Programming Principles

Midterm Solution

Friday, April 17 2015

Exercise 1: Counting Semaphore using Compare-and-Set (25 points)

A counting semaphore is a synchronization construct that is used to control accesses to a finite number of shared resources (e.g. processors, network ports), when there are multiple threads competing to access the resource.

Assume that we have `n` units of a shared resource. A semaphore for the resource maintains a counter to track the number of units that are available. It supports two operations: `lock` and `unlock`. The `lock` operation decrements by *one*, and the `unlock` operation increments by one the number of available units of the shared resource. The `unlock` method may cause the number of available units to exceed the initial `n`.
In case there no units available at the time of invoking the `lock` operation, the operation blocks until at least one unit becomes available.

You task is to implement the counting semaphore, as per the above description, using only an `AtomicReference` that has the methods `compareAndSet` and `get`, and *no* other synchronization primitive. In particular, you *cannot* use `synchronized`, `notify`, `wait`, or `fetchAndIncrement` operations. However, you are *allowed* to perform busy waiting, using loops and/or tail recursion for blocking.

Question (25 points)

Implement a counting semaphore based on the following template.

Answer

```
class CountingSemaphore(units: Int) {
  // declare any private variables needed here

  val semaphore = new AtomicReference(units)

  @tailrec def lock(): Unit = {
    val current_value = semaphore.value
    if(current_value == 0 ||
      !semaphore.compareAndSet(current_value, current_value - 1)) lock()
  }

  @tailrec def unlock(): Unit = {
    val current_value = semaphore.value
    if(!semaphore.compareAndSet(current_value, current_value + 1)) unlock()
  }
}
```

Exercise 2: Asynchronous histogram (25 points)

A well known probability and statistics professor is doing some ground-breaking work on distributions at EPFL. His research is almost ready to be published but his proof of concept needs some heavy and time-consuming computations.

As a remarkable concurrency student, he hired you to create a concurrent and non-blocking version of his code. Given the `magicGeneration` function and the starting point he wrote below, your task is to compute the **unit digits histogram**.

```
type Bin = Int
type Count = Int

def magicGeneration(): Future[Int] = /* Some computation */

val asyncData = List(
  magicGeneration(),
  magicGeneration(),
  // ...
  magicGeneration()
)
```

Question 1: Combining futures (10 points)

`asyncData` is currently a `List[Future[Int]]`. Your first task is to convert a list of future into a future of list.

```
def sequence(fs: List[Future[Int]]): Future[List[Int]] =
  fs.foldRight(Future.successful(List.empty[Int])) { case (f1, f2) =>
    for {
      r1 <- f1
      r2 <- f2
    } yield r1 :: r2
  }
```

Question 2: Asynchronous counting (10 points)

Now that you have a single piece of data, you can write a function that compute the number of integers falling in each bins. The `computeBin` parameter will return the corresponding bin given one integer.

Hint: do not forget that the computation is asynchonous.

```
def countByBin(f: Future[List[Int]], computeBin: Int => Bin): Future[Map[Bin, Count]] =
  f.map { res =>
    res
      .groupBy(computeBin)
      .map { case (bin, values) =>
        bin -> values.size
      }
  }
```

Question 3: Units histogram (5 points)

Finally using the starting point, you can glue everything together to compute the unit digits histogram of `asyncData`. For example `List(Future(1), Future(14), Future(21))` should print `Success(Map(1 -> 2, 4 -> 1))` once the computation is finished. You can assume that all numbers are positive.

```
val histogram: Future[Map[Bin, Count]] = countByBin(sequence(asyncData), v => v % 10)

histogram.onComplete { values =>
  println(values)
}
```

Exercise 3: Quorum and Vote (25 points)

An important voting must take place at EPFL. Students are asked if *they agree to double the tuition fee*. Each student, refered to as an `AssemblyMember`, has to vote on the matter, and an EPFL `Scrutineer` will handle the voting process. The `Scrutineer` must collect all votes and announce the outcome of the vote.
The voting session should start only when more than half of the assembly have gathered to vote. In other words, when the *quorum is met*, To be more precise, an absolute majority of the `AssemblyMember` must have informed the `Scrutineer` that they are present before the voting starts.

For this whole exercise, you are given the following setup and a `Timer` actor that notifies a requester after a desired duration. The case classes and objects defined below should be sufficient to solve this exercise.
Think twice before adding a new one. **Focus on the part you have to implement and pay attention to the methods signatures.** But you might find useful infos in the given code (regarding code structure and/or logic).

```
// These are the messages sent by the Scrutineer
case object VoteIsOver
case class IssueUnderVote(val question: String)

// These are the messages sent by an AssemblyMember
case object ImHere
sealed trait VoteOption
case object Yes extends VoteOption
case object No extends VoteOption
case object Blank extends VoteOption

// Message related to the Timer
case object TimesUp
case class StartTimer(duration: Long)

object Quorum extends App {
  val system = ActorSystem("Quorum")
  val AssemblySize: Int = 153
  val VotingItem: IssueUnderVote = IssueUnderVote("raise the EPFL tuition fee")

  val scrutineer =
    system.actorOf(Props(new Scrutineer(AssemblySize, VotingItem)))

  val AssemblyMembers = (0 until AssemblySize).map{
    id => system.actorOf(Props(new AssemblyMember(scrutineer,
        (10000 * Math.random()).toInt)), name = s"Assembly_member_$id")
  }
}

class Timer extends Actor {
  override def receive: Receive = {
    case StartTimer(duration: Long) =>
      Thread.sleep(duration)
      context.sender() ! TimesUp
  }
}
```

Question 1: Assembly Member Actor (10 points)

Implement the `AssemblyMember` actor by filling the template shown below. This actor has to do the following:

1. Announce itself to the `Scrutineer` (with an `ImHere` message)
2. Wait for receiving an `IssueUnderVote` message, and then start thinking. The actor must wait for `thinkDelay` ms by starting a `timer`.
3. Then, there are two options:

- When done with thinking, the actor must reply to the `Scrutineer` with a `VoteOption` given by the helper method `makeUpMind()` and start waiting for an issue again.
- When receiving a `VoteIsOver`, it means that the member did not vote in time. The actor must start waiting on an issue again directly. You can assume that the next issue will start long after the previous timer sends the ignored `TimesUp` message.

Do not use any `var` declarations, use multiple Receive methods instead

```
class AssemblyMember(scrutineer: ActorRef, thinkDelay: Int) extends Actor {
  val timer = context.actorOf(Props(new Timer))

  def makeUpMind(): VoteOption = {
      Math.random() match {
        case rng if rng < 0.45 => Yes
        case rng if rng < 0.9  => No
        case _                 => Blank
      }
  }

  // Announce itself to the scrutineer and implement the receive routine

  scrutineer ! ImHere

  override def receive = waiting()

  def waiting(): Receive = {
    case issue: IssueUnderVote =>
      timer ! StartTimer(thinkDelay)
      context.become(thinking())
  }

  def thinking(): Receive = {
    case VoteIsOver =>
      context.become(waiting())

    case TimesUp =>
      scrutineer ! makeUpMind()
      context.become(waiting())
  }
}
```

Question 2: Scrutineer Actor (15 points)

In this part, you are given a partial implementation of the `Scrutineer` actor over the next pages. Implement the missing parts (focus on them). The handling of votes is already implemented but you must ensure those are valid votes by adding a `case` filtering invalid `VoteOption` before the `case v: VoteOption` already implemented. Ensure the following points:

- The voting process must start only when *the quorum is met*, which means that a majority has gathered. Starting the vote means sending an `IssueUnderVote` message to all `presentMembers`.
- The voting process must be open for `VoteDuration` only. Use the given `Timer` actor, the `TimesUp` handling is already implemented for you.
- `AssemblyMember` arriving after the voting process started and before it ended should still have the opportunity to vote.
- No `AssemblyMember` should be able to vote twice even if they are disfunctioning.

Do not use any `var` declarations, use the different Receive methods instead

```
class Scrutineer(assemblySize: Int, votingItem: IssueUnderVote) extends Actor {
  val AbsoluteMajority = assemblySize/2 + 1
  val VoteDuration = 50000L
  val timer = context.actorOf(Props(new Timer))

  override def receive = waitingToStart()

  def waitingToStart(presentAssemblyMembers: Set[ActorRef] = Set()): Receive = {

    // Wait for the quorum to be met and start the voting process
    case ImHere =>
      val presentMembers = presentAssemblyMembers + context.sender()
      val currentItem = 0
      if (presentMembers.size > assemblySize/2) {
        startVoteOn(presentMembers)
        context.become(voting(alreadyVoted = Set(), presentMembers,
            yesNoVotes = (0, 0)))
      } else {
        context.become(waitingToStart(presentMembers))
      }
  }

  def startVoteOn(members: Set[ActorRef]): Unit = {
    println(s"Do you agree to ${votingItem.question}?")
    members foreach(_ ! votingItem)
    timer ! StartTimer(VoteDuration)
  }
```

```
def voting(alreadyVoted: Set[ActorRef],
    presentMembers: Set[ActorRef], yesNoVotes: (Int, Int)): Receive = {

  /* Take care of AssemblyMember arriving late
   * Make sure no one is voting twice by filtering
   * invalid received VoteOption in a separate case
   */

  case ImHere =>
    val newMember = context.sender()
    if (!presentMembers(newMember)) {
      newMember ! votingItem
      context.become(voting(alreadyVoted,
          presentMembers + newMember, yesNoVotes))
    }

  case _: VoteOption if alreadyVoted(context.sender()) =>
    println("You cannot vote twice on the same issue :" + context.sender())

  case v: VoteOption =>
    val (yesVotes, noVotes) = yesNoVotes
    val votes = v match {
      case Blank => yesNoVotes
      case Yes => (yesVotes + 1, noVotes)
      case No => (yesVotes, noVotes + 1)
    }

    val voteCasted = alreadyVoted + context.sender()

    tallyVotes(votes, voteCasted, presentMembers,
        presentMembers == voteCasted){ () =>
      context.become(voting(voteCasted, presentMembers, votes))
    }

  case TimesUp =>
    tallyVotes(yesNoVotes, alreadyVoted, presentMembers,
        isOver = true)(() => {})
}

def done(): Receive = {
  case _ => println(s"The voting session is over, it is too late to vote " +
      context.sender())
}
```

```
  /* Sets the scrutineer's Receive to done() if the voting can be decided,
     and calls the callback cont() otherwise.  */
  def tallyVotes(votes: (Int, Int), alreadyVoted: Set[ActorRef],
      members: Set[ActorRef], isOver: Boolean)(cont: () => Unit): Unit = {

    lazy val IssueUnderVote(question) = votingItem
    val (yesVotes, noVotes) = votes

    def endVote(message: String): Unit = {
      (members -- alreadyVoted) foreach(_ ! VoteIsOver)
      println(message)
      context.become(done())
    }

    if (yesVotes >= AbsoluteMajority)
      endVote(s"The Assembly agreed to $question by an absolute majority!")
    else if (noVotes >= AbsoluteMajority)
      endVote(s"The Assembly refused to $question by an absolute majority!")
    else if (isOver) {
      if (yesVotes > noVotes)
        endVote(s"The Assembly agreed to $question: $yesVotes against $noVotes votes!")
      else if (yesVotes == noVotes)
        endVote(s"The Assembly was unable to choose whether to $question or not")
      else
        endVote(s"The Assembly refused to $question: $noVotes against $yesVotes votes!")
    } else cont()
  }
}
```

Exercise 4: MMORPG Data Analysis (25 points)

In this exercise, you will be working with Spark to analyse data on a massively-multiplayer online role playing game (MMORPG). You may refer to the Spark API on page **??**. In order to obtain full points for each part, your solution not only need to be correct, but also ***efficient***! We give you enough space after each part to write down your answer. Please use that space.

In the game, characters are represented by the following case class:

```
case class Character(
    id: Int,
    name: String,
    guildId: Option[Int],
    level: Int,
    role: Role
)

sealed abstract class Role
case object Warrior extends Role
case object Mage extends Role
case object Priest extends Role
```

Each character has a unique identifier and a name. Each character also has a role, which defines its abilities in battle and a level, which indicates their power. Characters start at level 1 and can reach up to level 60. Characters in this game can join guilds, which are associations of characters. Joining a guild is completely optional.

Assume that we have a RDD containing all characters of the game:

```
val characters = sc.parallelize(List(
    Character(1, "Alek",    None,     60, Warrior),
    Character(2, "Beatrix", Some(17), 45, Priest),
    Character(3, "Cleop",   Some(17), 1,  Mage),
    Character(4, "Deadra",  Some(41), 60, Priest),
    ...
))
```

Assume that the data is too large to process on a single machine.

Question 1: Character Progression (10 points)

In a hurry, Tom, the lead designer of the game, storms your office and asks you to compute some statistics about the game for an important board meeting taking place very soon! Those numbers will help them gauge the popularity of the game.

1.1) Using `characters`, efficiently compute the number of characters which have reached level 60.

```
val numberOfLevel60: Long = characters.filter(_.level == 60).count()
```

1.2) Using `characters`, efficiently compute the average level of all characters in the game depending on their role.

```
val averageLevelPerRole: Map[Role, Double] =
  characters.map(c => (c.role, (c.level, 1)))
            .reduceByKey(mergePairs)
            .mapValues(p => p._1.toDouble / p._2)
            .collectAsMap()

def mergePairs(p1: (Int, Int), p2: (Int, Int)): (Int, Int) =
  (p1._1 + p2._1, p1._2 + p2._2)
```

Question 2: The Most Numerous Guild Award (5 points)

After his meeting, Tom tells you that the board has decided to give a special award to the guild with the most number of characters in it. It is your job to find out which guild deserves the award.

Using `characters`, efficiently compute the identifier of the guild with the most members. If some guilds have the same number of members, you may choose any of them as the result of your computation.

```
val mostNumerousGuildId: Int =
  characters.flatMap(c => c.guildId.map((_, 1)))
            .reduceByKey(_ + _)
            .sortBy(_._2, false)
            .take(1)
            .head
            ._1
```

Question 3: Final Boss (5 points)

Very satisfied with your job so far, Tom comes to you with a final request. Some players have been complaining that the final boss fight was too difficult for some characters. Tom suspects that the difficult of the fight strongly depends on the role of the character. To verify this, the company has been monitoring the boss fight for a week and collected records of the following form:

```
case class FightCompleted(characterId: Int, bossDefeated: Boolean)
```

Where `characterId` is the identifier of the character and `bossDefeated` indicates whether the boss was defeated by the character or not.

A week's worth of records have been collected so far in the following RDD:

```
val bossEvents = sc.parallelize(List(
  FightCompleted(1, false),
  FightCompleted(7, false),
  FightCompleted(4, true),
  ...
))
```

Efficiently compute, using `characters` and `bossEvents`, the ratio (between 0 and 1) of times the boss was defeated per encounter depending on the role of the character.

```
val bossDefeatedRatio: Map[Role, Double] = {
  val pairs = characters.map(c => (c.id, c))
  val partitioned = pairs.partitionBy(new RangePartitioner(100, pairs)).persist()

  partitioned.join(bossEvents.map(e => (e.characterId, e))).map {
    case (_, (c, e)) => (c.role, (if (e.bossDefeated) 1 else 0, 1))
  }
  .reduceByKey(mergePairs)
  .mapValues(p => p._1.toDouble / p._2)
  .collectAsMap()
}
```

Question 4: Shuffling (5 points)

During the coffee break, you meet with Tom again. He tells you that, being very impressed with your job, he has started learning to use Spark himself. During the conversation, he tells you that he doesn't quite understand what shuffling is and why it is relevant.

4.1) Using your own words, explain what shuffling is and why it is problematic. Limit yourself to 4 phrases at most.

Shuffling happens when data is sent over the network between the different nodes. This happens because tuples must be held in a specific partition, which might be different from the one they were created at. This is problematic because network communications are comparatively very slow.

4.2) What technique can be used to avoid unnecessary shuffling within Spark? Limit yourself to 2 phrases at most.

Range partitioning can be used to group together tuples with the same keys. Some methods, such as reduceByKey or join, involve way less shuffling when data is partitioned this way.