

# Programación Concurrente y de Tiempo Real

## Grado en Ingeniería Informática

### Asignación de Prácticas Número 4

Las primeras estrategias para el control de la exclusión mutua se basaron en algoritmos que compartían información utilizando variables comunes. Ello permitía que si una hebra deseaba ejecutar su sección crítica, pudiera saber qué hacían los demás antes de acceder a ella, y esperar en caso negativo mediante un bucle de espera activa, en el cual se está comprobando continuamente el estado de esas variables hasta encontrar la situación adecuada para el ingreso en la sección crítica. En esta práctica se le pide, al objeto de que se familiarice con esta clase de algoritmos de control de e.m. y la estrategia que emplean, que realice la implementación de varios de ellos. Documente todo su código con etiquetas (será sometido a análisis con `javadoc`). Si lo desea, puede también agrupar su código en un paquete de clases, aunque no es obligatorio.

## 1. Ejercicios

La literatura sobre programación concurrente siempre estudia el Algoritmo de *Dekker* y similares partiendo de un enfoque de refinamiento sucesivo de cuatro intentos incorrectos que utilizan alguna o ambas de las siguientes técnicas de sincronización, junto con el mecanismo de espera ocupada:

- ejecución de la sección crítica por turnos;
- ejecución de la sección crítica mediante *flags* de estado.

En las referencias bibliográficas propuestas para el Tema 3 del curso es posible encontrar descripciones en pseudocódigo de estos intentos incorrectos y de los algoritmos que funcionan, así como análisis de sus comportamientos y peculiaridades de cierta profundidad. Recomendamos por tanto la lectura, con carácter previo al desarrollo de la práctica, de al menos los capítulos dedicados a algoritmos de exclusión mutua con variables compartidas descritos en las siguientes obras:

- Ben-Ari, M. Principles of Concurrent and Distributed Programming, 2ª edición.

- Palma et al., Programación Concurrente, 2ª edición.

Durante la clase de prácticas se expondrán las dos primeras etapas del refinamiento sucesivo mediante código en Java que las soportan y que utilizan la primera la técnica de los turnos, y la segunda la técnica de los *flags* de estado.

1. Implemente, a partir de las referencias bibliográficas indicadas, las etapas tercera y cuarta del refinamiento sucesivo, y guárdelas en `tryThree.java` y `tryFour.java`; escriba un corto documento `analisis.pdf` utilizando  $\text{\LaTeX}$  que recoja el comportamiento obtenido y su interpretación del mismo. Utilice herencia de la clase `Thread`.

2. Nuevamente, a partir de las referencias bibliográficas indicadas, implemente en Java el algoritmo de *Dekker* para dos procesos. Guarde su trabajo en `algDekker.java` y añada al documento `analisis.pdf` su interpretación de los resultados de ejecución. Utilice herencia de la clase `Thread`.

3. Escriba un programa que implemente el Algoritmo de *Eisenberg-McGuire* para dos procesos. Cree dos hebras (que deberán compartir las variables de control comunes por implementación de la interfaz `Runnable`, y guarde su código en `algEisenbergMcGuire.java`. Utilice un ejecutor de tamaño fijo para procesar las tareas. Dedique ahora un apartado del documento `analisis.pdf` a este algoritmo que recoja el comportamiento obtenido y su interpretación del mismo.

4. Una solución incorrecta con variables comunes para el problema de la sección crítica fue en su día el algoritmo de *Hyman*. Impleméntelo utilizando tareas soportadas por la interfaz `Runnable` y analice su comportamiento en tiempo de ejecución. Guarde su código en `algHyman.java`. Dedique un apartado del documento `analisis.pdf` a este algoritmo que recoja el comportamiento obtenido y su interpretación del mismo.

## 2. Procedimiento de Entrega

PRODUCTOS A ENTREGAR:

- `tryThree.java`, `tryFour.java`, `algDekker.java`, `algEisenbergMcGuire.java`, `algHyman.java`, `analisis.pdf`.

MÉTODO DE ENTREGA: Tarea de Moodle.