



GRADO EN INGENIERÍA INFORMÁTICA

PROGRAMACIÓN CONCURRENTE Y DE
TIEMPO REAL

PRÁCTICA 4

Análisis

Autor:

Raúl Arcos Herrera

Fecha:

11 de Noviembre de 2021

Índice

1. Ejercicio 1	2
1.1. tryThree.java	2
1.2. tryFour.java	3
2. Ejercicio 2: algDekker.java	3
3. Ejercicio 3: algEinsenberMcGuire.java	4
4. Ejercicio 4: algHyman.java	4

1. Ejercicio 1

1.1. tryThree.java

En el algoritmo *Third Attempt* añadido a la introducción de las variables *want* de *Second Attempt*, se considera que estos indicadores deberían ser parte de la sección crítica.

El comportamiento, tal y como nos informa la bibliografía, está sujeto a *deadlock*, siguiendo el siguiente escenario: El resultado es el algoritmo provocando que am-

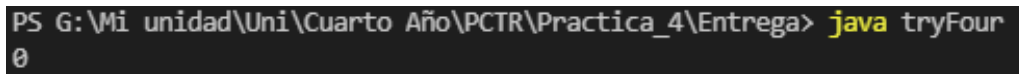
Proceso 1	Proceso 2	C1	C2
1.1: Sección no crítica	2.1 Sección no crítica	false	false
1.2: C1 <- true	2.1 Sección no crítica	false	false
1.2: C1 <- true	2.2 C2 <- true	false	false
1.3 Esperando a C2 <- false	2.2 C2 <- true	true	false
1.3 Esperando a C2 <- false	Esperando a C1	true	true

Cuadro 1: Comportamiento de tryThree.java

Los procesos esperen a que el otro se ejecute infinitamente, por lo que no llega a ejecutarse.

1.2. tryFour.java

A diferencia del algoritmo *Third Attempt*, en este caso el algoritmo puede ejecutarse, dando el resultado esperado. Aún dar el resultado esperado, esta solución fué descar-



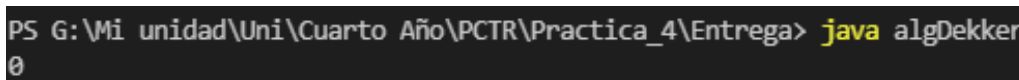
```
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java tryFour
0
```

Figura 1: Comportamiento de tryFour.java

tada debido a que no es posible asegurar que pueda ejecutarse de forma intervalada indefinidamente, tal y como cita en la bibliografía proporcionada.

2. Ejercicio 2: algDekker.java

Es una combinación de *First Attempt* y *Four Attempt*, de manera que cada proceso tiene el derecho de insistir en entrar, más que el derecho de entrar. El algoritmo de



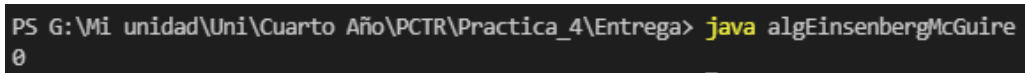
```
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algDekker
0
```

Figura 2: Comportamiento de algDekker.java

Dekker es correcto, satisface tanto el requerimiento de exclusión mutua como el requerimiento de progreso en la ejecución.

3. Ejercicio 3: algEinsenbergMcGuire.java

El comportamiento obtenido es el esperado, consta de 5 partes, cada una con su función única en la que se comprueba el permiso para entrar, si hay algún proceso en la sección crítica, y control de acceso, lo que lo convierte en un algoritmo que satisface tanto el requerimiento de exclusión mutua como el requerimiento de progreso en la ejecución.

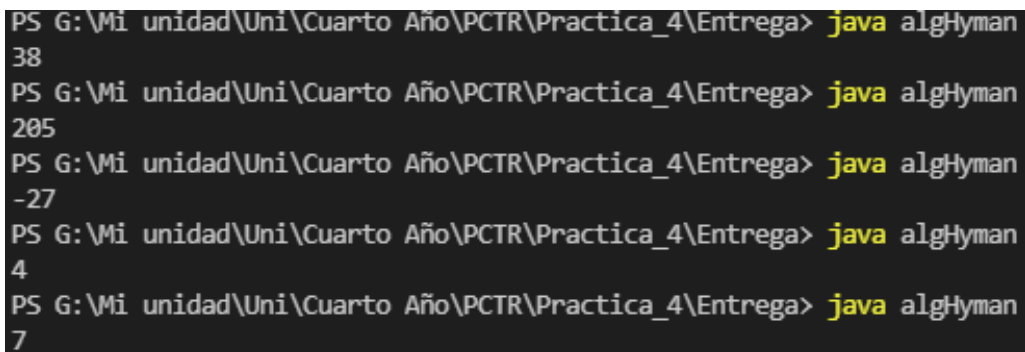


```
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algEinsenbergMcGuire
0
```

Figura 3: Comportamiento de algEinsenbergMcGuire.java

4. Ejercicio 4: algHyman.java

El algoritmo es incorrecto, el resultado obtenido no es el esperado y podemos ver resultados parecidos a los algoritmos programados en la práctica 2.



```
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algHyman
38
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algHyman
205
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algHyman
-27
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algHyman
4
PS G:\Mi unidad\Uni\Cuarto Año\PCTR\Practica_4\Entrega> java algHyman
7
```

Figura 4: Comportamiento de algHyman.java

Como podemos ver en *Primeras aproximaciones a la programación concurrente*:

```
P0                                P1
                                  states[1] = True
                                  ¿turn != 1? → True
                                  ¿states[0]? → False

states[0] = True
¿turn == 0? → True

                                  turn = 1
                                  ...
                                  ## BOOM! ##
```

Figura 5: Razón por lo que Hyman falla.