UNIVERS
DE MÁLA
Dpto. Lenguajes y Ciencias de la Computación

Programación avanzada II

Laboratorio 4

1. a) Escribe un programa Scala que tenga 3 hebras, además de la `main`. Cada hebra debe escribir **t** veces el carácter **c**, siendo **t** y **c** parámetros del constructor de la hebra. ¿Se mezclan los resultados?

   b) Modifica el programa anterior para que las letras A, B y C se escriban siguiendo el patrón: ABBCCC. Así, si la primera hebra escribe **t=3** As, la segunda **2*t=6** Bs y la tercera **3*t=9** Cs la única salida válida es ABBCCCABBCCCABBCCC. Para sincronizar las hebras, utiliza una variable compartida `turno` que asigne turnos a las hebras, de forma que cada hebra solo escriba su letra cuando le toca. Añade al constructor de las hebras un nuevo parámetro `miId` para establecer el turno de cada hebra. Cada hebra espera su turno mediante una instrucción de espera activa del tipo `while(turno! = miId) Thread.sleep(0)`.

2. a) Implementa el método **`def`** `periodico(t: Long)(b: =>Unit): Thread` que crea una hebra que ejecuta de forma indefinida el código `b` cada `t` milisegundos. Añade la función al **paquete objeto** desarrollado en clase para que puedas utilizarla en cualquier otro ejercicio.

   b) Crea dos hebras periódicas que escriban un mensaje en la pantalla cada 1000 y 3000 milisegundos, respectivamente.

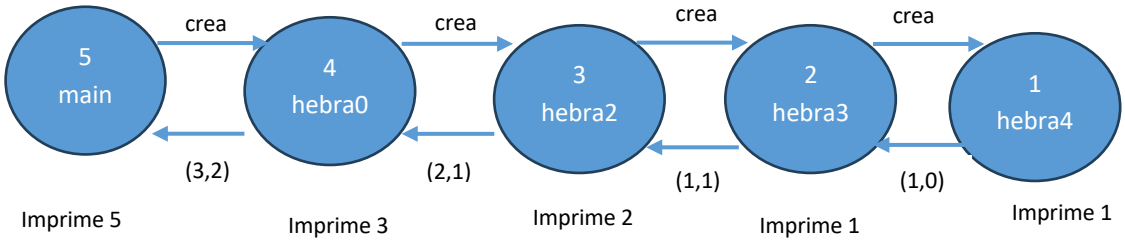
3. a) Define la función **`def`** `parallel[A,B](a: =>A, b: =>B):(A,B)` que crea dos hebras con los comportamientos dados por los parámetros `a` y `b`. El método devuelve los valores calculados por cada una de las hebras. Añade la función al **paquete objeto** desarrollado en clase para que puedas utilizarla en cualquier otro ejercicio. Nota: para inicializar una variable local capaz de registrar los valores calculados por una hebra, debes utilizar la instrucción `var a: A =` **`null`**`.asInstanceOf[A]`.

   b) Escribe un programa Scala que compruebe, si todas las componentes de una lista de booleanos son `true`. Para ello:
   a. Implementa el método **`def`** `todosTrue(inic:Int, fin:Int): Boolean` que devuelve `true` sii todas las componentes de `lista(inic..fin-1)` son `true`. Haz una implementación iterativa y otra recursiva de este método.
   b. Utiliza el método `parallel` para crear dos hebras que comprueben cada una de ellas si cada una de las mitades de la lista tiene todos sus componentes a `true`.
   c. Razona cómo se sincronizan la hebra principal y las dos hebras creadas en este ejercicio.

   Utiliza las funciones `fill` (de `List`) y `nextBoolean`, `nextInt` (de `scala.util.Random`) para crear listas aleatorias de booleanos mediante

```
List.fill(Random.nextInt(10))(Random.nextBoolean())
```

4. Escribe un programa Scala que **imprima por pantalla** los **n** primeros elementos de la serie de Fibonacci **$fib_0 = 0$**, **$fib_1 = 1$**, **$fib_2 = 1$**, **$fib_3 = 2$**, … Para ello, crea un pipeline de hebras como el que aparece en la figura para el caso **n = 5**.



Cada una de las hebras del pipeline llama a la función

```
def fibonacci(n: Int): (Int, Int)
```

con un valor **n > 1**. Esta función calcula el valor **fib_n** del modo siguiente: si **n = 1**, imprime **1** y devuelve (1,0); en otro caso, crea una nueva hebra que debe ejecutar la función `fibonacci(n-1)` y debe, por tanto, imprimir fib_{n-1} y devolver (fib_{n-1}, fib_{n-2}).

Nota: Este procedimiento para calcular los elementos de la serie de Fibonacci es muy ineficiente. El objetivo del ejercicio es mostrar cómo implementar una función como `fibonacci` que combina recursión y concurrencia, ya que cada llamada recursiva es ejecutada por una nueva hebra creada dinámicamente.

Si n = 7 la salida por pantalla del programa que se debe implementar debe ser parecida a:

main: fib(0) = 0
Thread-5: fib(1) = 1
Thread-4: fib(2) = 1
Thread-3: fib(3) = 2
Thread-2: fib(4) = 3
Thread-1: fib(5) = 5
Thread-0: fib(6) = 8
main: Fin del programa

5. Escribe un programa Scala que implemente una versión concurrente del algoritmo de *mergesort* (*ordenación por mezcla)*. El programa debe crear un árbol binario de hebras cuya profundidad depende del número de elementos a ordenar. Para ello debes implementar las funciones:

   a. `def mezclar(l1: List[Int], l2: List[Int]): List[Int]`

      que dadas dos listas ordenadas `l1` y `l2` construye una lista ordenada mezclando los elementos de `l1` y `l2`. Por ejemplo, dadas `l1 = List(1, 3, 5, 10)` y `l2 = List(2, 4, 7, 9)`, la función debe devolver la lista `List(1, 2, 3, 4, 5, 7, 9, 10)`.

b. **def** `ordenar(l: List[Int]): List[Int]`
que dadas una lista `l` la ordena utilizando una versión recursiva y concurrente del algoritmo *mergesort* del modo siguiente: si la lista `l` tiene un único elemento o está vacía, no hace nada (la lista ya está ordenada) ; ii) en otro caso, la función divide la lista `l` en dos mitades y crea dos hebras cada una de las cuales ejecuta el método ordenar sobre una de las dos mitades. Una vez que las hebras han ordenado las sublistas, la función las mezcla de forma ordenada y devuelve el resultado (la lista original ordenada).
Para implementar esta función puedes utilizar la función `splitAt(n)` de la clase `List` y la función `parallel` implementada en el ejercicio 3 de esta relación.

La lista aleatoria a ordenar puede crearse utilizando una instrucción del tipo
`List.fill(Random.nextInt(50))(Random.nextInt(100))`

Una ejecución de este programa debería producir una salida como

main: [40,5,34,5,35,25,73,22,39,55,58,7,21,2,97,55,53,19,78,99,35,68,21,45,58,29]
main: [2,5,5,7,19,21,21,22,25,29,34,35,35,39,40,45,53,55,55,58,58,68,73,78,97,99]