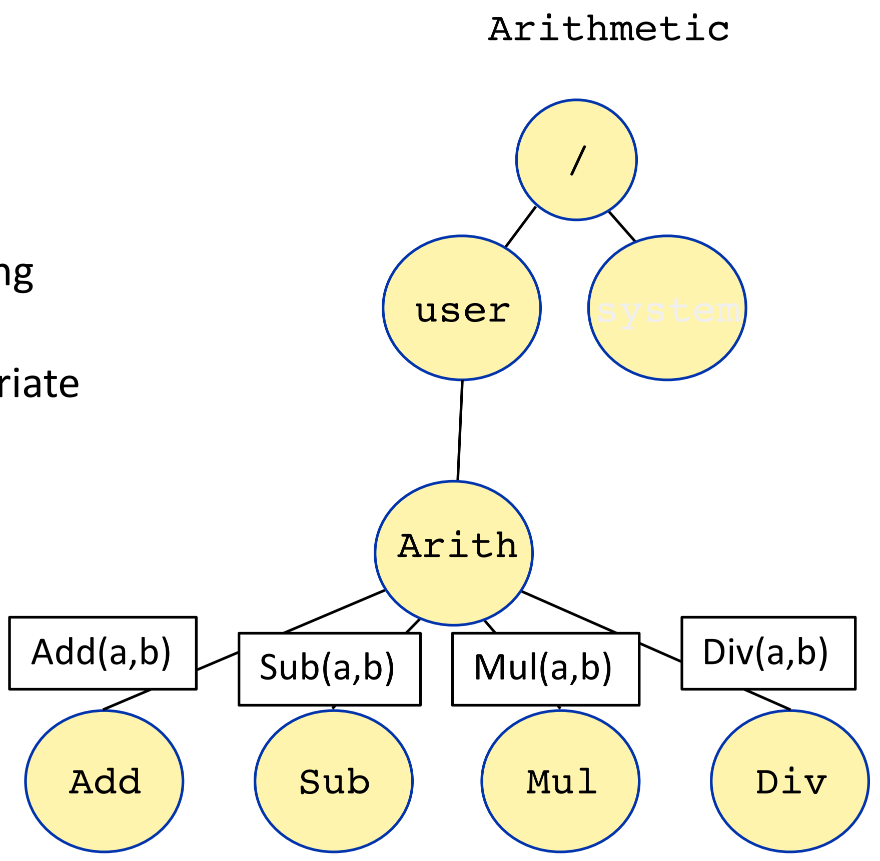
Fault Handing

Failure

- Actor systems use novel way of dealing with failures
- Failure considered to be part of "normality"
    - something that happens
- Applications should be written to expect failure
    - deal with it
    - recover in the most appropriate way
- Leads to robust applications
    - long running
- E.g. Erlang actor based systems
    - written to operate in telephone exchanges

Example

- Simple example
- Arithmetic actor
  - receives messages requesting add/sub/mul/div
  - forward message to appropriate "worker" actor



Example

```
case class Add(a:Int, b:Int)
case class Sub(a:Int, b:Int)
case class Mul(a:Int, b:Int)
case class Div(a:Int, b:Int)

class Arith extends Actor with ActorLogging {
 log.info("Creating Arith Actor")
 val adder = context.actorOf(Props[AddActor])
 val subber = context.actorOf(Props[SubtractActor])
 val multiplier = context.actorOf(Props[MultiplyActor])
 val divider = context.actorOf(Props[DivideActor])
 override def receive = {
  case m: Add => adder.forward(m)
  case m: Sub => subber.forward(m)
  case m: Mul => multiplier.forward(m)
  case m: Div => divider.forward(m)
 }
}
```

Example

```
class AddActor extends Actor with ActorLogging {
 log.info("Creating Add Actor")
 override def receive = {
  case Add(a,b) => log.info(s"$a + $b -> ${a+b}")
 }
}

class SubActor extends Actor with ActorLogging {
 …
}
…
```

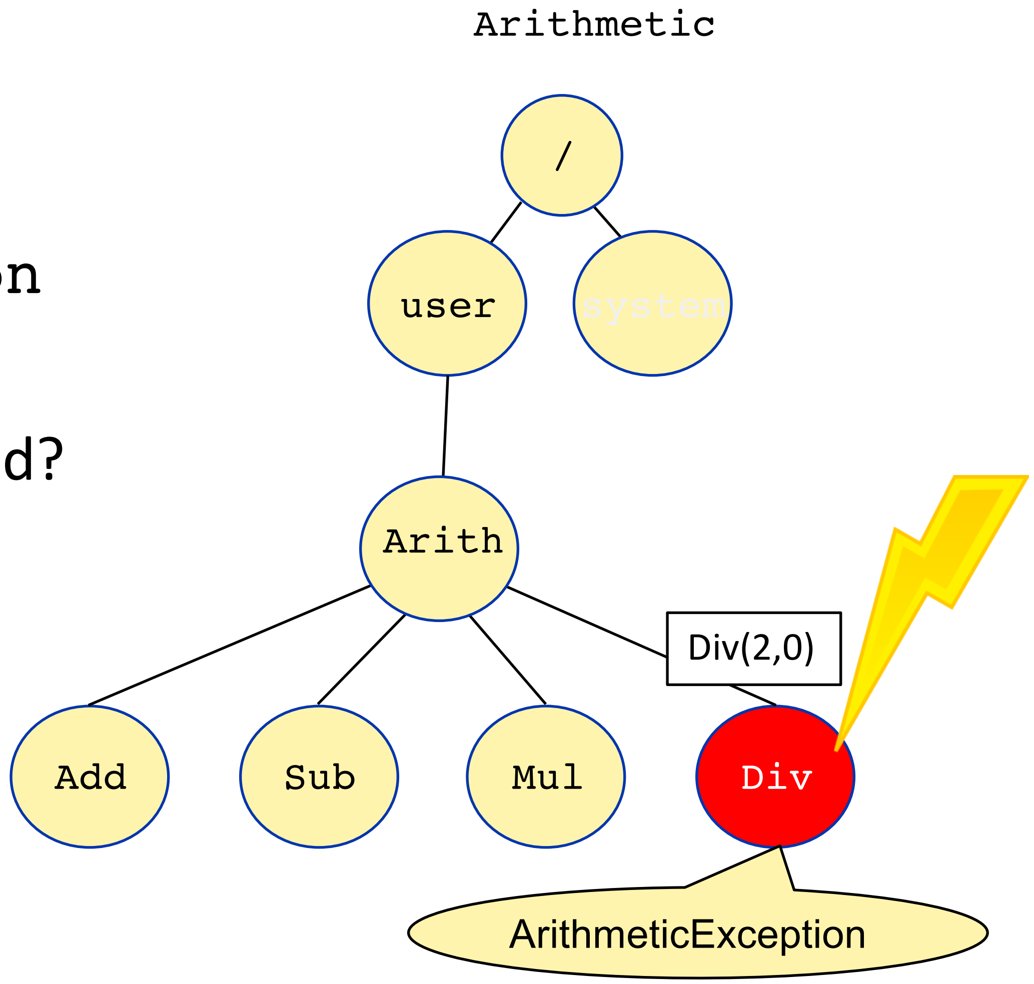
Example

```
object ArithActorApp2 extends App {

 val aSystem = ActorSystem("Arith")
 val arith2 = aSystem.actorOf(Props[Arith], "arithmetic")

 arith2 ! Add(1,3)
 arith2 ! Mul(2,4)

 Thread sleep 3000
 aSystem.shutdown
}
```

```
[INFO] … […/arithmetic] Creating Arith Actor
[INFO] … […/arithmetic/$a] Creating Add Actor
[INFO] … […/arithmetic/$b] Creating Sub Actor
[INFO] … […/arithmetic/$c] Creating Mult Actor
[INFO] … […/arithmetic/$d] Creating Divide Actor
[INFO] … […/arithmetic/$a] 1 + 3 -> 4
[INFO] … […/arithmetic/$c] 2 * 4 -> 8
```

Example

- Div(2,0)??
- Causes `ArithmeticException`
- How should this be handled?

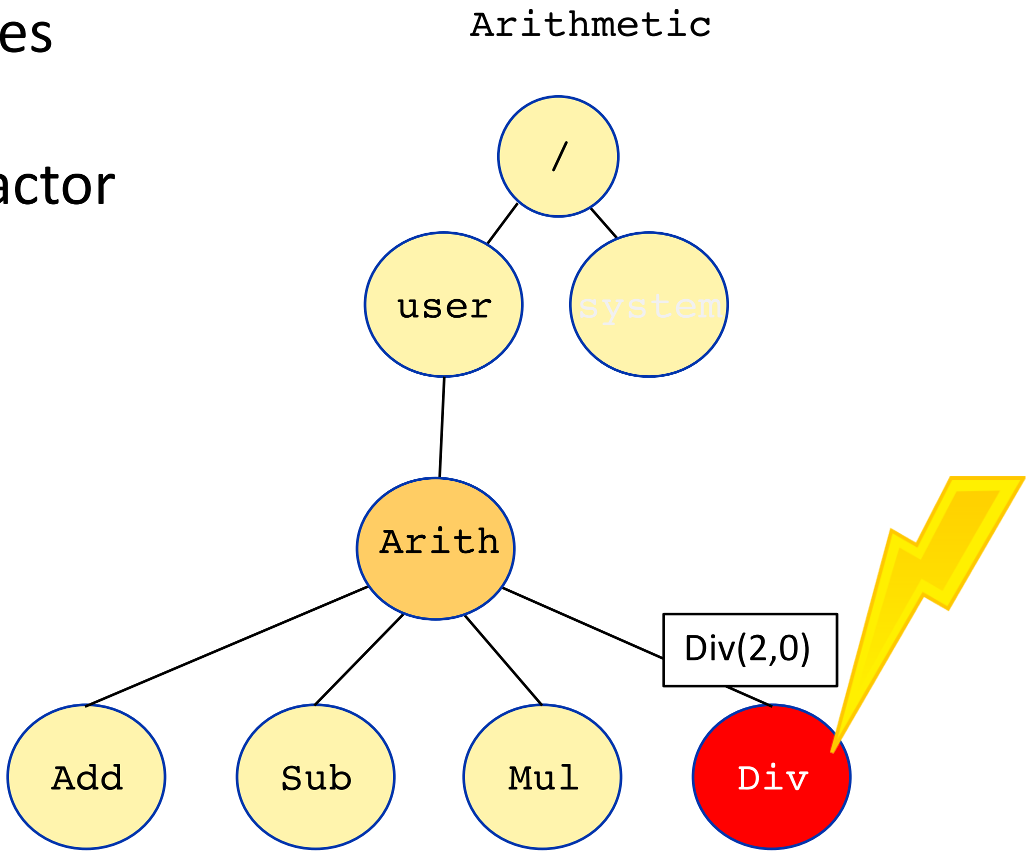


Dealing with the Failure

- `java.lang.ArithmeticException` is unchecked
    - in Scala all exceptions are unchecked!
- Normally application will terminate
    - without catch block for exception
- Actors are independent processing units
    - failing actor should not necessarily terminate application
- "Let it crash" approach
    - let actor crash
    - notice the crash
    - recover within the application

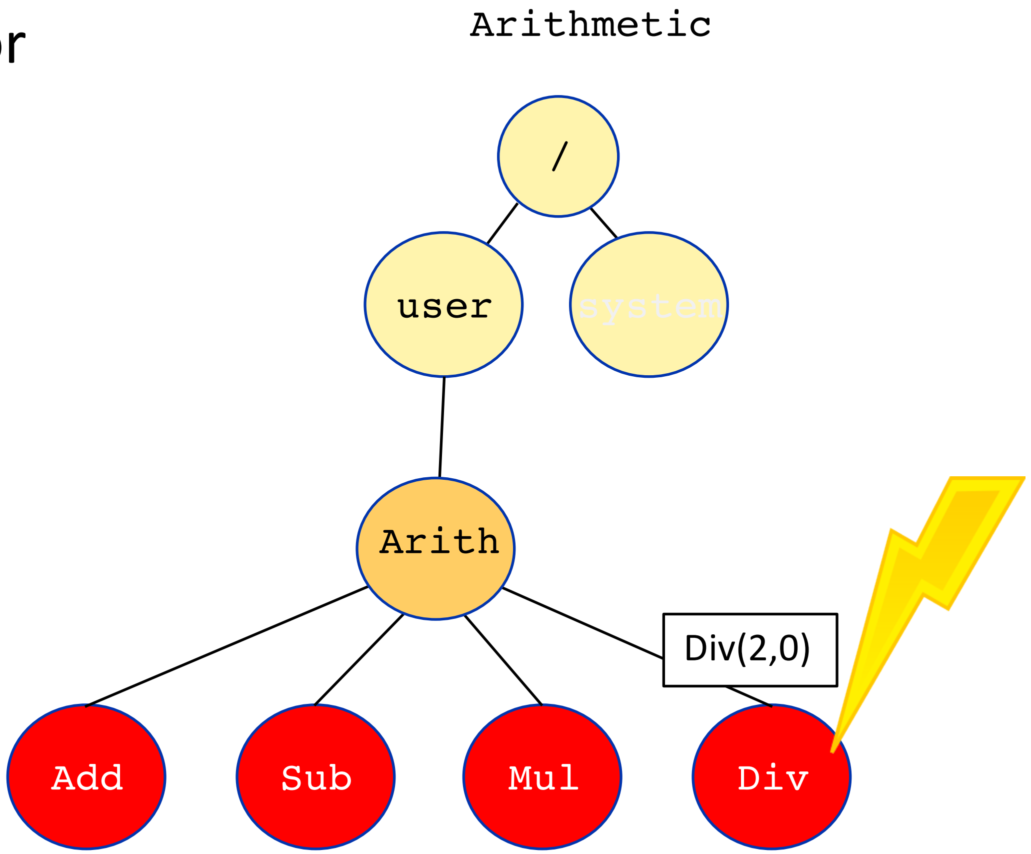
Fault Handling Options

- Parent actor decides
- Localise to failing actor
    - carry on?
    - restart?



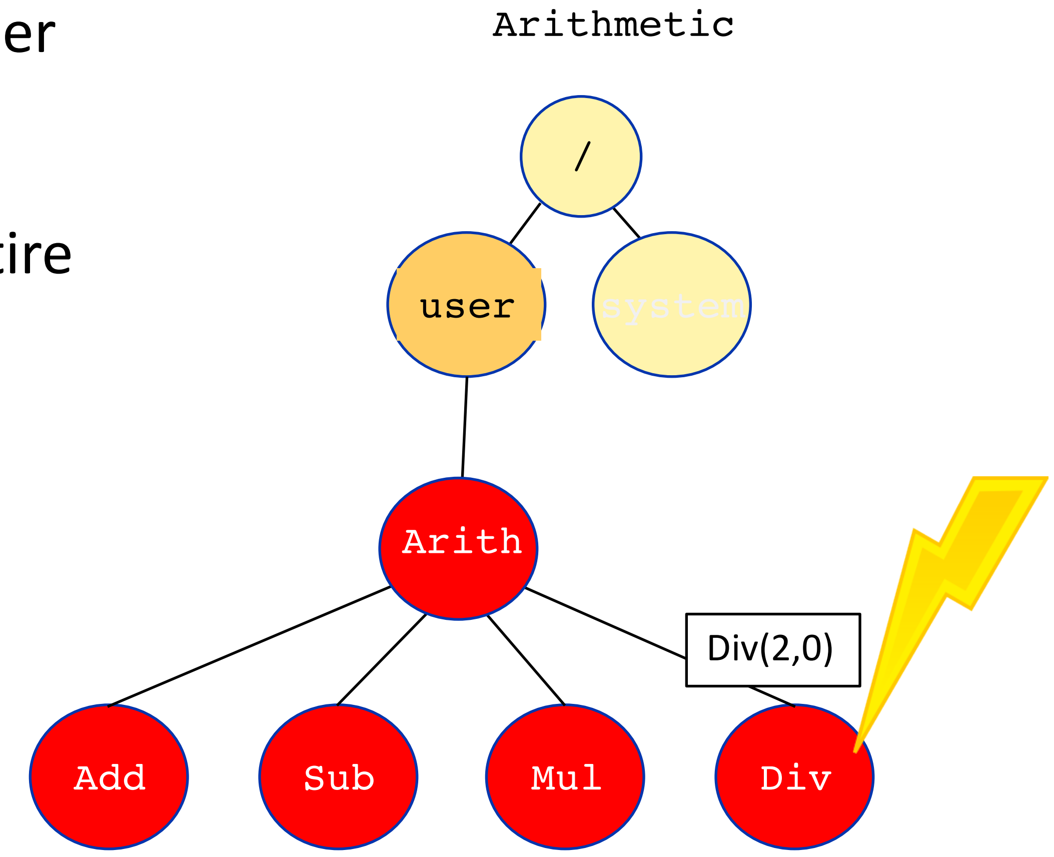
Fault Handling Options

- Failure affects actor and siblings



Fault Handling Options

- Propagate to higher level controller
- Failure affects entire processing tree



Default Fault Handling

- `ActorInitializationException`
  - stop the failing actor
- `ActorKilledException`
  - stop the failing actor
- Any other `Exception`
  - restart the failing actor
- Any other `Throwable`
  - escalate to parent supervisor
  - escalation from root actor causes guardian (user) to stop

Specifying Fault Handling

- Actor supervises its children
    - override `supervisorStrategy` to configure
- `OneForOneStrategy`
    - fault handling restricted to faulty actor
- `AllForOneStrategy`
    - all children affected by handling
- Define handling action using `Decider`
    - `PartialFunction[Throwable, Directive]`

Specifying Fault Handling

- Four possible `Directives`
- `Stop`
  - `stop` the actor(s)
  - default for initialisation problems or `ActorKilledException`
- `Restart`
  - create new actor(s) to replace failing one(s) and resume processing
  - default for Exceptions
- `Resume`
  - continue message processing (with next message)
- `Escalate`
  - delegate fault handling to supervisor's supervisor

Specifying Fault Handling

- Example

```
class Arith extends Actor with ActorLogging {
  import SupervisorStrategy._

  log.info("Creating Arith Actor")

  override val supervisorStrategy =
    OneForOneStrategy() {
    case _: ArithmeticException => Restart
    case _: RuntimeException    => Stop
  }

  …

}
```

Specifying Fault Handling

- Example – log messages

```
[INFO] … [akka://Arith/user/arithmetic] Creating Arith Actor
[INFO] … [akka://Arith/user/arithmetic/$a] Creating Add Actor
[INFO] … [akka://Arith/user/arithmetic/$b] Creating Sub Actor
[INFO] … [akka://Arith/user/arithmetic/$c] Creating Mult Actor
[INFO] … [akka://Arith/user/arithmetic/$d] Creating Divide Actor
[INFO] … [akka://Arith/user/arithmetic/$c] 2 * 4 -> 8
[ERROR] … [akka://Arith/user/arithmetic/$d] / by zero
java.lang.ArithmeticException: / by zero
       at
com.jgserv.DivideActor$$anonfun$receive$5.applyOrElse(ArithActorA
pp2.scala:110)
       at
akka.actor.ActorCell.receiveMessage(ActorCell.scala:425)
       at akka.actor.ActorCell.invoke(ActorCell.scala:386)
…

[INFO] … [akka://Arith/user/arithmetic/$d] Creating Divide Actor
[INFO] … [akka://Arith/user/arithmetic/$d] 2 / 2 -> 1
```

Specifying Fault Handling

- Example – handling IllegalArgumentException
  - Directive is to Stop

```
[INFO] … [akka://Arith/user/arithmetic] Creating Arith Actor
[INFO] … [akka://Arith/user/arithmetic/$a] Creating Add Actor
[INFO] … [akka://Arith/user/arithmetic/$b] Creating Sub Actor
[INFO] … [akka://Arith/user/arithmetic/$c] Creating Mult Actor
[INFO] … [akka://Arith/user/arithmetic/$d] Creating Divide Actor
[INFO] … [akka://Arith/user/arithmetic/$c] 2 * 4 -> 8
[ERROR] … [akka://Arith/user/arithmetic/$d] Bad karma
java.lang.IllegalArgumentException: Bad karma
       at
com.jgserv.DivideActor$$anonfun$receive$5.applyOrElse(ArithActorA
pp2.scala:109)
       at
akka.actor.ActorCell.receiveMessage(ActorCell.scala:425)
       at akka.actor.ActorCell.invoke(ActorCell.scala:386)
       at akka.dispatch.Mailbox.processMailbox(Mailbox.scala:230)
       at akka.dispatch.Mailbox.run(Mailbox.scala:212)
…
```

Following request for Div(2,2) is not processed

Restart Lifecycle Callback Methods

- `preRestart( t: Throwable,`
  `msg: Option[Any] ) : Unit`
    - called before restart process begins
    - default behaviour is to stop children and call `postStop`
- `postRestart( t: Throwable ) : Unit`
    - called after restart process finished
    - default behaviour is to call `preStart`
- Restarted actor created using factory method
    - continues with next message from mailbox

Fine Tuning the Fault Handling Strategy

- Danger of entering infinite loop
  - restarting a failing actor
- Strategy objects allow limits to be set on restarts
- `maxNrOfRetries (Int)`
  - number of times actor may be restarted
  - set to negative number for infinite
- `withinTimeRange (Duration)`
  - time window for `maxNrOfRetries`

```
override val supervisorStrategy =
                  OneForOneStrategy( 5, 1 minute ) { … }
```