Parallelism and Concurrency

Final Exam

Wednesday, May 30, 2018

Your points are *precious*, don't let them go to waste!

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated, some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Closed-book The exam is closed book.

No communication You are not allowed to use internet, mobile phones, laptops, smart watches, etc.

Exercise	**Points**	**Points Achieved**
1	20	
2	20	
3	20	
Total	60	

Exercise 1: Spark (20 points)

You are the lead data analyst for the e-commerce site *Macoop*. The company maintains logs of all activities on Spark.

Question 1

The online shop sells various kinds of items. Each item kind has a record of the form:

```
case class ItemKind(id: Int, name: String, price: Int)
```

Each record contains a unique `id`, a unique `name` and a price (in cents). The entire collection of item kinds is stored as an RDD named `items`:

```
val items: RDD[ItemKind] = ...
```

Your task:

Efficiently build an array of the name of all item kinds whose price is stricly larger than 1000 CHF (i.e., 100'000 cents). The order in which items appear in the array is not specified.

```
val expensiveItemsNames: Array[String] =
```

Question 2

The online shop also records all orders made by clients throughout the shop's history. Each order has a record of the following form:

```
case class Order(id: Int, date: Date, itemIds: Seq[Int])
```

Each order has a unique `id` and a `date`. Each record also contains a sequence of all item kind ids that are part of the order. When a client orders the same item multiple times, the corresponding item kind id appears multiple times in the sequence. The entire collection of orders is available as an RDD named `orders`:

```
val orders: RDD[Order] = ...
```

Your task:

Efficiently compute how many items were sold during the last *Black Friday*. You should use the helper function `isLastBlackFriday` to check if a date corresponds to the last *Black Friday*.

```
def isLastBlackFriday(date: Date): Boolean = ... // Given to you.
```

```
val itemsSoldLastBlackFriday: Int =
```

Question 3

Efficiently get the id and price (in cents) of the most expensive order ever made to the shop. The price of an order is the sum of the prices of all items ordered. In the case of a tie, you a free to return any of the most expensive orders.

```
val mostExpensiveOrder: (Int, Int) =
```

Exercise 2: Futures (20 points)

Suppose you are given the following API to query student grades from IS-Academia:

```
def getGrade(s: Student): Future[Double]
```

Under the hood `getGrade` sends an authenticated HTTP request to the IS-Academia servers. If IS-Academia is not broken that day it will answer with the student's grade for the parallelism and concurrency class.

Here is an example showing how `getGrade` can be used from the Scala console:

```
scala> val me = Student("STUDENTNAME", STUDENTSCIPER)
me: Student = Student(STUDENTNAME,STUDENTSCIPER)

scala> val future = getGrade(me)
future: scala.concurrent.Future[Double] = Future(<not completed>)

scala> future.foreach(println)
6.0 // Appears after ~500ms
```

Question 1

Given three students `s1`, `s2` and `s3`, use the `getGrade` API to emit three **sequential** requests to IS-Academia and compute the average of these three grades. One reason we might want to limit ourself to sequential requests would be to prevent flooding the IS-Academia server. In this context, we say that two requests are sequential if the second request is emitted *after* completion of the first request.

```
def computeAverage(s1: Student, s2: Student, s3: Student): Unit = {
  // TODO: emit 3 sequencial requests









  // TODO: compute the average grade
  val average: Future[Double] =









  average.foreach(println)
}
```

Question 2

Update your implementation to emit **concurrent** requests instead. In this context, we say that two requests are concurrent if they are emitted one after the other, without waiting for completion of first request to emit the second request.

```
def computeAverage2(s1: Student, s2: Student, s3: Student): Unit =  {
  // TODO: emit 3 concurrent requests









  // TODO: compute the average grade
  val average: Future[Double] =









  average.foreach(println)
}
```

Question 3

Implement the `zip` method on `Future`s.

Zip creates a new `Future` holding the tuple of results of the two `Future`s it's made from. If the first `Future` fails, the resulting `Future` is failed with the exception stored in the first `Future`. Otherwise, if the second `Future` fails, the resulting `Future` is failed with the exception stored in the second `Future`.

```
trait Future[A] {
  def zip[B](fb: Future[B]): Future[(A, B)] = {









  }
}
```

Exercise 3: Producer-consumer Problem (20 points)

You are an employee of the booming e-commerce company *Amigros*, a competitor of *Macoop*. To better handle fast-growing orders, the engineering team decides to introduce a *parallel order processing system*. The new system is designed around the following two concepts:

- *producer*: producers add new jobs to a shared queue.
- *consumer*: consumers remove jobs from the shared queue and handle them.

Multiple producers and multiple consumers are collaborating on a shared queue. By tuning the size of the job queue and the number of consumers, the infrastructure can scale easily to growing demands.

Realizing the core of the system will be the shared job queue, the engineering team agreed on the following design of the job queue:

```
case class Job(...)

abstract class JobQueue(maxSize: Int) {
  def put(job: Job): Unit
  def take: Job
}
```

The specification of the interface is as follows:

- **S1**: Producers will call `put` to schedule a job. The job should be appended to the queue when the call returns. The method should handle the case where the queue is full.
- **S2**: Consumers will call `take` to remove a job from the queue. It removes the first job from the queue and returns it. The method should handle the case where the queue is empty.
- **S3**: The system should not run into deadlock in the case of multiple consumers and producers.
- **S4**: The length of the job queue should not exceed `maxSize`.

Question 1

One of your colleagues proposed the following implementation of the job queue based on the idea of *circular buffer*:

```
class MyQueue(maxSize: Int) extends JobQueue(maxSize) {
  var front = 0
  var size = 0
  val buffer = new Array[Job](maxSize)

  def put(job: Job): Unit = synchronized {
    if (size == maxSize) wait()

    val rear = (front + size) % maxSize
    buffer(rear) = job
    size += 1
```

```
        notifyAll()
      }


      def take: Job = synchronized {
        if (size == 0) wait()

        val job = buffer(front)
        front = (front + 1) % maxSize
        size -= 1

        notifyAll()

        job
      }
    }
```

Does the implementation satisfy the specification S1-S4? If not, describe the changes needed to make it work? Justify your answer briefly.

Question 2

After the proposed system works well for some time, your manager asks for a web page to observe the job queue in real-time. To support this feature, the engineering team decides to add a new method `snapshot` to `JobQueue`:

```
abstract class JobQueue(maxSize: Int) {
  def put(job: Job): Unit
  def take: Job
  def snapshot: List[Job]
}
```

The specification for `JobQueue` is augmented as follows:

- **S5**: `snapshot` is called by watchers. It will return the content of the queue at some time-point during the call of `snapshot`.

Note that the requirement S5 **rejects** implementations that produce the following output:

1. The job queue is `j1, j2`
2. Watcher A calls `snapshot`
3. A consumer removes `j1`
4. A producer appends `j3`
5. `snapshot` returns `List(j1, j2, j3)` to watcher A

The output above is incorrect because there exists no instant that the job queue holds the jobs `j1`, `j2` and `j3`. In the schedule above, returning `List(j2)`, `List(j1, j2)` or `List(j2, j3)` would be *correct*. However, returning `List(j2, j1)` or `List(j3, j2)` would be *incorrect* because of the wrong ordering.

Knowing that you are an expert on concurrency, you are asked by the CTO to propose an improvement of `JobQueue` such that it satisfies all the requirements S1-S5. Please demonstrate your expertise.

```
def snapshot: List[Job] =
```

Question 3

After the snapshot feature is deployed in production, the maintenance team noticed that the system has a regression on performance. After some diagnosis, you realized that it is possible to have multiple watchers doing snapshot at the same time. You formulate the intution with the following specification:

- **S6**: multiple watchers should be able to execute `snapshot` in parallel.

Propose an improved version of `JobQueue` that implements the specifications S1-S6. For the methods `take` and `put`, you may write only the changed line(s).

```
class MyQueue(maxSize: Int) extends JobQueue(maxSize) {
  var front = 0
  var size = 0
  val buffer = new Array[Job](maxSize)




  def put(job: Job): Unit =
















  def take: Job =
  def snapshot: List[Job] =

}
```

Spark API

Relevant API for Spark `RDD[T]`:

- `def collect(): Array[T]`: Return an array that contains all of the elements in this RDD.
- `def count(): Long`: Return the number of elements in the RDD.
- `def distinct(): RDD[T]`: Return a new RDD containing the distinct elements in this RDD.
- `def filter(f: (T) => Boolean): RDD[T]`: Return a new RDD containing only the elements that satisfy a predicate.
- `def flatMap[U](f: (T) => TraversableOnce[U]): RDD[U]`: Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results. Collection types such as `Seq[U]`, `List[U]` and even `Option[U]` are valid `TraversableOnce[U]`.
- `def fold(zeroValue: T)(op: (T, T) => T): T`: Aggregate the elements of each partition, and then the results for all the partitions, using a given function and "zero value".
- `def groupBy[K](f: (T) => K): RDD[(K, Iterable[T])]`: Return an RDD of grouped items.
- `def map[U](f: (T) => U): RDD[U]`: Return a new RDD by applying a function to all elements of this RDD.
- `def maxBy[K](f: (T) => K): T`: Return a maximum element according to the given key function `f`.
- `def reduce(f: (T, T) => T): T`: Reduces the elements of this RDD using the specified commutative and associative binary operator.
- `def sortBy[K](f: (T) => K, ascending: Boolean = true): RDD[T]`: Return this RDD sorted by the given key function.
- `def take(num: Int): Array[T]`: Take the first num elements of the RDD.
- `def union(other: RDD[T]): RDD[T]`: Return the union of this RDD and another one.

Additional methods available to RDDs of type `RDD[(K, V)]`:

- `def groupByKey(): RDD[(K, Iterable[V])]`: Group the values for each key in the RDD into a single sequence.
- `def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]`: Return an RDD containing all pairs of elements with matching keys in this and other.
- `def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]`: Perform a left outer join of this and other.
- `def mapValues[U](f: (V) => U): RDD[(K, U)]`: Pass each value in the key-value pair RDD through a map function without changing the keys.
- `def reduceByKey(func: (V, V) => V): RDD[(K, V)]`: Merge the values for each key using an associative reduce function.
- `def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]`: Perform a right outer join of this and other.
- `def values(): RDD[V]`: Return a RDD containing only the values.

Future API

Relevant API for `Future[T]`:

- `def onComplete(callback: Try[T] => Unit): Unit`: When this future is completed, either through an exception, or a value, apply the provided function.
- `def flatMap[S](f: T => Future[S]): Future[S]`: Creates a new future by applying a function to the successful result of this future, and returns the result of the function as the new future.
- `def filter(p: T => Boolean): Future[T]`: Creates a new future by filtering the value of the current future with a predicate.
- `def map[S](f: T => S): Future[S]`: Creates a new future by applying a function to the successful result of this future.
- `def foreach(f: T => Unit): Unit`: Asynchronously processes the value in the future once the value becomes available.

A `Future` represents a value which may or may not *currently* be available, but will be available at some point, or an exception if that value could not be made available.

Try API

The `Try` type represents a computation that may either result in an exception, or return a successfully computed value. It's similar to, but semantically different from the `scala.util.Either` type.

Instances of `Try[T]`, are either an instance of `Success[T]` or `Failure[T]`:

```
sealed trait Try[T]
case class Failure[T](exception: Exception) extends Try[T]
case class Success[T](value: T) extends Try[T]
```