Prácticas Concurrencia y Distribución (22/23)

Arno Formella, Alba Nogueira Rodríguez, David Ruano Ordás

semana 10 abril – 14 abril

Práctica 7: Sincronización y exclusión

Objetivos: Objetos compartidos entre procesos con acceso sincronizado. Este ejercicio pretende reiterar el concepto de compartir objetos entre hilos y utilizar una referencia del objeto compartido para controlar el acceso de los procesos a las secciones críticas, es decir, a código que en un momento de tiempo, solamente un hilo puede ejecutar.

1. Implementa una clase `ClassA` que contenga un solo método `EnterAndWait()`. Este método debe hacer lo siguiente y en este orden:
   a) imprimir un mensaje indicando cual es el hilo que está comenzando a ejecutarlo;
   b) esperar unos segundos; y
   c) imprimir otro mensaje indicando el hilo que está acabando de ejecutar el método.
2. Implementa una clase `ClassB` que implemente `Runnable` y que se construya recibiendo como parámetro un objeto de la clase `ClassA`. Haz que en su método `run()` simplemente llame al método `EnterAndWait()` del objeto con cual ha sido construido.
3. Implementa una clase principal con un método `Main` en el que se crea un único objeto de la clase `ClassA` y varios objetos de la clase `ClassB` a los que se les pasa a todos como parámetro el objeto de la clase `ClassA`. Después se crearán y ejecutarán el mismo número de hilos (objetos de la clase `Thread`) que de objetos de tipo `ClassB` tengamos pasándoles como parámetros los objetos de la clase `ClassB`, de tal forma que cada método `run()` de cada objeto de la clase `ClassB` se ejecute en un hilo diferente.
4. Análiza: ¿Cuál es el resultado? ¿Cuántos hilos pueden estar simultáneamente ejecutando el método `EnterAndWait()`?

5. Limitamos ahora el acceso a la sección crítica (el método `EnterAndWait()` en nuestro caso) utilizando sincronización (el mecanismo empleando `synchronized` o `lock` que vimos la semana pasada). Realiza dos versiones: una con `synchronized` y otra con un cerrojo (`lock`). Explica lo que observas.

6. Aumenta el código de las dos clases `ClassA` y `ClassB` que ambas clases contengan un contador con el objetivo que el contador de la clase compartida se decrementa y el contador de la class no compartida se incrementa.

7. Modifica el método `EnterAndWait()` para que sea un `EnterAndDecrement()`, es decir, un método sincronizado que decremente el contador y devuelve `false` si ha llegado a 0 (`true` si no).

8. Usa ahora un bucle en el método `run()` de la clase `ClassB` que entre tantas veces en el método `EnterAndDecrement()` que sea posible y que cuente las veces conseguidos (con su contador), por ejemplo con un código como: `while(A.EnterAndDecrement()) { cnt.Increment(); }`

9. Imprime al final de `run()` cuentas veces se ha conseguido decrementar el contador compartido realizando experimentos con diferentes números de hilos y inicializaciones del contador compartido.

10. Averiguar el funcionamento de los métodos `wait()` y `notify()` (respectivamente `notifyAll()`), sobre todo: para que sirven...

11. Re-edita el código con las dos clases `ClassA` (objeto compartido) y `ClassB` (hilos trabajadores) y el método `EnterAndDecrement` con `synchronized` e incluye en el sitio adecuado una notificación (usando `notify()` o `notifyAll()` y a continuación una espera (usando `wait()`).

    - El propósito de está modificación es aliviar el hecho que observamos la semana pasda que en tal versión sin espera unos pocos hilos realizan todos los decrementos mientras otros nuncan llegan a hacer nada. (¡Cuidado que tu programa sigue funcionando quizá necesitas una condición antes de ir a `wait()` sino vas a durmir sin que nadie te despierta!)
    - ¿Observas una diferencia respecto a la distribución del *trabajo*? Analiza los valores mínimo, máximo, y varianza de los accesos.
    - ¿Observas una diferencia significativa entre el uso de `notify()` y `notifyAll()`?