Test of Concurrent Programming 2023-2024 – Module 2

DCC-FCUP, University of Porto
José Proença and Nelma Moreira

U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

18th June 2024 – duration: 2h00

Number: _____ Name: _____

You can use the following definitions, defined during lessons.

```
def thread(b:  =>Unit):  Thread = {
  //Runs ‘b‘ in a new thread
}
```

```
def execute(b:  =>Unit):  Unit = {
  //Runs ‘b‘ in the global execution context
}
```

You can use the following structures, covered during lessons.

```
new AtomicBoolean(bool:Boolean)
new AtomicInteger(i:Int)
new AtomicLong(l:Long)
new AtomicReference(ref:Any)
ActorSystem(name:  String)
actorSys.terminate()
actorSys.actorOf(p:Props,name:String)
```

```
context.actorOf(p:Props,name:String)
context.stop(...)
context.become(...)
context.children
context.parent
context.actorSelection(...)
```

Answer below each question. If you need more space, use the empty space at the end of the exam.
If needed ask for a new paper. A portuguese translation can be found at the end of the test.

Exercise 1. Write a small program with two parallel threads that use a single **volatile variable**. This program should **need** the volatile variable. Explain **(1)** the expected correct behaviour of the system, and **(2)** the possible incorrect behaviour if the volatile variable would **not** be volatile.

Exercise 2. Implement in Scala a `FIFO2` queue that can be shared among concurrent threads, defining the empty holes in the code below, such that:

- `blockingEnqueue(e)`: if the queue has less than 2 elements, then add `e` to the queue; if it has 2 elements then **block** until some element is removed.
- `blockingDequeue(e)`: if the queue is not empty, then return the oldest value `e` in the queue an remove it; otherwise **block** until an element is available.

```
class FIFO2[T] {
  // define the state of the queue
  [                                  ]
  [                                  ]
  [                                  ]
  [                                  ]
  [                                  ]
  [                                  ]
  [                                  ]
  def blockingEnqueue(elem: T): Unit {
    // implement enqueuing
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
  }
  def blockingDequeue(): T {
    // implement dequeuing
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
    [                                ]
  }
}
```

Exercise 3. The code below uses a class `Advertisement` to store the number of visualizations of a given advertisement, e.g., in a mobile device. The methods `atomicIncr` are used to safely increment this number by concurrent threads, and `atomicIncr2` also guarantees that both input counters are incremented in a single step without interference. Assume that the advertisement names are unique.

```scala
object CountingAdvertisement extends App {
  class Advertisement(name: String) {
    private var views: Int = 0
    def +=(more: Int) = views += more
    override def toString():String =
      s"$name:␣$views␣views"
  }

  def atomicIncr1(ad: Advertisement, more: Int) {
    ad.synchronised { ad += more }
  }

  def atomicIncr2(ad1: Advertisement, more1: Int
                 ,ad2: Advertisement, more2: Int) {
    ad1.synchronised {
      ad2.synchronised {
        ad1 += more1
        ad2 += more2
      }
    }
  }

  // Testing the execution
  val crush = new Advertisement("Candy␣Crush")
  val uEats = new Advertisement("Uber␣Eats")
  val temu  = new Advertisement("Temu")

  execute {
    atomicIncr2(crush, 4, temu, 3)
  }
  execute {
    atomicIncr1(uEats, 4)
  }
  execute {
    atomicIncr2(temu, 2, crush, 5)
  }
  execute {
    atomicIncr2(temu, 6, uEats, 7)
  }

  Thread.sleep(1000)
  println("$crush,␣$uEats,␣$temu")
}
```

3.1. What do you expect the execution above to do? Do you expect concurrency problems? If not, why not? If so, propose corrected definitions of the left side to avoid the problems.

3.2. Re-implement `atomicIncr1`, adapting `Advertisement` if needed, to use lock-free programming.

Exercise 4. An inexperienced developer produced and tested the implementation of `AdvertisementActor` below, where individual apps send the number of advertisements viewed to a centralised controller.

4.1. Draw (1) the actor hierarchy and (2) a sequence diagram explaining a possible exchange of messages. Also (3) explain what is expected to be printed in the terminal during this execution.

4.2. Enumerate aspects that should be improved in this code, explaining why each is problematic.

```
object AdvertisementActor extends App {
  var sentAds = 0
  class AppMgr extends Actor {
    def receive = {
      case ("start",counter:Actor) =>
        val a1 = context.actorOf(Props(new
            MyApp(1)))
        val a2 = context.actorOf(Props(new
            MyApp(2)))
        a1 ! ("start",sender)
        a2 ! ("start",sender)
    }
  }
  class MyApp(id:Int) extends Actor {
    def receive = {
      case ("start",counter:ActorRef) =>
       // simulate the sending of ads
       if (id==1) {
        counter ! ("moreAdds","Candy␣Crush",6)
        sentAds += 6
        Thread.sleep(2000) // wait 2 seconds
        counter ! ("moreAdds","Temu",2)
        sentAds += 2
       } else {
        counter ! ("moreAdds","Candy␣Crush",3)
        sentAds += 3
        Thread.sleep(2000) // wait 2 seconds
        counter ! ("moreAdds","Uber␣Eats",5)
        sentAds += 5
       }
    }
  }
  class AdCounter extends Actor {
    var receivedAds = 0
    def receive = {
      case "start" =>
        val appMgr  = sys.actorOf(Props[AppMgr])
        appMgr ! ("start",this)

      case ("moreAdds",prod:String,views:Int) =>
        receivedAds += views
        println(s"Got␣$views␣more␣adds␣from␣'$prod'␣
            (total:␣$receivedAds)")
        // throw exception if more adds received than
            sent
        assert(sentAds >= receivedAds)

      case "explode" =>
        throw new RuntimeException("Exploded.")
    }
  }


  // Running the system

  val sys = akka.actor.ActorSystem("AdCountingSystem")
  val counter = sys.actorOf(Props[AdCounter])
  counter ! "start"
  Thread.sleep(1000) // wait 1 second
  counter ! "bum"
  counter ! "explode"
  Thread.sleep(3000)
  sys.terminate()
}
```

Exercise 5. Explain what each of the concepts below means in Akka, and what kind of mechanisms does it provide.

1. Actor Supervision
2. DeathWatch

1. *Escreve um pequeno programa com 2 threads em paralelo que usem uma única variável volátil. Este programa deve precisar da variável volátil. Explique (1) o comportamento correto esperado, e (2) o possível comportamento incorreto caso a variável usada não seja volátil.*

2. *Implemente em Scala um fila FIFO2 que possa ser partilhada por threads concorrentes, preenchendo os espaços em branco no código abaixo, de tal forma que:*
- blockingEnqueue(e): se a fila tiver menos que 2 elementos, então acrescenta "e" para a fila; se tiver 2 elementos então bloqueia até que algum elemento seja removido.
- blockingDequeue(e): se a fila não estiver vazia, então devolve o valor "e" mais antigo e remove-o da fila; caso contrário bloqueia até algum elemento estiver disponível.

3. *O código abaixo usa a classe Advertisement para armazenar o número de visualizações de um dado anúncio, e.g., num dispositivo móvel. Os métodos atomicIncr são usados para incrementarem de forma segura por threads concorrentes. O método atomicIncr2 também garante que ambos os anúncios são incrementados num único passo sem interferência. Assuma que os nomes dos anúncios são todos diferentes.*
*- **3.1** O que espera que a execução do código abaixo faça? Espera encontrar problemas de concorrência? Se não, porque não? Se sim, proponha uma versão corrigida das definições do código à esquerda.*
*- **3.2** Re-implemente atomicIncr1 (e Advertisement se necessário) para usar programação sem locks (lock-free programming).*

4. *Um programador com pouca experiência produziu e testou a implementação AdvertisementActor abaixo, onde aplicações individuais enviam o número de visualizações de anúncios para um controlador central.*
*- **4.1** Desenhe (1) a hierarquia de atores e (2) um diagrama de sequência que explique uma possível troca de mensagens. Além disse (3) explique o que é esperado que seja impresso no terminal.*
*- **4.2** Enumere aspetos que devam ser melhorados neste código, e explique porque é que cada um destes pode ser problemático.*

5. *Explique o que é que cada um destes conceitos significa no Akka, e que mecanismos é que eles fornecem: (1) Supervisão de atores, (2) DeathWatch.*