TDT4165 Programming Languages
Scala Project Delivery 1

Random Group 4

Sondre T Bungum, Håvard R. Krogstie, Hermann Mørkrid

October 30, 2021

1   **Scala Introduction**

All the code for this task can be found in `task1/Main.scala`.

(d) The function `fibonacci(n: Int)` gives the n'th fibonacci number, with 0 being the zeroth, and 1 being the first. The rest of the seqence is implemented using the recursive definition of fibonacci.

The function return type is `BigInt` because the fibonacci sequence quickly exceeds $2^{31} - 1$, which is the maximum value representable as an `Int`. A `BigInt` however does not have a pre-determined range of possible integer values. It will allocate enough space to fit the value as it grows.

2   **Concurrency in Scala**

All the code for tasks **a**, **b** and **c** can be found in `task2/task2abc/Main.scala`. The code for the last subtask **d** is in `task2/task2d/Main.scala`.

(b) The `increaseCounter()` function and `printCounter()` both use the variable `counter`. They perform updates and reads respectivly. When we spawn 3 threads, we don't

know what order the scheduler is going to run our threads in. Even though we have the code

```
val thread1 = initializeThread(increaseCounter)
val thread2 = initializeThread(increaseCounter)
val thread3 = initializeThread(printCounter)

thread1.start()
thread2.start()
thread3.start()
```

we sometimes get 1 printed to the console. This is because `printCounter()` can be called before the `increaseCounter()` calls have done their job.

Another option is that the two calls to `increaseCounter()` happen on top of each other, in such a way that the two invocations of `counter += 1` don't see the effects of each other. They will both read the same old value of `counter`, and thus `counter` is only increased by one.

The possibilities of this occuring means the code is non-deterministic. This kind of problem is called a race condition, which is what happens when two threads are attempting to use the same memory address, at least one of the threads is writing to the address, and there is no **happens before**-relation between the two operations.

This can be very problematic in a banking system, if several deposits are made, and multiple deposits work with the old balance, the resulting balance will not see the effects of all the deposits.

We can avoid this by using syncronization atomics that create **happens before** relations across threads, or use locks to avoid multpiple threads ever working on the same memory at once.

(d) A deadlock occurs when two threads are waiting for a responce for one another. This can be avoided by creating order so that we are always sure that an action is finished or can be finished before or when it is required.