Library Design for Parallel and Asynchronous Computation

Most of the design considerations presented in Chapter 7 of the text book are not very well reflected in below. You need the context, in which the exercises have been posted, to understand the design story (which is definitely expected for a good grade). Read through the chapter text carefully while solving exercises.

We are using standard library lists (not fpinscala/adpro) lists in this module. There are rather few differences from what we constructed. Here is the API documentation: https://www.scala-lang.org/api/current/scala/collection/immutable/List$.html

If the test suite behaves weirdly (as if you are sure that the solutions are good but tests are failing), consider increasing the `TIMEOUT` value in the test file. Perhaps your computer is too slow, or too fast, to work with the configuration I suggested. If tests deadlock (seem to never terminate), it is likely that your solution is at fault.

All exercises are to be solved by extending the file `Par.scala`—the only file you should hand in. This exercise sheet is short, ranked for about 2 hours 15 minutes. However, this may be deceiving. It is essentially impossible to solve it without reading the chapter in parallel (which will take much more time). To make things more subtle, the chapter discusses two APIs that are very similar, but not identical. The simpler one is based on Java's `Future`. The problem with the API is that it cannot guarantee deadlock freeness when the number of threads in the pool is limited. The non-blocking API is introduced in Section 7.4.4, and all the exercise below assume the use of the non-blocking API. This mostly means that you do not have to call the `get` method of `Future` after calling `run`, but `run` returns the value of the computation directly. We chose to work with the non-blocking version, which is arguably slightly more complex, to avoid deadlocks in tests (which would cause you lots of frustration).

Hand-in: only `Par.scala`! Exercise 11 is the most important so do not (!) skip it.

Exercise 1. The book introduces the following function on the basic type `Par` in Section 7.1.1:

```
def unit[A] (a: =>A): Par[A]
```

Discuss in your group why the argument `a` is passed by-name to the `unit` function. Write a short explanation (max 5 lines) in the dedicated comment in `Par.scala` .

Exercise 2. Use `lazyUnit` to write a function that converts any function `A =>B` to one that evaluates its result asynchronously (so it spawns a separate thread).

```
def asyncF[A,B](f: A =>B): A =>Par[B]
```

A suitable place to start is marked Exercise 1 in `Par.scala`.[1]

Exercise 3. Find the implementation of `Par.map` in the chapter. Write in English how would you test it. What is a test-case and how would you decide that the test-case has passed. Write your response in English in the designated comment in the Scala file.

Expected size: 10-15 lines of text, but more (and less) is allowed.

Exercise 4. Write a function `sequence` that takes a list of parallel computations (`List[Par[B]]`) and returns a parallel computation producing a list (`Par[List[B]]`). No additional primitives are required. Don't call run, as we do not want this function to execute anything yet.[2]

[1] Exercise 7.4 [Chiusano, Bjarnason 2014]
[2] Exercise 7.5 [Chiusano, Bjarnason 2014]

```
def sequence[A](ps: List[Par[A]]): Par[List[A]]
```

Observe that `sequence` should not execute (force) any of the computations on the argument list. It should just 'repackage' the entire thing as a single parallel computation producing a list. Why is this useful? This allows us to continue building a computation on the entire list (for instance using `map` or `sortPar` from the chapter) without waiting for all the elements to terminate.

Exercise 5. Implement `parFilter`, which filters elements of a list in parallel (so the predicate `f` is executed in parallel for the various lists elements). [3]

```
def parFilter[A](as: List[A])(f: A =>Boolean): Par[List[A]]
```

Exercise 6. Implement `map3` using `map2`.

```
def map3[A,B,C,D] (pa :Par[A], pb: Par[B], pc: Par[C]) (f: (A,B,C) =>D)
```

Exercise 7. Implement `choiceN` and then `choice` in terms of `choiceN`.[4]

Exercise 8. Implement a general parallel computation `chooser`, and then use it to implement `choice` and `choiceN`. A `chooser` uses a parallel computation to obtain a selector for one of the available parallel computations in the range provided by `choices`:[5]

```
def chooser[A,B](pa: Par[A])(choices: A =>Par[B]): Par[B]
```

Note: In search for an "aha" moment, compare the type of the chooser, with the types of `Option.flatMap`, `Stream.flatMap`, `List.flatMap` and `State.flatMap`. Observe that the `chooser` is used to compose (sequence) to parallel computations here.

Exercise 9. Implement `join`. Can you see how to implement `flatMap` using `join`? And can you implement `join` using `flatMap`?

```
def join[A](a: Par[Par[A]]): Par[A]
```

Compare the type of `join` with the type of `List.flatten` (and the relation of `join` to `chooser` against the relation of `List.flatten` to `List.flatMap`).

Exercise 10. This exercise has nothing to do with parallelism, but it trains the general skill of expanding types using extension methods (as suggested in Chapter 7 of the text book). Implement extension methods for `Par` in Scala (using implicits) so that the following calls will work:

- If `l` is a `Par[A]` and `f` is a function `A =>B` then `l.map (f)` works the same way as `Par.map (l) (f)`
- If `l` (respectively `r`) is a `Par[A]` (respectively `Par[B]`) and `f` is a function `(A,B)=>C` then `l.map2 (r) (f)` works the same way as `Par.map2 (l,r) (f)`
- If `pa` is a `Par[A]` and `pm` is a function `A =>Par[B]` then `pa.chooser (pm)` works the same way as `Par.chooser (pa) (pm)`

Exercise 11. This is an open exercise that aims to help you operationalize the use of the `Par` API. Write a short function `wget` that, given a (variable length argument) list of URIs, downloads the files at URIs and returns the strings representing the HTML files at these addresses. For instance, we

[3] Exercise 7.6 [Chiusano, Bjarnason 2014]
[4] Exercise 7.11 [Chiusano, Bjarnason 2014]
[5] Exercise 7.13 [Chiusano, Bjarnason 2014]

could call:

```
wget ("http://www.dr.dk", "http://www.itu.dk")
```

This should return a list of two strings, the first one containing the file returned by DR, the second by ITU. You can use `scala.io.Source.fromURL` to download the file. It is important that downloading proceeds in parallel (so all the files in a list of arguments are downloaded over parallel connections), using the `Par` API.

There are no real tests for this exercise, but the test `"Exercise 11"` simply attempts to download three websites using your implementation of `wget` and prints the first line (up to 100 chars) in the test log. It will fail, if your code throws an exception.

Experiment with downloading sequentially and in parallel by modifying the test code. Can you observe a difference in speed?

Important: In a comment below the implementation of `wget` explain in a few sentences how your implementation achieves concurrency.