

TDT4165 Programming Languages

Scala Project Delivery 1

Random Group 4

Sondre T Bungum, Håvard R. Krogstie, Hermann Mørkrid

November 1, 2021

1

Scala Introduction

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

- (a) The `createIntegerArray()` function declares a new array of 50 items, and then assigns those items to the numbers 1 to 50.
- (b) The `sumIntegerArray(array)` function declares a variable `sum`, then loops over the provided array of integers to sum them.
- (c) The `recursiveSumIntegerArray(array)` function first checks for the exit condition of the array being empty. If not, it sums the first element of the array with a recursive call to the function with the remainder of the array.
- (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 sequence 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`.

- (a) The `initializeThread()` function takes a function as its argument, and returns an initialized, non-started thread with the `run()` method overridden by the provided function. When calling `start()` on the thread later, a call will be made to this `run()` function.
- (b) The `increaseCounter()` function and `printCounter()` both use the variable `counter`. They perform updates and reads respectively. 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 occurring 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 synchronization atomics that create **happens before** relations across threads, or use locks to avoid multiple threads ever working on the same memory at once.

- (c) `increaseCounter()` and `printCounter()` are made thread-safe through the implementation of `safeIncreaseCounter()` and `safePrintCounter()`, which each wrap the given functions in the `this.synchronized()` function. `this.synchronized()` makes sure that any variables operated on within the provided function are locked while operating on them, meaning that other functions are not allowed to operate on them at the same time. The `this` part of the function call binds it to the context of the surrounding object, in this case `Task2`.
- (d) A deadlock occurs when two threads are waiting for a response 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.

In our implementation, deadlock is caused by having two `lazy` variables refer back to each other.