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 varibles 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`).**

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 como lecturas del 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 la referencia de Ben-Ari, las etapas tercera y cuarta del refinamiento sucesivo, y guárdelas en `tryThree.java` y `tryFour.java`; escriba un corto documento `analisis.pdf` utilizando LaTeX que recoja el comportamiento obtenido y su interpretación del mismo. Utilice herencia de la clase `Thread`.

2. Nuevamente, a partir de la referencia de Ben-Ari, implemente en Java el algoritmo de *Dekker* para dos procesos. Guarde su trabajo en `algDekker.java` y añada al documento `analisis.pdf` su intepretación de los resultados de ejecución. Utilice herencia de la clase `Thread`.

3. Escriba un programa que implemente el algoritmo de *Peterson* para dos procesos (ver Ben-Ari). Cree dos hebras (que deberán compartir las variables de control comunes, utilice implementación de la interfaz `Runnable`, y guarde su código en `algPeterson.java`. Utilice en esta ocasión 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.

Notas de implementación:

- Codifigue los algoritmos de Dekker y *Peterson* utilizando bucles `while(true)` para el ciclo de vida principal del proceso, según se ilustra en el pseudocódigo de ambos algoritmos.
- Las secciones críticas imprimirán el identificador de las hebras mediante el método `Thread.getName()`,
- Si lo necesita, puede utilizar en sus códigos el método `Thread.yield()` donde lo crea oportuno.

2. Procedimiento de Entrega

PRODUCTOS A ENTREGAR:

- `tryThree.java`, `tryFour.java`, `algDekker.java`, `algPeterson.java`, `analisis.pdf`.

MÉTODO DE ENTREGA: Tarea de Moodle.